



CISC 3350

Workstation Programming OER Course Packet

Compiled by [Miriam Briskman](#)
CISC Department, Brooklyn College

Note: Each of the online sources included in this packet features its own usage permissions and licenses, which the readers shall follow. The packet itself, the order in which the sources are present herein, and lists of citations of the sources, however, are licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.



This packet compiled by [Miriam Briskman](#) is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](#).

Contents

| | |
|--|-------------|
| Topic 1: Intro to the Linux/UNIX OS Programming | 10 |
| Topic 2: Useful Linux Commands and Text Editors | 163 |
| Topic 3: Review of the C Language | 383 |
| Topic 4: File I/O | 457 |
| Topic 5: Buffered File I/O | 582 |
| Topic 6: Vectored I/O & Memory-Mapping | 685 |
| Topic 7: Processes | 769 |
| Topic 8: Threads | 946 |
| Topic 9: CPU Scheduling | 1034 |
| Topic 10: Signals | 1125 |
| Topic 11: Memory Management | 1237 |
| Topic 12: File and Directory Management | 1276 |

Detailed Contents

| | |
|--|-----------|
| Topic 1: Intro to the Linux/UNIX OS Programming | 10 |
| 1.1 Linux - Britannica Online Encyclopedia | 12 |
| 1.2 What is Linux - Opensource.com | 14 |
| 1.3 Lumen Learning - Personal Computers: Workstation | 19 |
| 1.4 Linux vs. Unix - Opensource.com | 21 |
| 1.5 System Programming | 32 |
| 1.6 Linux Syscall | 41 |
| 1.7 glibc(7) - overview of standard C libraries - Linux man page | 43 |
| 1.8 How to check glibc version on Linux | 44 |

| | | |
|--|--|------------|
| 1.9 | Guide to GNU C Compiler - Opensource.com | 48 |
| 1.10 | A 10-minute guide to the Linux ABI - Opensource.com | 57 |
| 1.11 | What is POSIX - Richard Stallman explains - Opensource.com | 67 |
| 1.12 | ROSEdu Techblog - Unix standards and implementations. Unix portability | 77 |
| 1.13 | LSB (Linux Standard Base) Introduction | 83 |
| 1.14 | An introduction to Linux filesystems - Opensource.com | 85 |
| 1.15 | An introduction to Linux's EXT4 filesystem - Opensource.com | 99 |
| 1.16 | ln(1) - Linux man page | 116 |
| 1.17 | A user's guide to links in the Linux filesystem - Opensource.com | 120 |
| 1.18 | Memory Overview | 133 |
| 1.19 | Stack Layout | 147 |
| 1.20 | Materials for Topic 1: Intro to the Linux/Unix Os Programming | 153 |
| 1.21 | helloworld.c | 158 |
| 1.22 | errno.c | 159 |
| Topic 2: Useful Linux Commands and Text Editors | | 163 |
| 2.1 | pwd(1) - Linux man page | 165 |
| 2.2 | whoami(1) - Linux man page | 168 |
| 2.3 | ls(1) - Linux man page | 171 |
| 2.4 | cd(1p) - Linux man page | 179 |
| 2.5 | mkdir(1) - Linux man page | 188 |
| 2.6 | touch(1) - Linux man page | 191 |
| 2.7 | rmdir(1) - Linux man page | 195 |
| 2.8 | rm(1) - Linux man page | 198 |
| 2.9 | cat(1) - Linux man page | 202 |
| 2.10 | clear(1) - Linux man page | 205 |
| 2.11 | cp(1) - Linux man page | 209 |
| 2.12 | mv(1) - Linux man page | 215 |
| 2.13 | sudo(8) - Linux man page | 219 |
| 2.14 | su(1) - Linux man page | 240 |
| 2.15 | man(1) - Linux man page | 247 |
| 2.16 | info(1) - read Info documents - Linux man page | 267 |
| 2.17 | ping(8) - Linux man page | 270 |
| 2.18 | wget(1) - Linux man page | 283 |
| 2.19 | grep(1) - Linux man page | 330 |
| 2.20 | gcc(1) - Linux man page | 348 |
| 2.21 | gdb(1) - Linux man page | 352 |
| 2.22 | exit(1p) - Linux man page | 361 |
| 2.23 | Materials for Topic 2: Useful Linux Commands and Text Editors | 366 |
| Topic 3: Review of the C Language | | 383 |
| 3.1 | C (Guide and Reference) | 385 |
| 3.2 | C Reference Card | 407 |
| 3.3 | Materials for Topic 3: Review of the C Language | 409 |
| 3.4 | variables.c | 411 |

| | | |
|-----------------------------------|---|------------|
| 3.5 | EXIT_SUCCESS(3const) - EXIT_SUCCESS and EXIT_FAILURE - Linux man page | 416 |
| 3.6 | conditions.c | 418 |
| 3.7 | loops.c | 420 |
| 3.8 | char_arrays_and_pointers.c | 423 |
| 3.9 | NULL(3const) - Linux man page | 432 |
| 3.10 | files.c | 435 |
| 3.11 | EOF(3const) - Linux man page | 441 |
| 3.12 | functions.c | 443 |
| 3.13 | structs.c | 448 |
| 3.14 | error_checking.c | 451 |
| 3.15 | command_line_arguments.c | 453 |
| Topic 4: File I/O | | 457 |
| 4.1 | stdin(3) - stdin, stdout, and stderr - Linux man page | 459 |
| 4.2 | errno(3) - Linux man page | 462 |
| 4.3 | open(2) - open() and creat() - Linux man page | 479 |
| 4.4 | read(2) - Linux man page | 503 |
| 4.5 | write(2) - Linux man page | 508 |
| 4.6 | fsync(2) - fsync() and fdatasync() - Linux man page | 514 |
| 4.7 | sync(2) - Linux man page | 518 |
| 4.8 | close(2) - Linux man page | 522 |
| 4.9 | lseek(2) - Linux man page | 527 |
| 4.10 | pread(2) - pread() and pwrite() - Linux man page | 532 |
| 4.11 | truncate(2) - truncate() and ftruncate() - Linux man page | 536 |
| 4.12 | select(2) - Linux man page | 541 |
| 4.13 | poll(2) - Linux man page | 553 |
| 4.14 | Materials for Topic 4: File I/O | 564 |
| 4.15 | file_io_syscalls.c | 566 |
| 4.16 | select_syscall_example.c | 577 |
| 4.17 | poll_syscall_example.c | 580 |
| Topic 5: Buffered File I/O | | 582 |
| 5.1 | time(1) - Linux man page | 584 |
| 5.2 | dd(1) - Linux man page | 590 |
| 5.3 | stat(1) - Linux man page | 595 |
| 5.4 | stdio(3) - standard i/o library - Linux man page | 601 |
| 5.5 | fopen(3) - Linux man page | 606 |
| 5.6 | fdopen(3p) - Linux man page | 613 |
| 5.7 | fclose(3) - Linux man page | 618 |
| 5.8 | fcloseall(3) - Linux man page | 621 |
| 5.9 | fgetc(3) - Linux man page | 623 |
| 5.10 | ungetc(3p) - Linux man page | 626 |
| 5.11 | fgets(3p) - Linux man page | 630 |
| 5.12 | fread(3) - fread() and fwrite() - Linux man page | 634 |
| 5.13 | ferror(3) - ferror(), feof(), and clearerr() - Linux man page | 638 |

| | | |
|---|--|------------|
| 5.14 | <code>fputc(3p)</code> - Linux man page | 641 |
| 5.15 | <code>fputs(3p)</code> - Linux man page | 645 |
| 5.16 | <code>fseek(3)</code> - Linux man page | 649 |
| 5.17 | <code>ftell(3p)</code> - Linux man page | 653 |
| 5.18 | <code>fsetpos(3p)</code> - Linux man page | 657 |
| 5.19 | <code>rewind(3p)</code> - Linux man page | 662 |
| 5.20 | <code>fflush(3)</code> - Linux man page | 665 |
| 5.21 | <code>fileno(3)</code> - Linux man page | 668 |
| 5.22 | <code>setvbuf(3p)</code> - Linux man page | 671 |
| 5.23 | <code>flockfile(3)</code> - <code>flockfile()</code> , <code>ftrylockfile()</code> , and <code>funlockfile()</code> - Linux man page | 675 |
| 5.24 | Materials for Topic 5: Buffered File I/O | 679 |
| 5.25 | <code>buffered_io.c</code> | 682 |
| Topic 6: Vectored I/O & Memory-Mapping | | 685 |
| 6.1 | <code>readv(2)</code> - <code>readv()</code> and <code>writev()</code> - Linux man page | 687 |
| 6.2 | <code>mmap(2)</code> - <code>mmap()</code> and <code>munmap()</code> - Linux man page | 694 |
| 6.3 | <code>sysconf(3)</code> - Linux man page | 710 |
| 6.4 | <code>getpagesize(2)</code> - Linux man page | 718 |
| 6.5 | <code>fstat(3p)</code> - Linux man page | 721 |
| 6.6 | <code>putchar(3p)</code> - Linux man page | 726 |
| 6.7 | <code>mremap(2)</code> - Linux man page | 729 |
| 6.8 | <code>mprotect(2)</code> - Linux man page | 735 |
| 6.9 | <code>msync(2)</code> - Linux man page | 742 |
| 6.10 | <code>madvise(2)</code> - Linux man page | 745 |
| 6.11 | Materials for Topic 6: Vectored I/O & Memory-Mapping | 759 |
| 6.12 | <code>writev_example.c</code> | 761 |
| 6.13 | <code>readv_example.c</code> | 763 |
| 6.14 | <code>mmap_example.c</code> | 766 |
| Topic 7: Processes | | 769 |
| 7.1 | <code>getpid(2)</code> - <code>getpid()</code> and <code>getppid()</code> - Linux man page | 771 |
| 7.2 | <code>exec(3)</code> - <code>exec()</code> function family - Linux man page | 774 |
| 7.3 | <code>execve(2)</code> - Linux man page | 780 |
| 7.4 | <code>fork(2)</code> - Linux man page | 794 |
| 7.5 | <code>exit(3)</code> - Linux man page | 801 |
| 7.6 | <code>_exit(2)</code> - Linux man page | 805 |
| 7.7 | <code>atexit(3)</code> - Linux man page | 809 |
| 7.8 | <code>wait(2)</code> - <code>wait()</code> and <code>waitpid()</code> - Linux man page | 813 |
| 7.9 | <code>system(3)</code> - Linux man page | 824 |
| 7.10 | <code>sh(1p)</code> - Linux man page | 829 |
| 7.11 | <code>passwd(1)</code> - Linux man page | 858 |
| 7.12 | <code>setuid(2)</code> - Linux man page | 865 |
| 7.13 | <code>setgid(2)</code> - Linux man page | 869 |
| 7.14 | <code>seteuid(2)</code> - Linux man page | 872 |

| | | |
|--|--|-------------|
| 7.15 | setegid(3p) - Linux man page | 875 |
| 7.16 | setuid(2) - setuid and seteuid - Linux man page | 878 |
| 7.17 | getuid(2) - getuid and geteuid - Linux man page | 882 |
| 7.18 | getgid(2) - getgid and getegid - Linux man page | 885 |
| 7.19 | setsid(2) - Linux man page | 888 |
| 7.20 | getsid(2) - Linux man page | 891 |
| 7.21 | setpgid(2) - setpgid and getpgid - Linux man page | 894 |
| 7.22 | dup(2) - Linux man page | 899 |
| 7.23 | daemon(3) - Linux man page | 904 |
| 7.24 | ps(1) - Linux man page | 907 |
| 7.25 | Materials for Topic 7: Processes | 932 |
| 7.26 | child_creation_example.c | 936 |
| 7.27 | wait_syscall_example.c | 940 |
| 7.28 | system_syscall_implementation.c | 942 |
| 7.29 | daemon_creation.c | 944 |
| Topic 8: Threads | | 946 |
| 8.1 | Threads and Concurrent Programming | 948 |
| 8.2 | Linux Kernel Support for Threads - Programming in Linux | 958 |
| 8.3 | Threads Programming in Linux - Examples - Programming in Linux | 962 |
| 8.4 | pthread_create(3) - Linux man page | 969 |
| 8.5 | pthread_self(3) - Linux man page | 978 |
| 8.6 | pthread_equal(3) - Linux man page | 981 |
| 8.7 | pthread_exit(3) - Linux man page | 984 |
| 8.8 | pthread_cancel(3) - Linux man page | 987 |
| 8.9 | pthread_setcancelstate(3) - pthread_setcancelstate() and pthread_setcanceltype() - Linux man page | 992 |
| 8.10 | pthread_join(3) - Linux man page | 997 |
| 8.11 | pthread_join(3) - Linux man page | 1001 |
| 8.12 | pthread_detach(3) - Linux man page | 1005 |
| 8.13 | pthread_mutex_lock(3p) - pthread_mutex_lock() and pthread_mutex_unlock() - Linux man page | 1008 |
| 8.14 | Materials for Topic 8: Threads | 1015 |
| 8.15 | unsynced_threads_experiment.c | 1017 |
| 8.16 | synced_threads_experiment.c | 1020 |
| 8.17 | unsynced_bank_withdraw.c | 1023 |
| 8.18 | synced_bank_withdraw.c | 1027 |
| 8.19 | threads_example.c | 1032 |
| Topic 9: CPU Scheduling | | 1034 |
| 9.1 | CFS: Completely fair process scheduling in Linux - Opensource.com | 1035 |
| 9.2 | sched_yield(2) - Linux man page | 1045 |
| 9.3 | nice(2) - Linux man page | 1048 |
| 9.4 | getpriority(2) - getpriority() and setpriority() - Linux man page | 1051 |

| | | |
|--|---|-------------|
| 9.5 | <code>sched_setaffinity(2)</code> - <code>sched_setaffinity()</code> and <code>sched_getaffinity()</code> - Linux man page | 1055 |
| 9.6 | <code>CPU_SET(3)</code> - CPU macro family - Linux man page | 1063 |
| 9.7 | <code>sched(7)</code> - overview of CPU scheduling - Linux man page | 1070 |
| 9.8 | <code>sched_setscheduler(2)</code> - <code>sched_setscheduler()</code> and <code>sched_getscheduler()</code> - Linux man page | 1087 |
| 9.9 | <code>sched_setparam(2)</code> - <code>sched_setparam()</code> and <code>sched_getparam()</code> - Linux man page | 1092 |
| 9.10 | <code>sched_get_priority_max(2)</code> - <code>sched_get_priority_max()</code> and <code>sched_get_priority_min()</code> - Linux man page | 1095 |
| 9.11 | <code>sched_rr_get_interval(2)</code> - Linux man page | 1098 |
| 9.12 | <code>getrlimit(2)</code> - <code>getrlimit()</code> and <code>setrlimit()</code> - Linux man page | 1101 |
| 9.13 | Materials for Topic 9: CPU Scheduling | 1114 |
| 9.14 | <code>scheduling_examples.c</code> | 1119 |
| Topic 10: Signals | | 1125 |
| 10.1 | <code>signal(7)</code> - overview of signals - Linux man page | 1127 |
| 10.2 | <code>signal(2)</code> - Linux man page | 1142 |
| 10.3 | <code>pause(2)</code> - Linux man page | 1147 |
| 10.4 | <code>strsignal(3)</code> - <code>strsignal()</code> and <code>sys_siglist</code> array - Linux man page | 1149 |
| 10.5 | <code>psignal(3)</code> - Linux man page | 1153 |
| 10.6 | <code>kill(2)</code> - Linux man page | 1156 |
| 10.7 | <code>raise(3)</code> - Linux man page | 1160 |
| 10.8 | <code>killpg(3)</code> - Linux man page | 1163 |
| 10.9 | <code>signal-safety(7)</code> - reentrant functions overview - Linux man page | 1166 |
| 10.10 | <code>sigsetops(3)</code> - <code>sigemptyset</code> , <code>sigfillset</code> , <code>sigaddset</code> , <code>sigdelset</code> , <code>sigismember</code> , <code>sigisemptyset</code> , <code>sigorset</code> , and <code>sigandset</code> - Linux man page | 1174 |
| 10.11 | <code>sigprocmask(2)</code> - Linux man page | 1178 |
| 10.12 | <code>sigpending(2)</code> - Linux man page | 1183 |
| 10.13 | <code>sigsuspend(2)</code> - Linux man page | 1186 |
| 10.14 | <code>sigaction(2)</code> - Linux man page | 1189 |
| 10.15 | <code>sigqueue(3)</code> - Linux man page | 1206 |
| 10.16 | <code>kill(1)</code> - command to terminate a process - Linux man page | 1210 |
| 10.17 | Materials for Topic 10: Signals | 1216 |
| 10.18 | <code>division_by_0.c</code> | 1219 |
| 10.19 | <code>handling_sigint.c</code> | 1221 |
| 10.20 | <code>more_signals.c</code> | 1223 |
| 10.21 | <code>sigaction_example.c</code> | 1225 |
| 10.22 | <code>issuing_signals.c</code> | 1227 |
| 10.23 | <code>payload_sending.c</code> | 1232 |
| Topic 11: Memory Management | | 1237 |
| 11.1 | <code>malloc(3)</code> - Linux man page | 1238 |
| 11.2 | <code>calloc(3p)</code> - Linux man page | 1245 |
| 11.3 | <code>realloc(3p)</code> - Linux man page | 1249 |

| | | |
|--|---|-------------|
| 11.4 | <code>free(3p)</code> - Linux man page | 1254 |
| 11.5 | <code>memset(3)</code> - Linux man page | 1257 |
| 11.6 | <code>memcpy(3)</code> - Linux man page | 1259 |
| 11.7 | <code>memmove(3)</code> - Linux man page | 1262 |
| 11.8 | <code>strcpy(3)</code> - Linux man page | 1264 |
| 11.9 | <code>memchr(3)</code> - Linux man page | 1267 |
| 11.10 | Materials for Topic 11: Memory Management | 1270 |
| 11.11 | <code>memory_functions.c</code> | 1271 |
| Topic 12: File and Directory Management | | 1276 |
| 12.1 | <code>inode(7)</code> - file inode information - Linux man page | 1279 |
| 12.2 | <code>stat(2)</code> - <code>stat()</code> , <code>fstat()</code> , and <code>lstat()</code> - Linux man page | 1287 |
| 12.3 | <code>stat(3type)</code> - file status structure - Linux man page | 1295 |
| 12.4 | <code>chmod(2)</code> - <code>chmod()</code> and <code>fchmod()</code> - Linux man page | 1299 |
| 12.5 | <code>chown(2)</code> - <code>chown()</code> , <code>fchown()</code> , and <code>lchown()</code> - Linux man page | 1305 |
| 12.6 | <code>getxattr(2)</code> - <code>getxattr()</code> , <code>fgetxattr()</code> , and <code>lgetxattr()</code> - Linux man page | 1313 |
| 12.7 | <code>setxattr(2)</code> - <code>setxattr()</code> , <code>fsetxattr()</code> , and <code>lsetxattr()</code> - Linux man page | 1317 |
| 12.8 | <code>listxattr(2)</code> - <code>listxattr()</code> , <code>flistxattr()</code> , and <code>llistxattr()</code> - Linux man page | 1321 |
| 12.9 | <code>removexattr(2)</code> - <code>removexattr()</code> , <code>fremovexattr()</code> , and <code>lremovexattr()</code> - Linux man page | 1328 |
| 12.10 | <code>getcwd(3)</code> - Linux man page | 1331 |
| 12.11 | <code>chdir(2)</code> - <code>chdir()</code> and <code>fchdir()</code> - Linux man page | 1337 |
| 12.12 | <code>passwd(5)</code> - <code>/etc/passwd</code> file - Linux man page | 1340 |
| 12.13 | <code>getent(1)</code> - Linux man page | 1344 |
| 12.14 | <code>mkdir(2)</code> - Linux man page | 1349 |
| 12.15 | <code>umask(1p)</code> - Linux man page | 1354 |
| 12.16 | <code>rmdir(2)</code> - Linux man page | 1362 |
| 12.17 | <code>opendir(3)</code> - Linux man page | 1365 |
| 12.18 | <code>dirfd(3)</code> - Linux man page | 1369 |
| 12.19 | <code>readdir(3)</code> - Linux man page | 1372 |
| 12.20 | <code>closedir(3)</code> - Linux man page | 1378 |
| 12.21 | <code>link(2)</code> - Linux man page | 1381 |
| 12.22 | <code>symlink(2)</code> - Linux man page | 1388 |
| 12.23 | <code>unlink(2)</code> - Linux man page | 1393 |
| 12.24 | <code>remove(3)</code> - Linux man page | 1399 |
| 12.25 | <code>rename(2)</code> - Linux man page | 1402 |
| 12.26 | <code>ioctl(2)</code> - Linux man page | 1410 |
| 12.27 | <code>less(1)</code> - Linux man page | 1414 |
| 12.28 | Materials for Topic 12: File and Directory Management | 1417 |
| 12.29 | <code>get_file_size.c</code> | 1422 |
| 12.30 | <code>find_file_type.c</code> | 1424 |
| 12.31 | <code>permissions_changing.c</code> | 1426 |

| | | |
|-------|------------------------------------|------|
| 12.32 | <code>extended_attributes.c</code> | 1431 |
| 12.33 | <code>switch_directories.c</code> | 1434 |
| 12.34 | <code>my_ls.c</code> | 1437 |
| 12.35 | <code>adding_links.c</code> | 1439 |
| 12.36 | <code>eject_cd_rom.c</code> | 1443 |

Topic 1: Intro to the Linux/UNIX OS Programming

Note: A copy of each one of the sources listed below is included in this packet. To jump to the location of a source copy, click the page link under “Page”.

In case a source isn’t included (e.g., a YouTube video,) click the ↗ (external link) symbol under “Page” to open the external source in a browser window.

| # | Citation & Source Link | Page |
|----|--|------|
| 1 | Britannica, The Editors of Encyclopaedia. "Linux". <i>Encyclopedia Britannica</i> , 18 Jan. 2024. Accessed 19 January 2024. URL: https://www.britannica.com/technology/Linux | 12 |
| 2 | "What Is Linux?" <i>Opensource.Com</i> URL: https://opensource.com/resources/linux | 14 |
| 3 | "Personal Computers: Workstation." <i>Information Literacy</i> . Lumen Learning. URL: https://courses.lumenlearning.com/suny-informationliteracy/chapter/personal-computers/#:~:text=Workstation | 19 |
| 4 | Estes, Phil. "Linux vs. Unix: What's the Difference?" <i>Opensource.Com</i> , 2018. URL: https://opensource.com/article/18/5/differences-between-linux-and-unix | 21 |
| 5 | Devopedia. 2022. "Systems Programming." Version 4, May 30. URL: https://devopedia.org/systems-programming | 32 |
| 6 | Rosenberg, Burt. <i>Linux Syscall</i> , University of Miami, 9 Sept. 2020. URL: https://www.cs.miami.edu/home/burt/learning/Csc421.151/workbook/linux_syscall.html | 41 |
| 7 | "Glibc(7): Overview of Standard C Libraries on - Linux Man Page." <i>die.net</i> , 2009. URL: https://linux.die.net/man/7/glibc | 43 |
| 8 | Nanni, Dan. "How to Check Glibc Version on Linux." How to Check Glibc Version on Linux, <i>Xmodulo.com</i> , 9 July 2020. URL: https://www.xmodulo.com/check-glibc-version-linux.html | 44 |
| 9 | Huttanagoudar, Jayashree. "A Programmer's Guide to GNU C Compiler." <i>Opensource.Com</i> , 20 May 2022. URL: https://opensource.com/article/22/5/gnu-c-compiler | 48 |
| 10 | Chaiken, Alison. "A 10-Minute Guide to the Linux Abi." <i>Opensource.Com</i> , 6 Dec. 2022. URL: https://opensource.com/article/22/12/linux-abi | 57 |
| 11 | Kenlon, Seth. "What Is POSIX? Richard Stallman Explains." <i>Opensource.Com</i> , 15 July 2019. URL: https://opensource.com/article/19/7/what-posix-richard-stallman-explains | 67 |
| 12 | Goia, Alexandru. "Unix Standards and Implementations. Unix Portability." <i>ROSEdu Techblog - Unix Standards and Implementations. Unix Portability</i> , 2 Feb. 2014. URL: https://techblog.rosedu.org/unix-standards.html | 77 |

| # | Citation & Source Link | Page |
|----|---|------|
| 13 | “LSB Introduction.” Lsb:Lsb-Introduction [Wiki], The Linux Foundation, 9 July 2016. URL: https://wiki.linuxfoundation.org/lsb/lsb-introduction | 83 |
| 14 | Both, David. “An Introduction to Linux Filesystems.” <i>Opensource.Com</i> , 31 Oct. 2016. URL: https://opensource.com/life/16/10/introduction-linux-filesystems | 85 |
| 15 | Both, David. “An Introduction to Linux’s Ext4 Filesystem.” <i>Opensource.Com</i> , 25 May 2017. URL: https://opensource.com/article/17/5/introduction-ext4-filesystem | 99 |
| 16 | Parker, Mike, and David MacKenzie. “ln(1) - Linux Man Page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/ln.1.html | 116 |
| 17 | Both, David. “A User’s Guide to Links in the Linux Filesystem.” <i>Opensource.Com</i> , 22 June 2017. URL: https://opensource.com/article/17/6/linking-linux-filesystem | 120 |
| 18 | Tychonievich, Luther. “An Overview of Memory.” CS 2130, University of Virginia, 2023. URL: https://www.cs.virginia.edu/~jh2jf/courses/cs2130/spring2023/readings/memory.html | 133 |
| 19 | Rosenberg, Burt. <i>Stack Layouts: with a look forward to virtual addressing</i> , University of Miami, Sept. 2015. URL: https://www.cs.miami.edu/home/burt/learning/Csc421.171/workbook/stack-memory.html | 147 |
| 20 | Briskman, Miriam. “Materials for Topic 1: Intro to the Linux/Unix Os Programming.” <i>Topic 1: Intro to the Linux/UNIX OS Programming — CISC 3350 Materials</i> , 2023. URL: www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_01/ | 153 |
| 21 | Briskman, Miriam. “Helloworld.c .” (C source code) 6 Jan. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_01/helloworld.c | 158 |
| 22 | Briskman, Miriam. “errno.c .” (C source code) 8 Feb. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_01/errno.c | 159 |

Linux

Linux, computer operating system created in the early 1990s by Finnish software engineer Linus Torvalds and the Free Software Foundation (FSF).

While still a student at the University of Helsinki, Torvalds started developing Linux to create a system similar to MINIX, a UNIX operating system. In 1991 he released version 0.02; Version 1.0 of the Linux kernel, the core of the operating system, was released in 1994. About the same time, American software developer Richard Stallman and the FSF made efforts to create an open-source UNIX-like operating system called GNU. In contrast to Torvalds, Stallman and the FSF started by creating utilities for the operating system first. These utilities were then added to the Linux kernel to create a complete system called GNU/Linux, or, less precisely, just Linux.

Linux grew throughout the 1990s because of the efforts of hobbyist developers. Although Linux is not as user-friendly as the popular Microsoft Windows and Mac OS operating systems, it is an efficient and reliable system that rarely crashes. Combined with Apache, an open-source Web server, Linux accounts for most of the servers used on the Internet. Because it is open-source, and thus modifiable for different uses, Linux is popular for systems as diverse as cellular telephones and supercomputers. Android, Google's operating system for mobile devices, has at its core a modified Linux kernel, and Chrome OS, Google's operating system that uses the Chrome browser, is also Linux-based. The addition of user-friendly desktop environments, office suites, Web browsers, and even games helped to increase Linux's popularity and make it more suitable for home and office desktops. New distributions (packages of Linux software) have been created since the 1990s. Some of the more well-known distributions include MX Linux, Manjaro, Linux Mint, and Ubuntu.

The Editors of Encyclopaedia Britannica This article was most recently revised and updated by Erik Gregersen.

Citation Information

Article Title: Linux

Website Name: Encyclopaedia Britannica

Publisher: Encyclopaedia Britannica, Inc.

Date Published: 18 January 2024

URL: <https://www.britannica.com><https://www.britannica.com/technology/Linux>

Access Date: January 19, 2024



LOG IN



opensource.com

What is Linux?



Image by: Opensource.com

Linux is the best-known and most-used [open source](#) operating system. As an operating system, Linux is software that sits underneath all of the other software on a computer, receiving requests from those programs and relaying these requests to the computer's hardware.

How does Linux differ from other operating systems?

In many ways, Linux is similar to other operating systems you may have used before, such as Windows, macOS (formerly OS X), or iOS. Like other operating systems, Linux has a graphical interface, and the same types of software you are accustomed to, such as word processors, photo editors, video editors, and so on. In many cases, a software's creator may have made a Linux version of the same program you use on other systems. In short: if you can use a computer or other electronic device, you can use Linux.

But Linux also is different from other operating systems in many important ways. First, and perhaps most importantly, Linux is open source software. The code used to create Linux is free and available to the public to view, edit, and—for users with the appropriate skills—to contribute to.

Linux is also different in that, although the core pieces of the Linux operating system are generally common, there are many *distributions* of Linux, which include different software options. This means that Linux is incredibly customizable, because not just applications, such as word processors and web browsers, can be swapped out. Linux users also can choose core components, such as which system displays graphics, and other user-interface components.

Who uses Linux?

You probably already use Linux, whether you know it or not. Depending on which user survey you look at, between one- and two-thirds of the webpages on the Internet are generated by servers running Linux.

Companies and individuals choose Linux for their servers because it's secure, flexible, and you can receive excellent support from a large community of users, in addition to companies like Canonical, SUSE, and Red Hat, each of which offer commercial support.

Many devices you probably own, such as Android phones and tablets and Chromebooks, digital storage devices, personal video recorders, cameras, wearables, and more, also run Linux. Your car has Linux running under the hood. Even Microsoft Windows features Linux components, as part of the [Windows Subsystem for Linux \(WSL\)](#).

Who “owns” Linux?

By virtue of its open source licensing, Linux is freely available to anyone. However, the trademark on the name “Linux” rests with its creator, Linus Torvalds. The source code for Linux is under copyright by its many individual authors, and licensed under the [GPLv2 license](#).

The term “Linux” technically refers to just the Linux kernel. Most people refer to the entire operating system as “Linux” because to most users an OS includes a bundle of programs, tools, and services (like a desktop, clock, an application menu, and so on). Some people, particularly members of the [Free Software Foundation](#), refer to this collection as GNU/Linux, because many vital tools included are GNU components. However, not all Linux installations use GNU components as a part of the operating system: [Android](#), for example, uses a Linux kernel but relies very little on GNU tools.

What is the difference between Unix and Linux?

You may have heard of Unix, which is an operating system developed in the 1970s at Bell Labs by Ken Thompson, Dennis Ritchie, and others. Unix and Linux are similar in many ways, and in fact, Linux was originally created to be indistinguishable from Unix. Both have similar tools for interfacing with the system, programming tools, filesystem layouts, and other key components. However, not all Unices are free and open source.

Over the years, a number of different operating systems have been created that attempted to be “unix-like” or “unix-compatible,” but Linux has been the most successful, far surpassing its predecessors in popularity.

How was Linux created?

Linux was created in 1991 by Linus Torvalds, a then-student at the University of Helsinki. Torvalds built Linux as a free and open source alternative to Minix, another Unix clone that was predominantly used in academic settings. He originally intended to name it “Freax,” but the administrator of the server Torvalds used to distribute the original code named his directory “Linux” after a combination of Torvalds’ first name and the word Unix, and the name stuck.

Linux cheat sheets

- 👉 [Linux networking](#)
- 👉 [SELinux](#)
- 👉 [Advanced Linux commands for developers](#)
- 👉 [Firewalls](#)

How can I get started using Linux?

There’s some chance you’re using Linux already and don’t know it, but if you’d like to install Linux on your home computer to try it out, the easiest way is to pick a popular distribution designed for your platform (for example, laptop or tablet device) and give it a try. Although there are numerous distributions available, most of the older, well-known distributions are good choices for beginners because they have large user communities that can help answer questions if you get stuck or can’t figure things out. Popular distributions include [Elementary OS](#), [Fedora](#), [Mint](#), and [Ubuntu](#), but there are many others. It’s a common saying that the best Linux distro is the one that works best on your computer, so try a few to see which one best suits your hardware and your style of working.

You can [install Linux](#) on your current computer (be sure to back-up your data first), or you can buy a [System76](#) or [Purism](#) computer with Linux already installed. If you’re not looking for the fastest computing experience possible, you can also install Linux on old computers, or buy a [Raspberry Pi](#).

Once you've installed Linux, read our article on [how to install applications on Linux](#), and check back often for news and tutorials on all the best applications open source has to offer. Ultimately, getting started with Linux is a matter of *getting started with Linux*. The sooner you try it, the sooner you'll get comfortable with it, and eventually you'll blissfully forget that non-open operating systems exist!

How can I contribute to Linux?

Most of the Linux kernel is written in the C programming language, with a little bit of assembly and other languages sprinkled in. If you're interested in writing code for the Linux kernel itself, a good place to get started is in the [Kernel Newbies FAQ](#), which will explain some of the concepts and processes you'll want to be familiar with.

But the Linux community is much more than the kernel, and needs contributions from lots of other people besides programmers. Every distribution contains hundreds or thousands of programs that can be distributed along with it, and each of these programs, as well as the distribution itself, need a variety of people and skill sets to make them successful, including:

- Testers to make sure everything works on different configurations of hardware and software, and to report the bugs when it does not.
- Designers to create user interfaces and graphics distributed with various programs.
- Writers who can create documentation, how-tos, and other important text distributed with software.
- Translators to take programs and documentation from their native languages and make them accessible to people around the world.
- Packagers to take software programs and put all the parts together to make sure they run flawlessly in different distributions.
- Enthusiasts to spread the word about Linux and open source in general.
- And of course developers to write the software itself.

Where can I learn more about Linux?

Opensource.com has a huge archive of Linux-related articles. To view our entire archive, browse our [Linux tag](#). Or check out some of our favorites below.

- [Do you need programming skills to learn Linux?](#) by Jen Wike Huger
- [How to create a bootable USB drive for Linux](#) by Don Watkins
- [Test drive Linux with nothing but a flash drive](#) by Scott Nesbitt
- [10 ways to try Linux](#) by Seth Kenlon

- [Want a fulfilling IT career? Learn Linux](#) by Shawn Powers
- [Install Linux on a used laptop](#) by Phil Shapiro
- [8 Linux file managers to try](#) by David Both
- [Who helps your Linux distribution run smoothly?](#) by Luis Ibanez
- [6 reasons people with disabilities should use Linux](#) by Spencer Hunley
- [The current state of video editing for Linux](#) by Seth Kenlon



This work is licensed under a Creative Commons Attribution-Share Alike 4.0 International License.

ABOUT THIS SITE

The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.

Opensource.com aspires to publish all content under a **Creative Commons license** but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.

A note on advertising: Opensource.com does not sell advertising on the site or in any of its newsletters.

opensource.com

Copyright ©2024 Red Hat, Inc.

[Privacy Policy](#)

[Terms of use](#)

[Cookie preferences](#)



At their introduction in 1981, the US \$1,795 price of the Osborne 1 and its competitor Kaypro was considered an attractive price point; these systems had text-only displays and only floppy disks for storage. By 1982, Michael Dell observed that a personal computer system selling at retail for about \$3,000 US was made of components that cost the dealer about \$600; typical gross margin on a computer unit was around \$1,000.^[52] The total value of personal computer purchases in the US in 1983 was about \$4 billion, comparable to total sales of pet food. By late 1998, the average selling price of personal computer systems in the United States had dropped below \$1,000.^[53]

For Microsoft Windows systems, the average selling price (ASP) showed a decline in 2008/2009, possibly due to low-cost netbooks, drawing \$569 for desktop computers and \$689 for laptops at U.S. retail in August 2008. In 2009, ASP had further fallen to \$533 for desktops and to \$602 for notebooks by January and to \$540 and \$560 in February.^[54] According to research firm NPD, the average selling price of all Windows portable PCs has fallen from \$659 in October 2008 to \$519 in October 2009.^[55]

Terminology

“PC” is an initialism for “personal computer.” However, it is used in a different sense: It means a personal computers with an Intel x86-compatible processor running Microsoft Windows (sometimes called Wintel). “PC” is used in contrast with “Mac”, an Apple Macintosh computer.^[56] This sense of the word is used in ***Get a Mac*** advertisement campaign that run between 2006 to 2009, as well as its rival, ***I'm a PC*** campaign, that appeared on 2008.



Types

Stationary

Workstation

A workstation is a high-end personal computer designed for technical, mathematical, or scientific applications. Intended primarily to be used by one person at a time, they are commonly connected to a local area network and run multi-user operating systems. Workstations are used for tasks such as computer-aided design, drafting and modeling, computation-intensive scientific and engineering calculations, image processing, architectural modeling, and computer graphics for animation and motion picture visual effects.^[57]



Sun SPARCstation 1+ from the early 1990s, with a 25 MHz RISC processor

Desktop Computer

Prior to the widespread usage of PCs, a computer that could fit on a desk was remarkably small, leading to the “desktop” nomenclature. More recently, the phrase usually indicates a particular style of computer case. Desktop computers come in a variety of styles ranging from large vertical tower cases to small models which can be tucked behind an LCD monitor. In this sense, the term “desktop” refers specifically to a horizontally oriented case, usually intended to have the display screen placed on top to save desk space. Most modern desktop computers have separate screens and keyboards.

Gaming Computer

A gaming computer is a standard desktop computer that typically has high-performance hardware, such as a more powerful video card, processor and memory, in





LOG IN



opensource.com

Linux vs. Unix: What's the difference?

Dive into the differences between these two operating systems that share much of the same heritage and many of the same goals.

By [Phil Estes](#)

October 21, 2020 | [8 Comments](#) | 13 min read

504 readers like this.



Image by: [Opensource.com](#)

If you are a software developer in your 20s or 30s, you've grown up in a world dominated by Linux. It has been a significant player in the data center for decades, and while it's hard to find definitive operating system market share reports, Linux's share of data center operating systems could be as high as 70%, with Windows variants carrying nearly all the remaining percentage. Developers using any major public cloud can expect the target system will run Linux. Evidence that Linux is everywhere has grown in recent years when you add in Android and Linux-based embedded systems in smartphones, TVs, automobiles, and many other devices.

Even so, most software developers, even those who have grown up during this venerable "Linux revolution" have at least heard of Unix. It sounds similar to Linux, and you've probably heard people



use these terms interchangeably. Or maybe you've heard Linux called a "Unix-like" operating system.

So, what is this Unix? The caricatures speak of wizard-like "graybeards" sitting behind glowing green screens, writing C code and shell scripts, powered by old-fashioned, drip-brewed coffee. But Unix has a much richer history beyond those bearded C programmers from the 1970s. While articles detailing the history of Unix and "Unix vs. Linux" comparisons abound, this article will offer a high-level background and a list of major differences between these complementary worlds.

Unix's beginnings

The history of Unix begins at AT&T Bell Labs in the late 1960s with a small team of programmers looking to write a multi-tasking, multi-user operating system for the PDP-7. Two of the most notable members of this team at the Bell Labs research facility were Ken Thompson and Dennis Ritchie. While many of Unix's concepts were derivative of its predecessor ([Multics](#)), the Unix team's decision early in the 1970s to rewrite this small operating system in the C language is what separated Unix from all others. At the time, operating systems were rarely, if ever, portable. Instead, by nature of their design and low-level source language, operating systems were tightly linked to the hardware platform for which they had been authored. By refactoring Unix on the C programming language, Unix could now be ported to many hardware architectures.

In addition to this new portability, which allowed Unix to quickly expand beyond Bell Labs to other research, academic, and even commercial uses, several key of the operating system's design tenets were attractive to users and programmers. For one, Ken Thompson's [Unix philosophy](#) became a powerful model of modular software design and computing. The Unix philosophy recommended utilizing small, purpose-built programs in combination to do complex overall tasks. Since Unix was designed around files and pipes, this model of "piping" inputs and outputs of programs together into a linear set of operations on the input is still in vogue today. In fact, the current cloud serverless computing model owes much of its heritage to the Unix philosophy.

Rapid growth and competition

Through the late 1970s and 80s, Unix became the root of a family tree that expanded across research, academia, and a growing commercial Unix operating system business. Unix was not open source software, and the Unix source code was licensable via agreements with its owner, AT&T. The first known software license was sold to the University of Illinois in 1975.

Unix grew quickly in academia, with Berkeley becoming a significant center of activity, given Ken Thompson's sabbatical there in the '70s. With all the activity around Unix at Berkeley, a new delivery of Unix software was born: the Berkeley Software Distribution, or BSD. Initially, BSD was not an alternative to AT&T's Unix, but an add-on with additional software and capabilities. By the time 2BSD (the Second Berkeley Software Distribution) arrived in 1979, Bill Joy, a Berkeley grad student, had added now-famous programs such as [vi](#) and the [C shell](#) (/bin/csh).

In addition to BSD, which became one of the most popular branches of the Unix family, Unix's commercial offerings exploded through the 1980s and into the '90s with names like HP-UX, IBM's AIX, Sun's Solaris, Sequent, and Xenix. As the branches grew from the original root, the "[Unix wars](#)"



began, and standardization became a new focus for the community. The POSIX standard was born in 1988, as well as other standardization follow-ons via The Open Group into the 1990s.

More Linux resources

- ▶ [Linux commands cheat sheet](#)
- ▶ [Advanced Linux commands cheat sheet](#)
- ▶ [Free online course: RHEL Technical Overview](#)
- ▶ [Linux networking cheat sheet](#)
- ▶ [SELinux cheat sheet](#)
- ▶ [Linux common commands cheat sheet](#)
- ▶ [What are Linux containers?](#)
- ▶ [Our latest Linux articles](#)

Around this time AT&T and Sun released System V Release 4 (SVR4), which was adopted by many commercial vendors. Separately, the [BSD family](#) of operating systems had grown over the years, leading to some open source variations that were released under the now-familiar [BSD license](#). This included FreeBSD, OpenBSD, and NetBSD, each with a slightly different target market in the Unix server industry. These Unix variants continue to have some usage today, although many have seen their server market share dwindle into the single digits (or lower). BSD may have the largest install base of any modern Unix system today. Also, every Apple Mac hardware unit shipped in recent history can be claimed by BSD, as its OS X (now macOS) operating system is a BSD-derivative.

While the full history of Unix and its academic and commercial variants could take many more pages, for the sake of our article focus, let's move on to the rise of Linux.

Enter Linux

What we call the Linux operating system today is really the combination of two efforts from the early 1990s. Richard Stallman was looking to create a truly free and open source alternative to the proprietary Unix system. He was working on the utilities and programs under the name GNU, a recursive acronym meaning "GNU's not Unix!" Although there was a kernel project underway, it turned out to be difficult going, and without a kernel, the free and open source operating system dream could not be realized. It was Linus Torvald's work—producing a working and viable kernel that he called Linux—that brought the complete operating system to life. Given that Linus was using several GNU tools (e.g., the GNU Compiler Collection, or [GCC](#)), the marriage of the GNU tools and the Linux kernel was a perfect match.

Linux distributions came to life with the components of GNU, the Linux kernel, MIT's X-Windows GUI, and other BSD components that could be used under the open source BSD license. The early popularity of distributions like Slackware and then Red Hat gave the "common PC user" of the 1990s

access to the Linux operating system and, with it, many of the proprietary Unix system capabilities and utilities they used in their work or academic lives.

Because of the free and open source standing of all the Linux components, anyone could create a Linux distribution with a bit of effort, and soon the total number of distros reached into the hundreds. Of course, many developers utilize Linux either via cloud providers or by using popular free distributions like Fedora, Canonical's Ubuntu, Debian, Arch Linux, Gentoo, and many other variants. Commercial Linux offerings, which provide support on top of the free and open source components, became viable as many enterprises, including IBM, migrated from proprietary Unix to offering middleware and software solutions atop Linux. Red Hat built a model of commercial support around Red Hat Enterprise Linux, as did German provider SUSE with SUSE Linux Enterprise Server (SLES).

Comparing Unix and Linux

So far, we've looked at the history of Unix and the rise of Linux and the GNU/Free Software Foundation underpinnings of a free and open source alternative to Unix. Let's examine the differences between these two operating systems that share much of the same heritage and many of the same goals.

From a user experience perspective, not very much is different! Much of the attraction of Linux was the operating system's availability across many hardware architectures (including the modern PC) and ability to use tools familiar to Unix system administrators and users.

Because of [POSIX](#) standards and compliance, software written on Unix could be compiled for a Linux operating system with a usually limited amount of porting effort. Shell scripts could be used directly on Linux in many cases. While some tools had slightly different flag/command-line options between Unix and Linux, many operated the same on both.

One side note is that the popularity of the macOS hardware and operating system as a platform for development that mainly targets Linux may be attributed to the BSD-like macOS operating system. Many tools and scripts meant for a Linux system work easily within the macOS terminal. Many open source software components available on Linux are easily available through tools like [Homebrew](#).

The remaining differences between Linux and Unix are mainly related to the licensing model: open source vs. proprietary, licensed software. Also, the lack of a common kernel within Unix distributions has implications for software and hardware vendors. For Linux, a vendor can create a device driver for a specific hardware device and expect that, within reason, it will operate across most distributions. Because of the commercial and academic branches of the Unix tree, a vendor might have to write different drivers for variants of Unix and have licensing and other concerns related to access to an SDK or a distribution model for the software as a binary device driver across many Unix variants.

As both communities have matured over the past decade, many of the advancements in Linux have been adopted in the Unix world. Many GNU utilities were made available as add-ons for Unix systems where developers wanted features from GNU programs that aren't part of Unix. For example, IBM's AIX offered an AIX Toolbox for Linux Applications with hundreds of GNU software packages (like Bash, GCC, OpenLDAP, and many others) that could be added to an AIX installation to ease the transition between Linux and Unix-based AIX systems.



Proprietary Unix is still alive and well and, with many major vendors promising support for their current releases well into the 2020s, it goes without saying that Unix will be around for the foreseeable future. Also, the BSD branch of the Unix tree is open source, and NetBSD, OpenBSD, and FreeBSD all have strong user bases and open source communities that may not be as visible or active as Linux, but are holding their own in recent server share reports, with well above the proprietary Unix numbers in areas like web serving.

Where Linux has shown a significant advantage over proprietary Unix is in its availability across a vast number of hardware platforms and devices. The Raspberry Pi, popular with hobbyists and enthusiasts, is Linux-driven and has opened the door for an entire spectrum of IoT devices running Linux. We've already mentioned Android devices, autos (with Automotive Grade Linux), and smart TVs, where Linux has large market share. Every cloud provider on the planet offers virtual servers running Linux, and many of today's most popular cloud-native stacks are Linux-based, whether you're talking about container runtimes or Kubernetes or many of the serverless platforms that are gaining popularity.

One of the most revealing representations of Linux's ascendancy is Microsoft's transformation in recent years. If you told software developers a decade ago that the Windows operating system would "run Linux" in 2016, most of them would have laughed hysterically. But the existence and popularity of the Windows Subsystem for Linux (WSL), as well as more recently announced capabilities like the Windows port of Docker, including LCOW (Linux containers on Windows) support, are evidence of the impact that Linux has had—and clearly will continue to have—across the software world.

This article was originally published in May 2018 and has been updated by the editor.

What to read next

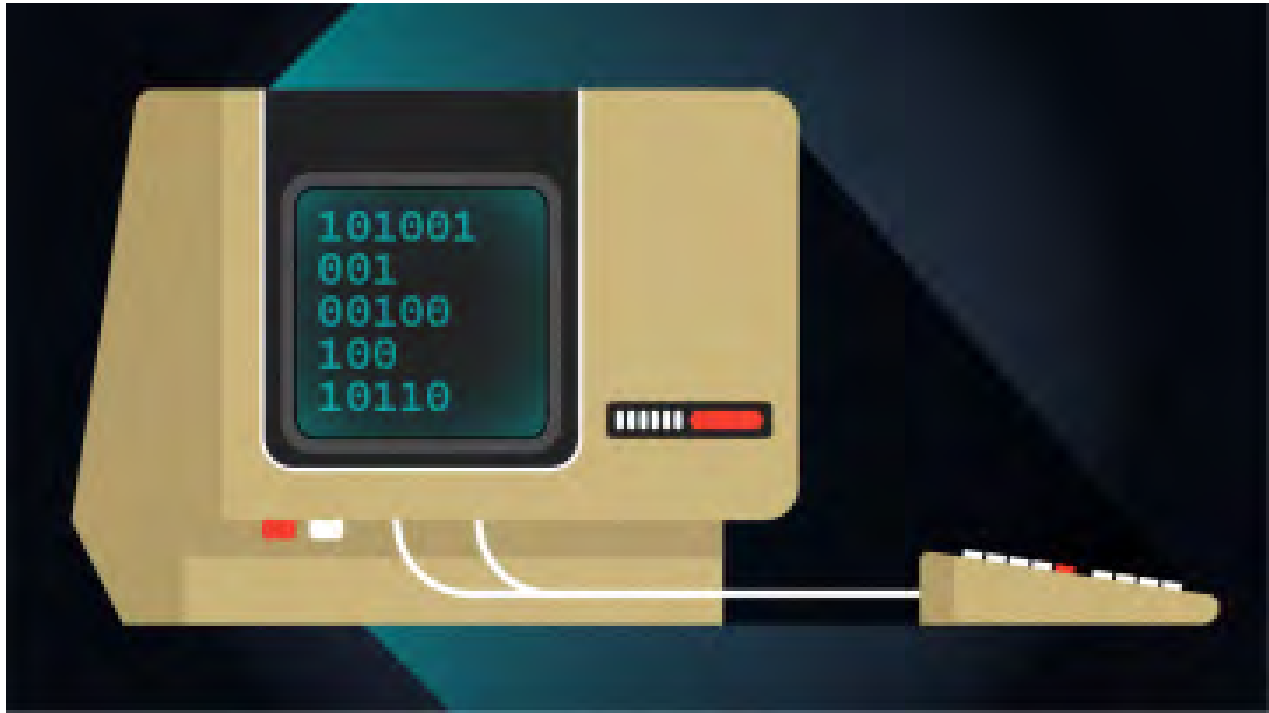


11 surprising ways you use Linux every day



What technology runs on Linux? You might be astonished to know just how often you use Linux in your daily life.

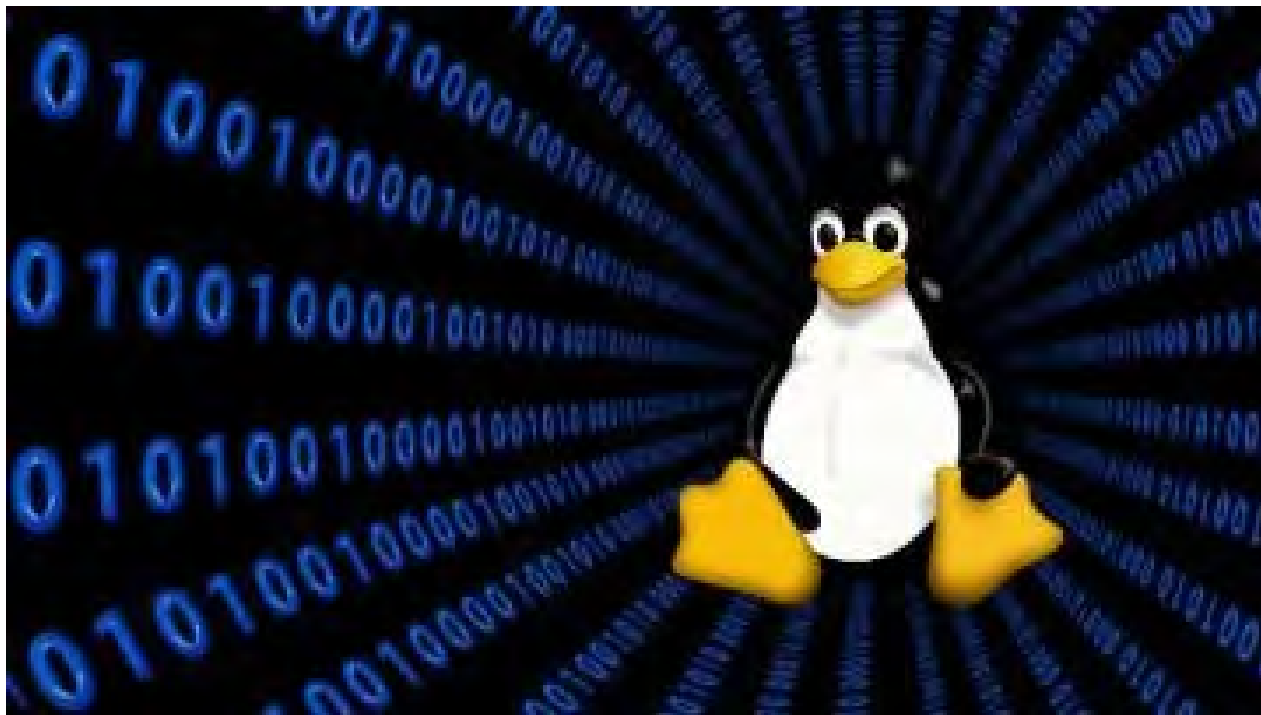
 [Don Watkins](#) (Correspondent)



Origin stories about Unix

Brian Kernighan, one of the original Unix gurus, shares his insights into the origins of Unix and its associated technology.

 [Jim Hall](#) (Correspondent)



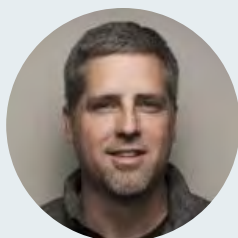
6 signs you might be a Linux user

If you're a heavy Linux user, you'll probably recognize these common tendencies.



[Seth Kenlon](#) (Team, Red Hat)

Tags:



Phil Estes

Phil is a Distinguished Engineer & CTO, Container and Linux OS Architecture Strategy for the IBM Watson and Cloud Platform division. Phil is currently an OSS maintainer in the Docker (now Moby) engine project, the CNCF containerd project, and is a member of both the Open Container Initiative (OCI) Technical Oversight Board and the Moby Technical Steering Committee.

[More about me](#)

8 Comments

These comments are closed.





[Matakaheru](#) | May 21, 2018

No readers like this yet.

Just as an aside, macOS *is* UNIX as validated by OpenGroup, the current owners of the UNIX brand and standardization outlines. Linux is "UNIX like" because no single distro has ever approached OpenGroup to certify their distro as an official UNIX.

Also, FreeBSD, OpenBSD and NetBSD are not currently "UNIX" either, even though they *are* BSD. The reasons are the same for Linux. The certification process through OpenGroup is expensive. The open distributions just don't see the point in putting money toward a brand badge to keep up with the likes of AIX.

The information for the price of being UNIX can be found here:

<https://www.opengroup.org/openbrand/Brandfees.htm>



Clarence | May 22, 2018

No readers like this yet.

Actually there are two Linux distros that have been certified as UNIX systems by The Open Group: Huawei EulerOS and Inspur K-UX.

See <https://www.opengroup.org/openbrand/register/>



[DarkMatter](#) | May 24, 2018

No readers like this yet.

"BSD may have the largest install base of any modern Unix system today. Also, every Apple Mac hardware unit shipped in recent history can be claimed by BSD, as its OS X (now macOS) operating system is a BSD-derivative."

I am so disappointed at the pace of "desktop" BSD in comparison to Desktop Linux. What's sad is that FreeBSD has the packaging, developers, cutting-edge technology and stability that you'd expect in Unix.

So why oh why can't TrueOS (formerly PC-BSD) deliver a solid stable desktop system? If I were them, I'd go straight with MATE after looking at Ubuntu Mate. This is what we need in BSD land on the desktop.

NetBSD would also make an excellent base for a desktop BSD release. It's solid, reliable and well designed. Heck, it's almost like there's a conspiracy to prevent a viable, thriving free desktop BSD from taking root.

However, GhostBSD looks promising with a Mate desktop. We shall see.





dgrb | May 21, 2018

No readers like this yet.

The early history of unix is a little more complex.

AT&T were part of the Multics project - they were responsible for the compilers - together with MIT (operating system) and GE (Hardware).

Ritchie and Thompson only began their Unics (original spelling) project after Bell withdrew from the Multics project. The reasons given depend on whom you ask.

Multics was, from the get-go, written in a high-level language, PL/1, so it too was in principle, portable - but then so was Burroughs MCP, also written in a HLL.



Jason L Gray | May 24, 2018

No readers like this yet.

Simple. Linux is an OS Kernel, Unix is a certification sold by the OpenGroup...



[Steve](#) | May 24, 2018

No readers like this yet.

Per management any Unix or AIX is like Linux. To me , Linux can stand by itself .



[David C.](#) | June 1, 2018

No readers like this yet.

The GNU project massively predates Linux. The Free Software Foundation developed and was distributing tons of software (all but a kernel) for use with all kinds of UNIX platforms throughout the 80's and 90's. I personally used GNU software on Sun (SunOS, later Solaris), HP (HP-UX), DEC (Ultrix) and IBM (AIX) platforms long before Linux ever existed.

And because of the (mostly) portable nature of GNU code, there were plenty of ports (of most of the packages) to non-UNIX platforms. I personally ran many GNU packages on MS-DOS and OS/2, neither of which even remotely resemble UNIX.

The distribution of GNU software with Linux was definitely critical to making Linux a viable platform, but that was facilitated by the fact that GNU has already been ported to nearly every UNIX and UNIX-like platform prior to Linux. In other words, Linux needed GNU to succeed but GNU did not need Linux, because it had already succeeded before Linux existed.



Oh, and BTW, the GNU project did eventually release its own kernel. They call it the HURD. Although it has a lot of interesting and cutting edge tech (and is well worth studying as a part of a college course on operating systems), it has not been a commercial success. If you're interested, it can be found here: <https://www.gnu.org/software/hurd/hurd.html>



[Dave](#) | July 15, 2018

No readers like this yet.

While the author states that ". . . the BSD branch of the Unix tree is open source . . ." , he omits to state that the SVR4 branch also lives on in open source as illumos, which in turn is a derivation of OpenSolaris (and Solaris before it). illumos is powering Samsung's public cloud build-out, as part of their ownership of Joyent:

<http://dtrace.org/blogs/bmc/2017/09/04/the-sudden-death-and-eternal-lif...>

Related Content



[What's new in GNOME 44?](#)



[5 reasons virtual machines still matter](#)



[Remove the background from an image with this Linux command](#)



This work is licensed under a Creative Commons Attribution-Share Alike 4.0 International License.

ABOUT THIS SITE

The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.

Opensource.com aspires to publish all content under a **Creative Commons license** but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.



A note on advertising: Opensource.com does not sell advertising on the site or in any of its newsletters.

opensource.com

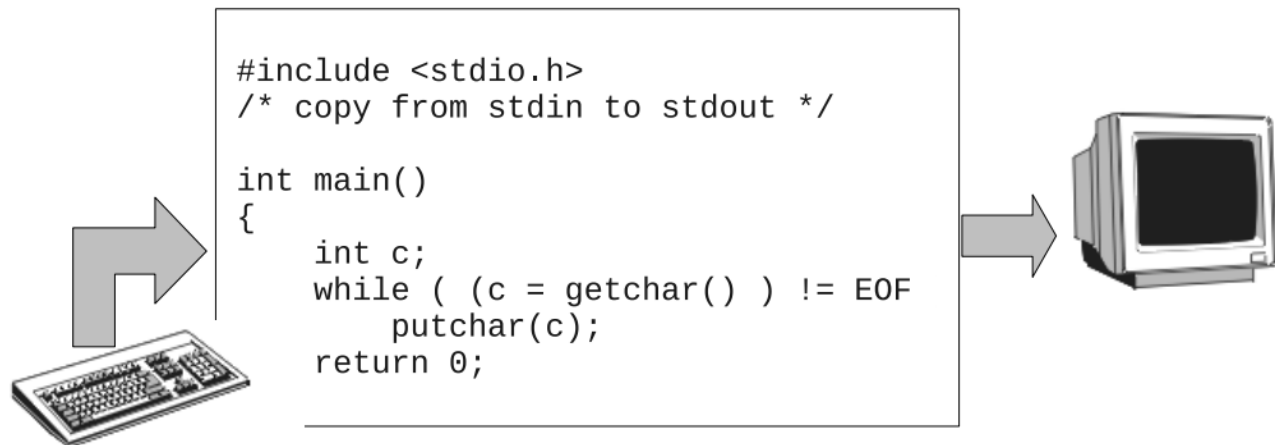
Copyright ©2024 Red Hat, Inc.


[Privacy Policy](#)

[Terms of use](#)

[Cookie preferences](#)

SYSTEMS PROGRAMMING



Application making system calls `getchar()` and `putchar()` to interface to hardware. Source: Weiss 2020, fig. 1.1.* 

Systems are built from hardware and software components. Systems programming is about implementing these components, their interfaces and the overall architecture. Individual components perform their prescribed functions and at the same time work together to form a stable and efficient system.*

Systems programming is distinct from application programming. System programs provide services to other software. Via abstractions, they expose API to simplify the development of applications. They're often optimized to low-level machine architecture. Unlike application software, most system software are not directly used by end users.**

Assembly and C language have been historically used for systems programming.** Go,* Rust,* Swift* and WebAssembly** are newer languages suited for systems programming.

Discussion

How is systems programming different from application programming?

Operating systems, device drivers, BIOS and other firmware, compilers, debuggers, web servers, database management systems, communication protocols and networking utilities are examples of system software.**

* As part of operating systems, memory management, scheduling, event handling and many more essential functions are done by system software. Examples of application software are Microsoft Office, web browsers, games, and graphics software.*

Application software generally don't directly access hardware or manage low-level resources. They do so via calls to system software. We may thus view system software as aiding application software with low-level access and management.** Application developers can therefore focus on the business logic of their applications.

**SYSTEM SOFTWARE
VERSUS
APPLICATION SOFTWARE**

| SYSTEM SOFTWARE | APPLICATION SOFTWARE |
|--|--|
| Computer software designed to provide a platform to other software | Software designed to perform a group of coordinated functions, tasks or activities for the benefit of the user |
| Manages resources and helps to run hardware and application software | Performs a specific task according to their type |
| Runs when the system starts and runs till the end | Runs when the user requires |
| Developed using languages like C, C++, Assembly | Developed using languages like Java, C, C++, Visual Basic |
| Essential for the proper functioning of a system | Not extremely important for the functioning of the system |
| Ex: operating system, language processors and device drivers | Ex: Word processor, Spreadsheet, Presentation software, web browsers, graphics software |

Visit www.PEDIAA.com

Comparing system software against application software. Source: Lithmee 2018.



While application developers may not build system software, they can become better developers by knowing more about the design and implementation of system software.* Specifically, by knowing and using system APIs correctly they can avoid implementing similar functions in their applications.*

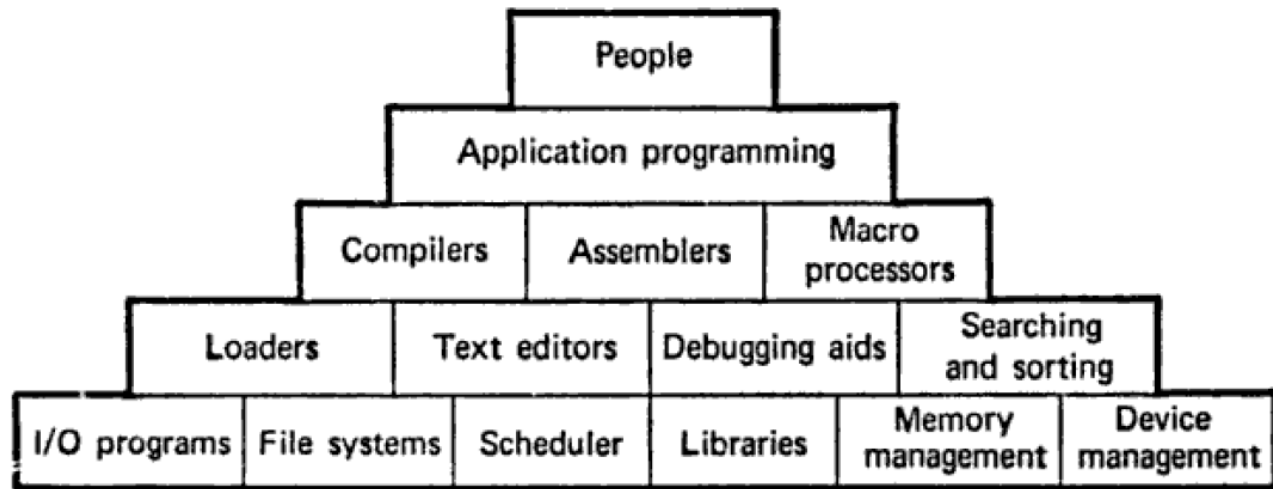
For example, a C application on UNIX/Linux calls the system API `getpid()`. The OS creates the process, allocates memory and assigns a process identifier.*

What are the main concerns of systems programming?

An operating system such as Linux is a collection of system programs. These deal with files and directories, manage processes for executable programs, enable I/O for those programs and allocate/release memory as needed. The OS manages users, groups and associated permissions. The OS prevents normal user programs from executing privileged operations. The shell is a special system program that allows users to interact with the system.* If processes need to communicate or respond to external events, signals (aka software interrupts) facilitate this.*

There are systems programs that transform other programs into machine-level instructions for execution: compilers, assemblers, macro processors, loaders and linkers. Their aim is to generate instructions optimized for speed or memory. Use of registers, loops, data structures, and algorithms are considered in these system programs.*





Foundations of systems programming. Source: Donovan 1972, fig. 1.1.*

Programming languages,* editors and debuggers are also system programs.* These are tools to write good and reliable system programs. They have to be easy to learn and productive for a developer while also being efficient and safe from a system perspective.**

Is a system programmer same as a system administrator?

A system programmer is one who write system software and this task is called system programming or systems programming.* But there's an older definition that's been used in the context of mainframes since the 1960s.

On a mainframe, a system programmer installs, upgrades, configures and maintains the operating system. She does capacity planning and evaluates new products. She's also skilled in optimizing the system for performance, troubleshooting problems and analyzing memory dumps.*

On the other hand, a system administrator handles day-to-day operations such as adding/removing users, configuring access, installing software, and monitoring performance. She deals with applications whereas a system programmer is more well-versed with the mainframe hardware.*

In small IT organizations, both roles may be performed by a single individual.*

What languages are suited to systems programming?

Wikipedia lists more than a dozen languages: C/C++, Ada, D, Nim, Go, Rust, Swift, and more. Notably, the following popular languages are absent: JavaScript, Java, C#, Python, Visual Basic, PHP, Perl, and Ruby.* Implemented in Haskell, COGENT is a functional programming language that's also suited for systems programming.*

System programming languages are required to provide direct access to hardware including memory and I/O access. Performance, explicit memory management and fine-grained control at bit level are essential capabilities.* Where they also offer high-level programming constructs, system programming languages (C, Rust, Swift, etc.) are also used for application programming.*

Since such languages give access to low-level hardware functions, there's a risk of introducing bugs. Rust was created to balance aspects of both safety and control. C sacrifices safety for control* while Java does the opposite.*

Scripting languages such as Python, JavaScript and Lua are not for systems programming. However, the introduction of static typing (for safety) and Just-in-Time (JIT) compilation (for speed) has seen these languages being used for systems programming.*



What's systems programming in the context of the web?

On the web, applications adopt the client-server architecture. Server-side logic may be implemented as many microservices distributed on a cloud platform. Applications make use of APIs served by different endpoints. In this context, we have systems programs that help create distributed applications for the cloud. System programs must be designed to address network topology changes,* security concerns, high latency, and poor network connections.*

Rob Pike commented that developers thought that Go was for systems programming. In fact, it was for writing any server-side code. Later he saw anything running on the cloud as systems software.* Ousterhout commented in 1998 that on the web Java was being used for systems programming.*

Bunardzic commented that "the only way to program a system is to program a network". Systems should be designed for concurrency, fault encapsulation/detection/recovery, upgrade without downtime, observability, and asynchronous communication. Developers who use system services and APIs must be free to choose their own technology stack. One service shouldn't depend on others.*

What are the best practices in systems programming?

A systems programmer needs to know the system APIs well. In addition, she should know how the OS kernel, programs and users interact with one another.*

Designing a good system is not a one-time task. The system should be designed for iteration and incremental improvements. System designers must be open to feedback.* Historically, many Linux system programmers blamed application developers for program crashes instead of seeing these as opportunities to improve the Linux kernel or system tools.*





All assumptions must be made explicit.* Systems software should get the abstractions right,* minimize if not avoid leaky abstractions. In other words, its users shouldn't need to know implementation details. For instance, ORM frameworks that interface between applications and databases are often leaky due to a conceptual mismatch between objects and relations.*

Before implementation, it's beneficial to do system modelling, analysis and simulations. Unified Modelling Language (UML) can help.* Small and simple systems are amenable to formal analysis. Anything else, we need to apply statistical analysis. More components there are, more complex becomes the system.*

Milestones

1960

Till early and even mid-1960s, the first concern in designing computer systems is the hardware itself. Programming them becomes a secondary concern. Programming techniques are chaotic. Often they're not as intended by the hardware designers. Systems programming as a discipline is only starting to emerge.*

1966

$$A \xrightarrow{a} T \rightarrow B$$

$$B := T(A)$$

Examples

| A | B | <u>T is called</u> |
|--|------------------------------------|--|
| (1) Binary Code (A ₂) | Results (A ₃) | Computer (or Interpreter) (T ₃) |
| (2) Symbolic Code (A ₁) | Binary Code (A ₂) | Assembler (T ₂) |
| (3) Phrase Structure Language (A ₀) | Symbolic Code (A ₁) | Compiler (T ₁) |

Translators are system programs. Source: Shaw 1966, sec. I-3.*

Shaw considers assemblers, interpreters, compilers, and monitors as **translators**; that is, they translate code in one form to another. Referring to the figure, translator T translates A to B. He defines the following.*

“ Systems Programming is the science of designing and constructing translators.

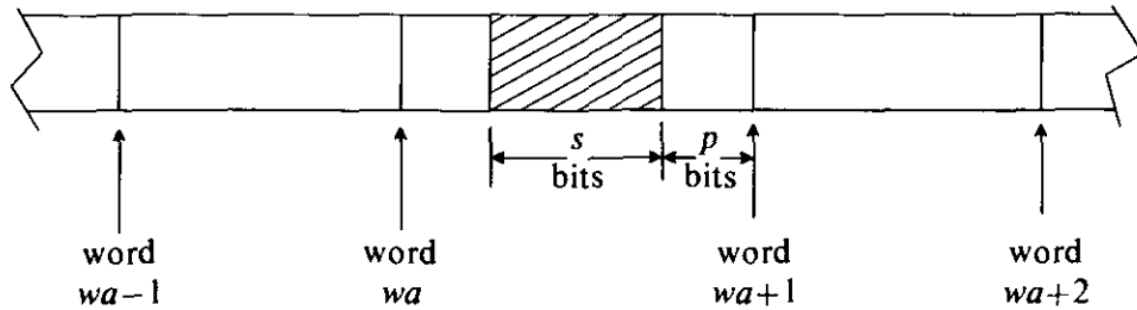
Oct
1968


At the NATO Conference on Software Engineering, the merits of **high-level languages** are discussed: cost, maintainability, correctness. System programmers however object to high-level languages. They don't like anything to get in between themselves and the machine.* Reconciling these two concerns is the main challenge in design a suitable systems programming language.*

1969

Unix operating system is invented at Bell Laboratories, first in assembly on PDP-7 (1969) and subsequently migrated to C on PDP-11 (1971). A decade later one of its inventors, Dennis Ritchie, comments that only the assembler is still written in assembler. Most other system programs are written in C language.*

1971



A 'pointer' in BLISS to access bits within a word. Source: Wulf et al. 1971, pp. 786.* 

Researchers at the Carnegie-Mellon University invent a new systems programming language that they name **BLISS**. They describe it as a general purpose high-level language for writing large software systems for specific machines. They enumerate the requirements of any good systems programming language: space/time economy, access to hardware features, data structures, control structures, understandability, debugging, etc. The figure shows an example of how BLISS enables bit-level access. Such low-level access is typical of systems software.*

1984

Weicker proposes **Dhrystone as a benchmark** for systems programming. Currently we have the Whetstone benchmark that's tuned to measure floating-point arithmetic. Systems programs often use enumerations, records and pointer data types.* Compared to numerical programs, systems programs have fewer loops, simpler compute statements, more conditional branching and more procedure calls. The Dhrystone benchmark accounts for these.*

1986

An IBM research report details the challenges and pitfalls in programming for a multi-CPU and multi-threaded environment. They refer to the System/370 architecture. Shared memory and data structures have to be handled carefully. Therefore, **parallelism** is a new concern for systems programming.* However, multi-programming was considered in the design of Unix back in the early 1970s.*

1990

This decade sees the wide adoption of **scripting languages** such as Perl, Tcl, Ruby, PHP, Python and JavaScript. They're seen as languages that "glue together" various components whereas systems programming languages are used to create those components. In the 2010s, the boundaries blur as some scripting languages are used to build large systems software.*

1992

Wolfe proposes changes to how systems programming must be taught to students. The curriculum would include small challenges in various aspects of systems programming and not just focus on assemblers and compilers. Currently, students perceive systems programming as boring, the techniques themselves stale, and the courses disconnected from systems programming jobs out there.*

2005

Brewer et al. note that 35 years after the invention of C language, we still don't have a suitable alternative for systems programming. Shapiro voices a similar complaint in 2006. Advances in programming languages have not motivated systems programmers to adopt them.*

2009


Mozilla begins sponsoring the development of **Rust**. Rust is announced to the world in 2010. The first stable release of the language happens in May 2015.* Today Rust is well-recognized as a systems programming language.*

2019

Crichton at Stanford University proposes that students of programming languages must be taught systems programming. Courses must not be about theory and formal methods alone. They should include practical programming while also providing effective mental models of computation and formal reasoning. He includes WebAssembly and Rust in the proposed curriculum.*

References



1. Beck, Leland L. 1997. "System Software: An Introduction to Systems Programming." Third Edition, Addison-Wesley. Accessed 2022-05-23. (<https://media.oaipdf.com/pdf/b48968d2-5b63-4612-9e0b-512123d069b8.pdf>).
2. Bunardzic, Alex. 2022. "An open source developer's guide to systems programming." Opensource.com, April 29. Accessed 2022-05-23. (<https://opensource.com/article/22/4/systems-programming>),  (<https://web.archive.org/web/20220530183226/https://opensource.com/article/22/4/systems-programming>).



3. Crichton, Will. 2018. "What is Systems Programming, Really?" September 9. Accessed 2022-05-23. (<https://willcrichton.net/notes/systems-programming/>)
4. Crichton, W. 2019. "From Theory to Systems: A Grounded Approach to Programming Language Education." arXiv, v1, April 14. Accessed 2022-05-23. (<https://arxiv.org/pdf/1904.06750.pdf>)
5. Donovan, John J. 1972. "Systems Programming." Tata McGraw-Hill Edition (1991). The McGraw-Hill, Inc. Accessed 2022-05-23. (<https://pdfcookie.com/download/system-programming-by-john-j-donovan-book-eg271k1qe420>)
6. Eberhardt, Colin. 2022. "WebAssembly & Modern Systems Programming Languages." OCon London, May 12. Accessed 2022-05-24. (<https://plus.gconferences.com/plus2022/track/webassembly-modern-systems-programming-languages>)
7. Encyclopaedia Britannica. 2021. "systems programming." Encyclopaedia Britannica, August 2. Accessed 2022-05-23. (<https://www.britannica.com/technology/systems-programming>)
8. Farkas, T., C. Neumann, and A. Hinnerichs. 2009. "An Integrative Approach for Embedded Software Design with UML and Simulink." 33rd Annual IEEE International Computer Software and Applications Conference, pp. 516-521. doi: 10.1109/COMPASAC.2009.185. Accessed 2022-05-30. (<https://sci-hub.hkvisa.net/10.1109/compsac.2009.185>)
9. GoLang Docs. 2020. "System programming in Go - 1." GoLang Docs, September 17. Accessed 2022-05-24. (<https://golangdocs.com/system-programming-in-go-1>)
10. IBM. 2008. "Mainframe concepts." z/OS Basic Skills Information Center, IBM Corporation. Accessed 2022-05-23. (https://www.ibm.com/docs/en/zosbasics/com.ibm.zos.zmainframe/zmainframe_book.pdf)
11. Jung, Ralf, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. "Safe Systems Programming in Rust." Communications of the ACM, vol. 64, no. 4, pp. 144-152. April. Accessed 2022-05-24. (<https://cacm.acm.org/magazines/2021/4/251364-safe-systems-programming-in-rust/fulltext>)
12. Kerr, Luka. 2020. "Extending a Purely Functional Language to Enable Low-Level Systems Programming." B.E. Thesis, The University of New South Wales, December 11. Accessed 2022-05-23. (<https://people.eng.unimelb.edu.au/rizkallahc/theses/luka-kerr-honours-thesis.pdf>)
13. Li, Peng. 2004. "Safe Systems Programming Languages." Department of Computer and Information Science, University of Pennsylvania, October 6. Accessed 2022-05-23. (<https://www.cs.uic.edu/pub/Main/PhDQualifyingExam/Sample4.pdf>)
14. Lithmee. 2018. "Difference Between System Software and Application Software." Pediaa, June 22. Accessed 2022-05-24. (<https://pediaa.com/difference-between-system-software-and-application-software/>)
15. Love, Robert. 2013. "Linux System Programming." Second Edition, O'Reilly. Accessed 2022-05-23. (<https://doc.lagout.org/programmation/unix/Linux%20System%20Programming%20Talking%20Directly%20to%20the%20Kernel%20and%20C%20Library.pdf>)
16. Madunuwan, Dumindu. 2021. "Why Rust?" Learning Rust, April 27. Accessed 2022-05-30. (https://learning-rust.github.io/docs/a1.why_rust.html)
17. McDaniel, Patrick. 2015. "Module: Systems Programming." CMPSC 311 - Introduction to Systems Programming, Penn State Univ. Accessed 2022-05-23. (<https://www.cse.psu.edu/~pdm12/cmpsc311-f15/slides/cmpsc311-systems-programming.pdf>)
18. PCMag. 2022. "System software." Encyclopedia, PCMag. Accessed 2022-05-26. (<https://www.pcmag.com/encyclopedia/term/system-software>)
19. Perkins, C. 2012. "Evolution of Systems Programming." Lecture 11 in Advanced Operating Systems, School of CS, Accessed 2022-05-23. (<https://csperkins.org/teaching/2011-2012/adv-os/lecture11.pdf>)
20. Ritchie, Dennis M. 1984. "The Evolution of the Unix Time-sharing System." Lucent Technologies Inc. Accessed 2022-05-23. (<https://www.bell-labs.com/usr/dmr/www/hist.html>)
21. Shapiro, Jonathan. 2006. "Programming language challenges in systems codes: why systems programmers still use C, and what to do about it." PLOS '06: Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems, October. doi: 10.1145/11215995.1216004. Accessed 2022-05-30. (https://www.researchgate.net/publication/220939001_Programming_language_challenges_in_systems_codes_Why_systems_programmers_still_use_C_and_what_to_do_about_it)
22. Shaw, Alan C. 1966. "Lecture notes on a course in systems programming." Technical Report No. 52, CS Dept., Stanford University, December 9. Accessed 2022-05-23. (<http://infolab.stanford.edu/pub/cstr/reports/cs/tr/66/52/CS-TR-66-52.pdf>)
23. Suda, Jon. 2015. "Stop Leak." Blog, Jon's Musings, July 25. Accessed 2022-05-30. (<https://jonsmusings.com/Stop-Leak>)
24. Swift. 2022. "About Swift." Swift. Accessed 2022-05-26. (<https://www.swift.org/about/>)
25. TIOBE. 2022. "TIOBE Index for May 2022." TIOBE. Accessed 2022-05-30. (<https://www.tiobe.com/tiobe-index/>)
26. Techopedia. 2015. "System Programming." Techopedia, September 28. Accessed 2022-05-23. (<https://www.techopedia.com/definition/9616/system-programming>)

27. Treiber, R. Kent. 1986. "Systems Programming: Coping with Parallelism." Research Report, RJ 5118, IBM Almaden Research Center, April 23. Accessed 2022-05-23. (<https://dominoweb.draco.res.ibm.com/reports/rj5118.pdf>)  (<https://web.archive.org/web/20220130231415/https://dominoweb.draco.res.ibm.com/reports/rj5118.pdf>)
28. Watts, Stephen. 2020. "What is Systems Programming?" Blog, BMC, April 24. Accessed 2022-05-23. (<https://www.bmc.com/blogs/systems-programming/>)  (<https://web.archive.org/web/20220530183156/https://www.bmc.com/blogs/systems-programming/>)
29. Weicker, Reinhold P. 1984. "Dhrystone: a synthetic systems programming benchmark." Communications of the ACM, vol. 27, no. 10, pp. 1013-1030, October. Accessed 2022-05-23. (<https://dl.acm.org/doi/pdf/10.1145/358274.358283>)  (<https://web.archive.org/web/20220530183310/https://dl.acm.org/doi/pdf/10.1145/358274.358283>)
30. Weicker, R. P. 1990. "An Overview of Common Benchmarks." IEEE Computer, vol. 23, no. 12, pp. 65-75, 1990, doi: 10.1109/2.62094. Accessed 2022-05-23. (<https://users.ece.utexas.edu/~ljohn/teaching/382m-15/reading/weicker.pdf>)  (<https://web.archive.org/web/20220530183458/https://users.ece.utexas.edu/~ljohn/teaching/382m-15/reading/weicker.pdf>)
31. Weiss, Stewart. 2020. "Chapter 1: Introduction to System Programming." Lecture notes, UNIX Application and System Programming, Department of Computer Science, Hunter College. Accessed 2022-05-23. (http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/unix_lecture_notes/chapter_01.pdf)  (https://web.archive.org/web/20220530183251/http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/unix_lecture_notes/chapter_01.pdf)
32. Wikipedia. 2022. "Systems programming language." Wikipedia, April 22. Accessed 2022-05-23. (https://en.wikipedia.org/w/index.php?title=System_programming_language&oldid=1084543674)  (https://web.archive.org/web/20220530183142/https://en.wikipedia.org/w/index.php?title=System_programming_language&oldid=1084543674)
33. Wolfe, James L. 1992. "Reviving systems programming." ACM SIGCSE Bulletin, vol. 24, no. 1, pp. 255-258, March. Accessed 2022-05-23. (<https://dl.acm.org/doi/pdf/10.1145/135250.134561>)  (<https://web.archive.org/web/20220530183528/https://dl.acm.org/doi/pdf/10.1145/135250.134561>)
34. Wulf, W.A., D.B. Russell, and A.N. Habermann. 1971. "BLISS: A Language for Systems Programming." Communications of the ACM, vol. 14, no. 12, pp. 780-790, December. Accessed 2022-05-23. (<https://dl.acm.org/doi/pdf/10.1145/362919.362936>)  (<https://web.archive.org/web/20220530183342/https://dl.acm.org/doi/pdf/10.1145/362919.362936>)



Further Reading



1. Crichton, Will. 2018. "What is Systems Programming, Really?" September 9. Accessed 2022-05-23. (<https://willcrichton.net/notes/systems-programming/>)
2. Donovan, John J. 1972. "Systems Programming." Tata McGraw-Hill Edition (1991). The McGraw-Hill, Inc. Accessed 2022-05-23. (<https://pdfcookie.com/download/system-programming-by-john-j-donovan-book-eg271k1qe420>)
3. Jung, Ralf, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. "Safe Systems Programming in Rust." Communications of the ACM, vol. 64, no. 4, pp. 144-152, April. Accessed 2022-05-24. (<https://cacm.acm.org/magazines/2021/4/251364-safe-systems-programming-in-rust/fulltext>)
4. Li, Peng. 2004. "Safe Systems Programming Languages." Department of Computer and Information Science, University of Pennsylvania, October 6. Accessed 2022-05-23. (<https://www.cs.uic.edu/pub/Main/PhDQualifyingExam/Sample4.pdf>)
5. Perkins, C. 2012. "Evolution of Systems Programming." Lecture 11 in Advanced Operating Systems, School of CS, Accessed 2022-05-23. (<https://csperkins.org/teaching/2011-2012/adv-os/lecture11.pdf>)
6. Robbins, Kay A., and Steven Robbins. 2003. Unix™ Systems Programming: Communication, Concurrency, and Threads." Pearson. Accessed 2022-05-23. (<https://cs.uwec.edu/~tan/priv/www-docs/cs452/USP.pdf>)

Article Stats

1801
Words
0
Chats

2
Authors
4
Likes

4
Edits
13K
Hits

Cite As

Devopedia. 2022. "Systems Programming." Version 4, May 30. Accessed 2023-11-12. <https://devopedia.org/systems-programming>

COPY CITATION

[ABOUT \(/SITE-INFO/ABOUT\)](#)
[TERMS OF USE \(/SITE-INFO/TERMS-OF-USE\)](#)
[PRIVACY POLICY \(/SITE-INFO/TERMS-OF-USE#PRIVACY-POLICY\)](#)
[MISSION \(/SITE-INFO/ABOUT#MISSION\)](#)
[VALUES \(/SITE-INFO/ABOUT#VALUES\)](#)
[LICENSING \(/SITE-INFO/ABOUT#LICENSING\)](#)

[FOUNDATION \(/SITE-INFO/FOUNDATION\)](#)
[TRUSTEES \(/SITE-INFO/FOUNDATION#TRUSTEES\)](#)
[DONATIONS \(/SITE-INFO/FOUNDATION#DONATIONS\)](#)
[EVENTS \(/SITE-MAP/COMMUNITY-OUTREACH?EVENTS\)](#)
[REPORT ISSUES](#)
 [\(HTTPS://GITHUB.COM/DEVOPEDIAORG/WEBAPP/ISSUES\)](https://github.com/devopediaorg/webapp/issues)
[OPEN SOURCE \(HTTPS://GITHUB.COM/DEVOPEDIAORG\)](https://github.com/devopediaorg)





[\(https://creativecommons.org/licenses/by-sa/4.0/\)](https://creativecommons.org/licenses/by-sa/4.0/)



[\(https://www.facebook.com/Devopedia/\)](https://www.facebook.com/Devopedia/)



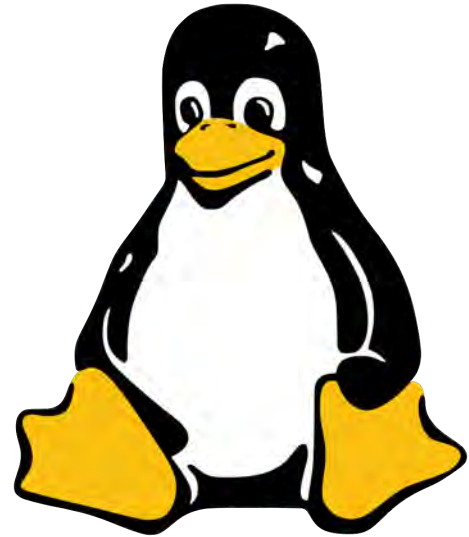
<https://www.devopedia.org/systems-programming>



Linux Syscall

by: burt rosenberg
at: university of miami
Revised:

9 september 2020, new formatting and
copyright
11 September 2014
adapted from [csc521 semester 101](#)



System services

The syscall and the syscall table are the fundamental mechanism for getting system services. The kernel of the operating system exposes an API. These are the operations that can be called on the kernel. Because a program is not linked to the kernel, the call to a system service is done through pre-assigned addresses and operation codes. Also, a call to a kernel function changes the mode of the CPU, and the thread moves from user mode to kernel mode at the instant of the call.

System calls on the Intel architecture involve a special trap instruction. A trap instruction works similar to a call instruction, in that it pushes a return address on the stack and branches to a new instruction address. However, a syscall trap also:

- Changes the state of the processor from user mode to kernel, enabling wide privileges over memory and hardware;
- Branches to an address pre-stored in a protected syscall vector table. For the security of the kernel, only the kernel decides its entry points;
- Switches from the user stack which resides in user memory to a kernel stack that resides in protected kernel memory;
- Will require a special return instruction that will restore the user privilege level and the user stack when the return is executed.

Because the trap instruction is the same for all unix implementations, and the operation codes are the same across all Linux platforms, an application program can be compiled separately from the kernel and the syscall will work properly. The trap will enter the kernel at the syscall handing code, and the operation code will be used in a table of function pointers to execute the proper syscall functionality.

Starting in linux kernel 3.3, the syscall operation code is vectored through a table at

- [linux/arch/x86/syscall_32.tbl](#)

(Prior to that, it was at [linux/arch/x86/kernel/syscall_table_32.S](#), and was written in assembler.) Each entry names a function. As an example the `sys_setgid` function is in

- [linux/kernel/sys.c](#).

The `SYSCALL_DEFINE1` symbol is a macro that sets up the signature properly, including changing the `setgid` argument into `sys_setgid` as the name of the function.

In the Intel x86 architecture, the code immediately run in response to the trap is determined by a complicated thing called the GDT, the Global Descriptor Table. A pointer to this table is installed in the CPU during linux boot, and it has entries for various sorts of traps. Each entry is called a call gate which directs the change of privilege and gives the address of the function that will respond to the trap. All of the CPU context can change through a call gate. In particular, the stack is changed from the user stack to the kernel stack, and new data segments are installed that allow for full permission over the entire 4G virtual memory space.

More on traps

The syscall depends on a trap. A trap is one of several call-like instructions that perform special operations as part of the call. What they all have in common is that an event causes a call-like instruction branch, with the return address saved on the stack, with additional changes in processor mode.

The names and taxonomy of this special sorts of calls will vary from processor to processor. Intel breaks them down in *interrupts* and *exceptions*:

- *Interrupts* are calls that are forced by outside events, such as a full network card buffer. Interrupts can be *maskable* or *non-maskable*, meaning they can be ignored or cannot be ignored. There is usually a facility for levels of interrupts, such that a higher level interrupt can interrupt a lower level interrupt, but interrupts of same or lower level must wait.
- *Exceptions* are calls that are forced by some internal event in the processor. They can be *programmed*, meaning the event is an actual call to the exception (these are the syscall traps) or *non-programmed*.

Exceptions are classified as *faults*, *traps* or *aborts*.

- A *fault* requires that the faulting instruction be retried to continue program flow.
- A *trap* requires that the instruction following the trap be the location to continue program flow. In general, these are the programmed exceptions, and the trap is an instruction. These are used for syscalls, but also for breakpoints in program debugging.
- An *abort* means that the instruction flow cannot be continued. The aborting instruction completed partially and cannot be restarted or continued.



This work is licensed under a [Creative Commons Attribution-ShareAlike 3.0 Unported License](https://creativecommons.org/licenses/by-sa/3.0/).



glibc(7) - Linux man page

Name

libc - overview of standard C libraries on Linux

Description

The term "libc" is commonly used as a shorthand for the "standard C library", a library of standard functions that can be used by all C programs (and sometimes by programs in other languages). Because of some history (see below), use of the term "libc" to refer to the standard C library is somewhat ambiguous on Linux.

glibc

By far the most widely used C library on Linux is the GNU C Library often referred to as *glibc*. This is the C library that is nowadays used in all major Linux distributions. It is also the C library whose details are documented in the relevant pages of the *man-pages* project (primarily in Section 3 of the manual). Documentation of glibc is also available in the glibc manual, available via the command *info libc*. Release 1.0 of glibc was made in September 1992. (There were earlier 0.x releases.) The next major release of glibc was 2.0, at the beginning of 1997.

The pathname */lib/libc.so.6* (or something similar) is normally a symbolic link that points to the location of the glibc library, and executing this pathname will cause glibc to display various information about the version installed on your system.

Linux libc

In the early to mid 1990s, there was for a while *Linux libc*, a fork of glibc 1.x created by Linux developers who felt that glibc development at the time was not sufficing for the needs of Linux. Often, this library was referred to (ambiguously) as just "libc". Linux libc released major versions 2, 3, 4, and 5 (as well as many minor versions of those releases). For a while, Linux libc was the standard C library in many Linux distributions. However, notwithstanding the original motivations of the Linux libc effort, by the time glibc 2.0 was released, it was clearly superior to Linux libc, and all major Linux distributions that had been using Linux libc soon switched back to glibc. (Since this switch occurred over a decade ago, *man-pages* no longer takes care to document Linux libc details. Nevertheless, the history is visible in vestiges of information about Linux libc that remain in some manual pages, in particular, references to *libc4* and *libc5*.)

Other C libraries

There are various other less widely used C libraries for Linux. These libraries are generally smaller than glibc, both in terms of features and memory footprint, and often intended for building small binaries, perhaps targeted at development for embedded Linux systems. Among such libraries are *uClibc* (<http://www.uclibc.org/>) and *dietlibc* (<http://www.fefe.de/dietlibc/>). Details of these libraries are generally not covered by the *man-pages* project.

See Also

[syscalls\(2\)](#), [feature_test_macros\(7\)](#), [man-pages\(7\)](#), [standards\(7\)](#)



How to check glibc version on Linux

Last updated on July 9, 2020 by Dan Nanni

Question: I need to find out the version of the GNU C library (`glibc`) that I have on my Linux system. How can I check `glibc` version on Linux?

The GNU C library (`glibc`) is the GNU implementation of the standard C library. `glibc` is a critical component of the GNU toolchain, which is used along with binutils and compiler to generate userspace application binaries for a target architecture.

When built from source, some Linux programs may be required to link against a particular version of `glibc`. In that case, you may want to check out the information about installed `glibc` to see if dependencies are met.

Here are simple ways to check `glibc` version on Linux.

Method One

A simple command-line to check the version of the GNU C library is as follows.

```
xterm  
  
$ ldd --version
```

```
alice@caillou:~$ ldd --version  
ldd (Ubuntu EGLIBC 2.19-0ubuntu6) 2.19  
Copyright (C) 2014 Free Software Foundation, Inc.  
This is free software; see the source for copying conditions. There is NO  
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  
Written by Roland McGrath and Ulrich Drepper.  
alice@caillou:~$
```



In this example, the version of `glibc` is 2.19.

Method Two

Another method is to "type" the `glibc` library (i.e., `libc.so.6`) from the command line as if it were a command.

The output will show more detailed information about `glibc` library, including the version of `glibc` and the GNU compiler used, as well as available `glibc` extensions. The location of `glibc` varies depending on distros and processor architectures.

On 64-bit Debian based system:

```
xterm  
$ /lib/x86_64-linux-gnu/libc.so.6
```

on 32-bit Debian based system:

```
xterm  
$ /lib/i386-linux-gnu/libc.so.6
```

On 64-bit Red Hat based system:

```
xterm  
$ /lib64/libc.so.6
```

On 32-bit Red Hat based systems:

```
xterm  
$ /lib/libc.so.6
```

Here is the sample output of typing `glibc` library.



```

alice@caillou:~$ /lib/x86_64-linux-gnu/libc.so.6
GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6) stable release version 2.19, by Roland McGrath et al.
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Compiled by GNU CC version 4.8.2.
Compiled on a Linux 3.13.9 system on 2014-04-12.
Available extensions:
  crypt add-on version 2.1 by Michael Glad and others
  GNU Libidn by Simon Josefsson
  Native POSIX Threads Library by Ulrich Drepper et al
  BIND-8.2.3-T5B
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/eglibc/+bugs>.
alice@caillou:~$

```

Support Xmodulo

This website is made possible by minimal ads and your gracious donation via [PayPal](#) or [credit card](#)

Please note that this article is published by Xmodulo.com under a [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you would like to use the whole or any part of this article, you need to cite this web page at Xmodulo.com as the original source.

0 Comments

 Login ▾

G

Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name



Share

Best Newest Oldest

Be the first to comment.

Subscribe

Privacy

Do Not Sell My Data

Xmodulo © 2021 – [About](#) – [Write for Us](#) – [Feed](#) – Powered by [DigitalOcean](#)



LOG IN



opensource.com

A programmer's guide to GNU C Compiler

Get a behind-the-scenes look at the steps it takes to produce a binary file so that when something goes wrong, you know how to step through the process to resolve problems.

By [Jayashree Huttanagoudar](#) (Correspondent, Red Hat)

May 20, 2022 | [0 Comments](#) | 5 min read

7 readers like this.



Image by: [Opensource.com](#)

C is a well-known programming language, popular with experienced and new programmers alike. Source code written in C uses standard English terms, so it's considered human-readable. However,

computers only understand binary code. To convert code into machine language, you use a tool called a *compiler*.

A very common compiler is GCC (GNU C Compiler). The compilation process involves several intermediate steps and adjacent tools.

Install GCC

To confirm whether GCC is already installed on your system, use the `gcc` command:

```
$ gcc --version
```

If necessary, install GCC using your packaging manager. On Fedora-based systems, use `dnf`:

```
$ sudo dnf install gcc libgcc
```

On Debian-based systems, use `apt`:

```
$ sudo apt install build-essential
```

After installation, if you want to check where GCC is installed, then use:

```
$ whereis gcc
```

Simple C program using GCC

Here's a simple C program to demonstrate how to compile code using GCC. Open your favorite text editor and paste in this code:

```
// hellogcc.c
#include <stdio.h>

int main() {
    printf("Hello, GCC!\n");
    return 0;
}
```

Save the file as `hellogcc.c` and then compile it:

```
$ ls
hellogcc.c

$ gcc hellogcc.c

$ ls -l
a.out
hellogcc.c
```

As you can see, `a.out` is the default executable generated as a result of compilation. To see the output of your newly-compiled application, just run it as you would any local binary:

```
$ ./a.out
Hello, GCC!
```

Name the output file

The filename `a.out` isn't very descriptive, so if you want to give a specific name to your executable file, you can use the `-o` option:

```
$ gcc -o hellogcc hellogcc.c

$ ls
a.out  hellogcc  hellogcc.c

$ ./hellogcc
Hello, GCC!
```

This option is useful when developing a large application that needs to compile multiple C source files.

Programming and development

- [Red Hat Developers Blog](#)
- [Programming cheat sheets](#)
- [Try for free: Red Hat Learning Subscription](#)
- [eBook: An introduction to programming with Bash](#)
- [Bash shell scripting cheat sheet](#)



- 👉 [eBook: Modernizing Enterprise Java](#)
- 👉 [An open source developer's guide to building applications](#)
- 👉 [Register for your free Red Hat account](#)

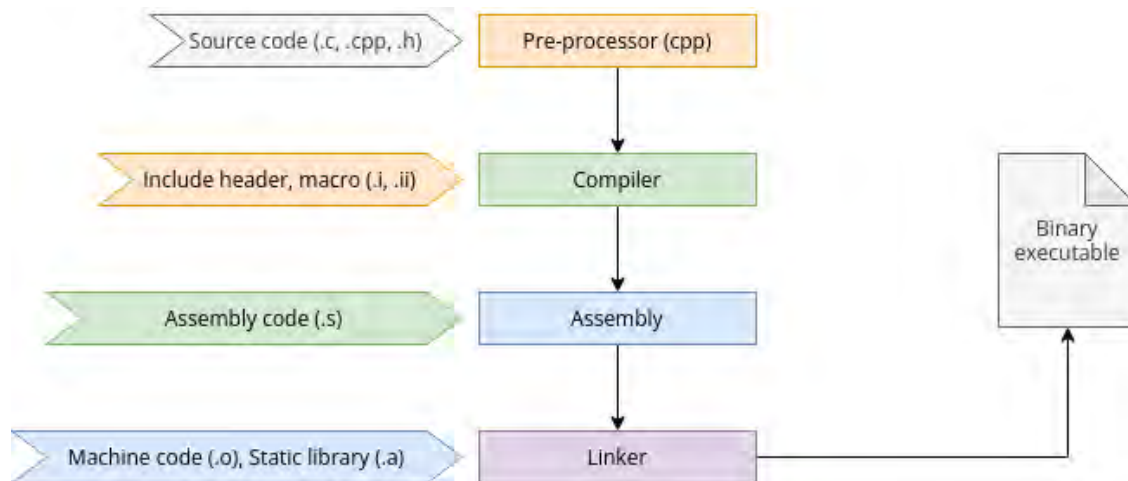
Intermediate steps in GCC compilation

There are actually four steps to compiling, even though GCC performs them automatically in simple use-cases.

1. Pre-Processing: The GNU C Preprocessor (`cpp`) parses the headers (**#include** statements), expands macros (**#define** statements), and generates an intermediate file such as `hellogcc.i` with expanded source code.
2. Compilation: During this stage, the compiler converts pre-processed source code into assembly code for a specific CPU architecture. The resulting assembly file is named with a `.s` extension, such as `hellogcc.s` in this example.
3. Assembly: The assembler (`as`) converts the assembly code into machine code in an object file, such as `hellogcc.o`.
4. Linking: The linker (`ld`) links the object code with the library code to produce an executable file, such as `hellogcc`.

When running GCC, use the `-v` option to see each step in detail.

```
$ gcc -v -o hellogcc hellogcc.c
```



(Jayashree Huttanagoudar, CC BY-SA 4.0)

Manually compile code

It can be useful to experience each step of compilation because, under some circumstances, you don't need GCC to go through all the steps.

First, delete the files generated by GCC in the current folder, except the source file.

```
$ rm a.out hellogcc.o  
  
$ ls  
hellogcc.c
```

Pre-processor

First, start the pre-processor, redirecting its output to `hellogcc.i`:

```
$ cpp hellogcc.c > hellogcc.i  
  
$ ls  
hellogcc.c hellogcc.i
```

Take a look at the output file and notice how the pre-processor has included the headers and expanded the macros.

Compiler

Now you can compile the code into assembly. Use the `-S` option to set GCC just to produce assembly code.

```
$ gcc -S hellogcc.i  
  
$ ls  
hellogcc.c hellogcc.i hellogcc.s  
  
$ cat hellogcc.s
```

Take a look at the assembly code to see what's been generated.

Assembly

Use the assembly code you've just generated to create an object file:

```
$ as -o hellogcc.o hellogcc.s
```



```
$ ls
hellogcc.c  hellogcc.i  hellogcc.o  hellogcc.s
```

Linking

To produce an executable file, you must link the object file to the libraries it depends on. This isn't quite as easy as the previous steps, but it's educational:

```
$ ld -o hellogcc hellogcc.o
ld: warning: cannot find entry symbol _start; defaulting to 0000000000401000
ld: hellogcc.o: in function `main`:
hellogcc.c:(.text+0xa): undefined reference to `puts'
```

An error referencing an `undefined puts` occurs after the linker is done looking at the `libc.so` library. You must find suitable linker options to link the required libraries to resolve this. This is no small feat, and it's dependent on how your system is laid out.

When linking, you must link code to core runtime (CRT) objects, a set of subroutines that help binary executables launch. The linker also needs to know where to find important system libraries, including `libc` and `libgcc`, notably within special start and end instructions. These instructions can be delimited by the `--start-group` and `--end-group` options or using paths to `crtbegin.o` and `crtend.o`.

This example uses paths as they appear on a RHEL 8 install, so you may need to adapt the paths depending on your system.

```
$ ld -dynamic-linker \
/lib64/ld-linux-x86-64.so.2 \
-o hello \
/usr/lib64/crt1.o /usr/lib64/crti.o \
--start-group \
-L/usr/lib/gcc/x86_64-redhat-linux/8 \
-L/usr/lib64 -L/lib64 hello.o \
-lgcc \
--as-needed -lgcc_s \
--no-as-needed -lc -lgcc \
--end-group
/usr/lib64/crtn.o
```

The same linker procedure on Slackware uses a different set of paths, but you can see the similarity in the process:



```
$ ld -static -o hello \  
-L/usr/lib64/gcc/x86_64-slackware-linux/11.2.0/ \  
/usr/lib64/crt1.o /usr/lib64/crti.o \  
hello.o /usr/lib64/crtn.o \  
--start-group -lc -lgcc -lgcc_eh \  
--end-group
```

Now run the resulting executable:

```
$ ./hello  
Hello, GCC!
```

Some helpful utilities

Below are a few utilities that help examine the file type, symbol table, and the libraries linked with the executable.

Use the `file` utility to determine the type of file:

```
$ file hellogcc.c  
hellogcc.c: C source, ASCII text  
  
$ file hellogcc.o  
hellogcc.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped  
  
$ file hellogcc  
hellogcc: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2
```

The use the `nm` utility to list symbol tables for object files:

```
$ nm hellogcc.o  
0000000000000000 T main  
U puts
```

Use the `ldd` utility to list dynamic link libraries:

```
$ ldd hellogcc  
linux-vdso.so.1 (0x00007ffe3bdd7000)
```

```
libc.so.6 => /lib64/libc.so.6 (0x00007f223395e000)
/lib64/ld-linux-x86-64.so.2 (0x00007f2233b7e000)
```

Wrap up

In this article, you learned the various intermediate steps in GCC compilation and the utilities to examine the file type, symbol table, and libraries linked with an executable. The next time you use GCC, you'll understand the steps it takes to produce a binary file for you, and when something goes wrong, you know how to step through the process to resolve problems.

Tags:

PROGRAMMING



Jayashree Huttanagoudar

Jayashree Huttanagoudar is a Senior Software Engineer at Red Hat India Pvt Ltd. She works with Middleware OpenJDK team. She is always curious to learn new things which adds to her work.

[More about me](#)

Comments are closed.

These comments are closed.

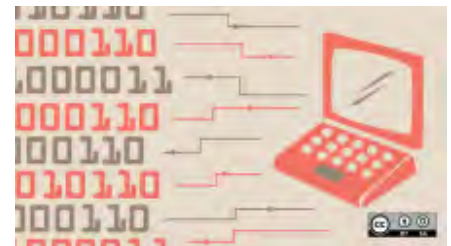
Related Content



[C vs. Go: Comparing programming languages](#)



[Learn Tcl/Tk and Wish with this simple game](#)



[BASIC vs. FORTRAN 77: Comparing programming](#)



blasts from the past



This work is licensed under a Creative Commons Attribution-Share Alike 4.0 International License.

ABOUT THIS SITE

The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.

Opensource.com aspires to publish all content under a **Creative Commons license** but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.

A note on advertising: Opensource.com does not sell advertising on the site or in any of its newsletters.

opensource.com

Copyright ©2024 Red Hat, Inc.

[Privacy Policy](#)

[Terms of use](#)

[Cookie preferences](#)





LOG IN



opensource.com

A 10-minute guide to the Linux ABI

Familiarize yourself with the concept of an ABI, why ABI stability matters, and what is included in Linux's stable ABI.

By [Alison Chaiken](#)

December 6, 2022 | [0 Comments](#) | 9 min read

7 readers like this.



Image by: *Internet Archive Book Images. Modified by Opensource.com. CC BY-SA 4.0*

Many Linux enthusiasts are familiar with Linus Torvalds' [famous admonition](#), "we don't break user space," but perhaps not everyone who recognizes the phrase is certain about what it means.

The "#1 rule" reminds developers about the stability of the applications' binary interface via which applications communicate with and configure the kernel. What follows is intended to familiarize



readers with the concept of an ABI, describe why ABI stability matters, and discuss precisely what is included in Linux's stable ABI. The ongoing growth and evolution of Linux necessitate changes to the ABI, some of which have been controversial.

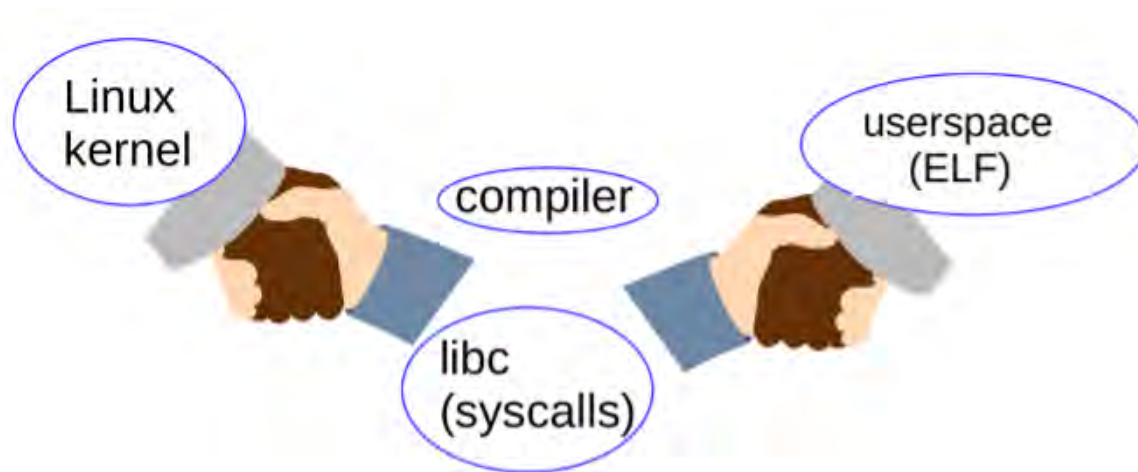
What is an ABI?

ABI stands for Applications Binary Interface. One way to understand the concept of an ABI is to consider what it is not. Applications Programming Interfaces (APIs) are more familiar to many developers. Generally, the headers and documentation of libraries are considered to be their API, as are standards documents like those for [HTML5](#), for example. Programs that call into libraries or exchange string-formatted data must comply with the conventions described in the API or expect unwanted results.

ABIs are similar to APIs in that they govern the interpretation of commands and exchange of binary data. For C programs, the ABI generally comprises the return types and parameter lists of functions, the layout of structs, and the meaning, ordering, and range of enumerated types. The Linux kernel remains, as of 2022, almost entirely a C program, so it must adhere to these specifications.

"[The kernel syscall interface](#)" is described by [Section 2 of the Linux man pages](#) and includes the C versions of familiar functions like "mount" and "sync" that are callable from middleware applications. The binary layout of these functions is the first major part of Linux's ABI. In answer to the question, "What is in Linux's stable ABI?" many users and developers will respond with "the contents of sysfs (/sys) and procfs (/proc)." In fact, the [official Linux ABI documentation](#) concentrates mostly on these [virtual filesystems](#).

The preceding text focuses on how the Linux ABI is exercised by programs but fails to capture the equally important human aspect. As the figure below illustrates, the functionality of the ABI requires a joint, ongoing effort by the kernel community, C compilers (such as [GCC](#) or [clang](#)), the developers who create the userspace C library (most commonly [glibc](#)) that implements system calls, and binary applications, which much be laid out in accordance with the Executable and Linking Format ([ELF](#)).



(Alison Chaiken, CC BY-SA 4.0)

Why do we care about the ABI?

The Linux ABI stability guarantee that comes from Torvalds himself enables Linux distros and individual users to update the kernel independently of the operating system.

If Linux did not have a stable ABI, then every time the kernel needed patching to address a security problem, a large part of the operating system, if not the entirety, would need to be reinstalled. Obviously, the stability of the binary interface is a major contributing factor to Linux's usability and wide adoption.

```
$ # Here is a simple example of the kernel-glibc interface.
$
$ # Linux kernel header which defines file permissions:
$ apt-file find -F /usr/include/linux/stat.h
linux-libc-dev: /usr/include/linux/stat.h
$
$ # glibc header which defines file permissions:
$ apt-file find -F /usr/include/x86_64-linux-gnu/sys/stat.h
libc6-dev: /usr/include/x86_64-linux-gnu/sys/stat.h
$
$ # The files are not the same:
$ shasum /usr/include/linux/stat.h /usr/include/x86_64-linux-gnu/sys/stat.h
167fe074b2e934ee0c6a0f4201f09969138f0dc3 /usr/include/linux/stat.h
f28662bbb962063901e7439f13717855cf500d1b /usr/include/x86_64-linux-gnu/sys/stat.h
$
$ # Implementation details with no stability guarantee are found in bits directory:
$ grep -i never /usr/include/x86_64-linux-gnu/bits/stat.h
# error "Never include <bits/stat.h> directly; use <sys/stat.h> instead."
```

(Alison Chaiken, CC BY-SA 4.0)

More Linux resources

- [Linux commands cheat sheet](#)
- [Advanced Linux commands cheat sheet](#)

- [Free online course: RHEL technical overview](#)
- [Linux networking cheat sheet](#)
- [SELinux cheat sheet](#)
- [Linux common commands cheat sheet](#)
- [What are Linux containers?](#)
- [Register for your free Red Hat account](#)
- [Our latest Linux articles](#)

As the second figure illustrates, both the kernel (in `linux-libc-dev`) and Glibc (in `libc6-dev`) provide bitmasks that define file permissions. Obviously the two sets of definitions must agree! The `apt` package manager identifies which software project provided each file. The potentially unstable part of Glibc's ABI is found in the `bits/` directory.

For the most part, the Linux ABI stability guarantee works just fine. In keeping with [Conway's Law](#), vexing technical issues that arise in the course of development most frequently occur due to misunderstandings or disagreements between different software development communities that contribute to Linux. The interface between communities is easy to envision via Linux package-manager metadata, as shown in the image above.

Y2038: An example of an ABI break

The Linux ABI is best understood by considering the example of the ongoing, [slow-motion](#) "Y2038" ABI break. In January 2038, 32-bit time counters will roll over to all zeroes, just like the odometer of an older vehicle. January 2038 sounds far away, but assuredly many IoT devices sold in 2022 will still be operational. Mundane products like [smart electrical meters](#) and [smart parking systems](#) installed this year may or may not have 32-bit processor architectures and may or may not support software updates.

The Linux kernel has already moved to a 64-bit `time_t` opaque data type internally to represent later timepoints. The implication is that system calls like `time()` have already changed their function signature on 64-bit systems. The arduousness of these efforts is on ready display in kernel headers like [time_types.h](#), which includes new and "_old" versions of data structures.



([marneejill](#), CC BY-SA 2.0)

The Glibc project also [supports 64-bit time](#), so yay, we're done, right? Unfortunately, no, as a [discussion on the Debian mailing list](#) makes clear. Distros are faced with the unenviable choice of either providing two versions of all binary packages for 32-bit systems or two versions of installation media. In the latter case, users of 32-bit time will have to recompile their applications and reinstall. As always, proprietary applications will be a real headache.

What precisely is in the Linux stable ABI anyway?

Understanding the stable ABI is a bit subtle. Consider that, while most of sysfs is stable ABI, the debug interfaces are guaranteed to be *unstable* since they expose kernel internals to userspace. In general, Linus Torvalds has pronounced that by "don't break userspace," he means to protect ordinary users who "just want it to work" rather than system programmers and kernel engineers, who should be able to read the kernel documentation and source code to figure out what has changed between releases. The distinction is illustrated in the figure below.





(Alison Chaiken, CC BY-SA 4.0)

Ordinary users are unlikely to interact with unstable parts of the Linux ABI, but system programmers may do so inadvertently. All of `sysfs` (`/sys`) and `procfs` (`/proc`) are guaranteed stable except for `/sys/kernel/debug`.

But what about other binary interfaces that are userspace-visible, including miscellaneous ABI bits like device files in `/dev`, the kernel log file (readable with the `dmesg` command), filesystem metadata, or "bootargs" provided on the kernel "command line" that are visible in a bootloader like GRUB or u-boot? Naturally, "it depends."

Mounting old filesystems

Next to observing a Linux system hang during the boot sequence, having a filesystem fail to mount is the greatest disappointment. If the filesystem resides on an SSD belonging to a paying customer, the matter is grave indeed. Surely a Linux filesystem that mounts with an old kernel version will still mount when the kernel is upgraded, right? Actually, "[it depends](#)."

In 2020 an aggrieved Linux developer [complained on the kernel's mailing list](#):

The kernel already accepted this as a valid mountable filesystem format, without a single error or warning of any kind, and has done so stably for years. . . . I was generally under the impression that mounting existing root filesystems fell under the scope of the kernel<->userspace or kernel<->existing-system boundary, as defined by what the kernel accepts and existing userspace has used successfully, and that upgrading the kernel should work with existing userspace and systems.



But there was a catch: The filesystems that failed to mount were created with a proprietary tool that relied on a flag that was defined but not used by the kernel. The flag did not appear in Linux's API header files or procfs/sysfs but was instead an [implementation detail](#). Therefore, interpreting the flag in userspace code meant relying on "[undefined behavior](#)," a phrase that will make software developers almost universally shudder. When the kernel community improved its internal testing and started making new consistency checks, the "[man 2 mount](#)" system call suddenly began rejecting filesystems with the proprietary format. Since the format creator was decidedly a software developer, he got little sympathy from kernel filesystem maintainers.



(Kernel developers working in-tree are protected from ABI changes. Alison Chaiken, CC BY-SA 4.0)

Threading the kernel dmesg log

Is the format of files in `/dev` guaranteed stable or not? The [command dmesg](#) reads from the file `/dev/kmsg`. In 2018, a developer [made output to dmesg threaded](#), enabling the kernel "to print a series of `printk()` messages to consoles without being disturbed by concurrent `printk()` from interrupts and/or other threads." Sounds excellent! Threading was made possible by adding a thread ID to each line of the `/dev/kmsg` output. Readers following closely will realize that the addition changed the ABI of `/dev/kmsg`, meaning that applications that parse that file needed to change too. Since many distros didn't compile their kernels with the new feature enabled, most users of `/bin/dmesg` won't have noticed, but the change broke the [GDB debugger](#)'s ability to read the kernel log.

Assuredly, astute readers will think users of GDB are out of luck because debuggers are developer tools. Actually, no, since the code that needed to be updated to support the new `/dev/kmsg` format was "[in-tree](#)," meaning part of the kernel's own Git source repository. The failure of programs within a single repo to work together is just an out-and-out bug for any sane project, and a [patch that made GDB work with threaded /dev/kmsg](#) was merged.

What about BPF programs?

[BPF](#) is a powerful tool to monitor and even configure the running kernel dynamically. BPF's original purpose was to support on-the-fly network configuration by allowing sysadmins to modify packet



filters from the command line instantly. [Alexei Starovoitov and others greatly extended BPF](#), giving it the power to trace arbitrary kernel functions. Tracing is clearly the domain of developers rather than ordinary users, so it is certainly not subject to any ABI guarantee (although the [bpf\(\) system call](#) has the same stability promise as any other). On the other hand, BPF programs that create new functionality present the possibility of "[replacing kernel modules as the de-facto means of extending the kernel](#)." Kernel modules make devices, filesystems, crypto, networks, and the like work, and therefore clearly are a facility on which the "just want it to work" user relies. The problem arises that BFP programs have not traditionally been "in-tree" as most open-source kernel modules are. A proposal in spring 2022 to [provide support to the vast array of human interface devices \(HIDs\) like mice and keyboards via tiny BPF programs](#) rather than patches to device drivers brought the issue into sharp focus.

A rather heated discussion followed, but the issue was apparently settled by [Torvalds' comments at Open Source Summit](#):

He specified if you break 'real user space tools, that normal (non-kernel developers) users use,' then you need to fix it, regardless of whether it is using eBPF or not.

A consensus appears to be forming that developers who expect their BPF programs to withstand kernel updates [will need to submit them to an as-yet unspecified place in the kernel source repository](#). Stay tuned to find out what policy the kernel community adopts regarding BPF and ABI stability.

Conclusion

The kernel ABI stability guarantee applies to procfs, sysfs, and the system call interface, with important exceptions. When "in-tree" code or userspace applications are "broken" by kernel changes, the offending patches are typically quickly reverted. When proprietary code relies on kernel implementation details that are incidentally accessible from userspace, it is not protected and garners little sympathy when it breaks. When, as with Y2038, there is no way to avoid an ABI break, the transition is made as thoughtfully and methodically as possible. Newer features like BPF programs present as-yet-unanswered questions about where exactly the ABI-stability border lies.

Acknowledgments

Thanks to [Akkana Peck](#), [Sarah R. Newman](#), and [Luke S. Crawford](#) for their helpful comments on early versions of this material.

Tags:

LINUX

SCALE





Alison Chaiken

Alison Chaiken is a software developer who rides bikes in Mountain View, CA. Her day job is maintaining the Linux kernel and writing operating-system-monitoring applications in C++ for Aurora Innovation. Alison has contributed upstream to u-boot, kernel, bazel and systemd and presented talks at Embedded Linux Conference, Usenix, linux.conf.au and Southern California Linux Expo.

[More about me](#)

Comments are closed.

These comments are closed.

Related Content



[What's new in GNOME 44?](#)



[5 reasons virtual machines still matter](#)



[Remove the background from an image with this Linux command](#)



This work is licensed under a Creative Commons Attribution-Share Alike 4.0 International License.

ABOUT THIS SITE



The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.

Opensource.com aspires to publish all content under a **Creative Commons license** but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.

A note on advertising: Opensource.com does not sell advertising on the site or in any of its newsletters.

opensource.com

Copyright ©2024 Red Hat, Inc.

[Privacy Policy](#)

[Terms of use](#)

[Cookie preferences](#)



What is POSIX - Richard Stallman explains - Opensource.com



LOG IN



opensource.com

What is POSIX? Richard Stallman explains

Discover what's behind the standards for operating system compatibility from a pioneer of computer freedom.

By [Seth Kenlon](#) (Team, Red Hat)

July 15, 2019 | [4 Comments](#) | 10 min read

 206 readers like this.



Image by: [Opensource.com](#)

What is [POSIX](#), and why does it matter? It's a term you've likely seen in technical writing, but it often gets lost in a sea of techno-initialisms and jargon-that-ends-in-X. I emailed Dr. [Richard Stallman](#) (better known in hacker circles as RMS) to find out more about the term's origin and the concept behind it.



Richard Stallman says "open" and "closed" are the wrong way to classify software. Stallman classifies programs as *freedom-respecting* ("free" or "libre") and *freedom-trampling* ("non-free" or "proprietary"). Open source discourse typically encourages certain practices for the sake of practical advantages, not as a moral imperative.

The free software movement, which Stallman launched in 1984, says more than *advantages* are at stake. Users of computers *deserve* control of their computing, so programs denying users control are an injustice to be rejected and eliminated. For users to have control, the program must give them the [four essential freedoms](#):

- The freedom to run the program as you wish, for any purpose (freedom 0).
- The freedom to study how the program works and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help others (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

About POSIX

Seth: The POSIX standard is a document released by the [IEEE](#) that describes a "portable operating system." As long as developers write programs to match this description, they have produced a POSIX-compliant program. In the tech industry, we call this a "specification" or "spec" for short. That's mostly understandable, as far as tech jargon goes, but what makes an operating system "portable"?

RMS: I think it was the *interface* that was supposed to be portable (among systems), rather than any one *system*. Indeed, various systems that are different inside do support parts of the POSIX interface spec.

Seth: So if two systems feature POSIX-compliant programs, then they can make assumptions about one another, enabling them to know how to "talk" to one another. I read that it was you who came up with the name "POSIX." How did you come up with the term, and how was it adopted by the IEEE?

RMS: The IEEE had finished developing the spec but had no concise name for it. The title said something like "portable operating system interface," though I don't remember the exact words. The committee put on "IEEEIX" as the concise name. I did not think that was a good choice. It is ugly to pronounce—it would sound like a scream of terror, "Ayeeee!"—so I expected people would instead call the spec "Unix."



Since [GNU's Not Unix](#), and it was intended to replace Unix, I did not want people to call GNU a "Unix system." I, therefore, proposed a concise name that people might actually use. Having no particular inspiration, I generated a name the unclever way: I took the initials of "portable operating system" and added "ix." The IEEE adopted this eagerly.

Seth: Does "operating system" in the POSIX acronym refer only to Unix and Unix-like systems such as GNU, or is the intent to encompass all operating systems?

RMS: The term "operating system," in the abstract, covers systems that are not at all like Unix, not at all close to the POSIX spec. However, the spec is meant for systems that are a lot like Unix; only such systems will fit the POSIX spec.

Seth: Are you involved in reviewing or updating the current POSIX standards?

RMS: Not now.

Seth: The GNU Autotools toolchain does a lot to make applications easier to port, at least in terms of when it comes time to build and install. Is Autotools an important part of building a portable infrastructure?

RMS: Yes, because even among systems that follow POSIX, there are lots of little differences. The Autotools make it easier for a program to adapt to those differences. By the way, if anyone wants to help in the development of the Autotools, please email me.

Seth: I imagine, way back when GNU was just starting to make people realize that a (not)Unix liberated from proprietary technology was possible, there must have been a void of clarity about how free software could possibly work together.

RMS: I don't think there was any void or any uncertainty. I was simply going to follow the interfaces of BSD.

Seth: Some GNU applications are POSIX-compliant, while others have GNU-specific features either not in the POSIX spec or lack features required by the spec. How important is POSIX compliance to GNU applications?

RMS: Following a standard is important to the extent it serves users. We do not treat a standard as an authority, but rather as a guide that may be useful to follow. Thus, we talk about following standards rather than "complying" with them. See the section [Non-GNU Standards](#) in the GNU Coding Standards.



We strive to be compatible with standards on most issues because, on most issues, that serves users best. But there are occasional exceptions.

For instance, POSIX specifies that some utilities measure disk space in units of 512 bytes. I asked the committee to change this to 1K, but it refused, saying that a bureaucratic rule compelled the choice of 512. I don't recall much attempt to argue that users would be pleased with that decision.

Since GNU's second priority, after users' freedom, is users' convenience, we made GNU programs measure disk space in blocks of 1K by default.

However, to defend against potential attacks by competitors who might claim that this deviation made GNU "noncompliant," we implemented optional modes that follow POSIX and ISO C to ridiculous extremes. For POSIX, setting the environment variable `POSIXLY_CORRECT` makes programs specified by POSIX list disk space in terms of 512 bytes. If anyone knows of a case of an actual use of `POSIXLY_CORRECT` or its GCC counterpart **--pedantic** that provides an actual benefit to some user, please tell me about it.

Seth: Are POSIX-compliant free software projects easier to port to other Unix-like systems?

RMS: I suppose so, but I decided in the 1980s not to spend my time on porting software to systems other than GNU. I focused on advancing the GNU system towards making it unnecessary to use any non-free software and left the porting of GNU programs to non-GNU-like systems to whoever wanted to run them on those systems.

Seth: Is POSIX important to software freedom?

RMS: At the fundamental level, it makes no difference. However, standardization by POSIX and ISO C surely made the GNU system easier to migrate to, and that helped us advance more quickly towards the goal of liberating users from non-free software. That was achieved in the early 1990s, when Linux was made free software and then filled the kernel-shaped gap in GNU.

GNU innovations adopted by POSIX

I also asked Dr. Stallman whether any GNU-specific innovations or conventions had later become adopted as a POSIX standard. He couldn't recall specific examples, but kindly emailed several developers on my behalf.

Developers Giacomo Catenazzi, James Youngman, Eric Blake, Arnold Robbins, and Joshua Judson Rosen all responded with memories of previous POSIX iterations as well as ones still in progress. POSIX is a "living" standard, so it's being updated and reviewed by industry professionals



continuously, and many developers who work on GNU projects propose the inclusion of GNU features.

For the sake of historical interest, here' are some popular GNU features that have made their way into POSIX.

Make

Some GNU **Make** features have been adopted into POSIX's definition of **make**. The relevant [specification](#) provides detailed attribution for features borrowed from existing implementations.

Diff and patch

Both the [diff](#) and [patch](#) commands have gotten **-u** and **-U** options added directly from GNU versions of those tools.

C library

Many features of the GNU C library, **glibc**, have been adopted in POSIX. Lineage is sometimes difficult to trace, but James Youngman wrote:

"I'm pretty sure there are a number of features of ISO C which were pioneered by GCC. For example, **_Noreturn** is new in C11, but GCC-1.35 had this feature (one used the **volatile** modifier on a function declaration). Also—though I'm not certain about this—GCC-1.35 supported Arrays of Variable Length which seem to be very similar to modern C's conformant arrays."

Giacomo Catenazzi cites the Open Group's [strftime article](#), pointing to this attribution: "This is based on a feature in some versions of GNU libc's **strftime()**."

Eric Blake notes that the **getline()** and various ***_l()** locale-based functions were definitely pioneered by GNU.

Joshua Judson Rosen adds to this, saying he clearly remembers being impressed by the adoption of **getline** functions after witnessing strangely familiar GNU-like behavior from code meant for a different OS entirely.

"Wait...that's GNU-specific... isn't it? Oh—not anymore, apparently."

Rosen pointed me to the [getline man page](#), which says:

Both **getline()** and **getdelim()** were originally GNU extensions. They were standardized in POSIX.1-2008.

Eric Blake sent me a list of other extensions that may be added in the next POSIX revision (codenamed Issue 8, currently due around 2021):

- [ppoll](#)
- [pthread_cond_clockwait et al.](#)
- [posix_spawn_file_actions_addchdir](#)
- [getlocalename_1](#)
- [reallocarray](#)

Userspace extensions

POSIX doesn't just define functions and features for developers. It defines standard behavior for userspace as well.

ls

The **-A** option is used to exclude the **.** (representing the current location) and **..** (representing the opportunity to go back one directory) notation from the results of an **ls** command. This was adopted for POSIX 2008.

find

The **find** command is a useful tool for ad hoc [for loops](#) and as a gateway into [parallel](#) processing.

A few conveniences made their way from GNU to POSIX, including the **-path** and **-perm** options.

The **-path** option lets you filter search results matching a filesystem path pattern and was available in GNU's version of **find** since before 1996 (the earliest record in **findutils**'s Git repository). James Youngman notes that [HP-UX](#) also had this option very early on, so whether it's a GNU or an HP-UX innovation (or both) is uncertain.

The **-perm** option lets you filter search results by file permission. This was in GNU's version of **find** by 1996 and arrived later in the POSIX standard "IEEE Std 1003.1, 2004 Edition."

The **xargs** command, part of the **findutils** package, had a **-p** option to put **xargs** into an interactive mode (the user is prompted whether to continue or not) by 1996, and it arrived in POSIX in "IEEE Std 1003.1, 2004 Edition."



Awk

Arnold Robbins, the maintainer of GNU **awk** (the **gawk** command in your **/usr/bin** directory, probably the destination of the symlink **awk**), says that **gawk** and **mawk** (another GPL **awk** implementation) allow **RS** to be a regular expression, which is the case when **RS** has a length greater than 1. This isn't a feature in POSIX yet, but there's an [indication that it will be](#):

The undefined behavior resulting from NULs in extended regular expressions allows future extensions for the GNU gawk program to process binary data.

The unspecified behavior from using multi-character RS values is to allow possible future extensions based on extended regular expressions used for record separators. Historical implementations take the first character of the string and ignore the others.

This is a significant enhancement because the **RS** notation defines a separator between records. It might be a comma or a semicolon or a dash or any such character, but if it's a *sequence* of characters, then only the first character is used unless you're working in **gawk** or **mawk**. Imagine parsing a document of IP addresses with records separated by an ellipsis (three dots in a row), only to get back results parsed at every dot in every IP address.

Mawk supported the feature first, but it was without a maintainer for several years, leaving **gawk** to carry the torch. (**Mawk** has since gained a new maintainer, so arguably credit can be shared for pushing this feature into the collective expectation.)

The POSIX spec

In general, Giacomo Catenzzi points out, "...because GNU utilities were used so much, a lot of other options and behaviors were aligned. At every change in shell, Bash is used as comparison (as a first-class citizen)." There's no requirement to cite GNU or any other influence when something is rolled into the POSIX spec, and it can safely be assumed that influences to POSIX come from many sources, with GNU being only one of many.

The significance of POSIX is consensus. A group of technologists working together toward common specifications to be shared by hundreds of uncommon developers lends strength to the greater movement toward software independence and developer and user freedom.



Seth Kenlon

Seth Kenlon is a UNIX geek, free culture advocate, independent multimedia artist, and D&D nerd. He has worked in the film and





computing industry, often at the same time.

[More about me](#)

4 Comments

These comments are closed.



[Novid Emami](#) | August 4, 2019

No readers like this yet.

Who's better than RMS to explain POSIX with this level of simplicity?! Thanks for this great interview Mr. Kenlon ?



[Seth Kenlon](#) | August 4, 2019

No readers like this yet.

Glad you enjoyed it! Thanks for reading.



Douglas Goodall | September 26, 2019

No readers like this yet.

I guess my nod to POOSIX is including stdlib and using EXIT_SUCCESS and EXIT_FAILURE as standard return codes in my software, rather than inventing something else.



[Seth Kenlon](#) | September 26, 2019

No readers like this yet.

Today, stdlib. Tomorrow, libposix!





This work is licensed under a Creative Commons Attribution-Share Alike 4.0 International License.

ABOUT THIS SITE

The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.

Opensource.com aspires to publish all content under a **Creative Commons license** but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.

A note on advertising: Opensource.com does not sell advertising on the site or in any of its newsletters.

opensource.com

Copyright ©2024 Red Hat, Inc.

[Privacy Policy](#)

[Terms of use](#)

[Cookie preferences](#)



ROSEdu Techblog

[ABOUT](#)

[PEOPLE](#)

[ARCHIVE](#)

[TAGS](#)

Latest Posts

[APPLICATION PRO...](#)

[HERE BE DRAGONS...](#)

[DAEMONIZING PRO...](#)

Unix standards and implementations. Unix portability

Published on February 2, 2014 by [Alexandru Goia](#)

Tagged: [Unix](#), [C](#), [portable code](#), [POSIX](#), [SUS](#), [libc](#)

The purpose of this article is to present in a general way the Unix standards and how can we write portable code on Unix systems, not only on Linux ones.

In the Unix world at present, there are three important standards:

- the C language (ISO C standard) and the standard C library (`libc`), which are included in the POSIX standard
- the POSIX standard (*Portable Operating System Interface for Unix*), which has the last version from 2008
- the SUS standard (*Single Unix Specification*), which includes as a subset the POSIX standard, with the last version from 2010 (SUSv4).

The POSIX standard consists of:

- POSIX.1: core services
- POSIX.1b: real-time extensions

- POSIX.1c: threads extensions
- POSIX.2: shell and utilities

We will be interested in this article only by POSIX.1 (last version: POSIX.1-2008, or IEEE Std 1003.1-2008) from the whole POSIX standard.

As implementations of the standard, we can name the GNU/Linux-based operating systems, the systems which descend from the BSD Unix version: FreeBSD, NetBSD, OpenBSD, DragonflyBSD, the certified and commercial UNIX-es, based on UNIX System V release 4 and, from case to case, with BSD elements: Oracle Solaris (known previously as Sun Solaris), HP-UX and Tru64 UNIX (HP), AIX (IBM), IRIX (SGI), Unixware and OpenServer (SCO), and also Mac OS X, which is also officially certified as a UNIX system, based on FreeBSD elements and not on UNIX System V.

The most “popular” Unix systems are at present Linux, FreeBSD, Solaris and Mac OS X. With regards to the C language, all these operating systems (Linux 3.x, FreeBSD >= 8.0, Mac OS X >= 10.6.8, Solaris >= 10) support the following LIB C headers:

- `assert.h` : verify program assertion
- `complex.h` : complex arithmetic support
- `ctype.h` : character classification and mapping support
- `errno.h` : error codes
- `fenv.h` : floating-point environment
- `float.h` : floating-point constants and characteristics
- `inttypes.h` : integer type format conversion
- `iso646.h` : macros for assignment, relational, and unary operators
- `limits.h` : implementation constants
- `locale.h` : locale categories and related definitions
- `math.h` : mathematical functions and type declarations and constants
- `setjmp.h` : nonlocal `goto`
- `signal.h` : signals
- `stdarg.h` : variable argument lists
- `stdbool.h` : boolean type and values
- `stddef.h` : standard definitions
- `stdint.h` : integer types
- `stdio.h` : standard I/O library
- `stdlib.h` : utility functions
- `string.h` : string operations
- `tgmath.h` : type-generic math macros
- `time.h` : time and date
- `wchar.h` : extended multibyte and wide character support



- `wctype.h` : wide character classification and mapping support

They also support the following POSIX headers (in the C language):

- `aio.h` : asynchronous I/O
- `cpio.h` : cpio archive values
- `dirent.h` : directory entries
- `dlfcn.h` : dynamic linking
- `fcntl.h` : file control
- `fnmatch.h` : filename-matching types
- `glob.h` : pathname pattern-matching and generations
- `grp.h` : group file
- `iconv.h` : codeset conversion utility
- `langinfo.h` : language information constants
- `monetary.h` : monetary types and functions
- `netdb.h` : network database operations
- `nl_types.h` : message catalogs
- `poll.h` : `poll()` function
- `pthread.h` : threads
- `pwd.h` : password file
- `regex.h` : regular expressions
- `sched.h` : execution scheduling
- `semaphore.h` : semaphores
- `strings.h` : string operations
- `tar.h` : tar archive values
- `termios.h` : terminal I/O
- `unistd.h` : symbolic constants
- `wordexp.h` : word-expansion definitions
- `arpa/inet.h` : Internet definitions
- `net/if.h` : socket local interfaces
- `netinet/in.h` : Internet address family
- `netinet/tcp.h` : TCP definitions
- `sys/mman.h` : memory management declarations
- `sys/select.h` : `select()` function
- `sys/socket.h` : sockets interface
- `sys/stat.h` : file status
- `sys/statvfs.h` : file system information
- `sys/times.h` : process times
- `sys/types.h` : primitive system data types
- `sys/un.h` : UNIX domain socket definitions
- `sys/utsname.h` : system name



- `sys/wait.h` : process control
- `fmtmsg.h` : message display structures
- `ftw.h` : file tree walking
- `libgen.h` : pathname management functions
- `ndbm.h` : database operations (*exception: Linux*)
- `search.h` : search tables
- `syslog.h` : system error logging
- `utmpx.h` : user accounting database (*exception: FreeBSD*)
- `sys/ipc.h` : inter-processes communication
- `sys/msg.h` : XSI message queues
- `sys/resource.h` : resource operations
- `sys/sem.h` : XSI semaphores
- `sys/shm.h` : XSI shared memory
- `sys/time.h` : time types
- `sys/uio.h` : vector I/O operations
- `mqueue.h` : message queues (exception: Mac OS X)
- `spawn.h` : real-time spawn interface.

The SUS standard (the whole set of UNIX functions and constants) can be found online for [SUSv2](#) (year 1997, naming UNIX 98), [SUSv3](#) (year 2001-2002, naming UNIX 03) and [SUSv4](#) (year 2010):

To write portable code which can be executed on any Unix systems we must know the C headers (defined by LIBC and by POSIX) which are recognized by the Unix systems. We can activate the operating system in order to use only POSIX.1 elements, or also SUSv1, SUSv2, SUSv3, or SUSv4 using the so-called “feature test macros”:

- `_POSIX_SOURCE` and `_POSIX_C_SOURCE`, to activate POSIX functionality
- `_XOPEN_SOURCE`, which activates SUSv1/2/3/4 functionality.

For older POSIX functionality we have to declare the following in our source file:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 1 /* for POSIX 1990 */
/* use 2 for POSIX C bindings 1003.2-1992 */
```

For POSIX 2008 functionality, we define:




```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 200809L
```

Or, we can compile with:

```
cc -D_POSIX_SOURCE -D_POSIX_C_SOURCE=200809L filename.c
```

If our code is written, or it will run on UNIX certified systems (hence on systems who follow SUSv1, SUSv2, SUSv3, or SUSv4), we must define also `_XOPEN_SOURCE` :

Thus, we would have to use

- for SUSv1:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 2
#define _XOPEN_SOURCE
#define _XOPEN_SOURCE_EXTENDED 1
```

- for SUSv2:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199506L
#define _XOPEN_SOURCE 500
```

- for SUSv3:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 200112L
#define _XOPEN_SOURCE 600
```



- for SUSv4:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 200809L
#define _XOPEN_SOURCE 700
```

If we write code only for Linux platforms, we will use the feature test macro `_GNU_SOURCE`, which will activate GNU LIBC functionality, which sometimes isn't POSIX compatible. There is also the feature test macro `_SVID_SOURCE` (to activate System V functionality) and `_BSD_SOURCE` (to activate BSD functionality). One important note is that a UNIX system (which follows SUSvX) can be activated to offer any SUSvX functionality.

This is the way we can write Unix portable code. Other methods to find out more about the operating system on which we compile are:

- LIBC functions: `sysconf(3)`, `pathconf(3)`, `fpathconf(3)` - functions which determine system constants
- `autoconf`, `automake` and `libtool`: utilities which determine at compile time, with scripts, what system and libc functions the operating system offers. (These will be part of the content of a following article.)

Happy Unix programming!

[Tweet this](#) - [Like this](#) - [Share on G+](#)

[comments powered by Disqus](#)

Social



Tech

This blog is created by ROSEdu community using Hekyll. See source on GitHub.

License



This work is licensed under a [Creative Commons Attribution 3.0 License](#).





LSB Introduction

An operating system's success is inextricably linked with the number and quality of applications that run on top of it. Linux and its variances between distributions, however, present ISVs and individual developers with a unique set of challenges: different distributions of Linux make use of different versions of libraries, important files stored in different locations, and so on. If an ISV wants to reach a global Linux audience, they must support more than one distribution of Linux. These challenges and variances make it difficult—and costly—for ISVs to target the Linux platform.

It is somewhat of an irony: choice is what drives Linux adoption, yet this very choice can make things difficult for application developers. The costs and resources involved in targeting multiple Linux distributions for application development is not something that can be taken lightly.

The Linux Standard Base was created to solve these challenges and lower the overall costs of supporting the Linux platform. By reducing the differences between individual Linux distributions, the LSB greatly reduces the costs involved with porting applications to different distributions, as well as lowers the cost and effort involved in after-market support of those applications.

The Linux Foundation and the Linux Standard Base were created to enable ISVs to cost effectively target the Linux platform, reducing their porting, support, and testing costs while helping them address a global market for their applications.

Benefits of the LSB

The LSB solution not only makes life easier for individual application developers and independent software vendors (ISVs), it also makes a huge positive impact on the entire Linux ecosystem.

Beyond the reduction of porting costs, the existence of the LSB solution means that Linux will also avoid the fate of the UNIX operating systems, where commercial interests caused the fragmentation of a single UNIX OS into several UNIX variants that, unlike Linux distributions, are very incompatible with each other.

There are other solid benefits of the LSB:

- Less complexity for ISVs to support Linux, thus reducing support costs.
- Greater reach for ISVs in more geographic markets, where a specific distribution may be more dominant.
- The ability to support additional Linux distributions with only a small increase in support and development costs.
- Support from the Linux Foundation and the Linux Developer Network to make ISVs' development process easier and their marketing more effective.

Clearly, the LSB solution is a positive force in the Linux environment. But what, exactly, is it? Understanding the components of the LSB is key to understanding all the things it can do for developers.

Understanding the LSB

The LSB is a core standard for the Linux operating system that encourages interoperability between applications and the platform. It includes a written binary interface specification, a set of test suites for both distributions and applications writing to the standard, and a sample implementation for testing purposes.

Currently, the distribution vendors are the enablers of the LSB standard. Without their participation, the standard cannot achieve any success. And without their participation in the creation of the LSB, their support for it would be unlikely. Luckily all major distribution vendors are certified to the LSB.

In order for the LSB to be successful, the Linux Foundation has built an organization that makes it as easy as possible for distribution vendors to come to consensus while balancing the needs of other constituents to have a timely and useful standard.

Community support is key, too. The open source community represents an amalgamation of software projects that are integrated into a single computing solution. It is important that the maintainers of those projects are aware of existing computing standards such as the LSB so they can work in a cooperative fashion to accelerate the adoption of their technology. All of these cooperative elements—distribution vendors, community developers, and application developers, work together to make the LSB happen. But what are the technical elements of the LSB? What's under the hood?

The key component of the LSB is the written binary interface specification, which informs developers of Linux applications the standard ways to build and configure their applications. Specifically, the specification lists:

- Common Packaging and Install Guidelines
- Common Shared Libraries and their Selection
- Configuration Files
- File Placement
- System Commands
- Application Binary Interfaces (ABIs) for System Interfaces (both application and platform levels)

Built from a foundation of existing standards, LSB delineates the binary interface between an application and a runtime environment. Existing standards that the LSB draws from include the Single UNIX Specification (SUS), the Standard C ABI, and the System V ABI. Other standards referenced include PAM, X11, and the desktop standards hosted on freedesktop.org. LSB formalizes the framework for interface availability within individual libraries and itemizes the data structures and constants associated with each interface. These components include shared libraries required by developers (including C++), file system hierarchies (defining where files are located in the system), specifications for the behavior of public interfaces, application packaging details, application behavior pre- and post-installation, and so on. The LSB provides backward compatibility at both the source and binary level beginning with LSB 3.0. In other words, applications that target version X.Y of the LSB (where X.Y \geq 3.0) will run on distributions certified to or compliant with LSB version X.Y //and newer//. For example, applications that target LSB 3.0 will run not only on LSB 3.0 certified/compliant distributions but also LSB 3.1, LSB 3.2, LSB 4.0 etc. certified/compliant distributions. To achieve backward compatibility, each subsequent version is purely additive--in other words, interfaces are only added, not removed (our https://www.linuxfoundation.org/en/Deprecation_Policy programmers looking to program on Linux.

- **LSB Build Tools.** LSB SDK enables developers to validate the binaries and RPM packages to ensure LSB compliance and monitor the `API` usage by the application while the build is taking place so that conformance is assured.
- **LSB Sample Implementation.** The LSB Sample Implementation (LSB-si), is a minimal LSB-conforming runtime environment used for testing purposes. LSB compliant applications should be tested inside the LSB-si to insure they haven't picked up any distribution-specific quirks. The LSB Certification program requires an application be tested under the LSB-si.
- **Tutorials and Blogs.** Get the latest how-tos and information for application portability, LSB compliance, and general Linux application development.
- **Forums and Mailing Lists.** Get real-time solutions and tips to many development problems.
- **LSB Certification and General Marketing Support.** No matter which path you choose, LSB or portability, resources will be available for developers to determine how to maximize their application's exposure in the Linux ecosystem. Certified applications can be included in the product directory.

Summary

The LSB gives application developers tools and a standardized foundation on which to take advantage of the global Linux opportunity. The Linux Developer Network is designed to leverage the full potential of the Linux Standard Base to assist application vendors and developers to maximize application portability across LSB-certified distributions.

lsb:lsb-introduction.txt · Last modified: 2016/07/19 01:23 (external edit)





LOG IN



opensource.com

An introduction to Linux filesystems

By [David Both](#) (Correspondent)

October 31, 2016 | [11 Comments](#) | 18 min read

 805 readers like this.



Image by: Original photo by Rikki Endsley. CC BY-SA 4.0

This article is intended to be a very high-level discussion of Linux filesystem concepts. It is not intended to be a low-level description of how a particular filesystem type, such as EXT4, works, nor is it intended to be a tutorial of filesystem commands.

More Linux resources

- [Linux commands cheat sheet](#)

- [Advanced Linux commands cheat sheet](#)
- [Free online course: RHEL Technical Overview](#)
- [Linux networking cheat sheet](#)
- [SELinux cheat sheet](#)
- [Linux common commands cheat sheet](#)
- [What are Linux containers?](#)
- [Our latest Linux articles](#)

Every general-purpose computer needs to store data of various types on a hard disk drive (HDD) or some equivalent, such as a USB memory stick. There are a couple reasons for this. First, RAM loses its contents when the computer is switched off. There are non-volatile types of RAM that can maintain the data stored there after power is removed (such as flash RAM that is used in USB memory sticks and solid state drives), but flash RAM is much more expensive than standard, volatile RAM like DDR3 and other, similar types.

The second reason that data needs to be stored on hard drives is that even standard RAM is still more expensive than disk space. Both RAM and disk costs have been dropping rapidly, but RAM still leads the way in terms of cost per byte. A quick calculation of the cost per byte, based on costs for 16GB of RAM vs. a 2TB hard drive, shows that the RAM is about 71 times more expensive per unit than the hard drive. A typical cost for RAM is around \$0.0000000043743750 per byte today.

For a quick historical note to put present RAM costs in perspective, in the very early days of computing, one type of memory was based on dots on a CRT screen. This was very expensive at about \$1.00 *per bit!*

Definitions

You may hear people talk about filesystems in a number of different and confusing ways. The word itself can have multiple meanings, and you may have to discern the correct meaning from the context of a discussion or document.

I will attempt to define the various meanings of the word "filesystem" based on how I have observed it being used in different circumstances. Note that while attempting to conform to standard "official" meanings, my intent is to define the term based on its various usages. These meanings will be explored in greater detail in the following sections of this article.

1. The entire Linux directory structure starting at the top (/) root directory.

2. A specific type of data storage format, such as EXT3, EXT4, BTRFS, XFS, and so on. Linux supports almost 100 types of filesystems, including some very old ones as well as some of the newest. Each of these filesystem types uses its own metadata structures to define how the data is stored and accessed.
3. A partition or logical volume formatted with a specific type of filesystem that can be mounted on a specified mount point on a Linux filesystem.

Basic filesystem functions

Disk storage is a necessity that brings with it some interesting and inescapable details. Obviously, a filesystem is designed to provide space for non-volatile storage of data; that is its ultimate function. However, there are many other important functions that flow from that requirement.

All filesystems need to provide a namespace—that is, a naming and organizational methodology. This defines how a file can be named, specifically the length of a filename and the subset of characters that can be used for filenames out of the total set of characters available. It also defines the logical structure of the data on a disk, such as the use of directories for organizing files instead of just lumping them all together in a single, huge conglomeration of files.

Once the namespace has been defined, a metadata structure is necessary to provide the logical foundation for that namespace. This includes the data structures required to support a hierarchical directory structure; structures to determine which blocks of space on the disk are used and which are available; structures that allow for maintaining the names of the files and directories; information about the files such as their size and times they were created, modified or last accessed; and the location or locations of the data belonging to the file on the disk. Other metadata is used to store high-level information about the subdivisions of the disk, such as logical volumes and partitions. This higher-level metadata and the structures it represents contain the information describing the filesystem stored on the drive or partition, but is separate from and independent of the filesystem metadata.

Filesystems also require an Application Programming Interface (API) that provides access to system function calls which manipulate filesystem objects like files and directories. APIs provide for tasks such as creating, moving, and deleting files. It also provides algorithms that determine things like where a file is placed on a filesystem. Such algorithms may account for objectives such as speed or minimizing disk fragmentation.

Modern filesystems also provide a security model, which is a scheme for defining access rights to files and directories. The Linux filesystem security model helps to ensure that users only have access to their own files and not those of others or the operating system itself.

The final building block is the software required to implement all of these functions. Linux uses a two-part software implementation as a way to improve both system and programmer efficiency.

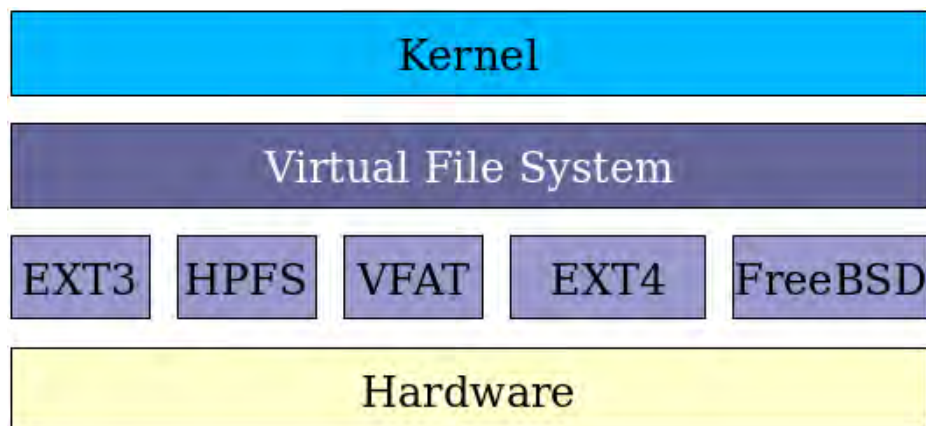


Figure 1: The Linux two-part filesystem software implementation.

The first part of this two-part implementation is the Linux virtual filesystem. This virtual filesystem provides a single set of commands for the kernel, and developers, to access all types of filesystems. The virtual filesystem software calls the specific device driver required to interface to the various types of filesystems. The filesystem-specific device drivers are the second part of the implementation. The device driver interprets the standard set of filesystem commands to ones specific to the type of filesystem on the partition or logical volume.

Directory structure

As a usually very organized Virgo, I like things stored in smaller, organized groups rather than in one big bucket. The use of directories helps me to be able to store and then locate the files I want when I am looking for them. Directories are also known as folders because they can be thought of as folders in which files are kept in a sort of physical desktop analogy.

In Linux and many other operating systems, directories can be structured in a tree-like hierarchy. The Linux directory structure is well defined and documented in the [Linux Filesystem Hierarchy Standard](#) (FHS). Referencing those directories when accessing them is accomplished by using the sequentially deeper directory names connected by forward slashes (/) such as `/var/log` and `/var/spool/mail`. These are called paths.

The following table provides a very brief list of the standard, well-known, and defined top-level Linux directories and their purposes.

| Directory | Description |
|----------------------------------|--|
| <code>/</code> (root filesystem) | The root filesystem is the top-level directory of the filesystem. It must contain all of the files required to boot the Linux system before other filesystems are mounted. It must include all of the required executables and |



| Directory | Description |
|-----------|---|
| | libraries required to boot the remaining filesystems. After the system is booted, all other filesystems are mounted on standard, well-defined mount points as subdirectories of the root filesystem. |
| /bin | The /bin directory contains user executable files. |
| /boot | Contains the static bootloader and kernel executable and configuration files required to boot a Linux computer. |
| /dev | This directory contains the device files for every hardware device attached to the system. These are not device drivers, rather they are files that represent each device on the computer and facilitate access to those devices. |
| /etc | Contains the local system configuration files for the host computer. |
| /home | Home directory storage for user files. Each user has a subdirectory in /home. |
| /lib | Contains shared library files that are required to boot the system. |
| /media | A place to mount external removable media devices such as USB thumb drives that may be connected to the host. |
| /mnt | A temporary mountpoint for regular filesystems (as in not removable media) that can be used while the administrator is repairing or working on a filesystem. |
| /opt | Optional files such as vendor supplied application programs should be located here. |
| /root | This is not the root (/) filesystem. It is the home directory for the root user. |
| /sbin | System binary files. These are executables used for system administration. |
| /tmp | Temporary directory. Used by the operating system and many programs to store temporary files. Users may also store files here temporarily. Note that files stored here may be deleted at any time without prior notice. |
| /usr | These are shareable, read-only files, including executable binaries and libraries, man files, and other types of documentation. |
| /var | Variable data files are stored here. This can include things like log files, MySQL, and other database files, web server data files, email inboxes, and |



| Directory | Description |
|-----------|-------------|
| | much more. |

Table 1: The top level of the Linux filesystem hierarchy.

The directories and their subdirectories shown in Table 1, along with their subdirectories, that have a teal background are considered an integral part of the root filesystem. That is, they cannot be created as a separate filesystem and mounted at startup time. This is because they (specifically, their contents) must be present at boot time in order for the system to boot properly.

The `/media` and `/mnt` directories are part of the root filesystem, but they should never contain any data. Rather, they are simply temporary mount points.

The remaining directories, those that have no background color in Table 1 do not need to be present during the boot sequence, but will be mounted later, during the startup sequence that prepares the host to perform useful work.

Be sure to refer to the official [Linux Filesystem Hierarchy Standard](#) (FHS) web page for details about each of these directories and their many subdirectories. Wikipedia also has a good description of the [FHS](#). This standard should be followed as closely as possible to ensure operational and functional consistency. Regardless of the filesystem types used on a host, this hierarchical directory structure is the same.

Linux unified directory structure

In some non-Linux PC operating systems, if there are multiple physical hard drives or multiple partitions, each disk or partition is assigned a drive letter. It is necessary to know on which hard drive a file or program is located, such as C: or D:. Then you issue the drive letter as a command, **D:**, for example, to change to the D: drive, and then you use the **cd** command to change to the correct directory to locate the desired file. Each hard drive has its own separate and complete directory tree.

The Linux filesystem unifies all physical hard drives and partitions into a single directory structure. It all starts at the top—the root (`/`) directory. All other directories and their subdirectories are located under the single Linux root directory. This means that there is only one single directory tree in which to search for files and programs.

This can work only because a filesystem, such as `/home`, `/tmp`, `/var`, `/opt`, or `/usr` can be created on separate physical hard drives, a different partition, or a different logical volume from the `/` (root) filesystem and then be mounted on a mountpoint (directory) as part of the root filesystem tree. Even removable drives such as a USB thumb drive or an external USB or ESATA hard drive will be mounted onto the root filesystem and become an integral part of that directory tree.



One good reason to do this is apparent during an upgrade from one version of a Linux distribution to another, or changing from one distribution to another. In general, and aside from any upgrade utilities like `dnf-upgrade` in Fedora, it is wise to occasionally reformat the hard drive(s) containing the operating system during an upgrade to positively remove any cruft that has accumulated over time. If `/home` is part of the root filesystem it will be reformatted as well and would then have to be restored from a backup. By having `/home` as a separate filesystem, it will be known to the installation program as a separate filesystem and formatting of it can be skipped. This can also apply to `/var` where database, email inboxes, website, and other variable user and system data are stored.

There are other reasons for maintaining certain parts of the Linux directory tree as separate filesystems. For example, a long time ago, when I was not yet aware of the potential issues surrounding having all of the required Linux directories as part of the `/` (root) filesystem, I managed to fill up my home directory with a large number of very big files. Since neither the `/home` directory nor the `/tmp` directory were separate filesystems but simply subdirectories of the root filesystem, the entire root filesystem filled up. There was no room left for the operating system to create temporary files or to expand existing data files. At first, the application programs started complaining that there was no room to save files, and then the OS itself started to act very strangely. Booting to single-user mode and clearing out the offending files in my home directory allowed me to get going again. I then reinstalled Linux using a pretty standard multi-filesystem setup and was able to prevent complete system crashes from occurring again.

I once had a situation where a Linux host continued to run, but prevented the user from logging in using the GUI desktop. I was able to log in using the command line interface (CLI) locally using one of the [virtual consoles](#), and remotely using SSH. The problem was that the `/tmp` filesystem had filled up and some temporary files required by the GUI desktop could not be created at login time. Because the CLI login did not require files to be created in `/tmp`, the lack of space there did not prevent me from logging in using the CLI. In this case, the `/tmp` directory was a separate filesystem and there was plenty of space available in the volume group the `/tmp` logical volume was a part of. I simply [expanded the `/tmp` logical volume](#) to a size that accommodated my fresh understanding of the amount of temporary file space needed on that host and the problem was solved. Note that this solution did not require a reboot, and as soon as the `/tmp` filesystem was enlarged the user was able to login to the desktop.

Another situation occurred while I was working as a lab administrator at one large technology company. One of our developers had installed an application in the wrong location (`/var`). The application was crashing because the `/var` filesystem was full and the log files, which are stored in `/var/log` on that filesystem, could not be appended with new messages due to the lack of space. However, the system remained up and running because the critical `/` (root) and `/tmp` filesystems did not fill up. Removing the offending application and reinstalling it in the `/opt` filesystem resolved that problem.

Filesystem types

Linux supports reading around 100 partition types; it can create and write to only a few of these. But it is possible—and very common—to mount filesystems of different types on the same root filesystem. In this context we are talking about filesystems in terms of the structures and metadata required to store and manage the user data on a partition of a hard drive or a logical volume. The complete list of filesystem partition types recognized by the Linux **fdisk** command is provided here, so that you can get a feel for the high degree of compatibility that Linux has with very many types of systems.

```

0 Empty 24 NEC DOS 81 Minix / old Lin bf Solaris 1 FAT12 27 Hidden NTFS Win 82 Linux swap / So c1
DRDOS/sec (FAT- 2 XENIX root 39 Plan 9 83 Linux c4 DRDOS/sec (FAT- 3 XENIX usr 3c PartitionMagic 84
OS/2 hidden or c6 DRDOS/sec (FAT- 4 FAT16 <32M 40 Venix 80286 85 Linux extended c7 Syrix 5 Extended
41 PPC PReP Boot 86 NTFS volume set da Non-FS data 6 FAT16 42 SFS 87 NTFS volume set db CP/M / CTOS /
. 7 HPFS/NTFS/exFAT 4d QNX4.x 88 Linux plaintext de Dell Utility 8 AIX 4e QNX4.x 2nd part 8e Linux
LVM df BootIt 9 AIX bootable 4f QNX4.x 3rd part 93 Amoeba e1 DOS access a OS/2 Boot Manag 50 OnTrack
DM 94 Amoeba BBT e3 DOS R/O b W95 FAT32 51 OnTrack DM6 Aux 9f BSD/OS e4 SpeedStor c W95 FAT32 (LBA)
52 CP/M a0 IBM Thinkpad hi ea Rufus alignment e W95 FAT16 (LBA) 53 OnTrack DM6 Aux a5 FreeBSD eb BeOS
fs f W95 Ext'd (LBA) 54 OnTrackDM6 a6 OpenBSD ee GPT 10 OPUS 55 EZ-Drive a7 NeXTSTEP ef EFI (FAT-
12/16/ 11 Hidden FAT12 56 Golden Bow a8 Darwin UFS f0 Linux/PA-RISC b 12 Compaq diagnost 5c Priam
Edisk a9 NetBSD f1 SpeedStor 14 Hidden FAT16 <3 61 SpeedStor ab Darwin boot f4 SpeedStor 16 Hidden
FAT16 63 GNU HURD or Sys af HFS / HFS+ f2 DOS secondary 17 Hidden HPFS/NTF 64 Novell Netware b7 BSDI
fs fb VMware VMFS 18 AST SmartSleep 65 Novell Netware b8 BSDI swap fc VMware VMKCORE 1b Hidden W95
FAT3 70 DiskSecure Mult bb Boot Wizard hid fd Linux raid auto 1c Hidden W95 FAT3 75 PC/IX bc Acronis
FAT32 L fe LANstep 1e Hidden W95 FAT1 80 Old Minix be Solaris boot ff BBT

```

The main purpose in supporting the ability to read so many partition types is to allow for compatibility and at least some interoperability with other computer systems' filesystems. The choices available when creating a new filesystem with Fedora are shown in the following list.

- btrfs
- **cramfs**
- **ext2**
- **ext3**
- **ext4**
- fat
- gfs2
- hfsplus



- minix
- **msdos**
- ntfs
- reiserfs
- **vfat**
- xfs

Other distributions support creating different filesystem types. For example, CentOS 6 supports creating only those filesystems highlighted in bold in the above list.

Mounting

The term "to mount" a filesystem in Linux refers back to the early days of computing when a tape or removable disk pack would need to be physically mounted on an appropriate drive device. After being physically placed on the drive, the filesystem on the disk pack would be logically mounted by the operating system to make the contents available for access by the OS, application programs and users.

A mount point is simply a directory, like any other, that is created as part of the root filesystem. So, for example, the home filesystem is mounted on the directory /home. Filesystems can be mounted at mount points on other non-root filesystems but this is less common.

The Linux root filesystem is mounted on the root directory (/) very early in the boot sequence. Other filesystems are mounted later, by the Linux startup programs, either **rc** under SystemV or by **systemd** in newer Linux releases. Mounting of filesystems during the startup process is managed by the /etc/fstab configuration file. An easy way to remember that is that fstab stands for "file system table," and it is a list of filesystems that are to be mounted, their designated mount points, and any options that might be needed for specific filesystems.

Filesystems are mounted on an existing directory/mount point using the **mount** command. In general, any directory that is used as a mount point should be empty and not have any other files contained in it. Linux will not prevent users from mounting one filesystem over one that is already there or on a directory that contains files. If you mount a filesystem on an existing directory or filesystem, the original contents will be hidden and only the content of the newly mounted filesystem will be visible.

Conclusion

I hope that some of the possible confusion surrounding the term filesystem has been cleared up by this article. It took a long time and a very helpful mentor for me to truly understand and appreciate

the complexity, elegance, and functionality of the Linux filesystem in all of its meanings.

If you have questions, please add them to the comments below and I will try to answer them.

Next month

Another important concept is that for Linux, everything is a file. This concept has some interesting and important practical applications for users and system admins. The reason I mention this is that you might want to read my "[Everything is a file](#)" article before the article I am planning for next month on the /dev directory.

Tags:

LINUX

SYSADMIN



David Both

David Both is an Open Source Software and GNU/Linux advocate, trainer, writer, and speaker. He has been working with Linux and Open Source Software since 1996 and with computers since 1969. He is a strong proponent of and evangelist for the "Linux Philosophy for System Administrators."

[More about me](#)

11 Comments

These comments are closed.



[Shawn H Corey](#) | October 31, 2016

 No readers like this yet.

To see what is mounted on your system, type:

```
mount | column -t
```



See ``man mount`` and ``man column`` for details.



[David Both](#) | October 31, 2016

No readers like this yet.

I have recently found the `lsblk` command and find that `lsblk -f` gives me a nice clean overview of the filesystems mounted on my hosts.

Thanks for your comment.



[FeRDNYC](#) | November 1, 2016

No readers like this yet.

`lsblk` is great for getting an overview of the physical devices mounted on the system, but it will miss any virtual filesystem mounts – most commonly, in-memory filesystems mounted as type `'tmpfs'`, of which modern Linux systems typically contain several.

`/tmp` and `/dev/shm` will be kept in memory for performance reasons on most systems (which is why `/var/tmp` should always be used instead of `/tmp` for large temporary files), and `systemd` installations will have `/run` and one or more `/run/user/$UID` mounted in memory as well.

`df` or `df -h` will provide a nicely-formatted listing of all mounts including virtual filesystems, or on distributions which provide it, I've come to prefer the output of `di`.



Evgeniy Yanyuk | November 2, 2016

No readers like this yet.

There are many possibilities to do it:

```
cat /etc/mstab or column -t /etc/mstab
```

```
cat /proc/mounts
```

```
findmnt
```

```
sudo disk -l
```

and I think there are more others...



[Bryan Behrenshausen](#) | October 31, 2016

No readers like this yet.

Invaluable. Thanks, David.



[Don Watkins](#) | October 31, 2016

No readers like this yet.

What an amazing article David!



[Ram Sambamurthy](#) | November 8, 2016

No readers like this yet.

Truly amazing, so clearly explained



Granato | October 31, 2016

No readers like this yet.

Very useful, thanks a lot.



[Edmund](#) | November 1, 2016

No readers like this yet.

Actually FHS has already 3.0 version released in June 3, 2015 and the main site is available at <http://refspecs.linuxfoundation.org/fhs.shtml>

This <http://www.pathname.com/fhs/> has an outdated 2.3 version.

Development sites can be found at:

* <https://wiki.linuxfoundation.org/en/FHS>

* <https://wiki.linuxfoundation.org/lstb/fhs>



[David Both](#) | November 1, 2016

No readers like this yet.

I did not know this. Thanks for pointing it out.





Mahesh Sreekandath | November 18, 2016

 No readers like this yet.

Hi David,

Great overview!

A while back I had worked on porting and benchmarking an embedded file system to Linux, back then the learning process was very bottom-up. Most of my initial time was spent exploring the grisly details of VFS data structures and kernel helper functions. It took a while to grasp how coherently all the subsystems from a posix call interface to the block driver adapted together.

In a way it helped that the initial study and benchmarks were done against relatively simple JFFS2, then UBIFS and finally on EXT4 with block driver. This was definitely in an increasing order of complexity. Exploring the Linux file system framework is definitely one of the most demanding learning experiences.

Also, had also written few related posts on my website illustrating these findings (<https://msreekan.com/2015/04/24/linux-storage-cache/>). It's sort of from an embedded systems engineering point of view. Now my work has moved on from Linux, but running into this article reminded me of the whole experience.

Related Content



What's new in GNOME 44?



5 reasons virtual machines still matter



Remove the background from an image with this Linux command





This work is licensed under a Creative Commons Attribution-Share Alike 4.0 International License.

ABOUT THIS SITE

The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.

Opensource.com aspires to publish all content under a **Creative Commons license** but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.

A note on advertising: Opensource.com does not sell advertising on the site or in any of its newsletters.

opensource.com

Copyright ©2024 Red Hat, Inc.

[Privacy Policy](#)

[Terms of use](#)

[Cookie preferences](#)





LOG IN



opensource.com

An introduction to Linux's EXT4 filesystem

Take a walk through EXT4's history, features, and optimal use, and learn how it differs from previous iterations of the EXT filesystem.

By [David Both](#) (Correspondent)

May 25, 2017 | [9 Comments](#) | 22 min read

 770 readers like this.



Image by: William Warby. Modified by Jason Baker. Creative Commons BY-SA 2.0.

In previous articles about Linux filesystems, I wrote [an introduction to Linux filesystems](#) and about some higher-level concepts such as [everything is a file](#). I want to go into more detail about the specifics of the EXT filesystems, but first, let's answer the question, "What is a filesystem?" A filesystem is all of the following:



1. **Data storage:** The primary function of any filesystem is to be a structured place to store and retrieve data.
2. **Namespace:** A naming and organizational methodology that provides rules for naming and structuring data.
3. **Security model:** A scheme for defining access rights.
4. **API:** System function calls to manipulate filesystem objects like directories and files.
5. **Implementation:** The software to implement the above.

This article concentrates on the first item in the list and explores the metadata structures that provide the logical framework for data storage in an EXT filesystem.

EXT filesystem history

Although written for Linux, the EXT filesystem has its roots in the Minix operating system and the Minix filesystem, which predate Linux by about five years, being first released in 1987. Understanding the EXT4 filesystem is much easier if we look at the history and technical evolution of the EXT filesystem family from its Minix roots.

More Linux resources

- 🔗 [Linux commands cheat sheet](#)
- 🔗 [Advanced Linux commands cheat sheet](#)
- 🔗 [Free online course: RHEL Technical Overview](#)
- 🔗 [Linux networking cheat sheet](#)
- 🔗 [SELinux cheat sheet](#)
- 🔗 [Linux common commands cheat sheet](#)
- 🔗 [What are Linux containers?](#)
- 🔗 [Our latest Linux articles](#)

Minix

When writing the original Linux kernel, Linus Torvalds needed a filesystem but didn't want to write one then. So he simply included the [Minix filesystem](#), which had been written by [Andrew S. Tanenbaum](#) and was a part of Tanenbaum's Minix operating system. [Minix](#) was a Unix-like operating system written for educational purposes. Its code was freely available and appropriately licensed to allow Torvalds to include it in his first version of Linux.

Minix has the following structures, most of which are located in the partition where the filesystem is generated:

- A **boot sector** in the first sector of the hard drive on which it is installed. The boot block includes a very small boot record and a partition table.
- The first block in each partition is a **superblock** that contains the metadata that defines the other filesystem structures and locates them on the physical disk assigned to the partition.
- An **inode bitmap block**, which determines which inodes are used and which are free.
- The **inodes**, which have their own space on the disk. Each inode contains information about one file, including the locations of the data blocks, i.e., zones belonging to the file.
- A **zone bitmap** to keep track of the used and free data zones.
- A **data zone**, in which the data is actually stored.

For both types of bitmaps, one bit represents one specific data zone or one specific inode. If the bit is zero, the zone or inode is free and available for use, but if the bit is one, the data zone or inode is in use.

What is an [inode](#)? Short for index-node, an inode is a 256-byte block on the disk and stores data about the file. This includes the file's size; the user IDs of the file's user and group owners; the file mode (i.e., the access permissions); and three timestamps specifying the time and date that: the file was last accessed, last modified, and the data in the inode was last modified.

The inode also contains data that points to the location of the file's data on the hard drive. In Minix and the EXT1-3 filesystems, this is a list of data zones or blocks. The Minix filesystem inodes supported nine data blocks, seven direct and two indirect. If you'd like to learn more, there is an excellent PDF with a detailed description of the [Minix filesystem structure](#) and a quick overview of the [inode pointer structure](#) on Wikipedia.

EXT

The original [EXT filesystem](#) (Extended) was written by [Rémy Card](#) and released with Linux in 1992 to overcome some size limitations of the Minix filesystem. The primary structural changes were to the metadata of the filesystem, which was based on the Unix filesystem (UFS), which is also known as the Berkeley Fast File System (FFS). I found very little published information about the EXT filesystem that can be verified, apparently because it had significant problems and was quickly superseded by the EXT2 filesystem.

EXT2

The [EXT2 filesystem](#) was quite successful. It was used in Linux distributions for many years, and it was the first filesystem I encountered when I started using Red Hat Linux 5.0 back in about 1997. The EXT2 filesystem has essentially the same metadata structures as the EXT filesystem, however EXT2 is more forward-looking, in that a lot of disk space is left between the metadata structures for future use.

Like Minix, EXT2 has a [boot sector](#) in the first sector of the hard drive on which it is installed, which includes a very small boot record and a partition table. Then there is some reserved space after the boot sector, which spans the space between the boot record and the first partition on the hard drive that is usually on the next cylinder boundary. [GRUB2](#)—and possibly GRUB1—uses this space for part of its boot code.

The space in each EXT2 partition is divided into cylinder groups that allow for more granular management of the data space. In my experience, the group size usually amounts to about 8MB. Figure 1, below, shows the basic structure of a cylinder group. The data allocation unit in a cylinder is the block, which is usually 4K in size.



Figure 1: The structure of a cylinder group in the EXT filesystems

The first block in the cylinder group is a superblock, which contains the metadata that defines the other filesystem structures and locates them on the physical disk. Some of the additional groups in the partition will have backup superblocks, but not all. A damaged superblock can be replaced by using a disk utility such as **dd** to copy the contents of a backup superblock to the primary superblock. It does not happen often, but once, many years ago, I had a damaged superblock, and I was able to restore its contents using one of the backup superblocks. Fortunately, I had been foresighted and used the **dumpe2fs** command to dump the descriptor information of the partitions on my system.

Following is the partial output from the **dumpe2fs** command. It shows the metadata contained in the superblock, as well as data about each of the first two cylinder groups in the filesystem.



```
# dumpe2fs /dev/sda1
Filesystem volume name: boot
Last mounted on: /boot
Filesystem UUID: 79fc5ed8-5bbc-4dfe-8359-b7b36be6eed3
Filesystem magic number: 0xEF53
Filesystem revision #: 1 (dynamic)
Filesystem features: has_journal ext_attr resize_inode dir_index fi
Filesystem flags: signed_directory_hash
Default mount options: user_xattr acl
Filesystem state: clean
Errors behavior: Continue
Filesystem OS type: Linux
Inode count: 122160
Block count: 488192
Reserved block count: 24409
Free blocks: 376512
Free inodes: 121690
First block: 0
Block size: 4096
Fragment size: 4096
Group descriptor size: 64
Reserved GDT blocks: 238
Blocks per group: 32768
Fragments per group: 32768
Inodes per group: 8144
Inode blocks per group: 509
Flex block group size: 16
Filesystem created: Tue Feb 7 09:33:34 2017
Last mount time: Sat Apr 29 21:42:01 2017
Last write time: Sat Apr 29 21:42:01 2017
Mount count: 25
Maximum mount count: -1
Last checked: Tue Feb 7 09:33:34 2017
Check interval: 0 (<none>)
Lifetime writes: 594 MB
Reserved blocks uid: 0 (user root)
Reserved blocks gid: 0 (group root)
First inode: 11
Inode size: 256
Required extra isize: 32
Desired extra isize: 32
Journal inode: 8
Default directory hash: half_md4
Directory Hash Seed: c780bac9-d4bf-4f35-b695-0fe35e8d2d60
Journal backup: inode blocks
Journal features: journal_64bit
Journal size: 32M
Journal length: 8192
Journal sequence: 0x00000213
Journal start: 0
```



```
Group 0: (Blocks 0-32767)
  Primary superblock at 0, Group descriptors at 1-1
  Reserved GDT blocks at 2-239
  Block bitmap at 240 (+240)
  Inode bitmap at 255 (+255)
  Inode table at 270-778 (+270)
  24839 free blocks, 7676 free inodes, 16 directories
  Free blocks: 7929-32767
  Free inodes: 440, 470-8144
Group 1: (Blocks 32768-65535)
  Backup superblock at 32768, Group descriptors at 32769-32769
  Reserved GDT blocks at 32770-33007
  Block bitmap at 241 (bg #0 + 241)
  Inode bitmap at 256 (bg #0 + 256)
  Inode table at 779-1287 (bg #0 + 779)
  8668 free blocks, 8142 free inodes, 2 directories
  Free blocks: 33008-33283, 33332-33791, 33974-33975, 34023-34092, 34094-34095
  Free inodes: 8147-16288
Group 2: (Blocks 65536-98303)
  Block bitmap at 242 (bg #0 + 242)
  Inode bitmap at 257 (bg #0 + 257)
  Inode table at 1288-1796 (bg #0 + 1288)
  6326 free blocks, 8144 free inodes, 0 directories
  Free blocks: 67042-67583, 72201-72994, 80185-80349, 81191-81919, 90112-90112
  Free inodes: 16289-24432
Group 3: (Blocks 98304-131071)

<snip>
```

Each cylinder group has its own inode bitmap that is used to determine which inodes are used and which are free within that group. The inodes have their own space in each group. Each inode contains information about one file, including the locations of the data blocks belonging to the file. The block bitmap keeps track of the used and free data blocks within the filesystem. Notice that there is a great deal of data about the filesystem in the output shown above. On very large filesystems the group data can run to hundreds of pages in length. The group metadata includes a listing of all of the free data blocks in the group.

The EXT filesystem implemented data-allocation strategies that ensured minimal file fragmentation. Reducing fragmentation improved filesystem performance. Those strategies are described below, in the section on EXT4.

The biggest problem with the EXT2 filesystem, which I encountered on some occasions, was that it could take many hours to recover after a crash because the **fsck** (file system check) program took a very long time to locate and correct any inconsistencies in the filesystem. It once took over 28

hours on one of my computers to fully recover a disk upon reboot after a crash—and that was when disks were measured in the low hundreds of megabytes in size.

EXT3

The [EXT3 filesystem](#) had the singular objective of overcoming the massive amounts of time that the **fsck** program required to fully recover a disk structure damaged by an improper shutdown that occurred during a file-update operation. The only addition to the EXT filesystem was the [journal](#), which records in advance the changes that will be performed to the filesystem. The rest of the disk structure is the same as it was in EXT2.

Instead of writing data to the disk's data areas directly, as in previous versions, the journal in EXT3 writes file data, along with its metadata, to a specified area on the disk. Once the data is safely on the hard drive, it can be merged in or appended to the target file with almost zero chance of losing data. As this data is committed to the data area of the disk, the journal is updated so that the filesystem will remain in a consistent state in the event of a system failure before all the data in the journal is committed. On the next boot, the filesystem will be checked for inconsistencies, and data remaining in the journal will then be committed to the data areas of the disk to complete the updates to the target file.

Journaling does reduce data-write performance, however there are three options available for the journal that allow the user to choose between performance and data integrity and safety. My personal preference is on the side of safety because my environments do not require heavy disk-write activity.

The journaling function reduces the time required to check the hard drive for inconsistencies after a failure from hours (or even days) to mere minutes, at the most. I have had many issues over the years that have crashed my systems. The details could fill another article, but suffice it to say that most were self-inflicted, like kicking out a power plug. Fortunately, the EXT journaling filesystems have reduced that bootup recovery time to two or three minutes. In addition, I have never had a problem with lost data since I started using EXT3 with journaling.

The journaling feature of EXT3 can be turned off and it then functions as an EXT2 filesystem. The journal itself still exists, empty and unused. Simply remount the partition with the mount command using the type parameter to specify EXT2. You may be able to do this from the command line, depending upon which filesystem you are working with, but you can change the type specifier in the **/etc/fstab** file and then reboot. I strongly recommend against mounting an EXT3 filesystem as EXT2 because of the additional potential for lost data and extended recovery times.

An existing EXT2 filesystem can be upgraded to EXT3 with the addition of a journal using the following command.

```
tune2fs -j /dev/sda1
```

Where **/dev/sda1** is the drive and partition identifier. Be sure to change the file type specifier in **/etc/fstab** and remount the partition or reboot the system to have the change take effect.

EXT4

The [EXT4 filesystem](#) primarily improves performance, reliability, and capacity. To improve reliability, metadata and journal checksums were added. To meet various mission-critical requirements, the filesystem timestamps were improved with the addition of intervals down to nanoseconds. The addition of two high-order bits in the timestamp field defers the [Year 2038 problem](#) until 2446—for EXT4 filesystems, at least.

In EXT4, data allocation was changed from fixed blocks to extents. An extent is described by its starting and ending place on the hard drive. This makes it possible to describe very long, physically contiguous files in a single inode pointer entry, which can significantly reduce the number of pointers required to describe the location of all the data in larger files. Other allocation strategies have been implemented in EXT4 to further reduce fragmentation.

EXT4 reduces fragmentation by scattering newly created files across the disk so that they are not bunched up in one location at the beginning of the disk, as many early PC filesystems did. The file-allocation algorithms attempt to spread the files as evenly as possible among the cylinder groups and, when fragmentation is necessary, to keep the discontinuous file extents as close as possible to others in the same file to minimize head seek and rotational latency as much as possible. Additional strategies are used to pre-allocate extra disk space when a new file is created or when an existing file is extended. This helps to ensure that extending the file will not automatically result in its becoming fragmented. New files are never allocated immediately after existing files, which also prevents fragmentation of the existing files.

Aside from the actual location of the data on the disk, EXT4 uses functional strategies, such as delayed allocation, to allow the filesystem to collect all the data being written to the disk before allocating space to it. This can improve the likelihood that the data space will be contiguous.

Older EXT filesystems, such as EXT2 and EXT3, can be mounted as EXT4 to make some minor performance gains. Unfortunately, this requires turning off some of the important new features of EXT4, so I recommend against this.

EXT4 has been the default filesystem for Fedora since Fedora 14. An EXT3 filesystem can be upgraded to EXT4 using the [procedure](#) described in the Fedora documentation, however its performance will still suffer due to residual EXT3 metadata structures. The best method for



upgrading to EXT4 from EXT3 is to back up all the data on the target filesystem partition, use the **mkfs** command to write an empty EXT4 filesystem to the partition, and then restore all the data from the backup.

Inode

The inode, described previously, is a key component of the metadata in EXT filesystems. Figure 2 shows the relationship between the inode and the data stored on the hard drive. This diagram is the directory and inode for a single file which, in this case, may be highly fragmented. The EXT filesystems work actively to reduce fragmentation, so it is very unlikely you will ever see a file with this many indirect data blocks or extents. In fact, as you will see below, fragmentation is extremely low in EXT filesystems, so most inodes will use only one or two direct data pointers and none of the indirect pointers.

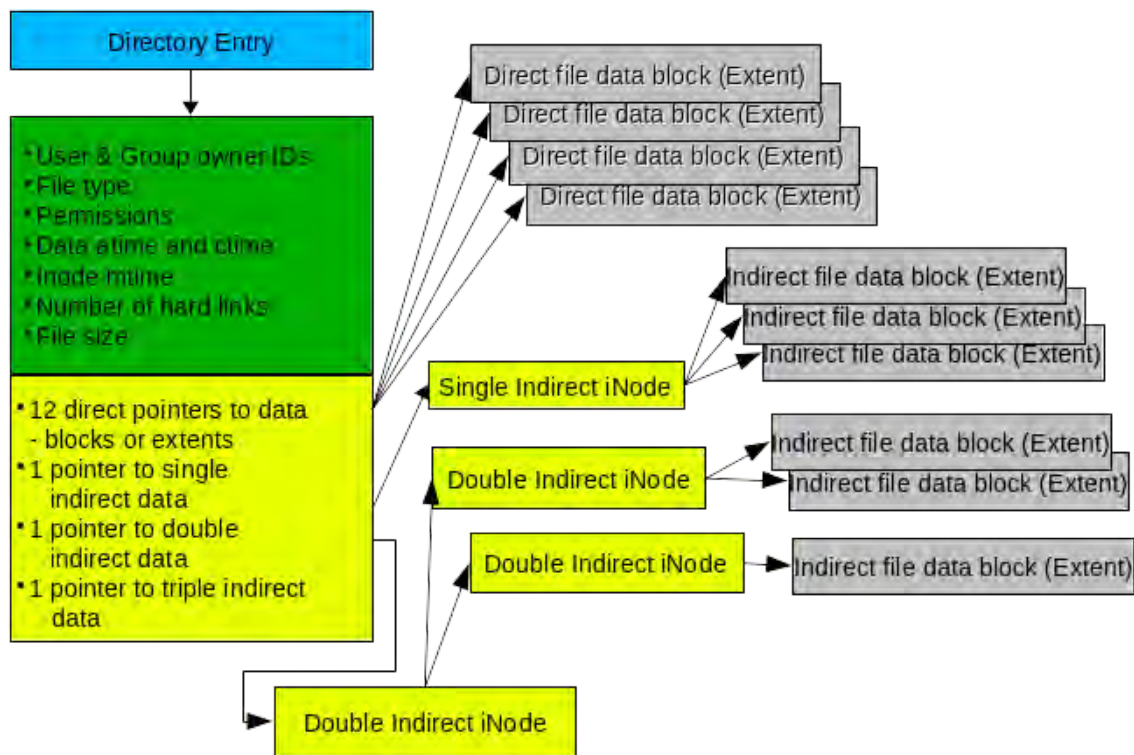


Figure 2: The inode stores information about each file and enables the EXT filesystem to locate all data belonging to it.

The inode does not contain the name of the file. Access to a file is via the directory entry, which itself is the name of the file and contains a pointer to the inode. The value of that pointer is the inode number. Each inode in a filesystem has a unique ID number, but inodes in other filesystems on the same computer (and even the same hard drive) can have the same inode number. This has implications for [links](#), and this discussion is beyond the scope of this article.

The inode contains the metadata about the file, including its type and permissions as well as its size. The inode also contains space for 15 pointers that describe the location and length of data

blocks or extents in the data portion of the cylinder group. Twelve of the pointers provide direct access to the data extents and should be sufficient to handle most files. However, for files that have significant fragmentation, it becomes necessary to have some additional capabilities in the form of indirect nodes. Technically these are not really inodes, so I use the term "node" here for convenience.

An indirect node is a normal data block in the filesystem that is used only for describing data and not for storage of metadata, thus more than 15 entries can be supported. For example, a block size of 4K can support 512 4-byte indirect nodes, allowing **12 (direct) + 512 (indirect) = 524** extents for a single file. Double and triple indirect node support is also supported, but most of us are unlikely to encounter files requiring that many extents.

Data fragmentation

For many older PC filesystems, such as FAT (and all its variants) and NTFS, fragmentation has been a significant problem resulting in degraded disk performance. Defragmentation became an industry in itself with different brands of defragmentation software that ranged from very effective to only marginally so.

Linux's extended filesystems use data-allocation strategies that help to minimize fragmentation of files on the hard drive and reduce the effects of fragmentation when it does occur. You can use the **fsck** command on EXT filesystems to check the total filesystem fragmentation. The following example checks the home directory of my main workstation, which was only 1.5% fragmented. Be sure to use the **-n** parameter, because it prevents **fsck** from taking any action on the scanned filesystem.

```
fsck -fn /dev/mapper/vg_01-home
```

I once performed some theoretical calculations to determine whether disk defragmentation might result in any noticeable performance improvement. While I did make some assumptions, the disk performance data I used were from a new 300GB, Western Digital hard drive with a 2.0ms track-to-track seek time. The number of files in this example was the actual number that existed in the filesystem on the day I did the calculation. I did assume that a fairly large amount of the fragmented files (20%) would be touched each day.

| | |
|--------------------|----------------|
| Total files | 271,794 |
| % fragmentation | 5.00% |
| Discontinuities | 13,590 |

| | |
|------------------------------------|--------------|
| | |
| % fragmented files touched per day | 20% (assume) |
| Number of additional seeks | 2,718 |
| Average seek time | 10.90 ms |
| Total additional seek time per day | 29.63 sec |
| | 0.49 min |
| | |
| Track-to-track seek time | 2.00 ms |
| Total additional seek time per day | 5.44 sec |
| | 0.091 min |

Table 1: The theoretical effects of fragmentation on disk performance

I have done two calculations for the total additional seek time per day, one based on the track-to-track seek time, which is the more likely scenario for most files due to the EXT file allocation strategies, and one for the average seek time, which I assumed would make a fair worst-case scenario.

As you can see from Table 1, the impact of fragmentation on a modern EXT filesystem with a hard drive of even modest performance would be minimal and negligible for the vast majority of applications. You can plug the numbers from your environment into your own similar spreadsheet to see what you might expect in the way of performance impact. This type of calculation most likely will not represent actual performance, but it can provide a bit of insight into fragmentation and its theoretical impact on a system.

Most of my partitions are around 1.5% or 1.6% fragmented; I do have one that is 3.3% fragmented but that is a large, 128GB filesystem with fewer than 100 very large ISO image files; I've had to expand the partition several times over the years as it got too full.

That is not to say that some application environments don't require greater assurance of even less fragmentation. The EXT filesystem can be tuned with care by a knowledgeable admin who can adjust the parameters to compensate for specific workload types. This can be done when the filesystem is created or later using the **tune2fs** command. The results of each tuning change should be tested, meticulously recorded, and analyzed to ensure optimum performance for the target environment. In the worst case, where performance cannot be improved to desired levels,



other filesystem types are available that may be more suitable for a particular workload. And remember that it is common to mix filesystem types on a single host system to match the load placed on each filesystem.

Due to the low amount of fragmentation on most EXT filesystems, it is not necessary to defragment. In any event, there is no safe defragmentation tool for EXT filesystems. There are a few tools that allow you to check the fragmentation of an individual file or the fragmentation of the remaining free space in a filesystem. There is one tool, **e4defrag**, which will defragment a file, directory, or filesystem as much as the remaining free space will allow. As its name implies, it only works on files in an EXT4 filesystem, and it does have some limitations.

If it becomes necessary to perform a complete defragmentation on an EXT filesystem, there is only one method that will work reliably. You must move all the files from the filesystem to be defragmented, ensuring that they are deleted after being safely copied to another location. If possible, you could then increase the size of the filesystem to help reduce future fragmentation. Then copy the files back onto the target filesystem. Even this does not guarantee that all the files will be completely defragmented.

Conclusions

The EXT filesystems have been the default for many Linux distributions for more than 20 years. They offer stability, high capacity, reliability, and performance while requiring minimal maintenance. I have tried other filesystems but always return to EXT. Every place I have worked with Linux has used the EXT filesystems and found them suitable for all the mainstream loads used on them. Without a doubt, the EXT4 filesystem should be used for most Linux systems unless there is a compelling reason to use another filesystem.

Tags:

[LINUX](#)[YEARBOOK](#)[2017 OPEN SOURCE YEARBOOK](#)

David Both

David Both is an Open Source Software and GNU/Linux advocate, trainer, writer, and speaker. He has been working with Linux and Open Source Software since 1996 and with computers since 1969. He is a strong proponent of and evangelist for the "Linux Philosophy for System Administrators."



[More about me](#)

9 Comments

These comments are closed.



[Seth Kenlon](#) | May 25, 2017

 No readers like this yet.

Amazing information. This really helps de-mystify the concept of the file system.

Given that EXT is a free and open source file system, I really do wish the major closed source OSES would integrate it into their systems as an option. I'd much rather use EXT4 over, say, HFS+ or FAT*.



[Don Watkins](#) | May 25, 2017

 No readers like this yet.

This is amazing. I never knew that EXT4 spread the information across the disk thereby reducing the likelihood of data loss. I have been impressed from the outset that journaled file systems were superior to FAT and NTFS. Great article David.



A. | May 26, 2017

 No readers like this yet.

Theodore Ts'o himself called EXT4 an evolutionary dead end and advised people to move to other filesystems, such as BTRFS. One reason is that journaling shortens the lifespan of a SSD considerably. Another reason is that adding snapshots to EXT4 is not easily feasible. Therefore, contrary to the authors final advise, I second Theodore Ts'o in urging people to abandon EXT4 for BTRFS.



Danglingpointer | May 26, 2017

 No readers like this yet.



Used ext4 for years since Ubuntu 9 jaunty. Have now switched to btrfs running compressed with lzo algo on the fly.



Danglingpointer | May 26, 2017

 No readers like this yet.

Used ext4 for years since Ubuntu 9 jaunty. Have now moved to btrfs and have never looked back. Been using it for 2 years now



Eddie G. | May 30, 2017

 No readers like this yet.

I have been using Linux since 2002 / '03 and I have never defragmented my drives, the files I need? I store off the main PC on a 4TB hard drive, and everything else I use daily?...doesn't cause me problems with space. I understand why the manufacturers and companies out there have to push the defrag software on the masses, its due to Microsoft's crappy filesystem. But since I'm no longer part of that ecosystem, defragmentation tools aren't a necessity for my machines anymore...(thank GOD!) Because when you look it up?...seems there are tons of various software that promise to "optimize" your disks for you....no thanks...not installing yet another program to do a job that shouldn't have to be done in the first place!.....I've even noticed there's BleachBit for Linux, and while it might seem effective, its pretty scary that harm you could possibly do to your machine using it...as it allows you to wipe certain directories and such that shouldn't be touched.



[David Both](#) | May 30, 2017

 No readers like this yet.

Eddie G. - I really never found a need to defrag at all since I started using OS/2 back in the late '80's. In fact, that calculation was first performed to show that the IBM OS/2 HPFS filesystem did not require fragmentation either. I have never used Windows as my primary operating system so fragmentation was never a problem for me except on DOS with FAT.

HPFS, like most modern filesystems, was designed to allocate space using algorithms that reduced fragmentation to negligible levels. For HPFS and EXT2/3/4, I have measured fragmentation at levels so low as to be irrelevant to most application environments.



I did, however, experiment to a couple defragmentation programs for HPFS on a test machine back in about 1991 or thereabout. It made no noticeable difference in performance and took about four hours to do the defrag. I never trusted defrag programs and always shudder when friends say they do it frequently on their non-Linux computers.

Thanks for your comment. - and thanks to all who have commented on this article.



[David Both](#) | May 30, 2017

 No readers like this yet.

Proponents of BTRFS take heart! I have tried BTRFS in the past and I will try it again at some point, because BTRFS. but I will start on a couple test hosts I have for that type of thing. The last time I tried BTRFS was a problem because the file system check program was not bug-free and I lost some data as a result.

So, in the meantime, would any of you BTRFS fans out there like to write an article similar to this one about BTRFS? Contact me via email: David.Both@Opensource.com if you want to do that. Thanks!



Leslie Satenstein | June 6, 2017

 No readers like this yet.

My system is a mix of SSD's and terrabyte drives. (2 +4). Until now, everything was based on using ext4. For a while, with the SSD's I was using xfs. Here is my experience over about 2 years of xfs use.

- a) xfs access is faster than ext4 in most cases.
- b) xfs meta data recovery is lacking. As the partition came close to being full, one could expand an xfs file system, but if one could not, performance and data recovery suffered if there was an unscheduled shutdown.

Since I use spinning disks for much of my volatile work, I believe that ext4 will do just fine.

With ext4 and journalling, recovery is just about 100%. At the worst, one file may be lost.

The article has interested me in converting some SSD partitions to btrfs. Notwithstanding that conversion, the SSD's I have will, according to my daily use, outlast my best hard disk

by 5 years. (eg, 10 year life for the SSD).

Related Content



[What's new in GNOME 44?](#)



[5 reasons virtual machines still matter](#)



[Remove the background from an image with this Linux command](#)



This work is licensed under a Creative Commons Attribution-Share Alike 4.0 International License.

ABOUT THIS SITE

The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.

Opensource.com aspires to publish all content under a **Creative Commons license** but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.

A note on advertising: Opensource.com does not sell advertising on the site or in any of its newsletters.

opensource.com



Copyright ©2024 Red Hat, Inc.

[Privacy Policy](#)

[Terms of use](#)

[Cookie preferences](#)

ln(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [AUTHOR](#) | [REPORTING BUGS](#) | [COPYRIGHT](#) | [SEE ALSO](#) | [COLOPHON](#)

LN(1)

User Commands

LN(1)**NAME** [top](#)

ln - make links between files

SYNOPSIS [top](#)

```
ln [OPTION]... [-T] TARGET LINK_NAME
ln [OPTION]... TARGET
ln [OPTION]... TARGET... DIRECTORY
ln [OPTION]... -t DIRECTORY TARGET...
```

DESCRIPTION [top](#)

In the 1st form, create a link to TARGET with the name LINK_NAME. In the 2nd form, create a link to TARGET in the current directory. In the 3rd and 4th forms, create links to each TARGET in DIRECTORY. Create hard links by default, symbolic links with **--symbolic**. By default, each destination (name of new link) should not already exist. When creating hard links, each TARGET must exist. Symbolic links can hold arbitrary text; if later resolved, a relative link is interpreted in relation to its parent directory.

Mandatory arguments to long options are mandatory for short options too.

--backup[=*CONTROL*]

make a backup of each existing destination file

- b** like **--backup** but does not accept an argument

- d, -F, --directory**
allow the superuser to attempt to hard link directories
(note: will probably fail due to system restrictions, even for the superuser)

- f, --force**
remove existing destination files

- i, --interactive**
prompt whether to remove destinations

- L, --logical**
dereference TARGETs that are symbolic links

- n, --no-dereference**
treat LINK_NAME as a normal file if it is a symbolic link to a directory

- P, --physical**
make hard links directly to symbolic links

- r, --relative**
with **-s**, create links relative to link location

- s, --symbolic**
make symbolic links instead of hard links

- S, --suffix=SUFFIX**
override the usual backup suffix

- t, --target-directory=DIRECTORY**
specify the DIRECTORY in which to create the links

- T, --no-target-directory**
treat LINK_NAME as a normal file always

- v, --verbose**
print name of each linked file

- help** display this help and exit



--version

output version information and exit

The backup suffix is '~', unless set with **--suffix** or `SIMPLE_BACKUP_SUFFIX`. The version control method may be selected via the **--backup** option or through the `VERSION_CONTROL` environment variable. Here are the values:

none, off

never make backups (even if **--backup** is given)

numbered, t

make numbered backups

existing, nil

numbered if numbered backups exist, simple otherwise

simple, never

always make simple backups

Using **-s** ignores **-L** and **-P**. Otherwise, the last option specified controls behavior when a `TARGET` is a symbolic link, defaulting to **-P**.

AUTHOR [top](#)

Written by Mike Parker and David MacKenzie.

REPORTING BUGS [top](#)

GNU coreutils online help:

<<https://www.gnu.org/software/coreutils/>>

Report any translation bugs to

<<https://translationproject.org/team/>>

COPYRIGHT [top](#)

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>.

This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO [top](#)

[link\(2\)](#), [symlink\(2\)](#)

Full documentation <<https://www.gnu.org/software/coreutils/ln>> or available locally via: info '(coreutils) ln invocation'

COLOPHON [top](#)

This page is part of the *coreutils* (basic file, shell and text manipulation utilities) project. Information about the project can be found at <<http://www.gnu.org/software/coreutils/>>. If you have a bug report for this manual page, see <<http://www.gnu.org/software/coreutils/>>. This page was obtained from the tarball *coreutils-9.4.tar.xz* fetched from <<http://ftp.gnu.org/gnu/coreutils/>> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

GNU coreutils 9.4

August 2023

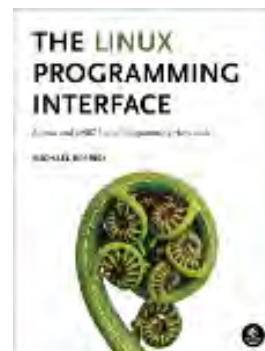
LN(1)

Pages that refer to this page: [dh_link\(1\)](#), [pmlogmv\(1\)](#), [update-alternatives\(1\)](#), [link\(2\)](#), [symlink\(2\)](#), [hier\(7\)](#), [symlink\(7\)](#), [sln\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).





LOG IN



opensource.com

A user's guide to links in the Linux filesystem

Learn how to use links, which make tasks easier by providing access to files from multiple locations in the Linux filesystem directory tree.

By [David Both](#) (Correspondent)

June 22, 2017 | [4 Comments](#) | 16 min read

 744 readers like this.



Image by: Paul Lewin. Modified by Opensource.com. CC BY-SA 2.0

In articles I have written about various aspects of Linux filesystems for Opensource.com, including [An introduction to Linux's EXT4 filesystem](#); [Managing devices in Linux](#); [An introduction to Linux filesystems](#); and [A Linux user's guide to Logical Volume Management](#), I have briefly mentioned an interesting feature of Linux filesystems that can make some tasks easier by providing access to files from multiple locations in the filesystem directory tree.



There are two types of Linux filesystem links: hard and soft. The difference between the two types of links is significant, but both types are used to solve similar problems. They both provide multiple directory entries (or references) to a single file, but they do it quite differently. Links are powerful and add flexibility to Linux filesystems because [everything is a file](#).

More Linux resources

- 👉 [Linux commands cheat sheet](#)
- 👉 [Advanced Linux commands cheat sheet](#)
- 👉 [Free online course: RHEL Technical Overview](#)
- 👉 [Linux networking cheat sheet](#)
- 👉 [SELinux cheat sheet](#)
- 👉 [Linux common commands cheat sheet](#)
- 👉 [What are Linux containers?](#)
- 👉 [Our latest Linux articles](#)

I have found, for instance, that some programs required a particular version of a library. When a library upgrade replaced the old version, the program would crash with an error specifying the name of the old, now-missing library. Usually, the only change in the library name was the version number. Acting on a hunch, I simply added a link to the new library but named the link after the old library name. I tried the program again and it worked perfectly. And, okay, the program was a game, and everyone knows the lengths that gamers will go to in order to keep their games running.

In fact, almost all applications are linked to libraries using a generic name with only a major version number in the link name, while the link points to the actual library file that also has a minor version number. In other instances, required files have been moved from one directory to another to comply with the Linux file specification, and there are links in the old directories for backwards compatibility with those programs that have not yet caught up with the new locations. If you do a long listing of the **/lib64** directory, you can find many examples of both.

```
lrwxrwxrwx. 1 root root      36 Dec  8 2016 cracklib_dict.hwm -> ../../usr/share/cracklib/pw_dict
lrwxrwxrwx. 1 root root      36 Dec  8 2016 cracklib_dict.pwd -> ../../usr/share/cracklib/pw_dict
lrwxrwxrwx. 1 root root      36 Dec  8 2016 cracklib_dict.pwi -> ../../usr/share/cracklib/pw_dict
lrwxrwxrwx. 1 root root      27 Jun  9 2016 libaccountsservice.so.0 -> libaccountsservice.so.0.0.
-rwxr-xr-x. 1 root root 288456 Jun  9 2016 libaccountsservice.so.0.0.0
lrwxrwxrwx  1 root root      15 May 17 11:47 libacl.so.1 -> libacl.so.1.1.0
```

```
-rwxr-xr-x  1 root root  36472 May 17 11:47 libacl.so.1.1.0
lrwxrwxrwx. 1 root root      15 Feb  4 2016 libaio.so.1 -> libaio.so.1.0.1
-rwxr-xr-x. 1 root root  6224 Feb  4 2016 libaio.so.1.0.0
-rwxr-xr-x. 1 root root  6224 Feb  4 2016 libaio.so.1.0.1
lrwxrwxrwx. 1 root root    30 Jan 16 16:39 libakonadi-calendar.so.4 -> libakonadi-calendar.so.4.
-rwxr-xr-x. 1 root root 816160 Jan 16 16:39 libakonadi-calendar.so.4.14.26
lrwxrwxrwx. 1 root root    29 Jan 16 16:39 libakonadi-contact.so.4 -> libakonadi-contact.so.4.14
```

A few of the links in the **/lib64** directory

The long listing of the **/lib64** directory above shows that the first character in the filemode is the letter "l," which means that each is a soft or symbolic link.

Hard links

In [An introduction to Linux's EXT4 filesystem](#), I discussed the fact that each file has one inode that contains information about that file, including the location of the data belonging to that file. [Figure 2](#) in that article shows a single directory entry that points to the inode. Every file must have at least one directory entry that points to the inode that describes the file. The directory entry is a hard link, thus every file has at least one hard link.

In Figure 1 below, multiple directory entries point to a single inode. These are all hard links. I have abbreviated the locations of three of the directory entries using the tilde (~) convention for the home directory, so that ~ is equivalent to **/home/user** in this example. Note that the fourth directory entry is in a completely different directory, **/home/shared**, which might be a location for sharing files between users of the computer.



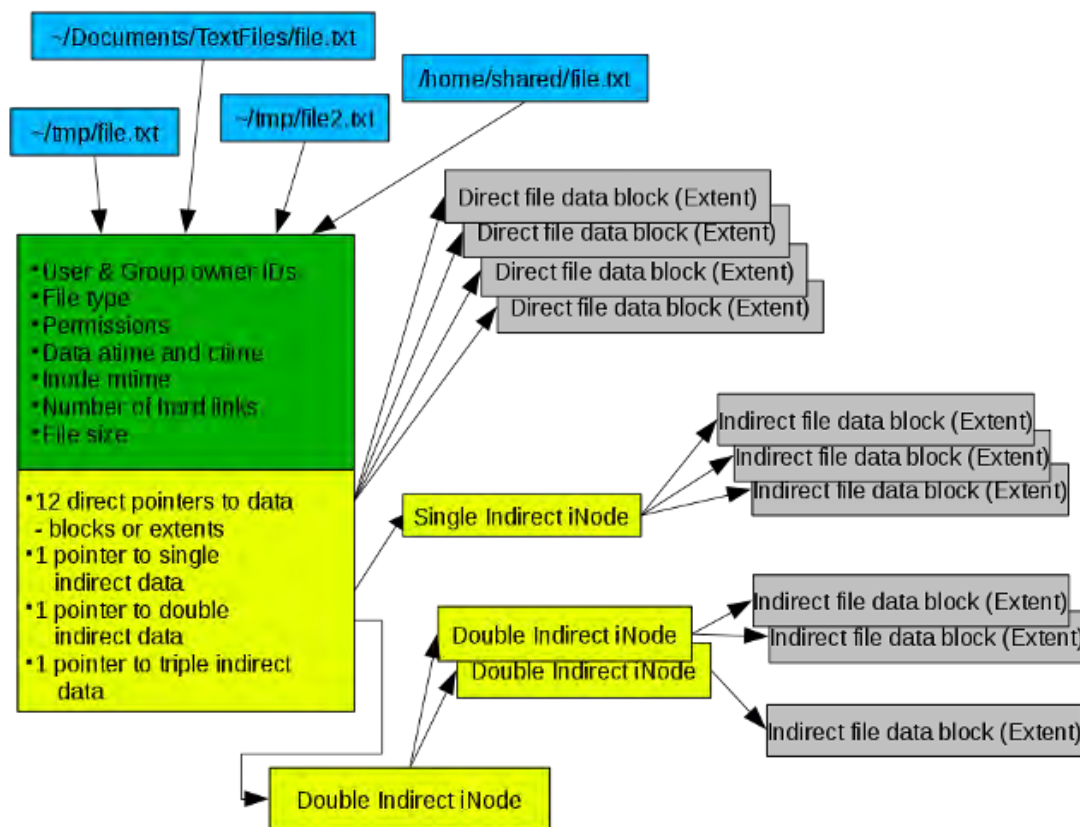


Figure 1

Hard links are limited to files contained within a single filesystem. "Filesystem" is used here in the sense of a partition or logical volume (LV) that is mounted on a specified mount point, in this case `/home`. This is because inode numbers are unique only within each filesystem, and a different filesystem, for example, `/var` or `/opt`, will have inodes with the same number as the inode for our file.

Because all the hard links point to the single inode that contains the metadata about the file, all of these attributes are part of the file, such as ownerships, permissions, and the total number of hard links to the inode, and cannot be different for each hard link. It is one file with one set of attributes. The only attribute that can be different is the file name, which is not contained in the inode. Hard links to a single **file/inode** located in the same directory must have different names, due to the fact that there can be no duplicate file names within a single directory.

The number of hard links for a file is displayed with the `ls -l` command. If you want to display the actual inode numbers, the command `ls -li` does that.

Symbolic (soft) links

The difference between a hard link and a soft link, also known as a symbolic link (or symlink), is that, while hard links point directly to the inode belonging to the file, soft links point to a directory entry, i.e., one of the hard links. Because soft links point to a hard link for the file and not the inode, they are not dependent upon the inode number and can work across filesystems, spanning partitions and LVs.



The downside to this is: If the hard link to which the symlink points is deleted or renamed, the symlink is broken. The symlink is still there, but it points to a hard link that no longer exists. Fortunately, the **ls** command highlights broken links with flashing white text on a red background in a long listing.

Lab project: experimenting with links

I think the easiest way to understand the use of and differences between hard and soft links is with a lab project that you can do. This project should be done in an empty directory as a *non-root user*. I created the **~/temp** directory for this project, and you should, too. It creates a safe place to do the project and provides a new, empty directory to work in so that only files associated with this project will be located there.

Initial setup

First, create the temporary directory in which you will perform the tasks needed for this project. Ensure that the present working directory (PWD) is your home directory, then enter the following command.

```
mkdir temp
```

Change into **~/temp** to make it the PWD with this command.

```
cd temp
```

To get started, we need to create a file we can link to. The following command does that and provides some content as well.

```
du -h > main.file.txt
```

Use the **ls -l** long list to verify that the file was created correctly. It should look similar to my results. Note that the file size is only 7 bytes, but yours may vary by a byte or two.

```
[dboth@david temp]$ ls -l
total 4
-rw-rw-r-- 1 dboth dboth 7 Jun 13 07:34 main.file.txt
```

Notice the number "1" following the file mode in the listing. That number represents the number of hard links that exist for the file. For now, it should be 1 because we have not created any additional links to our test file.

Experimenting with hard links

Hard links create a new directory entry pointing to the same inode, so when hard links are added to a file, you will see the number of links increase. Ensure that the PWD is still **~/temp**. Create a hard link to the file **main.file.txt**, then do another long list of the directory.

```
[dboth@david temp]$ ln main.file.txt link1.file.txt
[dboth@david temp]$ ls -l
total 8
-rw-rw-r-- 2 dboth dboth 7 Jun 13 07:34 link1.file.txt
-rw-rw-r-- 2 dboth dboth 7 Jun 13 07:34 main.file.txt
```

Notice that both files have two links and are exactly the same size. The date stamp is also the same. This is really one file with one inode and two links, i.e., directory entries to it. Create a second hard link to this file and list the directory contents. You can create the link to either of the existing ones: **link1.file.txt** or **main.file.txt**.

```
[dboth@david temp]$ ln link1.file.txt link2.file.txt ; ls -l
total 16
-rw-rw-r-- 3 dboth dboth 7 Jun 13 07:34 link1.file.txt
-rw-rw-r-- 3 dboth dboth 7 Jun 13 07:34 link2.file.txt
-rw-rw-r-- 3 dboth dboth 7 Jun 13 07:34 main.file.txt
```

Notice that each new hard link in this directory must have a different name because two files—really directory entries—cannot have the same name within the same directory. Try to create another link with a target name the same as one of the existing ones.

```
[dboth@david temp]$ ln main.file.txt link2.file.txt
ln: failed to create hard link 'link2.file.txt': File exists
```

Clearly that does not work, because **link2.file.txt** already exists. So far, we have created only hard links in the same directory. So, create a link in your home directory, the parent of the temp directory in which we have been working so far.

```
[dboth@david temp]$ ln main.file.txt ../main.file.txt ; ls -l ../main*
-rw-rw-r-- 4 dboth dboth 7 Jun 13 07:34 main.file.txt
```

The **ls** command in the above listing shows that the **main.file.txt** file does exist in the home directory with the same name as the file in the temp directory. Of course, these are not different

files; they are the same file with multiple links—directory entries—to the same inode. To help illustrate the next point, add a file that is not a link.

```
[dboth@david temp]$ touch unlinked.file ; ls -l
total 12
-rw-rw-r-- 4 dboth dboth 7 Jun 13 07:34 link1.file.txt
-rw-rw-r-- 4 dboth dboth 7 Jun 13 07:34 link2.file.txt
-rw-rw-r-- 4 dboth dboth 7 Jun 13 07:34 main.file.txt
-rw-rw-r-- 1 dboth dboth 0 Jun 14 08:18 unlinked.file
```

Look at the inode number of the hard links and that of the new file using the **-i** option to the **ls** command.

```
[dboth@david temp]$ ls -li
total 12
657024 -rw-rw-r-- 4 dboth dboth 7 Jun 13 07:34 link1.file.txt
657024 -rw-rw-r-- 4 dboth dboth 7 Jun 13 07:34 link2.file.txt
657024 -rw-rw-r-- 4 dboth dboth 7 Jun 13 07:34 main.file.txt
657863 -rw-rw-r-- 1 dboth dboth 0 Jun 14 08:18 unlinked.file
```

Notice the number **657024** to the left of the file mode in the example above. That is the inode number, and all three file links point to the same inode. You can use the **-i** option to view the inode number for the link we created in the home directory as well, and that will also show the same value. The inode number of the file that has only one link is different from the others. Note that the inode numbers will be different on your system.

Let's change the size of one of the hard-linked files.

```
[dboth@david temp]$ df -h >link2.file.txt ; ls -li
total 12
657024 -rw-rw-r-- 4 dboth dboth 1157 Jun 14 14:14 link1.file.txt
657024 -rw-rw-r-- 4 dboth dboth 1157 Jun 14 14:14 link2.file.txt
657024 -rw-rw-r-- 4 dboth dboth 1157 Jun 14 14:14 main.file.txt
657863 -rw-rw-r-- 1 dboth dboth 0 Jun 14 08:18 unlinked.file
```

The file size of all the hard-linked files is now larger than before. That is because there is really only one file that is linked to by multiple directory entries.

I know this next experiment will work on my computer because my **/tmp** directory is on a separate LV. If you have a separate LV or a filesystem on a different partition (if you're not using LVs), determine whether or not you have access to that LV or partition. If you don't, you can try to insert a USB memory stick and mount it. If one of those options works for you, you can do this experiment.



Try to create a link to one of the files in your **~/temp** directory in **/tmp** (or wherever your different filesystem directory is located).

```
[dboth@david temp]$ ln link2.file.txt /tmp/link3.file.txt
ln: failed to create hard link '/tmp/link3.file.txt' => 'link2.file.txt':
Invalid cross-device link
```

Why does this error occur? The reason is each separate mountable filesystem has its own set of inode numbers. Simply referring to a file by an inode number across the entire Linux directory structure can result in confusion because the same inode number can exist in each mounted filesystem.

There may be a time when you will want to locate all the hard links that belong to a single inode. You can find the inode number using the **ls -li** command. Then you can use the **find** command to locate all links with that inode number.

```
[dboth@david temp]$ find . -inum 657024
./main.file.txt
./link1.file.txt
./link2.file.txt
```

Note that the **find** command did not find all four of the hard links to this inode because we started at the current directory of **~/temp**. The **find** command only finds files in the PWD and its subdirectories. To find all the links, we can use the following command, which specifies your home directory as the starting place for the search.

```
[dboth@david temp]$ find ~ -samefile main.file.txt
/home/dboth/temp/main.file.txt
/home/dboth/temp/link1.file.txt
/home/dboth/temp/link2.file.txt
/home/dboth/main.file.txt
```

You may see error messages if you do not have permissions as a non-root user. This command also uses the **-samefile** option instead of specifying the inode number. This works the same as using the inode number and can be easier if you know the name of one of the hard links.

Experimenting with soft links

As you have just seen, creating hard links is not possible across filesystem boundaries; that is, from a filesystem on one LV or partition to a filesystem on another. Soft links are a means to answer that



problem with hard links. Although they can accomplish the same end, they are very different, and knowing these differences is important.

Let's start by creating a symlink in our `~/temp` directory to start our exploration.

```
[dboth@david temp]$ ln -s link2.file.txt link3.file.txt ; ls -li
total 12
657024 -rw-rw-r-- 4 dboth dboth 1157 Jun 14 14:14 link1.file.txt
657024 -rw-rw-r-- 4 dboth dboth 1157 Jun 14 14:14 link2.file.txt
658270 lrwxrwxrwx 1 dboth dboth  14 Jun 14 15:21 link3.file.txt ->
link2.file.txt
657024 -rw-rw-r-- 4 dboth dboth 1157 Jun 14 14:14 main.file.txt
657863 -rw-rw-r-- 1 dboth dboth   0 Jun 14 08:18 unlinked.file
```

The hard links, those that have the inode number **657024**, are unchanged, and the number of hard links shown for each has not changed. The newly created symlink has a different inode, number **658270**. The soft link named **link3.file.txt** points to **link2.file.txt**. Use the **cat** command to display the contents of **link3.file.txt**. The file mode information for the symlink starts with the letter "**l**" which indicates that this file is actually a symbolic link.

The size of the symlink **link3.file.txt** is only 14 bytes in the example above. That is the size of the text **link3.file.txt -> link2.file.txt**, which is the actual content of the directory entry. The directory entry **link3.file.txt** does not point to an inode; it points to another directory entry, which makes it useful for creating links that span file system boundaries. So, let's create that link we tried before from the **/tmp** directory.

```
[dboth@david temp]$ ln -s /home/dboth/temp/link2.file.txt
/tmp/link3.file.txt ; ls -l /tmp/link*
lrwxrwxrwx 1 dboth dboth 31 Jun 14 21:53 /tmp/link3.file.txt ->
/home/dboth/temp/link2.file.txt
```

Deleting links

There are some other things that you should consider when you need to delete links or the files to which they point.

First, let's delete the link **main.file.txt**. Remember that every directory entry that points to an inode is simply a hard link.

```
[dboth@david temp]$ rm main.file.txt ; ls -li
total 8
657024 -rw-rw-r-- 3 dboth dboth 1157 Jun 14 14:14 link1.file.txt
657024 -rw-rw-r-- 3 dboth dboth 1157 Jun 14 14:14 link2.file.txt
```




```
658270 lrwxrwxrwx 1 dboth dboth 14 Jun 14 15:21 link3.file.txt ->
link2.file.txt
657863 -rw-rw-r-- 1 dboth dboth 0 Jun 14 08:18 unlinked.file
```

The link **main.file.txt** was the first link created when the file was created. Deleting it now still leaves the original file and its data on the hard drive along with all the remaining hard links. To delete the file and its data, you would have to delete all the remaining hard links.

Now delete the **link2.file.txt** hard link.

```
[dboth@david temp]$ rm link2.file.txt ; ls -li
total 8
657024 -rw-rw-r-- 3 dboth dboth 1157 Jun 14 14:14 link1.file.txt
658270 lrwxrwxrwx 1 dboth dboth 14 Jun 14 15:21 link3.file.txt ->
link2.file.txt
657024 -rw-rw-r-- 3 dboth dboth 1157 Jun 14 14:14 main.file.txt
657863 -rw-rw-r-- 1 dboth dboth 0 Jun 14 08:18 unlinked.file
```

Notice what happens to the soft link. Deleting the hard link to which the soft link points leaves a broken link. On my system, the broken link is highlighted in colors and the target hard link is flashing. If the broken link needs to be fixed, you can create another hard link in the same directory with the same name as the old one, so long as not all the hard links have been deleted. You could also recreate the link itself, with the link maintaining the same name but pointing to one of the remaining hard links. Of course, if the soft link is no longer needed, it can be deleted with the **rm** command.

The **unlink** command can also be used to delete files and links. It is very simple and has no options, as the **rm** command does. It does, however, more accurately reflect the underlying process of deletion, in that it removes the link—the directory entry—to the file being deleted.

Final thoughts

I worked with both types of links for a long time before I began to understand their capabilities and idiosyncrasies. It took writing a lab project for a Linux class I taught to fully appreciate how links work. This article is a simplification of what I taught in that class, and I hope it speeds your learning curve.

Tags:

LINUX

David Both





David Both is an Open Source Software and GNU/Linux advocate, trainer, writer, and speaker. He has been working with Linux and Open Source Software since 1996 and with computers since 1969. He is a strong proponent of and evangelist for the "Linux Philosophy for System Administrators."

[More about me](#)

4 Comments

These comments are closed.



[Greg Pittman](#) | June 22, 2017

 No readers like this yet.

Linking is in some uses an alternative to or complement to making use of \$PATH, the advantage being you can make a link for any kind of file (especially libraries, which in my experience seems to be the most common need), whereas you specify a path for executables.



dgrb | June 23, 2017

 No readers like this yet.

There is a hard link "gotcha" which IMHO is worth mentioning.

If you use an editor which makes automatic backups - emacs certainly is one such - then you may end up with a new version of the edited file, while the backup is the linked copy, because the editor simply renames the file to the backup name (with emacs, test.c would be renamed test.c~) and the new version when saved under the old name is no longer linked.

Symbolic links avoid this problem, so I tend to use them for source code where required.



David Bowskill | June 26, 2017

 No readers like this yet.



Dear Administrator

I have a question as regards the /dev directory.

Why are there always a large number of empty device files ?

Would it be possible for the hardware to be interrogated on startup and just files corresponding to the actual hardware be generated ?

Thanks for any answers



[David Both](#) | June 27, 2017

 No readers like this yet.

The short answer is yes, Linux has been doing just as you suggest for quite some time, now. I am sure it seems that there are many more device files than actual devices, but most are actually used in one way or another.

My article "Managing devices in Linux" at <https://opensource.com/article/16/11/managing-devices-linux> does talk about exactly this.

Thanks for your question.

Related Content



[What's new in GNOME 44?](#)



[5 reasons virtual machines still matter](#)



[Remove the background from an image with this Linux command](#)



This work is licensed under a Creative Commons Attribution-Share Alike 4.0 International License.

ABOUT THIS SITE



The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.

Opensource.com aspires to publish all content under a **Creative Commons license** but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.

A note on advertising: Opensource.com does not sell advertising on the site or in any of its newsletters.

opensource.com

Copyright ©2024 Red Hat, Inc.

[Privacy Policy](#)

[Terms of use](#)

[Cookie preferences](#)



CS 2130

[Readings](#) / [Memory Overview](#)

An overview of memory

This is intended to be a practical guide (rather than an authoritative guide) to memory allocation and management in C, as implemented by clang and gcc for the x86-64 processor family under Linux.

Memory layout

Compiled binaries are typically *relocatable*, meaning they contain only guidelines about where code may be located in memory; the loader is responsible for assigning specific addresses prior to running the code.

Linux x86-64's loaders provide the following contents of memory, with large addresses at the top:

| Address range | Use |
|---------------------|---|
| above 0xFFFFFFFF | kernel memory (OS runs here; if your code accesses these segments as e.g. via <code>*(int *)-1</code> , your code crashes) |
| below 0xFFFFFFFF | user stack (grows into smaller addresses) |
| | empty space for future stack growth |
| | memory-mapped region (shared libraries) |
| | empty space for future heap growth |
| | run-time heap (grows into larger addresses) |
| | read/write segments (<code>.data</code> for initialized globals, <code>.bss</code> for uninitialized globals) |
| above 0x400000 | read-only code and data (<code>.init</code> run by loader, <code>.text</code> is your code, <code>.rodata</code> is string constants and such) |
| 0x0-0x400000 | unused segments, so that <code>*(int *)0</code> and the like crashes |

All of the user-access regions may be randomized (doing so is called ASLR: Address Space Layout Randomization) to prevent certain families of security vulnerabilities.

Pointers to `struct`s

It is common to use a pointer to a `struct`. In fact, it is uncommon to have a `struct` typed variable; almost all `struct`s are handled through a pointer. But this leads to a syntactic unpleasantness. Because `.` has higher precedence than `*`, `*a.b` means `*(a.b)`, so to get a field from a pointer to a `struct` requires parentheses: `(*a).b`.

Because of this, C has a special operator `a->b` that means `(*a).b`. We use it extensively in place of what Java or Python would do with a `.`.

Using the stack in C

Although the compiler may optimize this by placing variables in registers, conceptually all local variables (including function parameters) are stored on the stack. Because you've already learned to write programs with local variables, you've also already learned to use stack memory.

Stack memory is automatically "allocated" when a function is invoked and "de-allocated" when a function returns, although this does not actually entail much work beyond changing the contents of `%rsp`. Because of this, you should **never return the address of a local variable**.

Example: Consider the following program:

```
int *makeArray() {
    int answer[5];
    return answer;
}

void setTo(int *array, int length, int value) {
    for(int i=0; i<length; i+=1) array[i] = value;
}

int main(int argc, const char *argv[]) {
    int *a1 = makeArray();
    setTo(a1, 5, -2);
    return 0;
}
```

The function `makeArray` will allocate room for 5 `int`s on the stack (e.g., by adding `-20` to `%rsp`) and return the address of those `int`s. The function `setTo` will then be invoked, with its first argument being a pointer to *its own stack frame* as `setTo` reuses the same stack memory that `makeArray` used. If `setTo` stores `i` on the stack (as opposed to optimizing it into a register), `setTo` will not work properly. For example, we might see something like

| Address | <code>makeArray's</code> use | <code>setTo's</code> use |
|---------|-------------------------------------|---|
| ...200 | return address | return address |
| ...1F8 | saved copy of <code>%rbp</code> | saved copy of <code>%rbp</code> |
| ...1F4 | allocated as <code>answer[4]</code> | pointed to by <code>array[4]</code> and allocated as <code>i</code> |
| ...1F0 | allocated as <code>answer[3]</code> | pointed to by <code>array[3]</code> |
| ...1EC | allocated as <code>answer[2]</code> | pointed to by <code>array[2]</code> |



| Address | makeArray's use | setTo's use |
|---------|-------------------------------------|-------------------------------------|
| ...1E8 | allocated as <code>answer[1]</code> | pointed to by <code>array[1]</code> |
| ...1E4 | allocated as <code>answer[0]</code> | pointed to by <code>array[0]</code> |

... which would mean that in `setTo` the values of `i` will repeat an infinite loop: 0, 1, 2, 3, 4, in the usual way, but then iteration `i=4` will assign to `array[4]` which is address `...1F4` which is also the value of `i`, setting it to -2 and causing the loop to repeat as -1, 0, 1, 2, 3, 4, -1, 0, 1, ... forever.

Note that this bug may become invisible if we compile with optimizations and `i` is stored only in a register; we still did the wrong thing in `makeArray`, so this is still a bug, but we might not see it in this program.

Using global variables in C

Global variables in C are allocated in regions of memory that are accessible to all functions, which memory is set aside when the program is compiled and thus must be of a size the compiler can determine at compile time. Use of global variables can be an efficient way to program, provided you know in advance how much memory you'll need.

A common pattern in using global arrays is to (a) `#define` a maximum size and (b) use a variable to track how much has actually been used.

Example: The following is a simplified partial example of how one might collect a set of courses a student is interested in:

```

/* A function we'd need to define elsewhere that reads up to max chars *
 * from the keyboard into an array, returning how many were read      */
unsigned get_input(char *dest, unsigned max);

#define MAX_CLASS_SIZE 12
#define MAX_CLASSES 20

/* Note: 2D arrays are declared with sizes in the same order as indices */
char interest[MAX_CLASSES][MAX_CLASS_SIZE];
unsigned classes = 0;

int main(int argc, const char *argv[]) {
    while (classes < MAX_CLASSES) {
        puts("What class are you interested in? Press enter when done.");
        unsigned got = get_input(interest[classes], MAX_CLASS_SIZE);
        if (got == 0) break;
        else classes += 1;
    }
}

```



```
puts("You expressed interest in the following:");
for(int i=0; i<classes; i+=1) {
    puts(interest[i]);
}

return 0;
}
```

Correct implementation of functions like `get_input` will be a subject for a future part of this course.

Because global arrays are simple to program and efficient in practice, they are common in C code. Because a global array typically needs associated global information like used size, that means other global variables are also common.

A sizable (though not unanimous) majority of software engineering texts I have consulted explicitly state that "global variables are **bad**." One of the more readable examples I've found is <http://wiki.c2.com/?GlobalVariablesAreBad>. However, even when well written these tend to refer to topics like "coupling" and "namespace pollution" that are difficult to motivate properly before you've work on large software projects yourself.

Using the heap in C

If the stack cannot be used for long-term pointers and global variables have to be allocated at compile time and may also be bad for software maintenance, how do you allocate memory as the program runs and pass it around to different functions? The answer: put it in/on¹ the heap.

it roughly equally common to refer to the value as being "on the heap" or "in the heap". There appears to be a slight preference for "on" to refer to the memory itself and "in" to refer to the values stored using that memory, but many exceptions exist.

The heap is a region of memory where

- any number of chunks of memory of any size and purpose may co-exist
- new chunks can be added as the program runs
- each chunk remains until it is explicitly deallocated or the program terminates

"Heap" is used in two main ways in programming. When discussing memory, "the heap" is an unorganized region of memory made out of many heterogeneous chunks of memory with different purposes. When discussing data structures, "a heap" is a partially-organized tree structure where "small" things make their way to the top without requiring complete ordering of the whole. There is no relationship between these: *the* heap is not *a* heap, and



heap need not be stored in *the* heap. This course will only use it in the former way: a region of memory, not a data structure.

Managing memory

The operating system has final say on what memory, and how much of it, each program gets. It handles this via several concepts, including *virtual addresses* that ensure that two different processes cannot access one another's memory and *segments* that prevent you from jumping to an array or dereferencing a pointer to unallocated memory.

Operating systems typically allocate memory in large regions called "pages". It is common today² for pages to be 4KB – that is, 4096 bytes. Programs can ask the operating system for more pages of memory, or return pages of memory to the OS. However, programmers typically want to handle memory at finer-grained resolution, and C provides library functions to assist with this.

`malloc` (MEMORY ALLOCATE)

The library function `void *malloc(size_t size);` returns a pointer to the first byte of a `size`-byte region of memory that is allocated on the heap and not used by any previous purpose. It does this by

- Checking to see if it has enough space on partially-used heap page. If not, ask the OS to allocate new pages until it has enough unused heap space.
- Pick an address to return.
- Add that address and its allocated size in a special bookkeeping data structure.
- Return the address.

The internal bookkeeping data structure allows subsequent calls to `malloc` to be guaranteed not to return the same (or an overlapping) region a second time.

Example: It is typical to `malloc` a `struct` with `sizeof` and a pointer type cast, like

```
typedef struct student_s {
    const char *name;
    int credits;
} student;

student *enroll(const char *name, int transfer_credits) {
    student *ans = (student *)malloc(sizeof(student));
    ans->name = name;
    ans->credits = transfer_credits;
    return ans;
}
```



Example: It is typical to `malloc` an array with `sizeof` and a multiplier, like

```
typedef struct length_array_s {
    long *data;
    unsigned capacity;
    unsigned size;
} larray;

larray *make_array() {
    larray *ans = (larray *)malloc(sizeof(larray));
    ans->size = 0;
    ans->capacity = 16;
    ans->data = (long *)malloc(sizeof(long) * ans->capacity);
    return ans;
}
```

With very rare exceptions, we `malloc` to store one or more values of a given size inside the malloced memory; if you find yourself `malloc`ing *without* a `sizeof` inside, you almost certainly did something wrong.

`free`

The library function `void free(void *)`; accepts a pointer returned by `malloc` and marks it as no longer in use, and hence as available for future `malloc`s.

In small, short-running programs you may be able to get away with never `free`ing your data structures, but in larger and longer-running programs this can cause a program to hog all available memory on the computer, slowing all operations and possibly even crashing the program or entire computer. Programs that allocate memory and then forget about it without `free`ing it are said to have a [Memory leak]

`calloc` AND `realloc`

Two additional convenience functions can also be useful.

`x = calloc(n, s)`; is the same as `x = malloc(n * s)`; except that it (a) may be optimize for storing an array of `n` distinct `s`-byte values and (b) sets all bytes of allocated memory to `0`. Notably, `malloc` does *not* erase the memory it returns.

Example: After running the following code:

```
int *x = (int *)malloc(sizeof(int));
int *y = (int *)calloc(1, sizeof(int));
int a = *x;
int b = *y;
```



the value of `a` may be anything, while `b` is guaranteed to be `0`.

```
x = realloc(x, s);
```

- *either* extends the previously-allocated region pointed to by `x` to be `s` bytes long,
- *or*
 - 1 allocates a new `s`-byte region,
 - 2 copies the bytes previously pointed to by `x` into this new region,
 - 3 `free(x)`, and then
 - 4 returns the new region's address.

Example: After running the following code:

```
int *x = (int *)calloc(8, sizeof(int));
x[4] = 123;
int *x = (int *)realloc(x, 16*sizeof(int));
```

`x` is a pointer to an array of 16 `int`s. The first 8 elements are `{0, 0, 0, 0, 123, 0, 0, 0}` and the next 8 could be anything.

Garbage collectors

Many languages do not have an equivalent to `free`. They let you allocate memory, but never ask you to deallocate it. They avoid (most) memory leaks by adding to your program a *garbage collector*.

GARBAGE

Memory is **garbage** if it is (a) allocated on the heap and (b) will never be used in the future.

Memory is **unreachable** if it is (a) allocated on the heap and (b) it is not part of a *reachable* allocated memory block. A block of memory returned by a single call to `malloc` or its friends is **reachable** if any of the following are true:

- its address is in a program register, or
- its address is on the stack, or
- its address is in a reachable block of memory

All *unreachable* memory is *garbage*, but not all *garbage* is *unreachable*. Some sources call unreachable memory “syntactic garbage” and the more general category of garbage “semantic garbage”; confusingly, it is also not hard to find sources that use the word “garbage” to mean “unreachable” and ignore the existence of other kinds of garbage.

Example: Consider the following code:



```
int bad(int a) {
    int *list = calloc(a, sizeof(int));
    for(int i=0; i<a; i+=1) list[i] = (i+1)*(i+1);
    int sum = 0;
    for(int i=0; i<a; i+=1) sum += list[i];
    // midpoint
    int ans = 0;
    while (sum > 0) {
        ans += 1;
        ans >>= 1;
    }
    return ans;
}
```

The memory returned by the `calloc` call becomes garbage at the comment `// midpoint` because it is never used after that; it becomes unreachable after the function returns (and hence is a [memory leak](#)).

It's not related to memory, but what mathematical function does `bad(n)` compute?³

GARBAGE DETECTION

There are several well-known, well-studied, and carefully-implemented algorithms for performing garbage detection; almost all of these detect only unreachable garbage. A **garbage collector** is a process that detects garbage and then frees it; the most common model (technically a "tracing garbage collector") works as follows:

- inspect the entire contents of a program's memory
- flag as garbage all unreachable memory on the heap
- `free` that garbage

These steps require significant bookkeeping data structures and processing power, and periodically pause the entire program to perform a garbage detection hunt⁴. In general, this can slow down a program, and increase the memory it uses, and cause it to pause at awkward times. As garbage collectors become more sophisticated and computer memory becomes cheaper these concerns are decreasingly important, and the ability to write code that does not need to worry about `free`ing unused memory is a definite plus for software developers. However, because garbage collection always requires some space and time overhead, and because every byte and cycle always matters for some programmers somewhere, languages like C that do not have garbage collection remain common.

Detecting and avoiding bugs

This section is devoted to common mistakes people make when handling memory. Some of it is a duplication of material above, re-phrased here for inclusion in the general category of "bugs".

Using the "address sanitizer"



Most current compilers come with an option to compile in to the binary some additional information that can detect many common kinds of memory errors.

If you compile using `clang`, you can add `-fsanitize=address` to add in these checks. To get useful error messages when a problem is found with your code, you should also add `-g` and `-fno-omit-frame-pointer`

Note that the address sanitizer inserts bug detection code into the binary at compile time, but only actually detects bugs when the compiled program is run. Because of this, bugs that exist but that your program doesn't use (e.g., because they are in a branch of an `if` statement that your test cases do not exercise) are not detected.

Both `gcc` and `clang` have a variety of different categories of command-line flags. Often the first letter tells you something about the flag:

- `-f...` enables or disables a specific feature, such as an optimization, protection, or syntax extension.
- `-m...` changes some aspects of the ISA code is generated toward
- `-O...` selects a group of commonly-used optimizations (some of which are individually selectable using `-f...` options)
- `-W...` controls what warning messages are displayed
- `-g...` specifies how much debugging information should be generated and included in the binary

Common kinds of problems

The following are brief descriptions of several common memory bugs.

MEMORY LEAK

A memory leak occurs when you fail to `free` **garbage** or otherwise keep un-`free`d heap allocations of un-used memory.

The [address sanitizer](#) can detect this bug, but is somewhat conservative in what it looks for. Often it is necessary to explicitly change a pointer before the sanitizer notices the leak.

Example:

```
/** represents a mathematical expression */
typedef struct expr_s {
    char kind; // '=' for literal, or '+', '-', or '*' for operators
    long value; // only used by '='
    struct expr_s *left; // only used by operators
    struct expr_s *right; // only used by operators
} expr;

/** turns an expression of literals into a literal */
long flatten(expr *e) {
```



```
if (e->kind == '+') {
    e->kind = '=';
    e->value = flatten(e->left) + flatten(e->right);

    /* memory leak: remove pointers without free
     * asan will only notice it because of the explicit = NULL */
    e->left = e->right = NULL;
} else if (e->kind == '-') {
    e->kind = '=';
    e->value = flatten(e->left) - flatten(e->right);
    free(e->left);
    free(e->right);
    e->left = e->right = NULL;
} else if (e->kind == '*') {
    e->kind = '=';
    e->value = flatten(e->left) * flatten(e->right);
    free(e->left);
    free(e->right);
    e->left = e->right = NULL;
}
return e->value;
}
```

Memory leaks tend to make the program use more and more memory, becoming slower and slower the longer it runs. In the worst case, this can even cause your entire system to grind to a halt.

Garbage collectors are often said to remove the chance of memory leaks, but this not not strictly true: identifying all *garbage* is not possible in every case so they generally find and free *unreachable memory* instead. Even when writing in Java, Python, or other garbage-collected languages, make sure you set unused references to objects to `None`/`null` and otherwise don't maintain references to data you will not reuse.

UNINITIALIZED MEMORY

Because `malloc` and `realloc` do not initialize the memory they allocate, it is an error to access that memory before you initialize it. This is also true of local or global variables, structs, and arrays. Using uninitialized memory is a particularly tricky bug to notice because it is often the case that for many runs in a row the uninitialized memory just happens to be all `0` bytes, and then one time it happens to be other values instead, causing the bug to manifest itself intermittently.

Example:



```
int *allsum(int *y, int n) {
    int *x = (int *)malloc(sizeof(int)*n);
    for (int i=1; i<n; i+=1)
        for (int j=0; j<i; j+=1)
            x[j] += y[i];
    return x;
}
```

ACCIDENTAL CAST-TO-POINTER

If a function expects a pointer and you give it an integer instead, it will interpret the integer value as being an address. This is particularly problematic with variadic functions like `printf` and `scanf` that are harder for the compiler to type-check.

Example:

```
int x; scanf("%d", x); // should have been scanf("%d", &x);

int x; printf("%s", x); // %s means "char *" not "int"
```

WRONG USE OF `sizeof`

It is fairly common to make mistakes with `sizeof`, such as

- using `sizeof(T)` when you meant `sizeof(T *)`

```
int **A = (int **)malloc(sizeof(int) * n);
```

- using `sizeof(T)` when adding to a `T *`

```
int *find(int *p, int val) {
    while(*p && *p != val) p += sizeof(int);
    return p;
}
```

- failing to use `sizeof` when `malloc`ing

```
int *ten_ints = (int *)malloc(10);
```



UNARY OPERATOR PRECEDENCE MISTAKES

Most programmers have a hard time remembering the order of operations between prefix and postfix unary operators. Is `&a.b` `(&a).b` or `&(a.b)`? Is `*a++` `(*a)++` or `*(a++)`? Etc.

This lack of clarity leads to programmer mistakes that can cause many kinds of problems; when it includes modifying (or failing to modify) a pointer, those problems can become memory errors.

A few suggestions to avoid these:

- If you have prefix- and postfix-operators, always include parentheses to keep them separate
- Avoid postfix `--` and `++` unless you actually need their return-previous-value semantics
- Make use of `->` instead of a combination of `*` and `.` whenever you can

USE AFTER FREE

After you `free` a block of memory, using a pointer to it is an error.

The [address sanitizer](#) is usually able to detect this bug.

Example: The following is a minimal example

```
int *x = (int *)malloc(sizeof(int)*10);
int *y = &(x[5]);
free(x);
int z = *y;
```

More realistic examples generally hide the copying of the pointer and the freeing of its target memory inside other custom functions.

STACK BUFFER OVERFLOW

If you index past the end of a stack-allocated memory region, this is called a “stack buffer overflow”. This rarely crashes a program itself, but usually messes up what it will do in the future by changing the value of some other local variable or overwriting the return address.

Changing the return address usually causes a segfault when you `retq`, but stack buffer overflow can also allow malicious users to take over your program by intentionally supplying a return address that causes `retq` to jump to an address of code they included in the buffer overflow or some other code you didn’t want to run.

The [address sanitizer](#) is usually able to detect this bug.

Example:

```
char word[16];
scanf("%s", word); // overflows if type a 16+-character word
```



Since `scanf`'s `%s` format specifier reads a non-whitespace sequence of characters into `word`, this will be a buffer overflow if you type sixteen or more characters without any whitespace.

HEAP BUFFER OVERFLOW

If you index past the end of a heap-allocated memory region, this is called a "heap buffer overflow". This can "corrupt the heap" – that is, the program continues to run, but the overflow modified some other data structure, messing up some other part of your program.

The [address sanitizer](#) is usually able to detect this bug.

Example:

```
char word = (char *)malloc(16 * sizeof(char));
scanf("%s", word); // overflows if type a 16+-character word
```

GLOBAL BUFFER OVERFLOW

If you index past the end of global memory region, this is called a "global buffer overflow". This sometimes causes a segfault, or it might overwrite a different global variable.

The [address sanitizer](#) is usually able to detect this bug.

Example:

```
char word[16];
int f() {
    scanf("%s", word); // overflows if type a 16+-character word
}
```

USE AFTER RETURN

If you return the address of a local variable, and then later use that pointer, you have a use-after-return bug.

The [address sanitizer](#) is usually able to detect this bug.

See the worked example in the section [Using the stack in C] above.

UNINITIALIZED POINTER

If you dereference a pointer that you failed to initialize, you are likely to end up in an invalid code segment and get a segfault; however, you might by random bad luck end up with a pre-initialized value that points to valid memory and end up overwriting a value some other part of the program depends on.

Example:



```
int *x; int y = *x; // a fairly obvious bug...

printf("%s"); // printf's %s means "a char *" which we failed to supply
```

USE AFTER SCOPE

This is a nuance related to use-after-free.

Each set of braces and each for loop creates its own variable scope. The compiler is free to re-use that stack space after the scope ends if it wants. If you use a pointer to an out-of-scope variable, this creates a user-after-scope bug.

The [address sanitizer](#) is able to detect this bug, but requires a special additional flag during compilation to do so: `-fsanitize-address-use-after-scope`.

Example: The following code may or may not have this bug, depending on how the compiler chooses to optimize it.

```
int *p;
{
    int x = 0;
    p = &x;
}
*p = 5;
```

- 1 When a value is stored in memory that is part of the heap, ↩
- 2 a.d. 2018 ↩
- 3 Answer: `floor(log2((n)*(n+1)/2))+1` ↩
- 4 There are garbage collectors that run *while* the rest of the program is modifying memory, but doing so has various challenges that make them more complicated and have various possible drawbacks. See https://en.wikipedia.org/wiki/Tracing_garbage_collection#Stop-the-world_vs_incremental_vs_concurrent for more. ↩

Copyright © 2023 John Hott, portions Luther Tychonievich.
Released under the [CC-BY-NC-SA 4.0](#) license.



Stack Layouts:

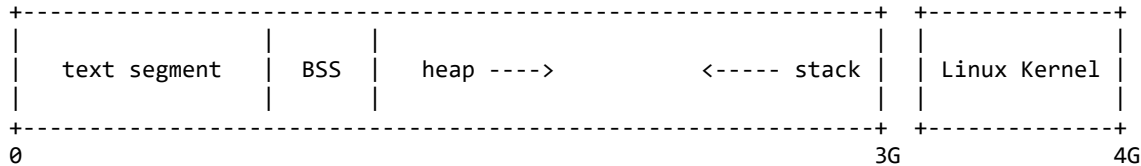
with a look forward to virtual addressing

by: burt rosenberg
 at: university of miami
 created: sept 2015
 Revised:

9 sept 2020 to add assembly output of demo .c program.
 (revised from [2015 version](#))
 (revised from [2012 version](#))

User memory is laid out in virtual address space from address 0 up to some maximum value that depends on a few things. To make things simple, a 32-bit machine running a traditional 32-bit linux kernel will have maximum user space address of 3-Gigabyte.

The space is laid out typically with the program in the lowest addresses, followed by pre-initialized data, followed by a variable sized heap segment that grows up, then starting at the top of the space, a stack segment that grows down.



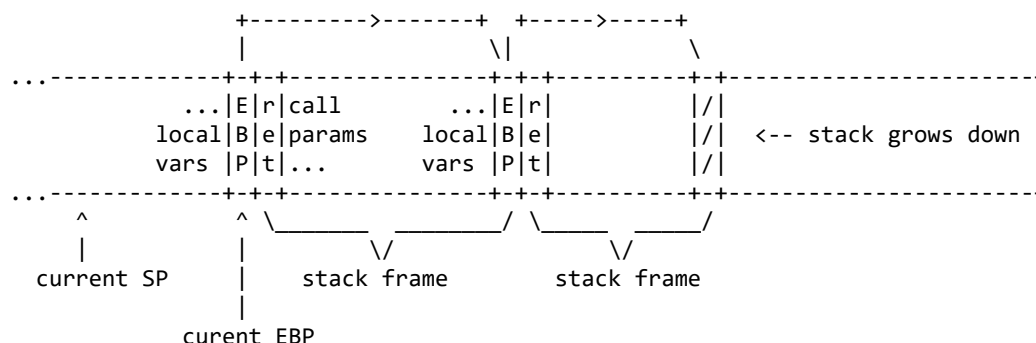
Most of the memory used during a program run is on the stack. The heap is for malloc's. The BSS would just be for a few variables like globals, and constant strings. The parameters to function calls (including argc and argv), and the local variables in scope of a function are all on the stack.

The stack is arranged in frames, with each frame associated with the instance of a function call. In IA-32 (x-86) architectures on linux, and in windows as well, a function call such as f(a,b,c) will push onto the stack the values of a, b and c, and then in calling f push onto the stack the address of the the instruction following f. This is called the return address, and is where code flow resumes when the called function executes the return.

The called function creates a new stack frame for the running of the function. A stack frame is identified by a memory address in a register called the base-pointer, EBP (this is specific to IA-32 architecture, but other CPU's might have a similar register). The EBP contains the base of the current stack frame, that is, the highest numbered address in the stack frame. All references to local variables in a function are not made by the name of the variable, but by a numerical offset to the EBP. The compiler will layout the stack frame and replace a local variable name, say i, with the offset -4, if it so happens that the variable i is stored at a location 4 bytes below the top of the stack frame.

To create a new stack frame, the called program pushes the old EBP onto the stack and then loads the EBP with the current stack point, ESP. It then drops the stack pointer by the number of bytes needed in the stack frame, which is determined by the compiler, and how the compiler has laid out and allocated the local variables.

The storing of the old EBP at the top of the current stack frame creates a linked list of stack frames, with the first element of the stack frame being the pointer to the previous stack frame. This allows a return to easily pop the stack frame, simply by reloading EBP with what the EBP points to.



Example

Here is a listing of a stack. Below is the program which created this stack. The stack frames are noted, along with the chain of stack frame pointers, the return addresses, the pushed parameters n, a, b, and c, and the local variables i, j and k.

```

addresses of subroutines:
  main= 0x8048506, f=0x80484ad, g=0x804844d

0xbff38538 0x00000000 end of chain of stack frames
0xbff38534 0x00000000
0xbff38530 0x08048560
0xbff3852c 0x00000007
0xbff38528 0x00000006
0xbff38524 0x00000005
0xbff38520 0x00000004
0xbff3851c 0x08048557 return address in main

0xbff38518 0xbff38538 next stack frame
0xbff38514 0x40021938
0xbff38510 0x00000001
0xbff3850c 0x0000000c int k
0xbff38508 0x0000000b int j
0xbff38504 0x0000000a int i
0xbff38500 0x401d7ac0
0xbff384fc 0x00000007 parameter c
0xbff384f8 0x00000006 parameter b
0xbff384f4 0x00000005 parameter a
0xbff384f0 0x00000003 parameter n
0xbff384ec 0x080484f1 return address in f

0xbff384e8 0xbff38518 next stack frame
0xbff384e4 0x400316c8
0xbff384e0 0xffffffff
0xbff384dc 0x0000000c int k
0xbff384d8 0x0000000b int j
0xbff384d4 0x0000000a int i
0xbff384d0 0x0804822c
0xbff384cc 0x00000007 parameter c
0xbff384c8 0x00000006 parameter b
0xbff384c4 0x00000005 parameter a
0xbff384c0 0x00000002 parameter n
0xbff384bc 0x080484f1 return address in f

0xbff384b8 0xbff384e8 next stack frame
0xbff384b4 0x00000001
0xbff384b0 0x00000001
0xbff384ac 0x0000000c int k
0xbff384a8 0x0000000b int j
0xbff384a4 0x0000000a int i
0xbff384a0 0xbff38500
0xbff3849c 0x00000007 parameter c
0xbff38498 0x00000006 parameter b
0xbff38494 0x00000005 parameter a
0xbff38490 0x00000001 parameter n
0xbff3848c 0x080484f1 return address in f

0xbff38488 0xbff384b8 next stack frame
0xbff38484 0x40021a94
0xbff38480 0xbff384e8
0xbff3847c 0x0000000c int k
0xbff38478 0x0000000b int j
0xbff38474 0x0000000a int i
0xbff38470 0xbff38530
0xbff3846c 0x00000007 parameter c
0xbff38468 0x00000006 parameter b

```



```

0xbff38464 0x00000005 parameter a
0xbff38460 0x00000000 parameter n
0xbff3845c 0x080484f1 return address in f

0xbff38458 0xbff38488 next stack frame
0xbff38454 0x00000001
0xbff38450 0xffffffff
0xbff3844c 0x0000000c int k
0xbff38448 0x0000000b int j
0xbff38444 0x0000000a int i
0xbff38440 0x40021938
0xbff3843c 0x0804824b
0xbff38438 0xbff38450
0xbff38434 0xbff38458
0xbff38430 0x00000064 parameter d
0xbff3842c 0x080484ff return address in f (following call to g)

0xbff38428 0xbff38458 next stack frame
0xbff38424 0x40021af0
0xbff38420 0x4002155c
0xbff3841c 0xbff3841c int * ip
0xbff38418 0x00000000 int i

```

done

```

#include<stdio.h>
#include<stdlib.h>

```

```

/*
* stack frame demonstration program
* author: bjr
* created: 11 Sept 2014
* lastupdate:
*
*/

```

```

int g(int d) {
    int i = d ;
    int * ip = &i ;
    ip += d ;
    for ( ; i>=0; i-- ) {
        printf("%p 0x%08x\n", ip, *ip) ;
        ip -= 1 ;
    }
    printf("done\n") ;
    return 0 ;
}

```

```

int f(int n ,int a, int b, int c) {
    int i = 0xa ;
    int j = 0xb ;
    int k = 0xc ;

    if ( n ) f(n-1, a, b, c ) ;
    else g(100) ;
    return 0 ;
}

```

```

int main(int argc, char * argv[]) {
    printf("addresses of subroutines:\n main= %p, f=%p, g=%p\n\n",

```



```

        main, f, g ) ;
    f(4,5,6,7) ;
    return 0 ;
}

```

// here is it in a binary compile: cc -S stack-demo.c, the .s file produced.

```

.file "stack-demo.c"
.section .rodata
.LC0:
.string "%p 0x%08x\n"
.LC1:
.string "done"
.text
.globl g
.type g, @function
g:
.LFB2:
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
subl $24, %esp
movl %gs:20, %eax
movl %eax, -12(%ebp)
xorl %eax, %eax
movl 8(%ebp), %eax
movl %eax, -20(%ebp)
leal -20(%ebp), %eax
movl %eax, -16(%ebp)
movl 8(%ebp), %eax
sall $2, %eax
addl %eax, -16(%ebp)
jmp .L2
.L3:
movl -16(%ebp), %eax
movl (%eax), %eax
subl $4, %esp
pushl %eax
pushl -16(%ebp)
pushl $.LC0
call printf
addl $16, %esp
subl $4, -16(%ebp)
movl -20(%ebp), %eax
subl $1, %eax
movl %eax, -20(%ebp)
.L2:
movl -20(%ebp), %eax
testl %eax, %eax
jns .L3
subl $12, %esp
pushl $.LC1
call puts
addl $16, %esp
movl $0, %eax
movl -12(%ebp), %edx
xorl %gs:20, %edx
je .L5
call __stack_chk_fail
.L5:
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret

```



```

.cfi_endproc
.LFE2:
.size    g, .-g
.globl   f
.type    f, @function
f:
.LFB3:
.cfi_startproc
pushl   %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl    %esp, %ebp
.cfi_def_cfa_register 5
subl    $24, %esp
movl    $10, -20(%ebp)
movl    $11, -16(%ebp)
movl    $12, -12(%ebp)
cmpl   $0, 8(%ebp)
je      .L7
movl    8(%ebp), %eax
subl    $1, %eax
pushl   20(%ebp)
pushl   16(%ebp)
pushl   12(%ebp)
pushl   %eax
call    f
addl    $16, %esp
jmp     .L8
.L7:
subl    $12, %esp
pushl   $100
call    g
addl    $16, %esp
.L8:
movl    $0, %eax
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE3:
.size    f, .-f
.section .rodata
.align 4
.LC2:
.string "addresses of subroutines:\n  main= %p, f=%p, g=%p\n\n"
.text
.globl   main
.type    main, @function
main:
.LFB4:
.cfi_startproc
leal    4(%esp), %ecx
.cfi_def_cfa 1, 0
andl    $-16, %esp
pushl   -4(%ecx)
pushl   %ebp
.cfi_escape 0x10,0x5,0x2,0x75,0
movl    %esp, %ebp
pushl   %ecx
.cfi_escape 0xf,0x3,0x75,0x7c,0x6
subl    $4, %esp
pushl   $g
pushl   $f
pushl   $main
pushl   $.LC2
call    printf
addl    $16, %esp

```



```

pushl   $7
pushl   $6
pushl   $5
pushl   $4
call    f
addl    $16, %esp
movl    $0, %eax
movl    -4(%ebp), %ecx
.cfi_def_cfa 1, 0
leave
.cfi_restore 5
leal    -4(%ecx), %esp
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE4:
.size   main, .-main
.ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609"
.section .note.GNU-stack,"",@progbits

```

A look forward to virtual memory

The memory layout described about might worry the reader. It seems that since all 4G of the memory is devoted to a single user program, how does the computer support multiple user's simultaneous use of memory?

One solution is called swapping. Swapping isn't used in this form anymore, but let me explain it this way, since it will work it will just be slow. To change from user program to user program, the entire 3G of memory is copied out to disk. That is called a swap out. Then a previously swapped out image is located on the disk and it is copied into memory, and the program picks up where it left off. This is called a swap in.

Another solution, what was used years ago, is to partition the memory, so that each running program was only allowed use a certain range of addresses. This was very inconvenient because the program needed to be recompiled so that all the instruction addresses and data addresses fit in a range of addresses that wouldn't be know until just before the program ran. Besides, 3G is not a lot, and to start sharing it between all running programs is a problem in itself.

These days, the problem is solved by virtual memory. The addresses that the CPU sees are virtual, they do not exactly correspond to the physical address where the data will be stored in physical memory. The operating system sets up a correspondance between the virtual addresses and available physical memory. The memory fetch and store logic is constantly translating between virtual and physical addresses, according to tables maintained by the operating system.

For more information see [Memory Management on Wiki](#).



This work is licensed under a [Creative Commons Attribution-ShareAlike 3.0 Unported License](#).

Materials for Topic 1: Intro to the Linux/UNIX OS Programming

Full C Programs

- [helloworld.c](#) - a C program that prints a short message to the terminal.
- [errno.c](#) - a C program that shows how to detect and print error messages to the terminal in case a function call or a system call fails.

Runnable Linux Commands

Quick Links:

- [gcc](#)
 - [./short_prompt](#)
 - [./long_prompt](#)
 - [ls -i](#)
 - [ls -i *](#)
 - [ln](#)
 - [ln -s](#)
-

- The command:

```
gcc -Wall -Wextra -O2 -g -o program program.c
```

compiles the C source code located inside the file `program.c` and created an executable (= binary) file called

program. The command `gcc` invokes the GNU C Compiler. Above, if we were to omit `"-o program"` from the command ("`o`" is the lowercase "Ohh" letter,) the executable's name would be the generic name `"a.out"`. The command options `"-Wall"` and `"-Wextra"` let the compiler display all the basic and extra possible compilation warnings to alert the user that a certain part of the program might not execute as intended in case of a warning showing up after running the command. The option `"-O2"` ("`O`" is the uppercase "Ohh" letter) sets the optimization level to 2 (= significant but sane) out of 5 possible optimization levels. Finally, the option `"-g"` allows the user to run the C debugger program, `gdb`, after the compilation in case debugging or program tracing is needed.

Example: The command `gcc -Wall -Wextra -O2 -g -o hello helloworld.c` will compile the source code file `helloworld.c` and produce a binary file named `hello` that you can run on the computer. You would run it in the following way: you would type `./hello` into the terminal and then press ENTER/RETURN for the program to run. Programs might prints messages to the terminal, such as `Hello World!`.

- The command:

```
./short_prompt
```

executes code inside a file named `short_prompt` and sources it (lets the changes indicated by the code take

effect by the command line interpreter.)

On my device, I put the following 2 lines of code into this file:

```
PS1="\[\033[0;32m\]\$ \[\033[0;37m\]"
```

```
echo "Prompt Updated!"
```

This code changes the terminal prompt to only: "\$ ", so that longer commands that we type in the terminal will most likely fit into a single line on the screen.

Once you create this file, make it executable by running the following command in the terminal:

```
chmod u+x .short_prompt
```

Finally, you would run this command by either typing:

```
./short_prompt
```

or

```
source ./short_prompt
```

into the terminal. Otherwise, if you just type:

`./short_prompt`, the change indicated by the code (changing the length of the command prompt) won't take effect in the terminal, and you'll keep seeing your old prompt.

The change made by the code in `.short_prompt` is good only for the duration of the current login session into the terminal; once you log out and log back in, you'll see your original, old prompt as before the change by

`.short_prompt`.

- The command:

```
. ./long_prompt
```

behaves similarly to `./short_prompt`, except that you can put code into `long_prompt` that will make the prompt 'fancier' than just "\$ ". On my device, `long_prompt` contains the following code:

```
magenta=$(tput setaf 5)
```

```
green=$(tput setaf 2)
```

```
reset=$(tput setaf 7) # White color
```

```
PS1='\[$magenta\][\D{%a %d/%m/%Y, %H:%M:%S}]\[$green\]
```

```
\u@\h:\w\$ \[$reset\]'
```

```
echo "Prompt Updated!"
```

- The command:

```
ls -i
```

shows the i-node number of every file and folder inside the current folder.

- The command:

```
ls -i *
```

shows the i-node number of every file inside the current folder, including those files within nested folders.

- The command:

```
ln file1 hardlinkForFile1
```

creates a hard link for `file1`. The name of the hard link is `hardlinkForFile1`. Once this command executes, two files will reside inside the current directory: `file1` and `hardlinkForFile1` (besides all the other files in the



directory.) These two files point at the exact same file. If the contents of the file are changed, this change will be reflected once you access either of `file1` or `hardlinkForFile1`.

Example: The command `ln message.txt linkMessage.txt` will create a hard link for `message.txt` called `linkMessage.txt`. Both files point at the exact same content: only one file is stored on the disk, but there are two ways of accessing its content (either by opening `message.txt`, or by opening `linkMessage.txt`.)

- The command:

```
ln -s file1 softlinkForFile1
```

creates a soft (= symbolic) link for `file1`. The name of the link is `softlinkForFile1`.

One difference between a hard and a soft link is that, when the file that has a soft link is deleted, the soft link still remains on the computer, and turns into what we usually call a 'broken link'. Conversely, if you delete a file that has a hard link, both the file and the hard link are deleted from the device.



This website by [Miriam Briskman](#) is licensed under [CC BY-NC 4.0](#).



```
1  /* Simple C program to display "Hello, World!"
2  *
3  *   Miriam Briskman, 1/6/2024
4  *   CISC 3350, Brooklyn College
5  *   Licensed under CC BY-NC 4.0
6  */
7
8  // A one-line C comment starts with: //
9  // Just like this line and the line above!
10
11 // A multiline C comment starts with /*
12 //   and ends with */ like:
13 /*
14     This
15     comment!
16 */
17 // Comments are skipped by the compiler,
18 //   so only non-comment code is executed.
19
20 // Include a reference to C's standard
21 //   input & output library without which no
22 //   messages can be printed to the screen.
23 #include <stdio.h>
24
25 // main function: the execution of program
26 //   begins here:
27 int main()
28 {
29     // Print "Hello, World!" followed by
30     //   a newline character:
31     printf ("Hello, World!\n");
32
33     // Exit the program with 0 as a message,
34     //   showing that the execution was
35     //   successful:
36     return 0;
37 }
38
```



```
1  /* A C program that shows how to detect and print
2  *   error messages in various ways.
3  *
4  *   We will attempt to issue a system call such
5  *   that it fails, intentionally, to test the
6  *   methods of C to detect errors, including
7  *   (1) checking the returned value of a system
8  *   call, and (2) checking the value of the
9  *   'errno' variable that C uses to indicate
10 *   what specific error happened.
11 *
12 *   Miriam Briskman, 2/8/2023
13 *   CISC 3350, Brooklyn College
14 *   Licensed under CC BY-NC 4.0
15 */
16
17
18 // Include a reference to C's standard input &
19 //   output library without which no messages
20 //   can be printed to the screen:
21 #include <stdio.h>
22 // Include the definition of the 'errno' variable
23 //   that is set every time an error occurs in a
24 //   function call:
25 #include <errno.h>
26 // Include the library that defines the close()
27 //   system call:
28 #include <unistd.h>
29 // The String library is needed to print error
30 //   messages, for example by using strerror().
31 #include <string.h>
32
33
34 // main function:
35 int main()
36 {
37     // Print the value of 'errno', which is
38     //   defined in the <errno.h> library that
39     //   was included above, before making any
40     //   system calls:
41     printf ("At the start of the program,"
42            " errno is: %d.\n", errno);
```



```
43 printf ("Now, we will attempt a faulty"
44         " call to close().\n");
45
46 // Setting 'errno' to 0 just in case:
47 errno = 0;
48
49 // Now, we attempt to call the close()
50 // system call, which is a function
51 // used to close an open file.
52 // However, to make it fail on purpose,
53 // we try to close a file that was
54 // never open. According to the manual
55 // page on close(), the errno variable
56 // can be set in a number of ways,
57 // including when the integer file
58 // descriptor that is passed to close()
59 // isn't a valid open file descriptor.
60 // The reason we decided to save these
61 // values in variable is that calls to
62 // perror() and strerror() might set
63 // errno to different values at any time.
64
65 // A file descriptor can only be 0, 1, 2, 3, ...
66 int return_value = close(-1);
67 int copy_of_errno = errno;
68
69 // Since the descriptor -1 above can never
70 // represent an open file, we expect that
71 // (1) the value of 'return_value' is -1
72 // since this is the value close()
73 // returns when it fails, and (2) errno
74 // was set within close() to a non-zero
75 // integer, which indicates an error.
76
77 // Now, we try printing a human-understandable
78 // statement about the error that happened.
79 // The message that should be printed to the
80 // terminal is of the form: "close: such and
81 // such error happened". The name 'perror'
82 // means 'print error'.
83 if (return_value == -1)
84     perror ("close");
85
```




```
86 // We also print return_value and
87 // copy_of_errno that we saved earlier:
88 printf ("After the call to close(),"
89         " the returned value by close() was: %d,"
90         " and errno was set to: %d.\n",
91         return_value, copy_of_errno);
92
93 // We also demonstrate how to print the error
94 // message using the strerror() function,
95 // which returns a pointer to the error
96 // string that perror() would print, but
97 // without printing it. strerror() accepts
98 // the errno value to be able to return
99 // the pointer.
100 char * error_string = strerror (copy_of_errno);
101
102 // Now, we print the string the pointer to
103 // which we obtained:
104 printf ("The error message that strerror()"
105         " returned is: %s.\n", error_string);
106
107 // For the sake of completion, the use of the
108 // thread-safe function strerror_r() is
109 // demonstrated:
110
111 // Creation of a 'buf' array of size 1024 bytes:
112 char buf[1024];
113 int strerror_r_returned_value
114     = strerror_r (copy_of_errno, buf, sizeof(buf));
115 printf ("The error message that strerror_r()"
116         " returned is: %s.\n", buf);
117 printf ("strerror_r() itself returned with"
118         " code %d.\n", strerror_r_returned_value);
119
120 // Just as a check, we show what the three
121 // functions return when there is no error
122 // (when errno remains zero:)
123 printf ("\nWe now check what perror(), strerror(),
124         " and strerror_r() return when"
125         " there is no error:\n");
126 errno = 0;
127 perror ("");
128 printf ("The message that strerror()"
```




```
129         " returned is: %s.\n", strerror (0));
130
131 // Another array for strerror_r() to use:
132 char buf2[1024];
133 strerror_r_returned_value
134     = strerror_r (0, buf2, sizeof(buf2));
135 printf ("The message that strerror_r()"
136         " returned is: %s.\n", buf2);
137 printf ("strerror_r() itself returned with"
138         " code %d.\n", strerror_r_returned_value);
139
140 // Exit the program with 0 as a message,
141 //     showing that the execution was successful.
142 return 0;
143 }
144
```



Topic 2: Useful Linux Commands and Text Editors

Note: A copy of each one of the sources listed below is included in this packet. To jump to the location of a source copy, click the page link under “Page”.

In case a source isn’t included (e.g., a YouTube video,) click the  (external link) symbol under “Page” to open the external source in a browser window.

| # | Citation & Source Link | Page |
|----|--|------|
| 1 | Meyering, Jim. “pwd(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/pwd.1.html | 165 |
| 2 | Mlynarik, Richard. “whoami(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/whoami.1.html | 168 |
| 3 | Stallman, Richard M. and MacKenzie, David. “ls(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/ls.1.html | 171 |
| 4 | “cd(1p) - Linux manual page”, <i>man7.org</i> , 2017. URL: https://man7.org/linux/man-pages/man1/cd.1p.html | 179 |
| 5 | MacKenzie, David. “mkdir(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/mkdir.1.html | 188 |
| 6 | Rubin, Paul, et al. “touch(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/touch.1.html | 191 |
| 7 | MacKenzie, David. “rmdir(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/rmdir.1.html | 195 |
| 8 | Rubin, Paul, et al. “rm(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/rm.1.html | 198 |
| 9 | Granlund, Torbjorn and Stallman, Richard M. “cat(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/cat.1.html | 202 |
| 10 | “clear(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/clear.1.html | 205 |
| 11 | Granlund, Torbjorn, et al. “cp(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/cp.1.html | 209 |
| 12 | Parker, Mike, et al. “mv(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/mv.1.html | 215 |
| 13 | Miller, Todd C. “sudo(8) - Linux manual page”, <i>man7.org</i> , 9 Aug. 2023. URL: https://man7.org/linux/man-pages/man8/sudo.8.html | 219 |
| 14 | MacKenzie, David. “su(1) - Linux manual page”, <i>man7.org</i> , 19 Jul. 2023. URL: https://man7.org/linux/man-pages/man1/su.1.html | 240 |
| 15 | Eaton, John W. “man(1) - Linux manual page”, <i>man7.org</i> , 23 Sep. 2023. URL: https://man7.org/linux/man-pages/man1/man.1.html | 247 |

| # | Citation & Source Link | Page |
|----|---|------|
| 16 | “info(1) - Linux manual page”, die.net, 2008. URL: https://linux.die.net/man/1/info | 267 |
| 17 | “ping(8) - Linux manual page”, <i>man7.org</i> , 26 Nov. 2022. URL: https://man7.org/linux/man-pages/man8/ping.8.html | 270 |
| 18 | Nikšić, Hrvoje. “wget(1) - Linux manual page”, <i>man7.org</i> , 22 Dec. 2023. URL: https://man7.org/linux/man-pages/man1/wget.1.html | 283 |
| 19 | “grep(1) - Linux manual page”, <i>man7.org</i> , 29 Dec. 2019. URL: https://man7.org/linux/man-pages/man1/grep.1.html | 330 |
| 20 | Anglin, John David, et al. “gcc(1) - Linux manual page”, <i>man7.org</i> , 27 May. 2022. URL: https://man7.org/linux/man-pages/man1/gcc.1.html | 348 |
| 21 | “gdb(1) - Linux manual page”, <i>man7.org</i> , 3 Dec. 2023. URL: https://man7.org/linux/man-pages/man1/gdb.1.html | 352 |
| 22 | “exit(1p) - Linux manual page”, <i>man7.org</i> , 2017. URL: https://man7.org/linux/man-pages/man1/exit.1p.html | 361 |
| 23 | SavvyNik, director. <i>Linux Absolute Basics Terminal Tutorial — What/How to Use</i> . YouTube, 29 Apr. 2020. URL: https://www.youtube.com/watch?v=Us3G-nJYru0 | ↗ |
| 24 | tutoriaLinux, director. <i>Vim Basics in 8 Minutes</i> . YouTube, 5 Oct. 2018. URL: https://www.youtube.com/watch?v=ggSyF1SVFr4 | ↗ |
| 25 | DistroTube, director. <i>The Basics of Emacs as a Text Editor</i> . YouTube, 3 Dec. 2019. URL: https://www.youtube.com/watch?v=jPkIaqSh3cA | ↗ |
| 26 | GradStudentTutorials, director. <i>Emacs Tutorial For Beginners - Simply Explained</i> . YouTube, 20 Jan. 2017. URL: https://www.youtube.com/watch?v=JdF1QZQqiTI | ↗ |
| 27 | SavvyNik, director. <i>Nano Text Editor Basics (pico) - How to Use Nano on Linux / Mac</i> . YouTube, 10 May. 2020. URL: https://www.youtube.com/watch?v=Jf0ZJZJ8j1I | ↗ |
| 28 | HackerSploit, director. <i>Nano Editor Fundamentals</i> . YouTube, 28 Jan. 2019. URL: https://www.youtube.com/watch?v=gyKiDczLIZ4 | ↗ |
| 29 | Briskman, Miriam. “Materials for Topic 2: Useful Linux Commands and Text Editors.” <i>Topic 2: Useful Linux Commands and Text Editors — CISC 3350 Materials</i> , 2023. URL: www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_02/ | 366 |

pwd(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [AUTHOR](#) | [REPORTING BUGS](#) | [COPYRIGHT](#) | [SEE ALSO](#) | [COLOPHON](#)

 PWD(1)

User Commands

PWD(1)**NAME** [top](#)

pwd - print name of current/working directory

SYNOPSIS [top](#)

pwd [*OPTION*]...

DESCRIPTION [top](#)

Print the full filename of the current working directory.

-L, --logical
use PWD from environment, even if it contains symlinks

-P, --physical
avoid all symlinks

--help display this help and exit

--version
output version information and exit

If no option is specified, **-P** is assumed.

NOTE: your shell may have its own version of pwd, which usually

supersedes the version described here. Please refer to your shell's documentation for details about the options it supports.

AUTHOR [top](#)

Written by Jim Meyering.

REPORTING BUGS [top](#)

GNU coreutils online help:
<<https://www.gnu.org/software/coreutils/>>
Report any translation bugs to
<<https://translationproject.org/team/>>

COPYRIGHT [top](#)

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>. This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO [top](#)

[getcwd\(3\)](#)

Full documentation <<https://www.gnu.org/software/coreutils/pwd>> or available locally via: info '(coreutils) pwd invocation'

COLOPHON [top](#)

This page is part of the *coreutils* (basic file, shell and text manipulation utilities) project. Information about the project can be found at <<http://www.gnu.org/software/coreutils/>>. If you have a bug report for this manual page, see <<http://www.gnu.org/software/coreutils/>>. This page was obtained from the tarball coreutils-9.4.tar.xz fetched from <<http://ftp.gnu.org/gnu/coreutils/>> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for



the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

GNU coreutils 9.4

August 2023

PWD(1)

Pages that refer to this page: [getcwd\(3\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



whoami(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [AUTHOR](#) | [REPORTING BUGS](#) | [COPYRIGHT](#) | [SEE ALSO](#) | [COLOPHON](#)

 WHOAMI(1)

User Commands

WHOAMI(1)**NAME** [top](#)

whoami - print effective user name

SYNOPSIS [top](#)

whoami [*OPTION*]...

DESCRIPTION [top](#)

Print the user name associated with the current effective user ID. Same as `id -un`.

--help display this help and exit

--version
output version information and exit

AUTHOR [top](#)

Written by Richard Mlynarik.

REPORTING BUGS [top](#)

GNU coreutils online help:
[<https://www.gnu.org/software/coreutils/>](https://www.gnu.org/software/coreutils/)

Report any translation bugs to
<<https://translationproject.org/team/>>

COPYRIGHT [top](#)

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>. This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO [top](#)

Full documentation
<<https://www.gnu.org/software/coreutils/whoami>>
or available locally via: info '(coreutils) whoami invocation'

COLOPHON [top](#)

This page is part of the *coreutils* (basic file, shell and text manipulation utilities) project. Information about the project can be found at <<http://www.gnu.org/software/coreutils/>>. If you have a bug report for this manual page, see <<http://www.gnu.org/software/coreutils/>>. This page was obtained from the tarball coreutils-9.4.tar.xz fetched from <<http://ftp.gnu.org/gnu/coreutils/>> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

GNU coreutils 9.4

August 2023

WHOAMI(1)

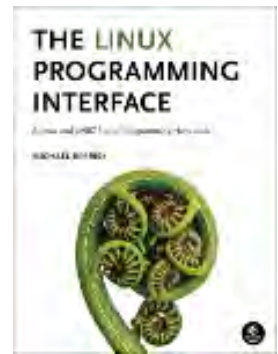
Pages that refer to this page: [seccomp\(2\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



ls(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [AUTHOR](#) | [REPORTING BUGS](#) | [COPYRIGHT](#) | [SEE ALSO](#) | [COLOPHON](#)

 LS(1)

User Commands

LS(1)**NAME** [top](#)

ls - list directory contents

SYNOPSIS [top](#)

ls [*OPTION*]... [*FILE*]...

DESCRIPTION [top](#)

List information about the FILES (the current directory by default). Sort entries alphabetically if none of **-cftuvSUX** nor **--sort** is specified.

Mandatory arguments to long options are mandatory for short options too.

-a, --all
do not ignore entries starting with **.**

-A, --almost-all
do not list implied **.** and **..**

--author
with **-l**, print the author of each file

-b, --escape

print C-style escapes for nongraphic characters

--block-size=SIZE

with **-l**, scale sizes by SIZE when printing them; e.g.,
'--block-size=M'; see SIZE format below

-B, --ignore-backups

do not list implied entries ending with ~

-c with **-lt**: sort by, and show, ctime (time of last change of file status information); with **-l**: show ctime and sort by name; otherwise: sort by ctime, newest first

-C list entries by columns

--color[=WHEN]

color the output WHEN; more info below

-d, --directory

list directories themselves, not their contents

-D, --dired

generate output designed for Emacs' dired mode

-f list all entries in directory order

-F, --classify[=WHEN]

append indicator (one of */=>@|) to entries WHEN

--file-type

likewise, except do not append '*'

--format=WORD

across **-x**, commas **-m**, horizontal **-x**, long **-l**,
single-column **-l**, verbose **-l**, vertical **-C**

--full-time

like **-l --time-style=full-iso**

-g like **-l**, but do not list owner

--group-directories-first

group directories before files; can be augmented with a
--sort option, but any use of **--sort=none (-U)** disables



grouping

- G, --no-group**
in a long listing, don't print group names
- h, --human-readable**
with **-l** and **-s**, print sizes like 1K 234M 2G etc.
- si** likewise, but use powers of 1000 not 1024
- H, --dereference-command-line**
follow symbolic links listed on the command line
- dereference-command-line-symlink-to-dir**
follow each command line symbolic link that points to a directory
- hide=PATTERN**
do not list implied entries matching shell PATTERN
(overridden by **-a** or **-A**)
- hyperlink[=WHEN]**
hyperlink file names WHEN
- indicator-style=WORD**
append indicator with style WORD to entry names: none
(default), slash (**-p**), file-type (**--file-type**), classify
(**-F**)
- i, --inode**
print the index number of each file
- I, --ignore=PATTERN**
do not list implied entries matching shell PATTERN
- k, --kibibytes**
default to 1024-byte blocks for file system usage; used
only with **-s** and per directory totals
- l** use a long listing format
- L, --dereference**
when showing file information for a symbolic link, show
information for the file the link references rather than

for the link itself

- m** fill width with a comma separated list of entries
- n, --numeric-uid-gid**
like **-l**, but list numeric user and group IDs
- N, --literal**
print entry names without quoting
- o** like **-l**, but do not list group information
- p, --indicator-style=*slash***
append / indicator to directories
- q, --hide-control-chars**
print ? instead of nongraphic characters
- show-control-chars**
show nongraphic characters as-is (the default, unless program is 'ls' and output is a terminal)
- Q, --quote-name**
enclose entry names in double quotes
- quoting-style=*WORD***
use quoting style *WORD* for entry names: literal, locale, shell, shell-always, shell-escape, shell-escape-always, c, escape (overrides QUOTING_STYLE environment variable)
- r, --reverse**
reverse order while sorting
- R, --recursive**
list subdirectories recursively
- s, --size**
print the allocated size of each file, in blocks
- S** sort by file size, largest first
- sort=*WORD***
sort by *WORD* instead of name: none (**-U**), size (**-S**), time (**-t**), version (**-v**), extension (**-X**), width



--time=WORD

select which timestamp used to display or sort; access time (-u): atime, access, use; metadata change time (-c): ctime, status; modified time (default): mtime, modification; birth time: birth, creation;

with **-l**, WORD determines which time to show; with **--sort=time**, sort by WORD (newest first)

--time-style=TIME_STYLE

time/date format with **-l**; see TIME_STYLE below

-t sort by time, newest first; see **--time**

-T, --tabsize=COLS

assume tab stops at each COLS instead of 8

-u with **-lt**: sort by, and show, access time; with **-l**: show access time and sort by name; otherwise: sort by access time, newest first

-U do not sort; list entries in directory order

-v natural sort of (version) numbers within text

-w, --width=COLS

set output width to COLS. 0 means no limit

-x list entries by lines instead of by columns

-X sort alphabetically by entry extension

-Z, --context

print any security context of each file

--zero end each output line with NUL, not newline

-1 list one file per line

--help display this help and exit

--version

output version information and exit



The `SIZE` argument is an integer and optional unit (example: `10K` is 10×1024). Units are `K`, `M`, `G`, `T`, `P`, `E`, `Z`, `Y`, `R`, `Q` (powers of 1024) or `KB`, `MB`, ... (powers of 1000). Binary prefixes can be used, too: `KiB`=`K`, `MiB`=`M`, and so on.

The `TIME_STYLE` argument can be `full-iso`, `long-iso`, `iso`, `locale`, or `+FORMAT`. `FORMAT` is interpreted like in [date\(1\)](#). If `FORMAT` is `FORMAT1`<newline>`FORMAT2`, then `FORMAT1` applies to non-recent files and `FORMAT2` to recent files. `TIME_STYLE` prefixed with `'posix-'` takes effect only outside the POSIX locale. Also the `TIME_STYLE` environment variable sets the default style to use.

The `WHEN` argument defaults to `'always'` and can also be `'auto'` or `'never'`.

Using color to distinguish file types is disabled both by default and with `--color=never`. With `--color=auto`, `ls` emits color codes only when standard output is connected to a terminal. The `LS_COLORS` environment variable can change the settings. Use the [dircolors\(1\)](#) command to set it.

Exit status:

- 0 if OK,
- 1 if minor problems (e.g., cannot access subdirectory),
- 2 if serious trouble (e.g., cannot access command-line argument).

AUTHOR [top](#)

Written by Richard M. Stallman and David MacKenzie.

REPORTING BUGS [top](#)

GNU coreutils online help:
<<https://www.gnu.org/software/coreutils/>>
Report any translation bugs to
<<https://translationproject.org/team/>>

COPYRIGHT [top](#)

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>. This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO [top](#)

[dircolors\(1\)](#)

Full documentation <<https://www.gnu.org/software/coreutils/ls>> or available locally via: info '(coreutils) ls invocation'

COLOPHON [top](#)

This page is part of the *coreutils* (basic file, shell and text manipulation utilities) project. Information about the project can be found at <<http://www.gnu.org/software/coreutils/>>. If you have a bug report for this manual page, see <<http://www.gnu.org/software/coreutils/>>. This page was obtained from the tarball *coreutils-9.4.tar.xz* fetched from <<http://ftp.gnu.org/gnu/coreutils/>> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

GNU coreutils 9.4

August 2023

LS(1)

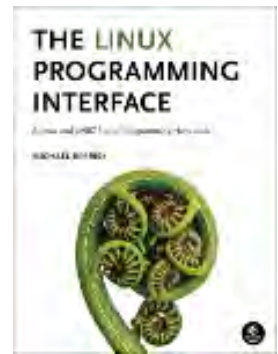
Pages that refer to this page: [column\(1\)](#), [find\(1\)](#), [namei\(1\)](#), [stat\(2\)](#), [statx\(2\)](#), [glob\(3\)](#), [strverscmp\(3\)](#), [core\(5\)](#), [dir_colors\(5\)](#), [passwd\(5\)](#), [proc\(5\)](#), [mq_overview\(7\)](#), [symlink\(7\)](#), [lsblk\(8\)](#), [lsuf\(8\)](#), [setfiles\(8\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



cd(1p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [OPERANDS](#) | [STDIN](#) | [INPUT FILES](#) | [ENVIRONMENT VARIABLES](#) | [ASYNCHRONOUS EVENTS](#) | [STDOUT](#) | [STDERR](#) | [OUTPUT FILES](#) | [EXTENDED DESCRIPTION](#) | [EXIT STATUS](#) | [CONSEQUENCES OF ERRORS](#) | [APPLICATION USAGE](#) | [EXAMPLES](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 CD(1P)

POSIX Programmer's Manual

CD(1P)**PROLOG**[top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME[top](#)

cd – change the working directory

SYNOPSIS[top](#)

```
cd [-L|-P] [directory]
```

```
cd -
```

DESCRIPTION[top](#)

The `cd` utility shall change the working directory of the current shell execution environment (see [Section 2.12, Shell Execution Environment](#)) by executing the following steps in sequence. (In the following steps, the symbol **curpath** represents an intermediate value used to simplify the description of the algorithm used by `cd`. There is no requirement that **curpath** be made visible to the application.)

1. If no *directory* operand is given and the `HOME` environment variable is empty or undefined, the default behavior is



implementation-defined and no further steps shall be taken.

2. If no *directory* operand is given and the *HOME* environment variable is set to a non-empty value, the *cd* utility shall behave as if the directory named in the *HOME* environment variable was specified as the *directory* operand.
3. If the *directory* operand begins with a <slash> character, set **curpath** to the operand and proceed to step 7.
4. If the first component of the *directory* operand is dot or dot-dot, proceed to step 6.
5. Starting with the first pathname in the <colon>-separated pathnames of *CDPATH* (see the ENVIRONMENT VARIABLES section) if the pathname is non-null, test if the concatenation of that pathname, a <slash> character if that pathname did not end with a <slash> character, and the *directory* operand names a directory. If the pathname is null, test if the concatenation of dot, a <slash> character, and the operand names a directory. In either case, if the resulting string names an existing directory, set **curpath** to that string and proceed to step 7. Otherwise, repeat this step with the next pathname in *CDPATH* until all pathnames have been tested.
6. Set **curpath** to the *directory* operand.
7. If the **-P** option is in effect, proceed to step 10. If **curpath** does not begin with a <slash> character, set **curpath** to the string formed by the concatenation of the value of *PWD*, a <slash> character if the value of *PWD* did not end with a <slash> character, and **curpath**.
8. The **curpath** value shall then be converted to canonical form as follows, considering each component from beginning to end, in sequence:
 - a. Dot components and any <slash> characters that separate them from the next component shall be deleted.
 - b. For each dot-dot component, if there is a preceding component and it is neither root nor dot-dot, then:
 - i. If the preceding component does not refer (in the context of pathname resolution with symbolic links followed) to a directory, then the *cd* utility shall display an appropriate error message and no further steps shall be taken.



- ii. The preceding component, all `<slash>` characters separating the preceding component from dot-dot, dot-dot, and all `<slash>` characters separating dot-dot from the following component (if any) shall be deleted.
 - c. An implementation may further simplify **curpath** by removing any trailing `<slash>` characters that are not also leading `<slash>` characters, replacing multiple non-leading consecutive `<slash>` characters with a single `<slash>`, and replacing three or more leading `<slash>` characters with a single `<slash>`. If, as a result of this canonicalization, the **curpath** variable is null, no further steps shall be taken.
9. If **curpath** is longer than `{PATH_MAX}` bytes (including the terminating null) and the *directory* operand was not longer than `{PATH_MAX}` bytes (including the terminating null), then **curpath** shall be converted from an absolute pathname to an equivalent relative pathname if possible. This conversion shall always be considered possible if the value of *PWD*, with a trailing `<slash>` added if it does not already have one, is an initial substring of **curpath**. Whether or not it is considered possible under other circumstances is unspecified. Implementations may also apply this conversion if **curpath** is not longer than `{PATH_MAX}` bytes or the *directory* operand was longer than `{PATH_MAX}` bytes.
 10. The *cd* utility shall then perform actions equivalent to the *chdir()* function called with **curpath** as the *path* argument. If these actions fail for any reason, the *cd* utility shall display an appropriate error message and the remainder of this step shall not be executed. If the **-P** option is not in effect, the *PWD* environment variable shall be set to the value that **curpath** had on entry to step 9 (i.e., before conversion to a relative pathname). If the **-P** option is in effect, the *PWD* environment variable shall be set to the string that would be output by *pwd -P*. If there is insufficient permission on the new directory, or on any parent of that directory, to determine the current working directory, the value of the *PWD* environment variable is unspecified.

If, during the execution of the above steps, the *PWD* environment variable is set, the *OLDPWD* environment variable shall also be set to the value of the old working directory (that is the current working directory immediately prior to the call to *cd*).



OPTIONS [top](#)

The *cd* utility shall conform to the Base Definitions volume of POSIX.1-2017, *Section 12.2, Utility Syntax Guidelines*.

The following options shall be supported by the implementation:

- L** Handle the operand dot-dot logically; symbolic link components shall not be resolved before dot-dot components are processed (see steps 8. and 9. in the DESCRIPTION).
- P** Handle the operand dot-dot physically; symbolic link components shall be resolved before dot-dot components are processed (see step 7. in the DESCRIPTION).

If both **-L** and **-P** options are specified, the last of these options shall be used and all others ignored. If neither **-L** nor **-P** is specified, the operand shall be handled dot-dot logically; see the DESCRIPTION.

OPERANDS [top](#)

The following operands shall be supported:

directory An absolute or relative pathname of the directory that shall become the new working directory. The interpretation of a relative pathname by *cd* depends on the **-L** option and the *CDPATH* and *PWD* environment variables. If *directory* is an empty string, the results are unspecified.

- When a <hyphen-minus> is used as the operand, this shall be equivalent to the command:

```
cd "$OLDPWD" && pwd
```

which changes to the previous working directory and then writes its name.

STDIN [top](#)

Not used.

INPUT FILES [top](#)

None.

ENVIRONMENT VARIABLES [top](#)

The following environment variables shall affect the execution of *cd*:

CDPATH A <colon>-separated list of pathnames that refer to directories. The *cd* utility shall use this list in its attempt to change the directory, as described in the DESCRIPTION. An empty string in place of a directory pathname represents the current directory. If *CDPATH* is not set, it shall be treated as if it were an empty string.

HOME The name of the directory, used when no *directory* operand is specified.

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of POSIX.1-2017, [Section 8.2, Internationalization Variables](#) for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).

LC_MESSAGES Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH Determine the location of message catalogs for the processing of *LC_MESSAGES*.

OLDPWD A pathname of the previous working directory, used by *cd -*.

PWD This variable shall be set as specified in the DESCRIPTION. If an application sets or unsets the value of *PWD*, the behavior of *cd* is unspecified.



ASYNCHRONOUS EVENTS [top](#)

Default.

STDOUT [top](#)

If a non-empty directory name from *CDPATH* is used, or if *cd -* is used, an absolute pathname of the new working directory shall be written to the standard output as follows:

```
"%s\n", <new directory>
```

Otherwise, there shall be no output.

STDERR [top](#)

The standard error shall be used only for diagnostic messages.

OUTPUT FILES [top](#)

None.

EXTENDED DESCRIPTION [top](#)

None.

EXIT STATUS [top](#)

The following exit values shall be returned:

- 0 The directory was successfully changed.
- >0 An error occurred.

CONSEQUENCES OF ERRORS [top](#)

The working directory shall remain unchanged.

The following sections are informative.

APPLICATION USAGE [top](#)



Since `cd` affects the current shell execution environment, it is always provided as a shell regular built-in. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(cd /tmp)
nohup cd
find . -exec cd {} \;
```

it does not affect the working directory of the caller's environment.

The user must have execute (search) permission in *directory* in order to change to it.

EXAMPLES [top](#)

The following template can be used to perform processing in the directory specified by *location* and end up in the current working directory in use before the first `cd` command was issued:

```
cd location
if [ $? -ne 0 ]
then
    print error message
    exit 1
fi
... do whatever is desired as long as the OLDPWD environment variable
    is not modified
cd -
```

RATIONALE [top](#)

The use of the `CDPATH` was introduced in the System V shell. Its use is analogous to the use of the `PATH` variable in the shell. The BSD C shell used a shell parameter `cdpath` for this purpose.

A common extension when `HOME` is undefined is to get the login directory from the user database for the invoking user. This does not occur on System V implementations.

Some historical shells, such as the KornShell, took special actions when the directory name contained a dot-dot component, selecting the logical parent of the directory, rather than the actual parent directory; that is, it moved up one level toward the `'/'` in the pathname, remembering what the user typed, rather than performing the equivalent of:

```
chdir("../");
```

In such a shell, the following commands would not necessarily produce equivalent output for all directories:

```
cd .. && ls      ls ..
```

This behavior is now the default. It is not consistent with the definition of dot-dot in most historical practice; that is, while this behavior has been optionally available in the KornShell, other shells have historically not supported this functionality. The logical pathname is stored in the *PWD* environment variable when the *cd* utility completes and this value is used to construct the next directory name if *cd* is invoked with the *-L* option.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

Section 2.12, Shell Execution Environment, [pwd\(1p\)](#)

The Base Definitions volume of POSIX.1-2017, *Chapter 8, Environment Variables*, *Section 12.2, Utility Syntax Guidelines*

The System Interfaces volume of POSIX.1-2017, [chdir\(3p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .



IEEE/The Open Group

2017

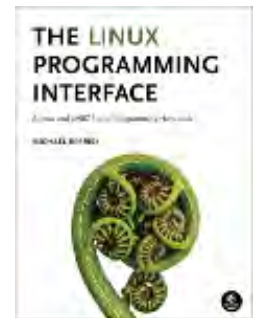
CD(1P)

Pages that refer to this page: [pwd\(1p\)](#), [sh\(1p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



mkdir(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [AUTHOR](#) | [REPORTING BUGS](#) | [COPYRIGHT](#) | [SEE ALSO](#) | [COLOPHON](#)

 MKDIR(1)

User Commands

MKDIR(1)**NAME** [top](#)

mkdir - make directories

SYNOPSIS [top](#)

mkdir [*OPTION*]... *DIRECTORY*...

DESCRIPTION [top](#)

Create the *DIRECTORY*(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.

- m, --mode=*MODE***
set file mode (as in `chmod`), not `a=rwx - umask`
- p, --parents**
no error if existing, make parent directories as needed, with their file modes unaffected by any **-m** option.
- v, --verbose**
print a message for each created directory
- Z** set SELinux security context of each created directory to the default type



- context**[=*CTX*]
like **-Z**, or if *CTX* is specified then set the SELinux or SMACK security context to *CTX*
- help** display this help and exit
- version**
output version information and exit

AUTHOR [top](#)

Written by David MacKenzie.

REPORTING BUGS [top](#)

GNU coreutils online help:
<<https://www.gnu.org/software/coreutils/>>
Report any translation bugs to
<<https://translationproject.org/team/>>

COPYRIGHT [top](#)

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>. This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO [top](#)

[mkdir\(2\)](#)

Full documentation <<https://www.gnu.org/software/coreutils/mkdir>> or available locally via: info '(coreutils) mkdir invocation'

COLOPHON [top](#)

This page is part of the *coreutils* (basic file, shell and text manipulation utilities) project. Information about the project can be found at <<http://www.gnu.org/software/coreutils/>>. If you

have a bug report for this manual page, see <http://www.gnu.org/software/coreutils/>. This page was obtained from the tarball coreutils-9.4.tar.xz fetched from <http://ftp.gnu.org/gnu/coreutils/> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

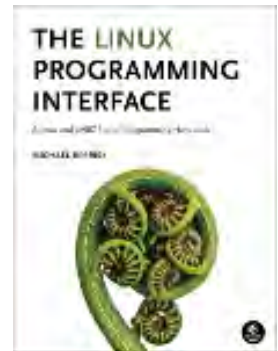
GNU coreutils 9.4**August 2023****MKDIR(1)**

Pages that refer to this page: [systemd-mount\(1\)](#), [mkdir\(2\)](#), [cpuset\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



touch(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [DATE STRING](#) | [AUTHOR](#) |
[REPORTING BUGS](#) | [COPYRIGHT](#) | [SEE ALSO](#) | [COLOPHON](#)

TOUCH(1)

User Commands

TOUCH(1)**NAME** [top](#)

touch - change file timestamps

SYNOPSIS [top](#)

touch [*OPTION*]... *FILE*...

DESCRIPTION [top](#)

Update the access and modification times of each FILE to the current time.

A FILE argument that does not exist is created empty, unless **-c** or **-h** is supplied.

A FILE argument string of **-** is handled specially and causes touch to change the times of the file associated with standard output.

Mandatory arguments to long options are mandatory for short options too.

-a change only the access time

-c, --no-create
do not create any files

- d, --date=STRING**
parse STRING and use it instead of current time
- f** (ignored)
- h, --no-dereference**
affect each symbolic link instead of any referenced file (useful only on systems that can change the timestamps of a symlink)
- m** change only the modification time
- r, --reference=FILE**
use this file's times instead of current time
- t STAMP**
use [[CC]YY]MMDDhhmm[.ss] instead of current time
- time=WORD**
change the specified time: WORD is access, atime, or use: equivalent to **-a** WORD is modify or mtime: equivalent to **-m**
- help** display this help and exit
- version**
output version information and exit

Note that the **-d** and **-t** options accept different time-date formats.

DATE STRING [top](#)

The **--date=STRING** is a mostly free format human readable date string such as "Sun, 29 Feb 2004 16:21:42 -0800" or "2004-02-29 16:21:42" or even "next Thursday". A date string may contain items indicating calendar date, time of day, time zone, day of week, relative time, relative date, and numbers. An empty string indicates the beginning of the day. The date string format is more complex than is easily documented here but is fully described in the info documentation.

AUTHOR [top](#)

Written by Paul Rubin, Arnold Robbins, Jim Kingdon, David MacKenzie, and Randy Smith.

REPORTING BUGS [top](#)

GNU coreutils online help:
<<https://www.gnu.org/software/coreutils/>>
Report any translation bugs to
<<https://translationproject.org/team/>>

COPYRIGHT [top](#)

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>. This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO [top](#)

Full documentation <<https://www.gnu.org/software/coreutils/touch>> or available locally via: info '(coreutils) touch invocation'

COLOPHON [top](#)

This page is part of the *coreutils* (basic file, shell and text manipulation utilities) project. Information about the project can be found at <<http://www.gnu.org/software/coreutils/>>. If you have a bug report for this manual page, see <<http://www.gnu.org/software/coreutils/>>. This page was obtained from the tarball coreutils-9.4.tar.xz fetched from <<http://ftp.gnu.org/gnu/coreutils/>> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org



Pages that refer to this page: [last\(1@@util-linux\)](#), [utime\(2\)](#), [utimensat\(2\)](#), [systemd-update-done.service\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



rmdir(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [AUTHOR](#) | [REPORTING BUGS](#) | [COPYRIGHT](#) | [SEE ALSO](#) | [COLOPHON](#)

RMDIR(1)

User Commands

RMDIR(1)**NAME** [top](#)

rmdir - remove empty directories

SYNOPSIS [top](#)

rmdir [*OPTION*]... *DIRECTORY*...

DESCRIPTION [top](#)

Remove the *DIRECTORY*(ies), if they are empty.

--ignore-fail-on-non-empty

ignore each failure to remove a non-empty directory

-p, --parents

remove *DIRECTORY* and its ancestors; e.g., 'rmdir -p a/b' is similar to 'rmdir a/b a'

-v, --verbose

output a diagnostic for every directory processed

--help display this help and exit

--version

output version information and exit



AUTHOR [top](#)

Written by David MacKenzie.

REPORTING BUGS [top](#)

GNU coreutils online help:

<<https://www.gnu.org/software/coreutils/>>

Report any translation bugs to

<<https://translationproject.org/team/>>

COPYRIGHT [top](#)

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>. This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO [top](#)

[rmdir\(2\)](#)

Full documentation <<https://www.gnu.org/software/coreutils/rmdir>> or available locally via: info '(coreutils) rmdir invocation'

COLOPHON [top](#)

This page is part of the *coreutils* (basic file, shell and text manipulation utilities) project. Information about the project can be found at <<http://www.gnu.org/software/coreutils/>>. If you have a bug report for this manual page, see <<http://www.gnu.org/software/coreutils/>>. This page was obtained from the tarball coreutils-9.4.tar.xz fetched from <<http://ftp.gnu.org/gnu/coreutils/>> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org



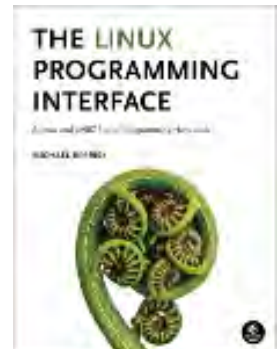
GNU coreutils 9.4**August 2023*****RMDIR(1)***

Pages that refer to this page: [rmdir\(2\)](#), [cpuset\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



rm(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [AUTHOR](#) | [REPORTING BUGS](#) | [COPYRIGHT](#) | [SEE ALSO](#) | [COLOPHON](#)

RM(1)

User Commands

RM(1)**NAME** [top](#)

rm - remove files or directories

SYNOPSIS [top](#)

rm [*OPTION*]... [*FILE*]...

DESCRIPTION [top](#)

This manual page documents the GNU version of **rm**. **rm** removes each specified file. By default, it does not remove directories.

If the *-I* or *--interactive=once* option is given, and there are more than three files or the *-r*, *-R*, or *--recursive* are given, then **rm** prompts the user for whether to proceed with the entire operation. If the response is not affirmative, the entire command is aborted.

Otherwise, if a file is unwritable, standard input is a terminal, and the *-f* or *--force* option is not given, or the *-i* or *--interactive=always* option is given, **rm** prompts the user for whether to remove the file. If the response is not affirmative, the file is skipped.

OPTIONS [top](#)

Remove (unlink) the FILE(s).

-f, --force

ignore nonexistent files and arguments, never prompt

-i prompt before every removal

-I prompt once before removing more than three files, or when removing recursively; less intrusive than **-i**, while still giving protection against most mistakes

--interactive[=WHEN]

prompt according to WHEN: never, once (**-I**), or always (**-i**); without WHEN, prompt always

--one-file-system

when removing a hierarchy recursively, skip any directory that is on a file system different from that of the corresponding command line argument

--no-preserve-root

do not treat '/' specially

--preserve-root[=all]

do not remove '/' (default); with 'all', reject any command line argument on a separate device from its parent

-r, -R, --recursive

remove directories and their contents recursively

-d, --dir

remove empty directories

-v, --verbose

explain what is being done

--help display this help and exit

--version

output version information and exit

By default, rm does not remove directories. Use the **--recursive** (**-r** or **-R**) option to remove each listed directory, too, along with all of its contents.



To remove a file whose name starts with a '-', for example '-foo', use one of these commands:

```
rm -- -foo
```

```
rm ./-foo
```

Note that if you use `rm` to remove a file, it might be possible to recover some of its contents, given sufficient expertise and/or time. For greater assurance that the contents are truly unrecoverable, consider using [shred\(1\)](#).

AUTHOR [top](#)

Written by Paul Rubin, David MacKenzie, Richard M. Stallman, and Jim Meyering.

REPORTING BUGS [top](#)

GNU coreutils online help:
<<https://www.gnu.org/software/coreutils/>>
Report any translation bugs to
<<https://translationproject.org/team/>>

COPYRIGHT [top](#)

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>. This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO [top](#)

[unlink\(1\)](#), [unlink\(2\)](#), [chattr\(1\)](#), [shred\(1\)](#)

Full documentation <<https://www.gnu.org/software/coreutils/rm>> or available locally via: `info '(coreutils) rm invocation'`

COLOPHON [top](#)

This page is part of the *coreutils* (basic file, shell and text manipulation utilities) project. Information about the project can be found at <http://www.gnu.org/software/coreutils/>. If you have a bug report for this manual page, see <http://www.gnu.org/software/coreutils/>. This page was obtained from the tarball *coreutils-9.4.tar.xz* fetched from <http://ftp.gnu.org/gnu/coreutils/> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

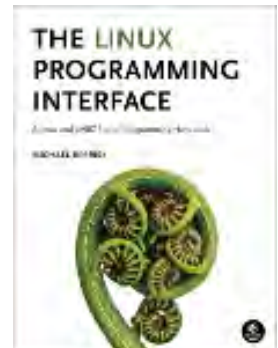
GNU coreutils 9.4**August 2023****RM(1)**

Pages that refer to this page: [rmdir\(2\)](#), [unlink\(2\)](#), [remove\(3\)](#), [mq_overview\(7\)](#), [symlink\(7\)](#), [debugfs\(8\)](#), [ls\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



cat(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [EXAMPLES](#) | [AUTHOR](#) | [REPORTING BUGS](#)
| [COPYRIGHT](#) | [SEE ALSO](#) | [COLOPHON](#)

 CAT(1)

User Commands

CAT(1)**NAME** [top](#)

cat - concatenate files and print on the standard output

SYNOPSIS [top](#)

cat [*OPTION*]... [*FILE*]...

DESCRIPTION [top](#)

Concatenate FILE(s) to standard output.

With no FILE, or when FILE is -, read standard input.

-A, --show-all
equivalent to **-vET**

-b, --number-nonblank
number nonempty output lines, overrides **-n**

-e equivalent to **-vE**

-E, --show-ends
display \$ at end of each line

-n, --number
number all output lines

- s, --squeeze-blank**
suppress repeated empty output lines
- t** equivalent to **-vT**
- T, --show-tabs**
display TAB characters as ^I
- u** (ignored)
- v, --show-nonprinting**
use ^ and M- notation, except for LFD and TAB
- help** display this help and exit
- version**
output version information and exit

EXAMPLES [top](#)

```
cat f - g
```

Output f's contents, then standard input, then g's contents.

```
cat
```

Copy standard input to standard output.

AUTHOR [top](#)

Written by Torbjorn Granlund and Richard M. Stallman.

REPORTING BUGS [top](#)

GNU coreutils online help:
<<https://www.gnu.org/software/coreutils/>>
Report any translation bugs to
<<https://translationproject.org/team/>>

COPYRIGHT [top](#)

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>. This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO [top](#)

[tac\(1\)](#)

Full documentation <<https://www.gnu.org/software/coreutils/cat>> or available locally via: info '(coreutils) cat invocation'

COLOPHON [top](#)

This page is part of the *coreutils* (basic file, shell and text manipulation utilities) project. Information about the project can be found at <<http://www.gnu.org/software/coreutils/>>. If you have a bug report for this manual page, see <<http://www.gnu.org/software/coreutils/>>. This page was obtained from the tarball *coreutils-9.4.tar.xz* fetched from <<http://ftp.gnu.org/gnu/coreutils/>> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

GNU coreutils 9.4

August 2023

CAT(1)

Pages that refer to this page: [pmlogrewrite\(1\)](#), [pv\(1\)](#), [systemd-socket-activate\(1\)](#), [tac\(1\)](#), [ul\(1\)](#), [proc\(5\)](#), [cpuset\(7\)](#), [time_namespaces\(7\)](#), [readprofile\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



clear(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [HISTORY](#) | [PORTABILITY](#) | [SEE ALSO](#) | [COLOPHON](#)

 [@CLEAR@\(1\)](#)

General Commands Manual

[@CLEAR@\(1\)](#)

NAME [top](#)

@CLEAR@ - clear the terminal screen

SYNOPSIS [top](#)

@CLEAR@ [-T*type*] [-V] [-x]

DESCRIPTION [top](#)

@CLEAR@ clears your terminal's screen if this is possible, including the terminal's scrollbar buffer (if the extended "E3" capability is defined). **@CLEAR@** looks in the environment for the terminal type given by the environment variable **TERM**, and then in the **terminfo** database to determine how to clear the screen.

@CLEAR@ writes to the standard output. You can redirect the standard output to a file (which prevents **@CLEAR@** from actually clearing the screen), and later **cat** the file to the screen, clearing it at that point.

OPTIONS [top](#)

-T *type*
indicates the *type* of terminal. Normally this option is unnecessary, because the default is taken from the

environment variable **TERM**. If **-T** is specified, then the shell variables **LINES** and **COLUMNS** will also be ignored.

- V** reports the version of ncurses which was used in this program, and exits. The options are as follows:
- x** do not attempt to clear the terminal's scrollbar buffer using the extended "E3" capability.

HISTORY [top](#)

A **clear** command appeared in 2.79BSD dated February 24, 1979. Later that was provided in Unix 8th edition (1985).

AT&T adapted a different BSD program (**tset**) to make a new command (**tput**), and used this to replace the **clear** command with a shell script which calls **tput clear**, e.g.,

```
/usr/bin/tput ${1:+-T$1} clear 2> /dev/null
exit
```

In 1989, when Keith Bostic revised the BSD **tput** command to make it similar to the AT&T **tput**, he added a shell script for the **clear** command:

```
exec tput clear
```

The remainder of the script in each case is a copyright notice.

The ncurses **clear** command began in 1995 by adapting the original BSD **clear** command (with terminfo, of course).

The **E3** extension came later:

- In June 1999, **xterm** provided an extension to the standard control sequence for clearing the screen. Rather than clearing just the visible part of the screen using

```
printf '\033[2J'
```

one could clear the *scrollback* using

```
printf '\033[3J'
```

This is documented in *XTerm Control Sequences* as a feature originating with **xterm**.

- A few other terminal developers adopted the feature, e.g., PuTTY in 2006.
- In April 2011, a Red Hat developer submitted a patch to the Linux kernel, modifying its console driver to do the same thing. The Linux change, part of the 3.0 release, did not mention **xterm**, although it was cited in the Red Hat bug report (#683733) which led to the change.
- Again, a few other terminal developers adopted the feature. But the next relevant step was a change to the **clear** program in 2013 to incorporate this extension.
- In 2013, the **E3** extension was overlooked in **@TPUT@** with the “clear” parameter. That was addressed in 2016 by reorganizing **@TPUT@** to share its logic with **@CLEAR@** and **@TSET@**.

PORTABILITY [top](#)

Neither IEEE Std 1003.1/The Open Group Base Specifications Issue 7 (POSIX.1-2008) nor X/Open Curses Issue 7 documents **@TSET@** or **@RESET@**.

The latter documents **tput**, which could be used to replace this utility either via a shell script or by an alias (such as a symbolic link) to run **@TPUT@** as **@CLEAR@**.

SEE ALSO [top](#)

@TPUT@(1), **terminfo(5)**, **xterm(1)**.

This describes **ncurses** version **@NCURSES_MAJOR@.@NCURSES_MINOR@** (patch **@NCURSES_PATCH@**).

COLOPHON [top](#)

This page is part of the *ncurses* (new curses) project. Information about the project can be found at <https://www.gnu.org/software/ncurses/ncurses.html>. If you have a bug report for this manual page, send it to bug-ncurses-request@gnu.org. This page was obtained from the project's upstream Git mirror of the CVS repository <https://github.com/mirror/ncurses.git> on 2023-12-22. (At that time, the date of the most recent commit that was found in the repository was 2023-03-12.) If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

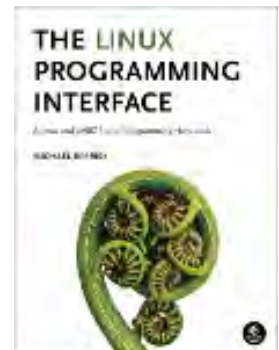
@CLEAR@(1)

Pages that refer to this page: [setterm\(1\)](#), [user_caps\(5\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



cp(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [AUTHOR](#) | [REPORTING BUGS](#) | [COPYRIGHT](#) | [SEE ALSO](#) | [COLOPHON](#)

 CP(1)

User Commands

CP(1)**NAME** [top](#)

cp - copy files and directories

SYNOPSIS [top](#)

```
cp [OPTION]... [-T] SOURCE DEST
cp [OPTION]... SOURCE... DIRECTORY
cp [OPTION]... -t DIRECTORY SOURCE...
```

DESCRIPTION [top](#)

Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

Mandatory arguments to long options are mandatory for short options too.

-a, --archive

same as **-dR --preserve=all**

--attributes-only

don't copy the file data, just the attributes

--backup[=CONTROL]

make a backup of each existing destination file

-b like **--backup** but does not accept an argument

- copy-contents**
copy contents of special files when recursive
- d** same as **--no-dereference** **--preserve=Links**
- debug**
explain how a file is copied. Implies **-v**
- f, --force**
if an existing destination file cannot be opened, remove it and try again (this option is ignored when the **-n** option is also used)
- i, --interactive**
prompt before overwrite (overrides a previous **-n** option)
- H** follow command-line symbolic links in SOURCE
- l, --link**
hard link files instead of copying
- L, --dereference**
always follow symbolic links in SOURCE
- n, --no-clobber**
do not overwrite an existing file (overrides a **-u** or previous **-i** option). See also **--update**
- P, --no-dereference**
never follow symbolic links in SOURCE
- p** same as **--preserve=mode,ownership,timestamps**
- preserve[=ATTR_LIST]**
preserve the specified attributes
- no-preserve=ATTR_LIST**
don't preserve the specified attributes
- parents**
use full source file name under DIRECTORY
- R, -r, --recursive**



copy directories recursively

--reflink[=*WHEN*]

control clone/CoW copies. See below

--remove-destination

remove each existing destination file before attempting to open it (contrast with **--force**)

--sparse=*WHEN*

control creation of sparse files. See below

--strip-trailing-slashes

remove any trailing slashes from each SOURCE argument

-s, --symbolic-link

make symbolic links instead of copying

-S, --suffix=*SUFFIX*

override the usual backup suffix

-t, --target-directory=*DIRECTORY*

copy all SOURCE arguments into DIRECTORY

-T, --no-target-directory

treat DEST as a normal file

--update[=*UPDATE*]

control which existing files are updated;
UPDATE={all,none,older(default)}. See below

-u equivalent to **--update**[=*older*]

-v, --verbose

explain what is being done

-x, --one-file-system

stay on this file system

-Z set SELinux security context of destination file to default type

--context[=*CTX*]

like **-Z**, or if CTX is specified then set the SELinux or



SMACK security context to CTX

--help display this help and exit

--version

output version information and exit

ATTR_LIST is a comma-separated list of attributes. Attributes are 'mode' for permissions (including any ACL and xattr permissions), 'ownership' for user and group, 'timestamps' for file timestamps, 'links' for hard links, 'context' for security context, 'xattr' for extended attributes, and 'all' for all attributes.

By default, sparse SOURCE files are detected by a crude heuristic and the corresponding DEST file is made sparse as well. That is the behavior selected by **--sparse=auto**. Specify **--sparse=always** to create a sparse DEST file whenever the SOURCE file contains a long enough sequence of zero bytes. Use **--sparse=never** to inhibit creation of sparse files.

UPDATE controls which existing files in the destination are replaced. 'all' is the default operation when an **--update** option is not specified, and results in all existing files in the destination being replaced. 'none' is similar to the **--no-clobber** option, in that no files in the destination are replaced, but also skipped files do not induce a failure. 'older' is the default operation when **--update** is specified, and results in files being replaced if they're older than the corresponding source file.

When **--reflink[=always]** is specified, perform a lightweight copy, where the data blocks are copied only when modified. If this is not possible the copy fails, or if **--reflink=auto** is specified, fall back to a standard copy. Use **--reflink=never** to ensure a standard copy is performed.

The backup suffix is '~', unless set with **--suffix** or SIMPLE_BACKUP_SUFFIX. The version control method may be selected via the **--backup** option or through the VERSION_CONTROL environment variable. Here are the values:

none, off

never make backups (even if **--backup** is given)



numbered, t
make numbered backups

existing, nil
numbered if numbered backups exist, simple otherwise

simple, never
always make simple backups

As a special case, cp makes a backup of SOURCE when the force and backup options are given and SOURCE and DEST are the same name for an existing, regular file.

AUTHOR [top](#)

Written by Torbjorn Granlund, David MacKenzie, and Jim Meyering.

REPORTING BUGS [top](#)

GNU coreutils online help:
<<https://www.gnu.org/software/coreutils/>>
Report any translation bugs to
<<https://translationproject.org/team/>>

COPYRIGHT [top](#)

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>. This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO [top](#)

[install\(1\)](#)

Full documentation <<https://www.gnu.org/software/coreutils/cp>> or available locally via: info '(coreutils) cp invocation'

COLOPHON [top](#)

This page is part of the *coreutils* (basic file, shell and text manipulation utilities) project. Information about the project can be found at <http://www.gnu.org/software/coreutils/>. If you have a bug report for this manual page, see <http://www.gnu.org/software/coreutils/>. This page was obtained from the tarball *coreutils-9.4.tar.xz* fetched from <http://ftp.gnu.org/gnu/coreutils/> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

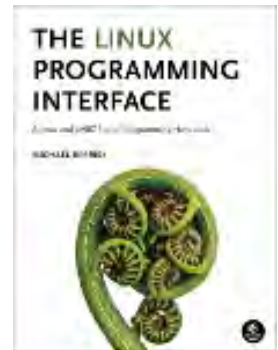
GNU coreutils 9.4**August 2023****CP(1)**

Pages that refer to this page: [install\(1\)](#), [pmllogmv\(1\)](#), [rsync\(1\)](#), [cpuset\(7\)](#), [symlink\(7\)](#), [e2image\(8\)](#), [readprofile\(8\)](#), [swapon\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



mv(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [AUTHOR](#) | [REPORTING BUGS](#) | [COPYRIGHT](#) | [SEE ALSO](#) | [COLOPHON](#)

 MV(1)

User Commands

MV(1)

NAME [top](#)

mv - move (rename) files

SYNOPSIS [top](#)

```
mv [OPTION]... [-T] SOURCE DEST
mv [OPTION]... SOURCE... DIRECTORY
mv [OPTION]... -t DIRECTORY SOURCE...
```

DESCRIPTION [top](#)

Rename SOURCE to DEST, or move SOURCE(s) to DIRECTORY.

Mandatory arguments to long options are mandatory for short options too.

--backup[=*CONTROL*]
make a backup of each existing destination file

-b like **--backup** but does not accept an argument

--debug
explain how a file is copied. Implies **-v**

-f, --force
do not prompt before overwriting

-i, --interactive
prompt before overwrite

-n, --no-clobber
do not overwrite an existing file

If you specify more than one of **-i**, **-f**, **-n**, only the final one takes effect.

--no-copy
do not copy if renaming fails

--strip-trailing-slashes
remove any trailing slashes from each SOURCE argument

-S, --suffix=*SUFFIX*
override the usual backup suffix

-t, --target-directory=*DIRECTORY*
move all SOURCE arguments into DIRECTORY

-T, --no-target-directory
treat DEST as a normal file

--update[=*UPDATE*]
control which existing files are updated;
UPDATE={all,none,older(default)}. See below

-u equivalent to **--update[=*older*]**

-v, --verbose
explain what is being done

-Z, --context
set SELinux security context of destination file to default type

--help display this help and exit

--version
output version information and exit

UPDATE controls which existing files in the destination are

replaced. 'all' is the default operation when an **--update** option is not specified, and results in all existing files in the destination being replaced. 'none' is similar to the **--no-clobber** option, in that no files in the destination are replaced, but also skipped files do not induce a failure. 'older' is the default operation when **--update** is specified, and results in files being replaced if they're older than the corresponding source file.

The backup suffix is '~', unless set with **--suffix** or `SIMPLE_BACKUP_SUFFIX`. The version control method may be selected via the **--backup** option or through the `VERSION_CONTROL` environment variable. Here are the values:

none, off

never make backups (even if **--backup** is given)

numbered, t

make numbered backups

existing, nil

numbered if numbered backups exist, simple otherwise

simple, never

always make simple backups

AUTHOR [top](#)

Written by Mike Parker, David MacKenzie, and Jim Meyering.

REPORTING BUGS [top](#)

GNU coreutils online help:

<<https://www.gnu.org/software/coreutils/>>

Report any translation bugs to

<<https://translationproject.org/team/>>

COPYRIGHT [top](#)

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>.

This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO [top](#)

[rename\(2\)](#)

Full documentation <<https://www.gnu.org/software/coreutils/mv>> or available locally via: info '(coreutils) mv invocation'

COLOPHON [top](#)

This page is part of the *coreutils* (basic file, shell and text manipulation utilities) project. Information about the project can be found at <<http://www.gnu.org/software/coreutils/>>. If you have a bug report for this manual page, see <<http://www.gnu.org/software/coreutils/>>. This page was obtained from the tarball *coreutils-9.4.tar.xz* fetched from <<http://ftp.gnu.org/gnu/coreutils/>> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

GNU coreutils 9.4

August 2023

MV(1)

Pages that refer to this page: [rename\(1\)](#), [sshfs\(1\)](#), [rename\(2\)](#), [inotify\(7\)](#), [symlink\(7\)](#), [lsof\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sudo(8) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [COMMAND EXECUTION](#) | [EXIT VALUE](#) | [SECURITY NOTES](#) | [ENVIRONMENT](#) | [FILES](#) | [EXAMPLES](#) | [DIAGNOSTICS](#) | [SEE ALSO](#) | [HISTORY](#) | [AUTHORS](#) | [CAVEATS](#) | [BUGS](#) | [SUPPORT](#) | [DISCLAIMER](#) | [COLOPHON](#)

 SUDO(8)

System Manager's Manual

SUDO(8)**NAME** [top](#)

sudo, **sudoedit** – execute a command as another user

SYNOPSIS [top](#)

```
sudo -h | -K | -k | -V sudo -v [-ABkNnS] [-g group] [-h host] [-p
prompt] [-u user] sudo -l [-ABkNnS] [-g group] [-h host] [-p
prompt] [-U user] [-u user] [command [arg ...]] sudo [-ABbEHnPS]
[-C num] [-D directory] [-g group] [-h host] [-p prompt] [-R
directory] [-T timeout] [-u user] [VAR=value] [-i | -s] [command
[arg ...]] sudoedit [-ABkNnS] [-C num] [-D directory] [-g group]
[-h host] [-p prompt] [-R directory] [-T timeout] [-u user]
file ...
```

DESCRIPTION [top](#)

allows a permitted user to execute a *command* as the superuser or another user, as specified by the security policy. The invoking user's real (*not* effective) user-ID is used to determine the user name with which to query the security policy.

supports a plugin architecture for security policies, auditing, and input/output logging. Third parties can develop and distribute their own plugins to work seamlessly with the front-



end. The default security policy is *sudoers*, which is configured via the file */etc/sudoers*, or via LDAP. See the “Plugins” section for more information.

The security policy determines what privileges, if any, a user has to run . The policy may require that users authenticate themselves with a password or another authentication mechanism. If authentication is required, will exit if the user's password is not entered within a configurable time limit. This limit is policy-specific; the default password prompt timeout for the *sudoers* security policy is 5 minutes.

Security policies may support credential caching to allow the user to run again for a period of time without requiring authentication. By default, the *sudoers* policy caches credentials on a per-terminal basis for 5 minutes. See the *timestamp_type* and *timestamp_timeout* options in *sudoers*(5) for more information. By running with the *-v* option, a user can update the cached credentials without running a *command*.

On systems where is the primary method of gaining superuser privileges, it is imperative to avoid syntax errors in the security policy configuration files. For the default security policy, *sudoers*(5), changes to the configuration files should be made using the *visudo*(8) utility which will ensure that no syntax errors are introduced.

When invoked as *sudoedit*, the *-e* option (described below), is implied.

Security policies and audit plugins may log successful and failed attempts to run . If an I/O plugin is configured, the running *command*'s input and output may be logged as well.

The options are as follows:

-A, --askpass

Normally, if requires a password, it will read it from the user's terminal. If the *-A* (*askpass*) option is specified, a (possibly graphical) helper program is executed to read the user's password and output the password to the standard output. If the *SUDO_ASKPASS* environment variable is set, it specifies the path to the helper program. Otherwise, if *sudo.conf*(5) contains a



line specifying the askpass program, that value will be used. For example:

```
# Path to askpass helper program
Path askpass /usr/X11R6/bin/ssh-askpass
```

If no askpass program is available, will exit with an error.

-B, --bell

Ring the bell as part of the password prompt when a terminal is present. This option has no effect if an askpass program is used.

-b, --background

Run the given *command* in the background. It is not possible to use shell job control to manipulate background processes started by `.` Most interactive *commands* will fail to work properly in background mode.

-C *num*, --close-from=*num*

Close all file descriptors greater than or equal to *num* before executing a *command*. Values less than three are not permitted. By default, will close all open file descriptors other than standard input, standard output, and standard error when executing a *command*. The security policy may restrict the user's ability to use this option. The *sudoers* policy only permits use of the `-C` option when the administrator has enabled the *closefrom_override* option.

-D *directory*, --chdir=*directory*

Run the *command* in the specified *directory* instead of the current working directory. The security policy may return an error if the user does not have permission to specify the working directory.

-E, --preserve-env

Indicates to the security policy that the user wishes to preserve their existing environment variables. The security policy may return an error if the user does not have permission to preserve the environment.

--preserve-env=*list*



Indicates to the security policy that the user wishes to add the comma-separated list of environment variables to those preserved from the user's environment. The security policy may return an error if the user does not have permission to preserve the environment. This option may be specified multiple times.

-e, --edit

Edit one or more *files* instead of running a *command*. In lieu of a path name, the string "sudoedit" is used when consulting the security policy. If the user is authorized by the policy, the following steps are taken:

1. Temporary copies are made of the files to be edited with the owner set to the invoking user.
2. The editor specified by the policy is run to edit the temporary files. The *sudoers* policy uses the SUDO_EDITOR, VISUAL and EDITOR environment variables (in that order). If none of SUDO_EDITOR, VISUAL or EDITOR are set, the first program listed in the *editor sudoers(5)* option is used.
3. If they have been modified, the temporary files are copied back to their original location and the temporary versions are removed.

To help prevent the editing of unauthorized files, the following restrictions are enforced unless explicitly allowed by the security policy:

- Symbolic links may not be edited (version 1.8.15 and higher).
- Symbolic links along the path to be edited are not followed when the parent directory is writable by the invoking user unless that user is root (version 1.8.16 and higher).
- Files located in a directory that is writable by the invoking user may not be edited unless that user is root (version 1.8.16 and higher).

Users are never allowed to edit device special files.



If the specified file does not exist, it will be created. Unlike most *commands* run by *sudo*, the editor is run with the invoking user's environment unmodified. If the temporary file becomes empty after editing, the user will be prompted before it is installed. If, for some reason, is unable to update a file with its edited version, the user will receive a warning and the edited copy will remain in a temporary file.

-g *group*, --group=*group*

Run the *command* with the primary group set to *group* instead of the primary group specified by the target user's password database entry. The *group* may be either a group name or a numeric group-ID (GID) prefixed with the '#' character (e.g., '#0' for GID 0). When running a *command* as a GID, many shells require that the '#' be escaped with a backslash ('\'). If no **-u** option is specified, the *command* will be run as the invoking user. In either case, the primary group will be set to *group*. The *sudoers* policy permits any of the target user's groups to be specified via the **-g** option as long as the **-P** option is not in use.

-H, --set-home

Request that the security policy set the HOME environment variable to the home directory specified by the target user's password database entry. Depending on the policy, this may be the default behavior.

-h, --help

Display a short help message to the standard output and exit.

-h *host*, --host=*host*

Run the *command* on the specified *host* if the security policy plugin supports remote *commands*. The *sudoers* plugin does not currently support running remote *commands*. This may also be used in conjunction with the **-l** option to list a user's privileges for the remote host.

-i, --login

Run the shell specified by the target user's password



database entry as a login shell. This means that login-specific resource files such as `.profile`, `.bash_profile`, or `.login` will be read by the shell. If a `command` is specified, it is passed to the shell as a simple `command` using the `-c` option. The `command` and any `args` are concatenated, separated by spaces, after escaping each character (including white space) with a backslash (`'\'`) except for alphanumerics, underscores, hyphens, and dollar signs. If no `command` is specified, an interactive shell is executed. `sudo` attempts to change to that user's home directory before running the shell. The `command` is run with an environment similar to the one a user would receive at log in. Most shells behave differently when a `command` is specified as compared to an interactive session; consult the shell's manual for details. The `Command environment` section in the `sudoers(5)` manual documents how the `-i` option affects the environment in which a `command` is run when the `sudoers` policy is in use.

-K, --remove-timestamp

Similar to the `-k` option, except that it removes every cached credential for the user, regardless of the terminal or parent process ID. The next time is run, a password must be entered if the security policy requires authentication. It is not possible to use the `-K` option in conjunction with a `command` or other option. This option does not require a password. Not all security policies support credential caching.

-k, --reset-timestamp

When used without a `command`, invalidates the user's cached credentials for the current session. The next time is run in the session, a password must be entered if the security policy requires authentication. By default, the `sudoers` policy uses a separate record in the credential cache for each terminal (or parent process ID if no terminal is present). This prevents the `-k` option from interfering with commands run in a different terminal session. See the `timestamp_type` option in `sudoers(5)` for more information. This option does not require a password, and was added to allow a user to revoke permissions from a `.Logout` file.

When used in conjunction with a `command` or an option that



may require a password, this option will cause to ignore the user's cached credentials. As a result, will prompt for a password (if one is required by the security policy) and will not update the user's cached credentials.

Not all security policies support credential caching.

-l, --list

If no *command* is specified, list the privileges for the invoking user (or the *user* specified by the **-U** option) on the current host. A longer list format is used if this option is specified multiple times and the security policy supports a verbose output format.

If a *command* is specified and is permitted by the security policy for the invoking user (or the, *user* specified by the **-U** option) on the current host, the fully-qualified path to the *command* is displayed along with any *args*. If **-l** is specified more than once (and the security policy supports it), the matching rule is displayed in a verbose format along with the *command*. If a *command* is specified but not allowed by the policy, will exit with a status value of 1.

-N, --no-update

Do not update the user's cached credentials, even if the user successfully authenticates. Unlike the **-k** flag, existing cached credentials are used if they are valid. To detect when the user's cached credentials are valid (or when no authentication is required), the following can be used:

```
sudo -Nnv
```

Not all security policies support credential caching.

-n, --non-interactive

Avoid prompting the user for input of any kind. If a password is required for the *command* to run, will display an error message and exit.

-P, --preserve-groups

Preserve the invoking user's group vector unaltered. By



default, the *sudoers* policy will initialize the group vector to the list of groups the target user is a member of. The real and effective group-IDs, however, are still set to match the target user.

-p *prompt*, --prompt=*prompt*

Use a custom password prompt with optional escape sequences. The following percent ('%') escape sequences are supported by the *sudoers* policy:

- %H expanded to the host name including the domain name (only if the machine's host name is fully qualified or the *fqdn* option is set in *sudoers*(5))
- %h expanded to the local host name without the domain name
- %p expanded to the name of the user whose password is being requested (respects the *rootpw*, *targetpw*, and *runaspw* flags in *sudoers*(5))
- %U expanded to the login name of the user the *command* will be run as (defaults to root unless the **-u** option is also specified)
- %u expanded to the invoking user's login name
- %% two consecutive '%' characters are collapsed into a single '%' character

The custom prompt will override the default prompt specified by either the security policy or the SUDO_PROMPT environment variable. On systems that use PAM, the custom prompt will also override the prompt specified by a PAM module unless the *passprompt_override* flag is disabled in *sudoers*.

-R *directory*, --chroot=*directory*

Change to the specified root *directory* (see *chroot*(8)) before running the *command*. The security policy may return an error if the user does not have permission to specify the root directory.

-S, --stdin



Write the prompt to the standard error and read the password from the standard input instead of using the terminal device.

-s, --shell

Run the shell specified by the SHELL environment variable if it is set or the shell specified by the invoking user's password database entry. If a *command* is specified, it is passed to the shell as a simple command using the **-c** option. The *command* and any *args* are concatenated, separated by spaces, after escaping each character (including white space) with a backslash (`'\'`) except for alphanumerics, underscores, hyphens, and dollar signs. If no *command* is specified, an interactive shell is executed. Most shells behave differently when a *command* is specified as compared to an interactive session; consult the shell's manual for details.

-U *user*, --other-user=*user*

Used in conjunction with the **-l** option to list the privileges for *user* instead of for the invoking user. The security policy may restrict listing other users' privileges. When using the *sudoers* policy, the **-U** option is restricted to the root user and users with either the "list" privilege for the specified *user* or the ability to run any *command* as root or *user* on the current host.

-T *timeout*, --command-timeout=*timeout*

Used to set a timeout for the *command*. If the timeout expires before the *command* has exited, the *command* will be terminated. The security policy may restrict the user's ability to set timeouts. The *sudoers* policy requires that user-specified timeouts be explicitly enabled.

-u *user*, --user=*user*

Run the *command* as a user other than the default target user (usually **root**). The *user* may be either a user name or a numeric user-ID (UID) prefixed with the '#' character (e.g., '#0' for UID 0). When running *commands* as a UID, many shells require that the '#' be escaped with a backslash (`'\'`). Some security policies may restrict UIDs to those listed in the password database. The *sudoers* policy allows UIDs that are not in the



password database as long as the *targetpw* option is not set. Other security policies may not support this.

-V, --version

Print the version string as well as the version string of any configured plugins. If the invoking user is already root, the **-V** option will display the options passed to configure when was built; plugins may display additional information such as default options.

-v, --validate

Update the user's cached credentials, authenticating the user if necessary. For the *sudoers* plugin, this extends the timeout for another 5 minutes by default, but does not run a *command*. Not all security policies support cached credentials.

-- The **--** is used to delimit the end of the options. Subsequent options are passed to the *command*.

Options that take a value may only be specified once unless otherwise indicated in the description. This is to help guard against problems caused by poorly written scripts that invoke **sudo** with user-controlled input.

Environment variables to be set for the *command* may also be passed as options to in the form *VAR=value*, for example *LD_LIBRARY_PATH=/usr/local/pkg/lib*. Environment variables may be subject to restrictions imposed by the security policy plugin. The *sudoers* policy subjects environment variables passed as options to the same restrictions as existing environment variables with one important difference. If the *setenv* option is set in *sudoers*, the *command* to be run has the SETENV tag set or the *command* matched is **ALL**, the user may set variables that would otherwise be forbidden. See *sudoers*(5) for more information.

COMMAND EXECUTION

[top](#)

When executes a *command*, the security policy specifies the execution environment for the *command*. Typically, the real and effective user and group and IDs are set to match those of the target user, as specified in the password database, and the group vector is initialized based on the group database (unless the **-P**



option was specified).

The following parameters may be specified by security policy:

- real and effective user-ID
- real and effective group-ID
- supplementary group-IDs
- the environment list
- current working directory
- file creation mode mask (umask)
- scheduling priority (aka nice value)

Process model

There are two distinct ways can run a *command*.

If an I/O logging plugin is configured to log terminal I/O, or if the security policy explicitly requests it, a new pseudo-terminal (“pty”) is allocated and *fork(2)* is used to create a second process, referred to as the *monitor*. The *monitor* creates a new terminal session with itself as the leader and the pty as its controlling terminal, calls *fork(2)* again, sets up the execution environment as described above, and then uses the *execve(2)* system call to run the *command* in the child process. The *monitor* exists to relay job control signals between the user's terminal and the pty the *command* is being run in. This makes it possible to suspend and resume the *command* normally. Without the *monitor*, the *command* would be in what POSIX terms an “orphaned process group” and it would not receive any job control signals from the kernel. When the *command* exits or is terminated by a signal, the *monitor* passes the *command*'s exit status to the main process and exits. After receiving the *command*'s exit status, the main process passes the *command*'s exit status to the security policy's close function, as well as the close function of any configured audit plugin, and exits. This mode is the default for sudo versions 1.9.14 and above when using the sudoers policy.

If no pty is used, calls *fork(2)*, sets up the execution environment as described above, and uses the *execve(2)* system



call to run the *command* in the child process. The main process waits until the *command* has completed, then passes the *command*'s exit status to the security policy's close function, as well as the close function of any configured audit plugins, and exits. As a special case, if the policy plugin does not define a close function, will execute the *command* directly instead of calling *fork(2)* first. The *sudoers* policy plugin will only define a close function when I/O logging is enabled, a pty is required, an SELinux role is specified, the *command* has an associated timeout, or the *pam_session* or *pam_setcred* options are enabled. Both *pam_session* and *pam_setcred* are enabled by default on systems using PAM. This mode is the default for sudo versions prior to 1.9.14 when using the sudoers policy.

On systems that use PAM, the security policy's close function is responsible for closing the PAM session. It may also log the *command*'s exit status.

Signal handling

When the *command* is run as a child of the process, will relay signals it receives to the *command*. The SIGINT and SIGQUIT signals are only relayed when the *command* is being run in a new pty or when the signal was sent by a user process, not the kernel. This prevents the *command* from receiving SIGINT twice each time the user enters control-C. Some signals, such as SIGSTOP and SIGKILL, cannot be caught and thus will not be relayed to the *command*. As a general rule, SIGTSTP should be used instead of SIGSTOP when you wish to suspend a *command* being run by .

As a special case, will not relay signals that were sent by the *command* it is running. This prevents the *command* from accidentally killing itself. On some systems, the *reboot(8)* utility sends SIGTERM to all non-system processes other than itself before rebooting the system. This prevents from relaying the SIGTERM signal it received back to *reboot(8)*, which might then exit before the system was actually rebooted, leaving it in a half-dead state similar to single user mode. Note, however, that this check only applies to the *command* run by and not any other processes that the *command* may create. As a result, running a script that calls *reboot(8)* or *shutdown(8)* via may cause the system to end up in this undefined state unless the *reboot(8)* or *shutdown(8)* are run using the *exec()* family of functions instead of *system()* (which interposes a shell between



the *command* and the calling process).

Plugins

Plugins may be specified via *Plugin* directives in the *sudo.conf*(5) file. They may be loaded as dynamic shared objects (on systems that support them), or compiled directly into the binary. If no *sudo.conf*(5) file is present, or if it doesn't contain any *Plugin* lines, will use *sudoers*(5) for the policy, auditing, and I/O logging plugins. See the *sudo.conf*(5) manual for details of the */etc/sudo.conf* file and the *sudo_plugin*(5) manual for more information about the plugin architecture.

EXIT VALUE [top](#)

Upon successful execution of a *command*, the exit status from will be the exit status of the program that was executed. If the *command* terminated due to receipt of a signal, will send itself the same signal that terminated the *command*.

If the **-l** option was specified without a *command*, will exit with a value of 0 if the user is allowed to run and they authenticated successfully (as required by the security policy). If a *command* is specified with the **-l** option, the exit value will only be 0 if the *command* is permitted by the security policy, otherwise it will be 1.

If there is an authentication failure, a configuration/permission problem, or if the given *command* cannot be executed, exits with a value of 1. In the latter case, the error string is printed to the standard error. If cannot *stat*(2) one or more entries in the user's PATH, an error is printed to the standard error. (If the directory does not exist or if it is not really a directory, the entry is ignored and no error is printed.) This should not happen under normal circumstances. The most common reason for *stat*(2) to return "permission denied" is if you are running an automounter and one of the directories in your PATH is on a machine that is currently unreachable.

SECURITY NOTES [top](#)

tries to be safe when executing external *commands*.



To prevent command spoofing, checks "." and "" (both denoting current directory) last when searching for a *command* in the user's PATH (if one or both are in the PATH). Depending on the security policy, the user's PATH environment variable may be modified, replaced, or passed unchanged to the program that executes.

Users should *never* be granted privileges to execute files that are writable by the user or that reside in a directory that is writable by the user. If the user can modify or replace the *command* there is no way to limit what additional *commands* they can run.

By default, will only log the *command* it explicitly runs. If a user runs a *command* such as 'sudo su' or 'sudo sh', subsequent *commands* run from that shell are not subject to **sudo**'s security policy. The same is true for *commands* that offer shell escapes (including most editors). If I/O logging is enabled, subsequent *commands* will have their input and/or output logged, but there will not be traditional logs for those *commands*. Because of this, care must be taken when giving users access to *commands* via to verify that the *command* does not inadvertently give the user an effective root shell. For information on ways to address this, see the *Preventing shell escapes* section in *sudoers(5)*.

To prevent the disclosure of potentially sensitive information, disables core dumps by default while it is executing (they are re-enabled for the *command* that is run). This historical practice dates from a time when most operating systems allowed set-user-ID processes to dump core by default. To aid in debugging crashes, you may wish to re-enable core dumps by setting "disable_coredump" to false in the *sudo.conf(5)* file as follows:

```
Set disable_coredump false
```

See the *sudo.conf(5)* manual for more information.

ENVIRONMENT [top](#)

utilizes the following environment variables. The security policy has control over the actual content of the *command*'s environment.



| | |
|--------------|---|
| EDITOR | Default editor to use in -e (sudoedit) mode if neither SUDO_EDITOR nor VISUAL is set. |
| MAIL | Set to the mail spool of the target user when the -i option is specified, or when <i>env_reset</i> is enabled in <i>sudoers</i> (unless MAIL is present in the <i>env_keep</i> list). |
| HOME | Set to the home directory of the target user when the -i or -H options are specified, when the -s option is specified and <i>set_home</i> is set in <i>sudoers</i> , when <i>always_set_home</i> is enabled in <i>sudoers</i> , or when <i>env_reset</i> is enabled in <i>sudoers</i> and HOME is not present in the <i>env_keep</i> list. |
| LOGNAME | Set to the login name of the target user when the -i option is specified, when the <i>set_Logname</i> option is enabled in <i>sudoers</i> , or when the <i>env_reset</i> option is enabled in <i>sudoers</i> (unless LOGNAME is present in the <i>env_keep</i> list). |
| PATH | May be overridden by the security policy. |
| SHELL | Used to determine shell to run with -s option. |
| SUDO_ASKPASS | Specifies the path to a helper program used to read the password if no terminal is available or if the -A option is specified. |
| SUDO_COMMAND | Set to the <i>command</i> run by sudo, including any <i>args</i> . The <i>args</i> are truncated at 4096 characters to prevent a potential execution error. |
| SUDO_EDITOR | Default editor to use in -e (sudoedit) mode. |
| SUDO_GID | Set to the group-ID of the user who invoked sudo. |
| SUDO_PROMPT | Used as the default password prompt unless the -p option was specified. |
| SUDO_PS1 | If set, PS1 will be set to its value for the program being run. |



| | |
|-----------|---|
| SUDO_UID | Set to the user-ID of the user who invoked sudo. |
| SUDO_USER | Set to the login name of the user who invoked sudo. |
| USER | Set to the same value as LOGNAME, described above. |
| VISUAL | Default editor to use in -e (sudoedit) mode if SUDO_EDITOR is not set. |

FILES [top](#)

/etc/sudo.conf front-end configuration

EXAMPLES [top](#)

The following examples assume a properly configured security policy.

To get a file listing of an unreadable directory:

```
$ sudo ls /usr/local/protected
```

To list the home directory of user yaz on a machine where the file system holding ~yaz is not exported as root:

```
$ sudo -u yaz ls ~yaz
```

To edit the *index.html* file as user www:

```
$ sudoedit -u www ~www/htdocs/index.html
```

To view system logs only accessible to root and users in the adm group:

```
$ sudo -g adm more /var/log/syslog
```

To run an editor as jim with a different primary group:

```
$ sudoedit -u jim -g audio ~jim/sound.txt
```



To shut down a machine:

```
$ sudo shutdown -r +15 "quick reboot"
```

To make a usage listing of the directories in the /home partition. The *commands* are run in a sub-shell to allow the 'cd' command and file redirection to work.

```
$ sudo sh -c "cd /home ; du -s * | sort -rn > USAGE"
```

DIAGNOSTICS [top](#)

Error messages produced by include:

editing files in a writable directory is not permitted

By default, **sudoedit** does not permit editing a file when any of the parent directories are writable by the invoking user. This avoids a race condition that could allow the user to overwrite an arbitrary file. See the *sudoedit_checkdir* option in *sudoers*(5) for more information.

editing symbolic links is not permitted

By default, **sudoedit** does not follow symbolic links when opening files. See the *sudoedit_follow* option in *sudoers*(5) for more information.

effective uid is not 0, is sudo installed setuid root?

was not run with root privileges. The binary must be owned by the root user and have the set-user-ID bit set. Also, it must not be located on a file system mounted with the 'nosuid' option or on an NFS file system that maps uid 0 to an unprivileged uid.

effective uid is not 0, is sudo on a file system with the 'nosuid' option set or an NFS file system without root privileges?

was not run with root privileges. The binary has the proper owner and permissions but it still did not run with root privileges. The most common reason for this is that the file system the binary is located on is mounted with the 'nosuid' option or it is an NFS file system that maps



uid 0 to an unprivileged uid.

fatal error, unable to load plugins

An error occurred while loading or initializing the plugins specified in *sudo.conf*(5).

invalid environment variable name

One or more environment variable names specified via the **-E** option contained an equal sign (`'='`). The arguments to the **-E** option should be environment variable names without an associated value.

no password was provided

When tried to read the password, it did not receive any characters. This may happen if no terminal is available (or the **-S** option is specified) and the standard input has been redirected from */dev/null*.

a terminal is required to read the password

needs to read the password but there is no mechanism available for it to do so. A terminal is not present to read the password from, has not been configured to read from the standard input, the **-S** option was not used, and no askpass helper has been specified either via the *sudo.conf*(5) file or the SUDO_ASKPASS environment variable.

no writable temporary directory found

sudoedit was unable to find a usable temporary directory in which to store its intermediate files.

The “no new privileges” flag is set, which prevents sudo from running as root.

was run by a process that has the Linux “no new privileges” flag is set. This causes the set-user-ID bit to be ignored when running an executable, which will prevent from functioning. The most likely cause for this is running within a container that sets this flag. Check the documentation to see if it is possible to configure the container such that the flag is not set.

sudo must be owned by uid 0 and have the setuid bit set

was not run with root privileges. The binary does not have the correct owner or permissions. It must be owned by the root user and have the set-user-ID bit set.



sudoedit is not supported on this platform

It is only possible to run **sudoedit** on systems that support setting the effective user-ID.

timed out reading password

The user did not enter a password before the password timeout (5 minutes by default) expired.

you do not exist in the passwd database

Your user-ID does not appear in the system passwd database.

you may not specify environment variables in edit mode

It is only possible to specify environment variables when running a *command*. When editing a file, the editor is run with the user's environment unmodified.

SEE ALSO [top](#)

su(1), *stat(2)*, *login_cap(3)*, *passwd(5)*, *sudo.conf(5)*,
sudo_plugin(5), *sudoers(5)*, *sudoers_timestamp(5)*, *sudoreplay(8)*,
visudo(8)

HISTORY [top](#)

See the HISTORY.md file in the distribution (<https://www.sudo.ws/about/history/>) for a brief history of sudo.

AUTHORS [top](#)

Many people have worked on over the years; this version consists of code written primarily by:

Todd C. Miller

See the CONTRIBUTORS.md file in the distribution (<https://www.sudo.ws/about/contributors/>) for an exhaustive list of people who have contributed to .

CAVEATS [top](#)

There is no easy way to prevent a user from gaining a root shell if that user is allowed to run arbitrary *commands* via `.` Also, many programs (such as editors) allow the user to run *commands* via shell escapes, thus avoiding `sudo`'s checks. However, on most systems it is possible to prevent shell escapes with the `sudoers(5)` plugin's *noexec* functionality.

It is not meaningful to run the `'cd'` *command* directly via `sudo`, e.g.,

```
$ sudo cd /usr/local/protected
```

since when the *command* exits the parent process (your shell) will still be the same. The `-D` option can be used to run a *command* in a specific *directory*.

Running shell scripts via `sudo` can expose the same kernel bugs that make set-user-ID shell scripts unsafe on some operating systems (if your OS has a `/dev/fd/` directory, set-user-ID shell scripts are generally safe).

BUGS [top](#)

If you believe you have found a bug in `sudo`, you can submit a bug report at <https://bugzilla.sudo.ws/>

SUPPORT [top](#)

Limited free support is available via the `sudo-users` mailing list, see <https://www.sudo.ws/mailman/listinfo/sudo-users> to subscribe or search the archives.

DISCLAIMER [top](#)

`sudo` is provided “AS IS” and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. See the LICENSE.md file distributed with or <https://www.sudo.ws/about/license/> for complete details.



COLOPHON [top](#)

This page is part of the *sudo* (execute a command as another user) project. Information about the project can be found at <https://www.sudo.ws/>. If you have a bug report for this manual page, see [\(https://bugzilla.sudo.ws/\)](https://bugzilla.sudo.ws/). This page was obtained from the project's upstream Git repository [\(<https://github.com/sudo-project/sudo>](https://github.com/sudo-project/sudo)) on 2023-12-22. (At that time, the date of the most recent commit that was found in the repository was 2023-12-21.) If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

Sudo 1.9.15p4

August 9, 2023

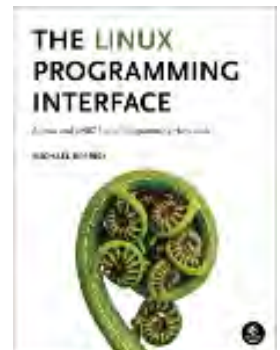
SUDO(8)

Pages that refer to this page: [homectl\(1\)](#), [journalctl\(1\)](#), [localectl\(1\)](#), [loginctl\(1\)](#), [machinectl\(1\)](#), [portablectl\(1\)](#), [setpriv\(1\)](#), [systemctl\(1\)](#), [systemd\(1\)](#), [systemd-analyze\(1\)](#), [systemd-ask-password\(1\)](#), [systemd-inhibit\(1\)](#), [systemd-nspawn\(1\)](#), [systemd-vmspawn\(1\)](#), [timedatectl\(1\)](#), [uid0\(1\)](#), [userdbctl\(1\)](#), [nsswitch.conf\(5\)](#), [credentials\(7\)](#), [systemd-tmpfiles\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



Another version of this page is provided by the [shadow-utils](#) project

su(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [SIGNALS](#) | [CONFIG FILES](#) | [EXIT STATUS](#) | [FILES](#) | [NOTES](#) | [HISTORY](#) | [SEE ALSO](#) | [REPORTING BUGS](#) | [AVAILABILITY](#)

 SU(1)

User Commands

SU(1)**NAME** [top](#)

`su` - run a command with substitute user and group ID

SYNOPSIS [top](#)

`su` [`options`] [`-`] [`user` [`argument...`]]

DESCRIPTION [top](#)

`su` allows commands to be run with a substitute user and group ID.

When called with no `user` specified, `su` defaults to running an interactive shell as `root`. When `user` is specified, additional `arguments` can be supplied, in which case they are passed to the shell.

For backward compatibility, `su` defaults to not change the current directory and to only set the environment variables **HOME** and **SHELL** (plus **USER** and **LOGNAME** if the target `user` is not root). It is recommended to always use the `--login` option (instead of its shortcut `-`) to avoid side effects caused by mixing environments.

This version of `su` uses PAM for authentication, account and

session management. Some configuration options found in other **su** implementations, such as support for a wheel group, have to be configured via PAM.

su is mostly designed for unprivileged users, the recommended solution for privileged users (e.g., scripts executed by root) is to use non-set-user-ID command `runuser(1)` that does not require authentication and provides separate PAM configuration. If the PAM session is not required at all then the recommended solution is to use command `setpriv(1)`.

Note that **su** in all cases uses PAM (`pam_getenvlist(3)`) to do the final environment modification. Command-line options such as `--login` and `--preserve-environment` affect the environment before it is modified by PAM.

Since version 2.38 **su** resets process resource limits `RLIMIT_NICE`, `RLIMIT_RTPRIO`, `RLIMIT_FSIZE`, `RLIMIT_AS` and `RLIMIT_NOFILE`.

OPTIONS [top](#)

- c, --command=command**
Pass *command* to the shell with the `-c` option.
- f, --fast**
Pass `-f` to the shell, which may or may not be useful, depending on the shell.
- g, --group=group**
Specify the primary group. This option is available to the root user only.
- G, --supp-group=group**
Specify a supplementary group. This option is available to the root user only. The first specified supplementary group is also used as a primary group if the option `--group` is not specified.
- , -l, --login**
Start the shell as a login shell with an environment similar to a real login:
 - clears all the environment variables except **TERM** and



variables specified by **--whitelist-environment**

- initializes the environment variables **HOME**, **SHELL**, **USER**, **LOGNAME**, and **PATH**
- changes to the target user's home directory
- sets `argv[0]` of the shell to '-' in order to make the shell a login shell

-m, -p, --preserve-environment

Preserve the entire environment, i.e., do not set **HOME**, **SHELL**, **USER** or **LOGNAME**. This option is ignored if the option **--login** is specified.

-P, --pty

Create a pseudo-terminal for the session. The independent terminal provides better security as the user does not share a terminal with the original session. This can be used to avoid **TIOCTI** ioctl terminal injection and other security attacks against terminal file descriptors. The entire session can also be moved to the background (e.g., **su --pty -username -c application &**). If the pseudo-terminal is enabled, then **su** works as a proxy between the sessions (sync stdin and stdout).

This feature is mostly designed for interactive sessions. If the standard input is not a terminal, but for example a pipe (e.g., **echo "date" | su --pty**), then the **ECHO** flag for the pseudo-terminal is disabled to avoid messy output.

-s, --shell=shell

Run the specified *shell* instead of the default. The shell to run is selected according to the following rules, in order:

- the shell specified with **--shell**
- the shell specified in the environment variable **SHELL**, if the **--preserve-environment** option is used
- the shell listed in the `passwd` entry of the target user
- `/bin/sh`



If the target user has a restricted shell (i.e., not listed in */etc/shells*), the **--shell** option and the **SHELL** environment variables are ignored unless the calling user is root.

--session-command=command

Same as **-c**, but do not create a new session. (Discouraged.)

-w, --whitelist-environment=list

Don't reset the environment variables specified in the comma-separated *list* when clearing the environment for **--login**. The whitelist is ignored for the environment variables **HOME**, **SHELL**, **USER**, **LOGNAME**, and **PATH**.

-h, --help

Display help text and exit.

-V, --version

Print version and exit.

SIGNALS

[top](#)

Upon receiving either **SIGINT**, **SIGQUIT** or **SIGTERM**, **su** terminates its child and afterwards terminates itself with the received signal. The child is terminated by **SIGTERM**, after unsuccessful attempt and 2 seconds of delay the child is killed by **SIGKILL**.

CONFIG FILES

[top](#)

su reads the */etc/default/su* and */etc/login.defs* configuration files. The following configuration items are relevant for **su**:

FAIL_DELAY (number)

Delay in seconds in case of an authentication failure. The number must be a non-negative integer.

ENV_PATH (string)

Defines the **PATH** environment variable for a regular user. The default value is */usr/local/bin:/bin:/usr/bin*.

ENV_ROOTPATH (string), **ENV_SUPATH** (string)

Defines the **PATH** environment variable for root. **ENV_SUPATH** takes precedence. The default value is



/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin.

ALWAYS_SET_PATH (boolean)

If set to *yes* and **--login** and **--preserve-environment** were not specified **su** initializes **PATH**.

The environment variable **PATH** may be different on systems where */bin* and */sbin* are merged into */usr*; this variable is also affected by the **--login** command-line option and the PAM system setting (e.g., `pam_env(8)`).

EXIT STATUS [top](#)

su normally returns the exit status of the command it executed. If the command was killed by a signal, **su** returns the number of the signal plus 128.

Exit status generated by **su** itself:

- 1
Generic error before executing the requested command
- 126
The requested command could not be executed
- 127
The requested command was not found

FILES [top](#)

- /etc/pam.d/su*
default PAM configuration file
- /etc/pam.d/su-l*
PAM configuration file if **--login** is specified
- /etc/default/su*
command specific logindef config file
- /etc/login.defs*
global logindef config file



NOTES [top](#)

For security reasons, **su** always logs failed log-in attempts to the *btmp* file, but it does not write to the *lastlog* file at all. This solution can be used to control **su** behavior by PAM configuration. If you want to use the [pam_lastlog\(8\)](#) module to print warning message about failed log-in attempts then [pam_lastlog\(8\)](#) has to be configured to update the *lastlog* file as well. For example by:

```
session required pam_lastlog.so nowtmp
```

HISTORY [top](#)

This **su** command was derived from coreutils' **su**, which was based on an implementation by David MacKenzie. The util-linux version has been refactored by Karel Zak.

SEE ALSO [top](#)

[setpriv\(1\)](#), [login.defs\(5\)](#), [shells\(5\)](#), [pam\(8\)](#), [runuser\(1\)](#)

REPORTING BUGS [top](#)

For bug reports, use the issue tracker at <https://github.com/util-linux/util-linux/issues>.

AVAILABILITY [top](#)

The **su** command is part of the util-linux package which can be downloaded from Linux Kernel Archive <<https://www.kernel.org/pub/linux/utils/util-linux/>>. This page is part of the *util-linux* (a random collection of Linux utilities) project. Information about the project can be found at <<https://www.kernel.org/pub/linux/utils/util-linux/>>. If you have a bug report for this manual page, send it to util-linux@vger.kernel.org. This page was obtained from the project's upstream Git repository <[git://git.kernel.org/pub/scm/utils/util-linux/util-linux.git](https://git.kernel.org/pub/scm/utils/util-linux/util-linux.git)> on 2023-12-22. (At that time, the date of the most recent commit



that was found in the repository was 2023-12-14.) If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

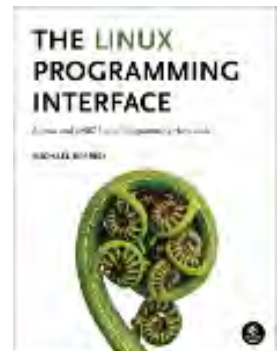
util-linux 2.39.594-1e0ad**2023-07-19****SU(1)**

Pages that refer to this page: [flock\(1\)](#), [homectl\(1\)](#), [login\(1\)](#), [login\(1@@shadow-utils\)](#), [machinectl\(1\)](#), [newgrp\(1\)](#), [runuser\(1\)](#), [setpriv\(1\)](#), [sg\(1\)](#), [updatedb\(1\)](#), [pam\(3\)](#), [pts\(4\)](#), [crontab\(5\)](#), [login.defs\(5\)](#), [passwd\(5\)](#), [passwd\(5@@shadow-utils\)](#), [shadow\(5\)](#), [suauth\(5\)](#), [credentials\(7\)](#), [environ\(7\)](#), [PAM\(8\)](#), [pam_rootok\(8\)](#), [pam_xauth\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



man(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [EXAMPLES](#) | [OVERVIEW](#) | [DEFAULTS](#) | [OPTIONS](#) | [EXIT STATUS](#) | [ENVIRONMENT](#) | [FILES](#) | [STANDARDS](#) | [SEE ALSO](#) | [HISTORY](#) | [BUGS](#) | [COLOPHON](#)

 MAN(1)

Manual pager utils

MAN(1)**NAME** [top](#)

`man` - an interface to the system reference manuals

SYNOPSIS [top](#)

```
man [man options] [[section] page ...] ...
man -k [apropos options] regex ...
man -K [man options] [section] term ...
man -f [whatis options] page ...
man -l [man options] file ...
man -w|-W [man options] page ...
```

DESCRIPTION [top](#)

`man` is the system's manual pager. Each *page* argument given to `man` is normally the name of a program, utility or function. The *manual page* associated with each of these arguments is then found and displayed. A *section*, if provided, will direct `man` to look only in that *section* of the manual. The default action is to search in all of the available *sections* following a pre-defined order (see **DEFAULTS**), and to show only the first *page* found, even if *page* exists in several *sections*.

The table below shows the *section* numbers of the manual followed by the types of pages they contain.



- 1 Executable programs or shell commands
- 2 System calls (functions provided by the kernel)
- 3 Library calls (functions within program libraries)
- 4 Special files (usually found in */dev*)
- 5 File formats and conventions, e.g. */etc/passwd*
- 6 Games
- 7 Miscellaneous (including macro packages and conventions),
e.g. [man\(7\)](#), [groff\(7\)](#), [man-pages\(7\)](#)
- 8 System administration commands (usually only for root)
- 9 Kernel routines [Non standard]

A manual *page* consists of several sections.

Conventional section names include **NAME**, **SYNOPSIS**, **CONFIGURATION**, **DESCRIPTION**, **OPTIONS**, **EXIT STATUS**, **RETURN VALUE**, **ERRORS**, **ENVIRONMENT**, **FILES**, **VERSIONS**, **STANDARDS**, **NOTES**, **BUGS**, **EXAMPLE**, **AUTHORS**, and **SEE ALSO**.

The following conventions apply to the **SYNOPSIS** section and can be used as a guide in other sections.

| | |
|---------------------------|--|
| bold text | type exactly as shown. |
| <i>italic text</i> | replace with appropriate argument. |
| [-abc] | any or all arguments within [] are optional. |
| -a -b | options delimited by cannot be used together. |
| <i>argument</i> ... | <i>argument</i> is repeatable. |
| [<i>expression</i>] ... | entire <i>expression</i> within [] is repeatable. |

Exact rendering may vary depending on the output device. For instance, man will usually not be able to render italics when running in a terminal, and will typically use underlined or coloured text instead.

The command or function illustration is a pattern that should match all possible invocations. In some cases it is advisable to illustrate several exclusive invocations as is shown in the **SYNOPSIS** section of this manual page.

EXAMPLES [top](#)



man *ls*

Display the manual page for the *item* (program) *ls*.

man *man.7*

Display the manual page for macro package *man* from section 7. (This is an alternative spelling of "**man 7 man**".)

man '*man(7)*'

Display the manual page for macro package *man* from section 7. (This is another alternative spelling of "**man 7 man**". It may be more convenient when copying and pasting cross-references to manual pages. Note that the parentheses must normally be quoted to protect them from the shell.)

man **-a** *intro*

Display, in succession, all of the available *intro* manual pages contained within the manual. It is possible to quit between successive displays or skip any of them.

man **-t** *bash* | *lpr -Pps*

Format the manual page for *bash* into the default **troff** or **groff** format and pipe it to the printer named *ps*. The default output for **groff** is usually PostScript. **man --help** should advise as to which processor is bound to the **-t** option.

man **-l -Tdvi** *./foo.1x.gz* > *./foo.1x.dvi*

This command will decompress and format the nroff source manual page *./foo.1x.gz* into a **device independent (dvi)** file. The redirection is necessary as the **-T** flag causes output to be directed to **stdout** with no pager. The output could be viewed with a program such as **xdvi** or further processed into PostScript using a program such as **dvips**.

man **-k** *printf*

Search the short descriptions and manual page names for the keyword *printf* as regular expression. Print out any matches. Equivalent to **apropos printf**.

man **-f** *smail*

Lookup the manual pages referenced by *smail* and print out the short descriptions of any found. Equivalent to **whatis smail**.



OVERVIEW [top](#)

Many options are available to **man** in order to give as much flexibility as possible to the user. Changes can be made to the search path, section order, output processor, and other behaviours and operations detailed below.

If set, various environment variables are interrogated to determine the operation of **man**. It is possible to set the "catch-all" variable **\$MANOPT** to any string in command line format, with the exception that any spaces used as part of an option's argument must be escaped (preceded by a backslash). **man** will parse **\$MANOPT** prior to parsing its own command line. Those options requiring an argument will be overridden by the same options found on the command line. To reset all of the options set in **\$MANOPT**, **-D** can be specified as the initial command line option. This will allow **man** to "forget" about the options specified in **\$MANOPT**, although they must still have been valid.

Manual pages are normally stored in [nroff\(1\)](#) format under a directory such as */usr/share/man*. In some installations, there may also be preformatted *cat pages* to improve performance. See [manpath\(5\)](#) for details of where these files are stored.

This package supports manual pages in multiple languages, controlled by your *locale*. If your system did not set this up for you automatically, then you may need to set **\$LC_MESSAGES**, **\$LANG**, or another system-dependent environment variable to indicate your preferred locale, usually specified in the **POSIX** format:

```
<Language>[_<territory>[.<character-set>[,<version>]]]
```

If the desired page is available in your *locale*, it will be displayed in lieu of the standard (usually American English) page.

If you find that the translations supplied with this package are not available in your native language and you would like to supply them, please contact the maintainer who will be coordinating such activity.

Individual manual pages are normally written and maintained by the maintainers of the program, function, or other topic that



they document, and are not included with this package. If you find that a manual page is missing or inadequate, please report that to the maintainers of the package in question.

For information regarding other features and extensions available with this manual pager, please read the documents supplied with the package.

DEFAULTS [top](#)

The order of sections to search may be overridden by the environment variable `$MANSECT` or by the **SECTION** directive in `/usr/local/etc/man_db.conf`. By default it is as follows:

```
1 n 1 8 3 0 2 3type 5 4 9 6 7
```

The formatted manual page is displayed using a *pager*. This can be specified in a number of ways, or else will fall back to a default (see option **-P** for details).

The filters are deciphered by a number of means. Firstly, the command line option **-p** or the environment variable `$MANROFFSEQ` is interrogated. If **-p** was not used and the environment variable was not set, the initial line of the `nroff` file is parsed for a preprocessor string. To contain a valid preprocessor string, the first line must resemble

```
'\" <string>
```

where **string** can be any combination of letters described by option **-p** below.

If none of the above methods provide any filter information, a default set is used.

A formatting pipeline is formed from the filters and the primary formatter (`nroff` or `[tg]roff` with **-t**) and executed. Alternatively, if an executable program `mandb_nfmt` (or `mandb_tfmt` with **-t**) exists in the man tree root, it is executed instead. It gets passed the manual source file, the preprocessor string, and optionally the device specified with **-T** or **-E** as arguments.



OPTIONS [top](#)

Non-argument options that are duplicated either on the command line, in `$MANOPT`, or both, are not harmful. For options that require an argument, each duplication will override the previous argument value.

General options

-C *file*, **--config-file=***file*

Use this user configuration file rather than the default of `~/.manpath`.

-d, **--debug**

Print debugging information.

-D, **--default**

This option is normally issued as the very first option and resets **man's** behaviour to its default. Its use is to reset those options that may have been set in `$MANOPT`. Any options that follow **-D** will have their usual effect.

--warnings[=*warnings*]

Enable warnings from *groff*. This may be used to perform sanity checks on the source text of manual pages.

warnings is a comma-separated list of warning names; if it is not supplied, the default is "mac". To disable a *groff* warning, prefix it with "!": for example,

--warnings=mac,!break enables warnings in the "mac" category and disables warnings in the "break" category.

See the "Warnings" node in **info groff** for a list of available warning names.

Main modes of operation

-f, **--whatis**

Approximately equivalent to **whatis**. Display a short description from the manual page, if available. See [whatis\(1\)](#) for details.

-k, **--apropos**

Approximately equivalent to **apropos**. Search the short manual page descriptions for keywords and display any matches. See [apropos\(1\)](#) for details.

-K, **--global-apropos**



Search for text in all manual pages. This is a brute-force search, and is likely to take some time; if you can, you should specify a section to reduce the number of pages that need to be searched. Search terms may be simple strings (the default), or regular expressions if the **--regex** option is used.

Note that this searches the *sources* of the manual pages, not the rendered text, and so may include false positives due to things like comments in source files, or false negatives due to things like hyphens being written as "\-" in source files. Searching the rendered text would be much slower.

-l, --local-file

Activate "local" mode. Format and display local manual files instead of searching through the system's manual collection. Each manual page argument will be interpreted as an nroff source file in the correct format. No cat file is produced. If '-' is listed as one of the arguments, input will be taken from stdin.

If this option is not used, then **man** will also fall back to interpreting manual page arguments as local file names if the argument contains a "/" character, since that is a good indication that the argument refers to a path on the file system.

-w, --where, --path, --location

Don't actually display the manual page, but do print the location of the source nroff file that would be formatted. If the **-a** option is also used, then print the locations of all source files that match the search criteria.

-W, --where-cat, --location-cat

Don't actually display the manual page, but do print the location of the preformatted cat file that would be displayed. If the **-a** option is also used, then print the locations of all preformatted cat files that match the search criteria.

If **-w** and **-W** are both used, then print both source file and cat file separated by a space. If all of **-w**, **-W**, and **-a** are used, then do this for each possible match.



-c, --catman

This option is not for general use and should only be used by the **catman** program.

-R *encoding*, --recode=*encoding*

Instead of formatting the manual page in the usual way, output its source converted to the specified *encoding*. If you already know the encoding of the source file, you can also use [manconv\(1\)](#) directly. However, this option allows you to convert several manual pages to a single encoding without having to explicitly state the encoding of each, provided that they were already installed in a structure similar to a manual page hierarchy.

Consider using [man-recode\(1\)](#) instead for converting multiple manual pages, since it has an interface designed for bulk conversion and so can be much faster.

Finding manual pages**-L *locale*, --locale=*locale***

man will normally determine your current locale by a call to the C function [setlocale\(3\)](#) which interrogates various environment variables, possibly including **\$LC_MESSAGES** and **\$LANG**. To temporarily override the determined value, use this option to supply a *locale* string directly to **man**. Note that it will not take effect until the search for pages actually begins. Output such as the help message will always be displayed in the initially determined locale.

-m *system*[,...], --systems=*system*[,...]

If this system has access to other operating systems' manual pages, they can be accessed using this option. To search for a manual page from NewOS's manual page collection, use the option **-m NewOS**.

The *system* specified can be a combination of comma delimited operating system names. To include a search of the native operating system's manual pages, include the system name **man** in the argument string. This option will override the **\$SYSTEM** environment variable.

-M *path*, --manpath=*path*

Specify an alternate manpath to use. By default, **man** uses **manpath** derived code to determine the path to search. This option overrides the **\$MANPATH** environment variable and causes option **-m** to be ignored.

A path specified as a manpath must be the root of a manual page hierarchy structured into sections as described in the man-db manual (under "The manual page system"). To view manual pages outside such hierarchies, see the **-l** option.

-S list, -s list, --sections=list

The given *list* is a colon- or comma-separated list of sections, used to determine which manual sections to search and in what order. This option overrides the **\$MANSECT** environment variable. (The **-s** spelling is for compatibility with System V.)

-e sub-extension, --extension=sub-extension

Some systems incorporate large packages of manual pages, such as those that accompany the **Tcl** package, into the main manual page hierarchy. To get around the problem of having two manual pages with the same name such as **exit(3)**, the **Tcl** pages were usually all assigned to section **1**. As this is unfortunate, it is now possible to put the pages in the correct section, and to assign a specific "extension" to them, in this case, **exit(3tcl)**. Under normal operation, **man** will display **exit(3)** in preference to **exit(3tcl)**. To negotiate this situation and to avoid having to know which section the page you require resides in, it is now possible to give **man** a *sub-extension* string indicating which package the page must belong to. Using the above example, supplying the option **-e tcl** to **man** will restrict the search to pages having an extension of ***tcl**.

-i, --ignore-case

Ignore case when searching for manual pages. This is the default.

-I, --match-case

Search for manual pages case-sensitively.

--regex



Show all pages with any part of either their names or their descriptions matching each *page* argument as a regular expression, as with [apropos\(1\)](#). Since there is usually no reasonable way to pick a "best" page when searching for a regular expression, this option implies **-a**.

--wildcard

Show all pages with any part of either their names or their descriptions matching each *page* argument using shell-style wildcards, as with [apropos\(1\) --wildcard](#). The *page* argument must match the entire name or description, or match on word boundaries in the description. Since there is usually no reasonable way to pick a "best" page when searching for a wildcard, this option implies **-a**.

--names-only

If the **--regex** or **--wildcard** option is used, match only page names, not page descriptions, as with [whatis\(1\)](#). Otherwise, no effect.

-a, --all

By default, **man** will exit after displaying the most suitable manual page it finds. Using this option forces **man** to display all the manual pages with names that match the search criteria.

-u, --update

This option causes **man** to update its database caches of installed manual pages. This is only needed in rare situations, and it is normally better to run [mandb\(8\)](#) instead.

--no-subpages

By default, **man** will try to interpret pairs of manual page names given on the command line as equivalent to a single manual page name containing a hyphen or an underscore. This supports the common pattern of programs that implement a number of subcommands, allowing them to provide manual pages for each that can be accessed using similar syntax as would be used to invoke the subcommands themselves. For example:

```
$ man -aw git diff
```




```
/usr/share/man/man1/git-diff.1.gz
```

To disable this behaviour, use the **--no-subpages** option.

```
$ man -aw --no-subpages git diff
/usr/share/man/man1/git.1.gz
/usr/share/man/man3/Git.3pm.gz
/usr/share/man/man1/diff.1.gz
```

Controlling formatted output

-P pager, --pager=pager

Specify which output pager to use. By default, **man** uses **less**, falling back to **cat** if **less** is not found or is not executable. This option overrides the **\$MANPAGER** environment variable, which in turn overrides the **\$PAGER** environment variable. It is not used in conjunction with **-f** or **-k**.

The value may be a simple command name or a command with arguments, and may use shell quoting (backslashes, single quotes, or double quotes). It may not use pipes to connect multiple commands; if you need that, use a wrapper script, which may take the file to display either as an argument or on standard input.

-r prompt, --prompt=prompt

If a recent version of **less** is used as the pager, **man** will attempt to set its prompt and some sensible options. The default prompt looks like

```
Manual page name(sec) line x
```

where *name* denotes the manual page name, *sec* denotes the section it was found under and *x* the current line number. This is achieved by using the **\$LESS** environment variable.

Supplying **-r** with a string will override this default. The string may contain the text **\$MAN_PN** which will be expanded to the name of the current manual page and its section name surrounded by "(" and ")". The string used to produce the default could be expressed as

```
\ Manual\ page\ \ $MAN_PN\ ?ltline\ %lt?L/%L.:
byte\ %bB?s/%s..?\ (END):?pB\ %pB\%..
```



(press h for help or q to quit)

It is broken into three lines here for the sake of readability only. For its meaning see the [less\(1\)](#) manual page. The prompt string is first evaluated by the shell. All double quotes, back-quotes and backslashes in the prompt must be escaped by a preceding backslash. The prompt string may end in an escaped \$ which may be followed by further options for less. By default **man** sets the **-ix8** options.

The **\$MANLESS** environment variable described below may be used to set a default prompt string if none is supplied on the command line.

-7, --ascii

When viewing a pure [ascii\(7\)](#) manual page on a 7 bit terminal or terminal emulator, some characters may not display correctly when using the [latin1\(7\)](#) device description with **GNU nroff**. This option allows pure *ascii* manual pages to be displayed in *ascii* with the *latin1* device. It will not translate any *latin1* text. The following table shows the translations performed: some parts of it may only be displayed properly when using **GNU nroff**'s [latin1\(7\)](#) device.

| Description | Octal | latin1 | ascii |
|------------------------|-------|--------|-------|
| continuation hyphen | 255 | - | - |
| bullet (middle dot) | 267 | • | o |
| acute accent | 264 | ´ | ' |
| multiplication sign | 327 | x | x |

If the *latin1* column displays correctly, your terminal may be set up for *latin1* characters and this option is not necessary. If the *latin1* and *ascii* columns are identical, you are reading this page using this option or **man** did not format this page using the *latin1* device description. If the *latin1* column is missing or corrupt, you may need to view manual pages with this option.



This option is ignored when using options **-t**, **-H**, **-T**, or **-Z** and may be useless for **nroff** other than **GNU's**.

-E *encoding*, --encoding=*encoding*

Generate output for a character encoding other than the default. For backward compatibility, *encoding* may be an **nroff** device such as **ascii**, **latin1**, or **utf8** as well as a true character encoding such as **UTF-8**.

--no-hyphenation, --nh

Normally, **nroff** will automatically hyphenate text at line breaks even in words that do not contain hyphens, if it is necessary to do so to lay out words on a line without excessive spacing. This option disables automatic hyphenation, so words will only be hyphenated if they already contain hyphens.

If you are writing a manual page and simply want to prevent **nroff** from hyphenating a word at an inappropriate point, do not use this option, but consult the **nroff** documentation instead; for instance, you can put **"\%"** inside a word to indicate that it may be hyphenated at that point, or put **"\%"** at the start of a word to prevent it from being hyphenated.

--no-justification, --nj

Normally, **nroff** will automatically justify text to both margins. This option disables full justification, leaving justified only to the left margin, sometimes called "ragged-right" text.

If you are writing a manual page and simply want to prevent **nroff** from justifying certain paragraphs, do not use this option, but consult the **nroff** documentation instead; for instance, you can use the **".na"**, **".nf"**, **".fi"**, and **".ad"** requests to temporarily disable adjusting and filling.

-p *string*, --preprocessor=*string*

Specify the sequence of preprocessors to run before **nroff** or **troff/groff**. Not all installations will have a full set of preprocessors. Some of the preprocessors and the letters used to designate them are: **eqn** (**e**), **grap** (**g**), **pic** (**p**), **tbl** (**t**), **vgrind** (**v**), **refer** (**r**). This option



overrides the `$MANROFFSEQ` environment variable. `zsoelim` is always run as the very first preprocessor.

-t, --troff

Use `groff -mandoc` to format the manual page to stdout. This option is not required in conjunction with `-H`, `-T`, or `-Z`.

-T[*device*], --troff-device[=*device*]

This option is used to change `groff` (or possibly `troff's`) output to be suitable for a device other than the default. It implies `-t`. Examples (provided with Groff-1.17) include `dvi`, `latin1`, `ps`, `utf8`, `X75` and `X100`.

-H[*browser*], --html[=*browser*]

This option will cause `groff` to produce HTML output, and will display that output in a web browser. The choice of browser is determined by the optional `browser` argument if one is provided, by the `$BROWSER` environment variable, or by a compile-time default if that is unset (usually `lynx`). This option implies `-t`, and will only work with **GNU troff**.

-X[*dpi*], --gxditview[=*dpi*]

This option displays the output of `groff` in a graphical window using the `gxditview` program. The `dpi` (dots per inch) may be 75, 75-12, 100, or 100-12, defaulting to 75; the -12 variants use a 12-point base font. This option implies `-T` with the `X75`, `X75-12`, `X100`, or `X100-12` device respectively.

-Z, --ditroff

`groff` will run `troff` and then use an appropriate post-processor to produce output suitable for the chosen device. If `groff -mandoc` is `groff`, this option is passed to `groff` and will suppress the use of a post-processor. It implies `-t`.

Getting help

-?, --help

Print a help message and exit.

--usage

Print a short usage message and exit.



-V, --version

Display version information.

EXIT STATUS [top](#)

- 0** Successful program execution.
- 1** Usage, syntax or configuration file error.
- 2** Operational error.
- 3** A child process returned a non-zero exit status.
- 16** At least one of the pages/files/keywords didn't exist or wasn't matched.

ENVIRONMENT [top](#)**MANPATH**

If **\$MANPATH** is set, its value is used as the path to search for manual pages.

See the **SEARCH PATH** section of [manpath\(5\)](#) for the default behaviour and details of how this environment variable is handled.

MANROFFOPT

Every time **man** invokes the formatter (**nroff**, **troff**, or **groff**), it adds the contents of **\$MANROFFOPT** to the formatter's command line.

MANROFFSEQ

If **\$MANROFFSEQ** is set, its value is used to determine the set of preprocessors to pass each manual page through. The default preprocessor list is system dependent.

MANSECT

If **\$MANSECT** is set, its value is a colon-delimited list of sections and it is used to determine which manual sections to search and in what order. The default is "1 n l 8 3 0 2 3type 5 4 9 6 7", unless overridden by the **SECTION** directive in */usr/local/etc/man_db.conf*.



MANPAGER, PAGER

If **\$MANPAGER** or **\$PAGER** is set (**\$MANPAGER** is used in preference), its value is used as the name of the program used to display the manual page. By default, **less** is used, falling back to **cat** if **less** is not found or is not executable.

The value may be a simple command name or a command with arguments, and may use shell quoting (backslashes, single quotes, or double quotes). It may not use pipes to connect multiple commands; if you need that, use a wrapper script, which may take the file to display either as an argument or on standard input.

MANLESS

If **\$MANLESS** is set, its value will be used as the default prompt string for the **less** pager, as if it had been passed using the **-r** option (so any occurrences of the text **\$MAN_PN** will be expanded in the same way). For example, if you want to set the prompt string unconditionally to “my prompt string”, set **\$MANLESS** to **'-Pmy prompt string'**. Using the **-r** option overrides this environment variable.

BROWSER

If **\$BROWSER** is set, its value is a colon-delimited list of commands, each of which in turn is used to try to start a web browser for **man --html**. In each command, **%s** is replaced by a filename containing the HTML output from **groff**, **%** is replaced by a single percent sign (%), and **%c** is replaced by a colon (:).

SYSTEM If **\$SYSTEM** is set, it will have the same effect as if it had been specified as the argument to the **-m** option.

MANOPT If **\$MANOPT** is set, it will be parsed prior to **man's** command line and is expected to be in a similar format. As all of the other **man** specific environment variables can be expressed as command line options, and are thus candidates for being included in **\$MANOPT** it is expected that they will become obsolete. N.B. All spaces that should be interpreted as part of an option's argument must be escaped.



MANWIDTH

If `$MANWIDTH` is set, its value is used as the line length for which manual pages should be formatted. If it is not set, manual pages will be formatted with a line length appropriate to the current terminal (using the value of `$COLUMNS`, and `ioctl(2)` if available, or falling back to 80 characters if neither is available). Cat pages will only be saved when the default formatting can be used, that is when the terminal line length is between 66 and 80 characters.

MAN_KEEP_FORMATTING

Normally, when output is not being directed to a terminal (such as to a file or a pipe), formatting characters are discarded to make it easier to read the result without special tools. However, if `$MAN_KEEP_FORMATTING` is set to any non-empty value, these formatting characters are retained. This may be useful for wrappers around `man` that can interpret formatting characters.

MAN_KEEP_STDERR

Normally, when output is being directed to a terminal (usually to a pager), any error output from the command used to produce formatted versions of manual pages is discarded to avoid interfering with the pager's display. Programs such as `groff` often produce relatively minor error messages about typographical problems such as poor alignment, which are unsightly and generally confusing when displayed along with the manual page. However, some users want to see them anyway, so, if `$MAN_KEEP_STDERR` is set to any non-empty value, error output will be displayed as usual.

MAN_DISABLE_SECCOMP

On Linux, `man` normally confines subprocesses that handle untrusted data using a `seccomp(2)` sandbox. This makes it safer to run complex parsing code over arbitrary manual pages. If this goes wrong for some reason unrelated to the content of the page being displayed, you can set `$MAN_DISABLE_SECCOMP` to any non-empty value to disable the sandbox.

PIPELINE_DEBUG

If the `$PIPELINE_DEBUG` environment variable is set to "1",



then **man** will print debugging messages to standard error describing each subprocess it runs.

LANG, LC_MESSAGES

Depending on system and implementation, either or both of **\$LANG** and **\$LC_MESSAGES** will be interrogated for the current message locale. **man** will display its messages in that locale (if available). See [setlocale\(3\)](#) for precise details.

FILES [top](#)

/usr/local/etc/man_db.conf
man-db configuration file.

/usr/share/man
A global manual page hierarchy.

STANDARDS [top](#)

POSIX.1-2001, POSIX.1-2008, POSIX.1-2017.

SEE ALSO [top](#)

[apropos\(1\)](#), [groff\(1\)](#), [less\(1\)](#), [manpath\(1\)](#), [nroff\(1\)](#), [troff\(1\)](#), [whatis\(1\)](#), [zsoelim\(1\)](#), [manpath\(5\)](#), [man\(7\)](#), [catman\(8\)](#), [mandb\(8\)](#)

Documentation for some packages may be available in other formats, such as **info(1)** or HTML.

HISTORY [top](#)

1990, 1991 – Originally written by John W. Eaton (jwe@che.utexas.edu).

Dec 23 1992: Rik Faith (faith@cs.unc.edu) applied bug fixes supplied by Willem Kasdorp (wkasdo@nikhef.nikef.nl).

30th April 1994 – 23rd February 2000: Wilf. (G.Wilford@ee.surrey.ac.uk) has been developing and maintaining this package with the help of a few dedicated people.



30th October 1996 – 30th March 2001: Fabrizio Polacco <fpolacco@debian.org> maintained and enhanced this package for the Debian project, with the help of all the community.

31st March 2001 – present day: Colin Watson <cjwatson@debian.org> is now developing and maintaining man-db.

BUGS [top](#)

<https://gitlab.com/man-db/man-db/-/issues>
<https://savannah.nongnu.org/bugs/?group=man-db>

COLOPHON [top](#)

This page is part of the *man-db* (manual pager suite) project. Information about the project can be found at <<http://www.nongnu.org/man-db/>>. If you have a bug report for this manual page, send it to man-db-devel@nongnu.org. This page was obtained from the project's upstream Git repository <<https://gitlab.com/cjwatson/man-db>> on 2023-12-22. (At that time, the date of the most recent commit that was found in the repository was 2023-12-18.) If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

2.12.0

2023-09-23

MAN(1)

Pages that refer to this page: [apropos\(1\)](#), [git\(1\)](#), [intro\(1\)](#), [lexgrog\(1\)](#), [manconv\(1\)](#), [manpath\(1\)](#), [man-recode\(1\)](#), [systemd-analyze\(1\)](#), [ul\(1\)](#), [whatis\(1\)](#), [zsoelim\(1\)](#), [manpath\(5\)](#), [environ\(7\)](#), [man\(7\)](#), [man-pages\(7\)](#), [catman\(8\)](#), [mandb\(8\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



info(1) - Linux man page

Name

info - read Info documents

Synopsis

info [*OPTION*]... [*MENU-ITEM*...]

Description

Read documentation in Info format.

Options

- k, --apropos=STRING**
look up STRING in all indices of all manuals.
- d, --directory=DIR**
add DIR to INFOPATH.
- dribble=FILENAME**
remember user keystrokes in FILENAME.
- f, --file=FILENAME**
specify Info file to visit.
- h, --help**
display this help and exit.
- index-search=STRING**
go to node pointed by index entry STRING.
- n, --node=NODENAME**
specify nodes in first visited Info file.
- o, --output=FILENAME**
output selected nodes to FILENAME.
- R, --raw-escapes**
output "raw" ANSI escapes (default).
- no-raw-escapes**
output escapes as literal text.
- restore=FILENAME**



read initial keystrokes from FILENAME.

-O, --show-options, --usage

go to command-line options node.

--subnodes

recursively output menu items.

--vi-keys

use vi-like and less-like key bindings.

--version

display version information and exit.

-w, --where, --location

print physical location of Info file.

The first non-option argument, if present, is the menu entry to start from; it is searched for in all 'dir' files along INFOPATH. If it is not present, info merges all 'dir' files and shows the result. Any remaining arguments are treated as the names of menu items relative to the initial node visited.

For a summary of key bindings, type h within Info.

Examples

info

show top-level dir menu

info info

show the general manual for Info readers

info info-stdn

show the manual specific to this Info program

info emacs

start at emacs node from top-level dir

info emacs buffers

start at buffers node within emacs manual

info **--show-options** emacs

start at node with emacs' command line options

info **--subnodes -o** out.txt emacs

dump entire manual to out.txt

info **-f** ./foo.info

show file ./foo.info, not searching dir



Reporting Bugs

Email bug reports to bug-texinfo@gnu.org, general questions and discussion to help-texinfo@gnu.org. Texinfo home page: <http://www.gnu.org/software/texinfo/>

Copyright

Copyright © 2008 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<http://gnu.org/licenses/gpl.html>>

This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

Referenced By

[amd\(8\)](#), [amd.conf\(5\)](#), [amq\(8\)](#), [automount2amd\(8\)](#), [eplain\(1\)](#), [fixmount\(8\)](#), [fsinfo\(8\)](#), [groff\(1\)](#), [groff\(7\)](#), [groff_diff\(7\)](#), [groff_tmac\(5\)](#), [groff_trace\(7\)](#), [hlfsd\(8\)](#), [info\(5\)](#), [ldp\(7\)](#), [mk-amd-map\(8\)](#), [parted\(8\)](#), [pawd\(1\)](#), [texinfo\(5\)](#), [troff\(1\)](#), [wire-test\(8\)](#), [x11vnc\(1\)](#)



ping(8) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [IPV6 LINK-LOCAL DESTINATIONS](#) | [ICMP PACKET DETAILS](#) | [DUPLICATE AND DAMAGED PACKETS](#) | [ID COLLISIONS](#) | [TRYING DIFFERENT DATA PATTERNS](#) | [TTL DETAILS](#) | [BUGS](#) | [SEE ALSO](#) | [HISTORY](#) | [SECURITY](#) | [AVAILABILITY](#) | [COLOPHON](#)

 PING(8)

iputils

PING(8)**NAME** [top](#)

ping - send ICMP ECHO_REQUEST to network hosts

SYNOPSIS [top](#)

```
ping [-aAbBdCDfhHLnOqrRUvV46] [-c count] [-e identifier]
     [-F flowlabel] [-i interval] [-I interface] [-l preload]
     [-m mark] [-M pmtudisc_option] [-N nodeinfo_option]
     [-w deadline] [-W timeout] [-p pattern] [-Q tos]
     [-s packetsize] [-S sndbuf] [-t ttl] [-T timestamp_option]
     [hop...] {destination}
```

DESCRIPTION [top](#)

ping uses the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a host or gateway. ECHO_REQUEST datagrams (“pings”) have an IP and ICMP header, followed by a struct timeval and then an arbitrary number of “pad” bytes used to fill out the packet.

ping works with both IPv4 and IPv6. Using only one of them explicitly can be enforced by specifying **-4** or **-6**.



ping can also send IPv6 Node Information Queries (RFC4620). Intermediate *hops* may not be allowed, because IPv6 source routing was deprecated (RFC5095).

OPTIONS

[top](#)

-4

Use IPv4 only.

-6

Use IPv6 only.

-a

Audible ping.

-A

Adaptive ping. Interpacket interval adapts to round-trip time, so that effectively not more than one (or more, if preload is set) unanswered probe is present in the network. Minimal interval is 200msec unless super-user. On networks with low RTT this mode is essentially equivalent to flood mode.

-b

Allow pinging a broadcast address.

-B

Do not allow **ping** to change source address of probes. The address is bound to one selected when **ping** starts.

-c *count*

Stop after sending *count* ECHO_REQUEST packets. With *deadline* option, **ping** waits for *count* ECHO_REPLY packets, until the timeout expires.

-C

Call connect() syscall on socket creation.

-d

Set the SO_DEBUG option on the socket being used. Essentially, this socket option is not used by Linux kernel.

-D



Print timestamp (unix time + microseconds as in `gettimeofday`) before each line.

-e *identifier*

Set the identification field of ECHO_REQUEST. Value 0 implies using *raw socket* (not supported on *ICMP datagram socket*). The value of the field may be printed with **-v** option.

-f

Flood ping. For every ECHO_REQUEST sent a period "." is printed, while for every ECHO_REPLY received a backspace is printed. This provides a rapid display of how many packets are being dropped. If interval is not given, it sets interval to zero and outputs packets as fast as they come back or one hundred times per second, whichever is more. Only the super-user may use this option with zero interval.

-F *flow label*

IPv6 only. Allocate and set 20 bit flow label (in hex) on echo request packets. If value is zero, kernel allocates random flow label.

-h

Show help.

-H

Force DNS name resolution for the output. Useful for numeric destination, or **-f** option, which by default do not perform it. Override previously defined **-n** option.

-i *interval*

Wait *interval* seconds between sending each packet. Real number allowed with dot as a decimal separator (regardless locale setup). The default is to wait for one second between each packet normally, or not to wait in flood mode. Only super-user may set interval to values less than 2 ms. Broadcast and multicast ping have even higher limitation for regular user: minimum is 1 sec.

-I *interface*

interface is either an address, an interface name or a VRF name. If *interface* is an address, it sets source address to specified interface address. If *interface* is an interface name, it sets source interface to specified interface. If



interface is a VRF name, each packet is routed using the corresponding routing table; in this case, the **-I** option can be repeated to specify a source address. NOTE: For IPv6, when doing ping to a link-local scope address, link specification (by the '%'-notation in *destination*, or by this option) can be used but it is no longer required.

-l *preload*

If *preload* is specified, **ping** sends that many packets not waiting for reply. Only the super-user may select preload more than 3.

-L

Suppress loopback of multicast packets. This flag only applies if the ping destination is a multicast address.

-m *mark*

use *mark* to tag the packets going out. This is useful for variety of reasons within the kernel such as using policy routing to select specific outbound processing.

-M *pmtudisc_opt*

Select Path MTU Discovery strategy. *pmtudisc_option* may be either *do* (set DF flag but subject to PMTU checks by kernel, packets too large will be rejected), *want* (do PMTU discovery, fragment locally when packet size is large), *probe* (set DF flag and bypass PMTU checks, useful for probing), or *dont* (do not set DF flag).

-N *nodeinfo_option*

IPv6 only. Send IPv6 Node Information Queries (RFC4620), instead of Echo Request. CAP_NET_RAW capability is required.

help

Show help for NI support.

name

Queries for Node Names.

ipv6

Queries for IPv6 Addresses. There are several IPv6 specific flags.

ipv6-global



Request IPv6 global-scope addresses.

ipv6-sitelocal

Request IPv6 site-local addresses.

ipv6-linklocal

Request IPv6 link-local addresses.

ipv6-all

Request IPv6 addresses on other interfaces.

ipv4

Queries for IPv4 Addresses. There is one IPv4 specific flag.

ipv4-all

Request IPv4 addresses on other interfaces.

subject-ipv6=*ipv6addr*

IPv6 subject address.

subject-ipv4=*ipv4addr*

IPv4 subject address.

subject-name=*nodename*

Subject name. If it contains more than one dot, fully-qualified domain name is assumed.

subject-fqdn=*nodename*

Subject name. Fully-qualified domain name is always assumed.

-n

Numeric output only. No attempt will be made to lookup symbolic names for host addresses (no reverse DNS resolution). This is the default for numeric destination or **-f** option. Override previously defined **-H** option.

-O

Report outstanding ICMP ECHO reply before sending next packet. This is useful together with the timestamp **-D** to log output to a diagnostic file and search for missing answers.

-p *pattern*

You may specify up to 16 “pad” bytes to fill out the packet you send. This is useful for diagnosing data-dependent problems in a network. For example, **-p ff** will cause the sent packet to be filled with all ones.

-q

Quiet output. Nothing is displayed except the summary lines at startup time and when finished.

-Q tos

Set Quality of Service -related bits in ICMP datagrams. *tos* can be decimal (**ping** only) or hex number.

In RFC2474, these fields are interpreted as 8-bit Differentiated Services (DS), consisting of: bits 0-1 (2 lowest bits) of separate data, and bits 2-7 (highest 6 bits) of Differentiated Services Codepoint (DSCP). In RFC2481 and RFC3168, bits 0-1 are used for ECN.

Historically (RFC1349, obsoleted by RFC2474), these were interpreted as: bit 0 (lowest bit) for reserved (currently being redefined as congestion control), 1-4 for Type of Service and bits 5-7 (highest bits) for Precedence.

-r

Bypass the normal routing tables and send directly to a host on an attached interface. If the host is not on a directly-attached network, an error is returned. This option can be used to ping a local host through an interface that has no route through it provided the option **-I** is also used.

-R

ping only. Record route. Includes the RECORD_ROUTE option in the ECHO_REQUEST packet and displays the route buffer on returned packets. Note that the IP header is only large enough for nine such routes. Many hosts ignore or discard this option.

-s packetsize

Specifies the number of data bytes to be sent. The default is 56, which translates into 64 ICMP data bytes when combined with the 8 bytes of ICMP header data.

-S sndbuf

Set socket sndbuf. If not specified, it is selected to buffer not more than one packet.

-t *tll*

ping only. Set the IP Time to Live.

-T *timestamp option*

Set special IP timestamp options. *timestamp option* may be either *tsonly* (only timestamps), *tsandaddr* (timestamps and addresses) or *tsprespec host1 [host2 [host3 [host4]]]* (timestamp prespecified hops).

-U

Print full user-to-user latency (the old behaviour). Normally **ping** prints network round trip time, which can be different f.e. due to DNS failures.

-v

Verbose output. Do not suppress DUP replies when pinging multicast address.

-V

Show version and exit.

-w *deadline*

Specify a timeout, in seconds, before **ping** exits regardless of how many packets have been sent or received. In this case **ping** does not stop after *count* packet are sent, it waits either for *deadline* expire or until *count* probes are answered or for some error notification from network.

-W *timeout*

Time to wait for a response, in seconds. The option affects only timeout in absence of any responses, otherwise **ping** waits for two RTTs. Real number allowed with dot as a decimal separator (regardless locale setup). 0 means infinite timeout.

When using **ping** for fault isolation, it should first be run on the local host, to verify that the local network interface is up and running. Then, hosts and gateways further and further away should be “pinged”. Round-trip times and packet loss statistics are computed. If duplicate packets are received, they are not included in the packet loss calculation, although the round trip



time of these packets is used in calculating the minimum/average/maximum/mdev round-trip time numbers.

Population standard deviation (mdev), essentially an average of how far each ping RTT is from the mean RTT. The higher mdev is, the more variable the RTT is (over time). With a high RTT variability, you will have speed issues with bulk transfers (they will take longer than is strictly speaking necessary, as the variability will eventually cause the sender to wait for ACKs) and you will have middling to poor VoIP quality.

When the specified number of packets have been sent (and received) or if the program is terminated with a SIGINT, a brief summary is displayed. Shorter current statistics can be obtained without termination of process with signal SIGQUIT.

If **ping** does not receive any reply packets at all it will exit with code 1. If a packet *count* and *deadline* are both specified, and fewer than *count* packets are received by the time the *deadline* has arrived, it will also exit with code 1. On other error it exits with code 2. Otherwise it exits with code 0. This makes it possible to use the exit code to see if a host is alive or not.

This program is intended for use in network testing, measurement and management. Because of the load it can impose on the network, it is unwise to use **ping** during normal operations or from automated scripts.

IPV6 LINK-LOCAL DESTINATIONS [top](#)

For IPv6, when the destination address has link-local scope and **ping** is using *ICMP datagram sockets*, the output interface must be specified. When **ping** is using *raw sockets*, it is not strictly necessary to specify the output interface but it should be done to avoid ambiguity when there are multiple possible output interfaces.

There are two ways to specify the output interface:

- using the *% notation*
The destination address is postfixed with *%* and the output interface name or ifindex, for example:

```
ping fe80::5054:ff:fe70:67bc%eth0
```

```
ping fe80::5054:ff:fe70:67bc%2
```

- using the *-I option*

When using *ICMP datagram sockets*, this method is supported since the following kernel versions: 5.17, 5.15.19, 5.10.96, 5.4.176, 4.19.228, 4.14.265. Also it is not supported on musl libc.

ICMP PACKET DETAILS [top](#)

An IP header without options is 20 bytes. An ICMP ECHO_REQUEST packet contains an additional 8 bytes worth of ICMP header followed by an arbitrary amount of data. When a *packetsize* is given, this indicates the size of this extra piece of data (the default is 56). Thus the amount of data received inside of an IP packet of type ICMP ECHO_REPLY will always be 8 bytes more than the requested data space (the ICMP header).

If the data space is at least of size of struct `timeval` **ping** uses the beginning bytes of this space to include a timestamp which it uses in the computation of round trip times. If the data space is shorter, no round trip times are given.

DUPLICATE AND DAMAGED PACKETS [top](#)

ping will report duplicate and damaged packets. Duplicate packets should never occur, and seem to be caused by inappropriate link-level retransmissions. Duplicates may occur in many situations and are rarely (if ever) a good sign, although the presence of low levels of duplicates may not always be cause for alarm.

Damaged packets are obviously serious cause for alarm and often indicate broken hardware somewhere in the **ping** packet's path (in the network or in the hosts).

ID COLLISIONS [top](#)



Unlike TCP and UDP, which use port to uniquely identify the recipient to deliver data, ICMP uses identifier field (ID) for identification. Therefore, if on the same machine, at the same time, two ping processes use the same ID, echo reply can be delivered to a wrong recipient. This is a known problem due to the limited size of the 16-bit ID field. That is a historical limitation of the protocol that cannot be fixed at the moment unless we encode an ID into the ping packet payload. **ping** prints *DIFFERENT ADDRESS* error and packet loss is negative.

ping uses PID to get unique number. The default value of */proc/sys/kernel/pid_max* is 32768. On the systems that use ping heavily and with *pid_max* greater than 65535 collisions are bound to happen.

TRYING DIFFERENT DATA PATTERNS [top](#)

The (inter)network layer should never treat packets differently depending on the data contained in the data portion. Unfortunately, data-dependent problems have been known to sneak into networks and remain undetected for long periods of time. In many cases the particular pattern that will have problems is something that doesn't have sufficient “transitions”, such as all ones or all zeros, or a pattern right at the edge, such as almost all zeros. It isn't necessarily enough to specify a data pattern of all zeros (for example) on the command line because the pattern that is of interest is at the data link level, and the relationship between what you type and what the controllers transmit can be complicated.

This means that if you have a data-dependent problem you will probably have to do a lot of testing to find it. If you are lucky, you may manage to find a file that either can't be sent across your network or that takes much longer to transfer than other similar length files. You can then examine this file for repeated patterns that you can test using the **-p** option of **ping**.

TTL DETAILS [top](#)

The TTL value of an IP packet represents the maximum number of IP routers that the packet can go through before being thrown away. In current practice you can expect each router in the Internet to



decrement the TTL field by exactly one.

The TTL field for TCP packets may take various values. The maximum possible value of this field is 255, a recommended initial value is 64. For more information, see the TCP/Lower-Level Interface section of RFC9293.

In normal operation **ping** prints the TTL value from the packet it receives. When a remote system receives a ping packet, it can do one of three things with the TTL field in its response:

- Not change it; this is what Berkeley Unix systems did before the 4.3BSD Tahoe release. In this case the TTL value in the received packet will be 255 minus the number of routers in the round-trip path.
- Set it to 255; this is what current Berkeley Unix systems do. In this case the TTL value in the received packet will be 255 minus the number of routers in the path **from** the remote system **to** the **pinging** host.
- Set it to some other value. Some machines use the same value for ICMP packets that they use for TCP packets, for example either 30 or 60. Others may use completely wild values.

BUGS

[top](#)

- Many Hosts and Gateways ignore the RECORD_ROUTE option.
- The maximum IP header length is too small for options like RECORD_ROUTE to be completely useful. There's not much that can be done about this, however.
- Flood pingging is not recommended in general, and flood pingging the broadcast address should only be done under very controlled conditions.

SEE ALSO

[top](#)

[ip\(8\)](#), [ss\(8\)](#).



HISTORY [top](#)

The **ping** command appeared in 4.3BSD.

The version described here is its descendant specific to Linux.

As of version s20150815, the **ping6** binary doesn't exist anymore. It has been merged into **ping**. Creating a symlink named **ping6** pointing to **ping** will result in the same functionality as before.

SECURITY [top](#)

ping requires CAP_NET_RAW capability to be executed 1) if the program is used for non-echo queries (see **-N** option) or when the identification field set to 0 for ECHO_REQUEST (see **-e**), or 2) if kernel does not support ICMP datagram sockets, or 3) if the user is not allowed to create an ICMP echo socket. The program may be used as set-uid root.

AVAILABILITY [top](#)

ping is part of *iputils* package.

COLOPHON [top](#)

This page is part of the *iputils* (IP utilities) project. Information about the project can be found at <http://www.skbuff.net/iputils/>. If you have a bug report for this manual page, send it to yoshfuji@skbuff.net, netdev@vger.kernel.org. This page was obtained from the project's upstream Git repository <https://github.com/iputils/iputils.git> on 2023-12-22. (At that time, the date of the most recent commit that was found in the repository was 2023-12-22.) If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

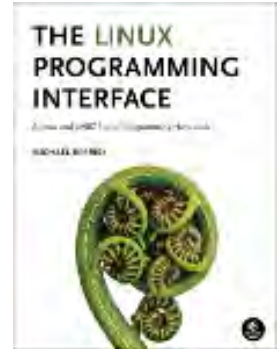


Pages that refer to this page: [arping\(8\)](#), [clockdiff\(8\)](#), [tracepath\(8\)](#), [traceroute\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



wget(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [ENVIRONMENT](#) | [EXIT STATUS](#) | [FILES](#) | [BUGS](#) | [SEE ALSO](#) | [AUTHOR](#) | [COPYRIGHT](#) | [COLOPHON](#)

 WGET(1)

GNU Wget

WGET(1)**NAME** [top](#)

Wget - The non-interactive network downloader.

SYNOPSIS [top](#)

```
wget [option]... [URL]...
```

DESCRIPTION [top](#)

GNU Wget is a free utility for non-interactive download of files from the Web. It supports HTTP, HTTPS, and FTP protocols, as well as retrieval through HTTP proxies.

Wget is non-interactive, meaning that it can work in the background, while the user is not logged on. This allows you to start a retrieval and disconnect from the system, letting Wget finish the work. By contrast, most of the Web browsers require constant user's presence, which can be a great hindrance when transferring a lot of data.

Wget can follow links in HTML, XHTML, and CSS pages, to create local versions of remote web sites, fully recreating the directory structure of the original site. This is sometimes referred to as "recursive downloading." While doing that, Wget respects the Robot Exclusion Standard (*/robots.txt*). Wget can be instructed to convert the links in downloaded files to point at the local files, for offline viewing.

Wget has been designed for robustness over slow or unstable network connections; if a download fails due to a network problem, it will keep retrying until the whole file has been

retrieved. If the server supports regetting, it will instruct the server to continue the download from where it left off.

OPTIONS [top](#)

Option Syntax

Since Wget uses GNU getopt to process command-line arguments, every option has a long form along with the short one. Long options are more convenient to remember, but take time to type. You may freely mix different option styles, or specify options after the command-line arguments. Thus you may write:

```
wget -r --tries=10 http://fly.srk.fer.hr/ -o log
```

The space between the option accepting an argument and the argument may be omitted. Instead of **-o log** you can write **-olog**.

You may put several options that do not require arguments together, like:

```
wget -drc <URL>
```

This is completely equivalent to:

```
wget -d -r -c <URL>
```

Since the options can be specified after the arguments, you may terminate them with **--**. So the following will try to download URL **-x**, reporting failure to *Log*:

```
wget -o log -- -x
```

The options that accept comma-separated lists all respect the convention that specifying an empty list clears its value. This can be useful to clear the *.wgetrc* settings. For instance, if your *.wgetrc* sets "exclude_directories" to */cgi-bin*, the following example will first reset it, and then set it to exclude */~nobody* and */~somebody*. You can also clear the lists in *.wgetrc*.

```
wget -X "" -X /~nobody,/~somebody
```

Most options that do not accept arguments are *boolean* options, so named because their state can be captured with a yes-or-no ("boolean") variable. For example, **--follow-ftp** tells Wget to follow FTP links from HTML files and, on the other hand, **--no-glob** tells it not to perform file globbing on FTP URLs. A boolean option is either *affirmative* or *negative* (beginning with **--no**). All such options share several properties.



Unless stated otherwise, it is assumed that the default behavior is the opposite of what the option accomplishes. For example, the documented existence of **--follow-ftp** assumes that the default is to *not* follow FTP links from HTML pages.

Affirmative options can be negated by prepending the **--no-** to the option name; negative options can be negated by omitting the **--no-** prefix. This might seem superfluous---if the default for an affirmative option is to not do something, then why provide a way to explicitly turn it off? But the startup file may in fact change the default. For instance, using "follow_ftp = on" in *.wgetrc* makes Wget *follow* FTP links by default, and using **--no-follow-ftp** is the only way to restore the factory default from the command line.

Basic Startup Options

-V

--version

Display the version of Wget.

-h

--help

Print a help message describing all of Wget's command-line options.

-b

--background

Go to background immediately after startup. If no output file is specified via the **-o**, output is redirected to *wget-Log*.

-e *command*

--execute *command*

Execute *command* as if it were a part of *.wgetrc*. A command thus invoked will be executed *after* the commands in *.wgetrc*, thus taking precedence over them. If you need to specify more than one wgetrc command, use multiple instances of **-e**.

Logging and Input File Options

-o *logfile*

--output-file=logfile

Log all messages to *logfile*. The messages are normally reported to standard error.

-a *logfile*

--append-output=logfile

Append to *logfile*. This is the same as **-o**, only it appends to *logfile* instead of overwriting the old log file. If *logfile* does not exist, a new file is created.



-d

--debug

Turn on debug output, meaning various information important to the developers of Wget if it does not work properly. Your system administrator may have chosen to compile Wget without debug support, in which case **-d** will not work. Please note that compiling with debug support is always safe---Wget compiled with the debug support will *not* print any debug info unless requested with **-d**.

-q

--quiet

Turn off Wget's output.

-v

--verbose

Turn on verbose output, with all the available data. The default output is verbose.

-nv

--no-verbose

Turn off verbose without being completely quiet (use **-q** for that), which means that error messages and basic information still get printed.

--report-speed=type

Output bandwidth as *type*. The only accepted value is **bits**.

-i file

--input-file=file

Read URLs from a local or external *file*. If **-** is specified as *file*, URLs are read from the standard input. (Use **./-** to read from a file literally named **-**.)

If this function is used, no URLs need be present on the command line. If there are URLs both on the command line and in an input file, those on the command lines will be the first ones to be retrieved. If **--force-html** is not specified, then *file* should consist of a series of URLs, one per line.

However, if you specify **--force-html**, the document will be regarded as **html**. In that case you may have problems with relative links, which you can solve either by adding "**<base href="url">**" to the documents or by specifying **--base=url** on the command line.

If the *file* is an external one, the document will be automatically treated as **html** if the Content-Type matches



text/html. Furthermore, the *file*'s location will be implicitly used as base href if none was specified.

--input-metalink=*file*

Downloads files covered in local Metalink *file*. Metalink version 3 and 4 are supported.

--keep-badhash

Keeps downloaded Metalink's files with a bad hash. It appends `.badhash` to the name of Metalink's files which have a checksum mismatch, except without overwriting existing files.

--metalink-over-http

Issues HTTP HEAD request instead of GET and extracts Metalink metadata from response headers. Then it switches to Metalink download. If no valid Metalink metadata is found, it falls back to ordinary HTTP download. Enables **Content-Type: application/metalink4+xml** files download/processing.

--metalink-index=*number*

Set the Metalink **application/metalink4+xml** metaurl ordinal NUMBER. From 1 to the total number of "application/metalink4+xml" available. Specify 0 or **inf** to choose the first good one. Metaurls, such as those from a **--metalink-over-http**, may have been sorted by priority key's value; keep this in mind to choose the right NUMBER.

--preferred-location

Set preferred location for Metalink resources. This has effect if multiple resources with same priority are available.

--xattr

Enable use of file system's extended attributes to save the original URL and the Referer HTTP header value if used.

Be aware that the URL might contain private information like access tokens or credentials.

-F

--force-html

When input is read from a file, force it to be treated as an HTML file. This enables you to retrieve relative links from existing HTML files on your local disk, by adding "`<base href=url>`" to HTML, or using the **--base** command-line option.

-B *URL*

--base=*URL*

Resolves relative links using *URL* as the point of reference,



when reading links from an HTML file specified via the **-i/--input-file** option (together with **--force-html**, or when the input file was fetched remotely from a server describing it as HTML). This is equivalent to the presence of a "BASE" tag in the HTML input file, with *URL* as the value for the "href" attribute.

For instance, if you specify **http://foo/bar/a.html** for *URL*, and Wget reads **../baz/b.html** from the input file, it would be resolved to **http://foo/baz/b.html** .

--config=FILE

Specify the location of a startup file you wish to use instead of the default one(s). Use **--no-config** to disable reading of config files. If both **--config** and **--no-config** are given, **--no-config** is ignored.

--rejected-log=logfile

Logs all URL rejections to *logfile* as comma separated values. The values include the reason of rejection, the URL and the parent URL it was found in.

Download Options

--bind-address=ADDRESS

When making client TCP/IP connections, bind to *ADDRESS* on the local machine. *ADDRESS* may be specified as a hostname or IP address. This option can be useful if your machine is bound to multiple IPs.

--bind-dns-address=ADDRESS

[libcares only] This address overrides the route for DNS requests. If you ever need to circumvent the standard settings from `/etc/resolv.conf`, this option together with **--dns-servers** is your friend. *ADDRESS* must be specified either as IPv4 or IPv6 address. Wget needs to be built with libcares for this option to be available.

--dns-servers=ADDRESSES

[libcares only] The given address(es) override the standard nameserver addresses, e.g. as configured in `/etc/resolv.conf`. *ADDRESSES* may be specified either as IPv4 or IPv6 addresses, comma-separated. Wget needs to be built with libcares for this option to be available.

-t number

--tries=number

Set number of tries to *number*. Specify `0` or **inf** for infinite retrying. The default is to retry 20 times, with the exception of fatal errors like "connection refused" or "not found" (404), which are not retried.



-O *file***--output-document=***file*

The documents will not be written to the appropriate files, but all will be concatenated together and written to *file*. If *-* is used as *file*, documents will be printed to standard output, disabling link conversion. (Use *./-* to print to a file literally named *.-*.)

Use of **-O** is *not* intended to mean simply "use the name *file* instead of the one in the URL;" rather, it is analogous to shell redirection: `wget -O file http://foo` is intended to work like `wget -O - http://foo > file`; *file* will be truncated immediately, and *all* downloaded content will be written there.

For this reason, **-N** (for timestamp-checking) is not supported in combination with **-O**: since *file* is always newly created, it will always have a very new timestamp. A warning will be issued if this combination is used.

Similarly, using **-r** or **-p** with **-O** may not work as you expect: Wget won't just download the first file to *file* and then download the rest to their normal names: *all* downloaded content will be placed in *file*. This was disabled in version 1.11, but has been reinstated (with a warning) in 1.11.2, as there are some cases where this behavior can actually have some use.

A combination with **-nc** is only accepted if the given output file does not exist.

Note that a combination with **-k** is only permitted when downloading a single document, as in that case it will just convert all relative URIs to external ones; **-k** makes no sense for multiple URIs when they're all being downloaded to a single file; **-k** can be used only when the output is a regular file.

-nc**--no-clobber**

If a file is downloaded more than once in the same directory, Wget's behavior depends on a few options, including **-nc**. In certain cases, the local file will be *clobbered*, or overwritten, upon repeated download. In other cases it will be preserved.

When running Wget without **-N**, **-nc**, **-r**, or **-p**, downloading the same file in the same directory will result in the original copy of *file* being preserved and the second copy being named



file.1. If that file is downloaded yet again, the third copy will be named *file.2*, and so on. (This is also the behavior with **-nd**, even if **-r** or **-p** are in effect.) When **-nc** is specified, this behavior is suppressed, and Wget will refuse to download newer copies of *file*. Therefore, "**no-clobber**" is actually a misnomer in this mode--it's not clobbering that's prevented (as the numeric suffixes were already preventing clobbering), but rather the multiple version saving that's prevented.

When running Wget with **-r** or **-p**, but without **-N**, **-nd**, or **-nc**, re-downloading a file will result in the new copy simply overwriting the old. Adding **-nc** will prevent this behavior, instead causing the original version to be preserved and any newer copies on the server to be ignored.

When running Wget with **-N**, with or without **-r** or **-p**, the decision as to whether or not to download a newer copy of a file depends on the local and remote timestamp and size of the file. **-nc** may not be specified at the same time as **-N**.

A combination with **-O/--output-document** is only accepted if the given output file does not exist.

Note that when **-nc** is specified, files with the suffixes **.html** or **.htm** will be loaded from the local disk and parsed as if they had been retrieved from the Web.

--backups=backups

Before (over)writing a file, back up an existing file by adding a **.1** suffix (**_1** on VMS) to the file name. Such backup files are rotated to **.2**, **.3**, and so on, up to *backups* (and lost beyond that).

--no-netrc

Do not try to obtain credentials from **.netrc** file. By default **.netrc** file is searched for credentials in case none have been passed on command line and authentication is required.

-c

--continue

Continue getting a partially-downloaded file. This is useful when you want to finish up a download started by a previous instance of Wget, or by another program. For instance:

```
wget -c ftp://sunsite.doc.ic.ac.uk/ls-lR.Z
```

If there is a file named *ls-lR.Z* in the current directory, Wget will assume that it is the first portion of the remote file, and will ask the server to continue the retrieval from



an offset equal to the length of the local file.

Note that you don't need to specify this option if you just want the current invocation of Wget to retry downloading a file should the connection be lost midway through. This is the default behavior. **-c** only affects resumption of downloads started *prior* to this invocation of Wget, and whose local files are still sitting around.

Without **-c**, the previous example would just download the remote file to *ls-LR.Z.1*, leaving the truncated *ls-LR.Z* file alone.

If you use **-c** on a non-empty file, and the server does not support continued downloading, Wget will restart the download from scratch and overwrite the existing file entirely.

Beginning with Wget 1.7, if you use **-c** on a file which is of equal size as the one on the server, Wget will refuse to download the file and print an explanatory message. The same happens when the file is smaller on the server than locally (presumably because it was changed on the server since your last download attempt)---because "continuing" is not meaningful, no download occurs.

On the other side of the coin, while using **-c**, any file that's bigger on the server than locally will be considered an incomplete download and only "(length(remote) - length(local))" bytes will be downloaded and tacked onto the end of the local file. This behavior can be desirable in certain cases---for instance, you can use **wget -c** to download just the new portion that's been appended to a data collection or log file.

However, if the file is bigger on the server because it's been *changed*, as opposed to just *appended* to, you'll end up with a garbled file. Wget has no way of verifying that the local file is really a valid prefix of the remote file. You need to be especially careful of this when using **-c** in conjunction with **-r**, since every file will be considered as an "incomplete download" candidate.

Another instance where you'll get a garbled file if you try to use **-c** is if you have a lame HTTP proxy that inserts a "transfer interrupted" string into the local file. In the future a "rollback" option may be added to deal with this case.

Note that **-c** only works with FTP servers and with HTTP servers that support the "Range" header.



--start-pos=OFFSET

Start downloading at zero-based position *OFFSET*. Offset may be expressed in bytes, kilobytes with the ``k'` suffix, or megabytes with the ``m'` suffix, etc.

--start-pos has higher precedence over **--continue**. When **--start-pos** and **--continue** are both specified, wget will emit a warning then proceed as if **--continue** was absent.

Server support for continued download is required, otherwise **--start-pos** cannot help. See **-c** for details.

--progress=type

Select the type of the progress indicator you wish to use. Legal indicators are "dot" and "bar".

The "bar" indicator is used by default. It draws an ASCII progress bar graphics (a.k.a "thermometer" display) indicating the status of retrieval. If the output is not a TTY, the "dot" bar will be used by default.

Use **--progress=dot** to switch to the "dot" display. It traces the retrieval by printing dots on the screen, each dot representing a fixed amount of downloaded data.

The progress *type* can also take one or more parameters. The parameters vary based on the *type* selected. Parameters to *type* are passed by appending them to the type separated by a colon (:) like this: **--progress=type:parameter1:parameter2**.

When using the dotted retrieval, you may set the *style* by specifying the type as **dot:style**. Different styles assign different meaning to one dot. With the "default" style each dot represents 1K, there are ten dots in a cluster and 50 dots in a line. The "binary" style has a more "computer"-like orientation---8K dots, 16-dots clusters and 48 dots per line (which makes for 384K lines). The "mega" style is suitable for downloading large files---each dot represents 64K retrieved, there are eight dots in a cluster, and 48 dots on each line (so each line contains 3M). If "mega" is not enough then you can use the "giga" style---each dot represents 1M retrieved, there are eight dots in a cluster, and 32 dots on each line (so each line contains 32M).

With **--progress=bar**, there are currently two possible parameters, *force* and *noscroll*.

When the output is not a TTY, the progress bar always falls



back to "dot", even if **--progress=bar** was passed to Wget during invocation. This behaviour can be overridden and the "bar" output forced by using the "force" parameter as **--progress=bar:force**.

By default, the **bar** style progress bar scroll the name of the file from left to right for the file being downloaded if the filename exceeds the maximum length allotted for its display. In certain cases, such as with **--progress=bar:force**, one may not want the scrolling filename in the progress bar. By passing the "noscroll" parameter, Wget can be forced to display as much of the filename as possible without scrolling through it.

Note that you can set the default style using the "progress" command in *.wgetrc*. That setting may be overridden from the command line. For example, to force the bar output without scrolling, use **--progress=bar:force:noscroll**.

--show-progress

Force wget to display the progress bar in any verbosity.

By default, wget only displays the progress bar in verbose mode. One may however, want wget to display the progress bar on screen in conjunction with any other verbosity modes like **--no-verbose** or **--quiet**. This is often a desired a property when invoking wget to download several small/large files. In such a case, wget could simply be invoked with this parameter to get a much cleaner output on the screen.

This option will also force the progress bar to be printed to *stderr* when used alongside the **--output-file** option.

-N

--timestamping

Turn on time-stamping.

--no-if-modified-since

Do not send If-Modified-Since header in **-N** mode. Send preliminary HEAD request instead. This has only effect in **-N** mode.

--no-use-server-timestamps

Don't set the local file's timestamp by the one on the server.

By default, when a file is downloaded, its timestamps are set to match those from the remote file. This allows the use of **--timestamping** on subsequent invocations of wget. However, it is sometimes useful to base the local file's timestamp on



when it was actually downloaded; for that purpose, the **--no-use-server-timestamps** option has been provided.

-S

--server-response

Print the headers sent by HTTP servers and responses sent by FTP servers.

--spider

When invoked with this option, Wget will behave as a Web *spider*, which means that it will not download the pages, just check that they are there. For example, you can use Wget to check your bookmarks:

```
wget --spider --force-html -i bookmarks.html
```

This feature needs much more work for Wget to get close to the functionality of real web spiders.

-T seconds

--timeout=seconds

Set the network timeout to *seconds* seconds. This is equivalent to specifying **--dns-timeout**, **--connect-timeout**, and **--read-timeout**, all at the same time.

When interacting with the network, Wget can check for timeout and abort the operation if it takes too long. This prevents anomalies like hanging reads and infinite connects. The only timeout enabled by default is a 900-second read timeout. Setting a timeout to 0 disables it altogether. Unless you know what you are doing, it is best not to change the default timeout settings.

All timeout-related options accept decimal values, as well as subsecond values. For example, **0.1** seconds is a legal (though unwise) choice of timeout. Subsecond timeouts are useful for checking server response times or for testing network latency.

--dns-timeout=seconds

Set the DNS lookup timeout to *seconds* seconds. DNS lookups that don't complete within the specified time will fail. By default, there is no timeout on DNS lookups, other than that implemented by system libraries.

--connect-timeout=seconds

Set the connect timeout to *seconds* seconds. TCP connections that take longer to establish will be aborted. By default, there is no connect timeout, other than that implemented by system libraries.



--read-timeout=seconds

Set the read (and write) timeout to *seconds* seconds. The "time" of this timeout refers to *idle time*: if, at any point in the download, no data is received for more than the specified number of seconds, reading fails and the download is restarted. This option does not directly affect the duration of the entire download.

Of course, the remote server may choose to terminate the connection sooner than this option requires. The default read timeout is 900 seconds.

--limit-rate=amount

Limit the download speed to *amount* bytes per second. Amount may be expressed in bytes, kilobytes with the **k** suffix, or megabytes with the **m** suffix. For example, **--limit-rate=20k** will limit the retrieval rate to 20KB/s. This is useful when, for whatever reason, you don't want Wget to consume the entire available bandwidth.

This option allows the use of decimal numbers, usually in conjunction with power suffixes; for example, **--limit-rate=2.5k** is a legal value.

Note that Wget implements the limiting by sleeping the appropriate amount of time after a network read that took less time than specified by the rate. Eventually this strategy causes the TCP transfer to slow down to approximately the specified rate. However, it may take some time for this balance to be achieved, so don't be surprised if limiting the rate doesn't work well with very small files.

-w seconds**--wait=seconds**

Wait the specified number of seconds between the retrievals. Use of this option is recommended, as it lightens the server load by making the requests less frequent. Instead of in seconds, the time can be specified in minutes using the "m" suffix, in hours using "h" suffix, or in days using "d" suffix.

Specifying a large value for this option is useful if the network or the destination host is down, so that Wget can wait long enough to reasonably expect the network error to be fixed before the retry. The waiting interval specified by this function is influenced by "--random-wait", which see.

--waitretry=seconds

If you don't want Wget to wait between *every* retrieval, but



only between retries of failed downloads, you can use this option. Wget will use *linear backoff*, waiting 1 second after the first failure on a given file, then waiting 2 seconds after the second failure on that file, up to the maximum number of *seconds* you specify.

By default, Wget will assume a value of 10 seconds.

--random-wait

Some web sites may perform log analysis to identify retrieval programs such as Wget by looking for statistically significant similarities in the time between requests. This option causes the time between requests to vary between 0.5 and $1.5 * \textit{wait}$ seconds, where *wait* was specified using the **--wait** option, in order to mask Wget's presence from such analysis.

A 2001 article in a publication devoted to development on a popular consumer platform provided code to perform this analysis on the fly. Its author suggested blocking at the class C address level to ensure automated retrieval programs were blocked despite changing DHCP-supplied addresses.

The **--random-wait** option was inspired by this ill-advised recommendation to block many unrelated users from a web site due to the actions of one.

--no-proxy

Don't use proxies, even if the appropriate `*_proxy` environment variable is defined.

-Q quota

--quota=quota

Specify download quota for automatic retrievals. The value can be specified in bytes (default), kilobytes (with **k** suffix), or megabytes (with **m** suffix).

Note that quota will never affect downloading a single file. So if you specify `wget -Q10k https://example.com/ls-LR.gz`, all of the `ls-LR.gz` will be downloaded. The same goes even when several URLs are specified on the command-line. The quota is checked only at the end of each downloaded file, so it will never result in a partially downloaded file. Thus you may safely type `wget -Q2m -i sites---`download will be aborted after the file that exhausts the quota is completely downloaded.

Setting quota to 0 or to **inf** unlimits the download quota.

--no-dns-cache



Turn off caching of DNS lookups. Normally, Wget remembers the IP addresses it looked up from DNS so it doesn't have to repeatedly contact the DNS server for the same (typically small) set of hosts it retrieves from. This cache exists in memory only; a new Wget run will contact DNS again.

However, it has been reported that in some situations it is not desirable to cache host names, even for the duration of a short-running application like Wget. With this option Wget issues a new DNS lookup (more precisely, a new call to "gethostbyname" or "getaddrinfo") each time it makes a new connection. Please note that this option will *not* affect caching that might be performed by the resolving library or by an external caching layer, such as NSCD.

If you don't understand exactly what this option does, you probably won't need it.

--restrict-file-names=*modes*

Change which characters found in remote URLs must be escaped during generation of local filenames. Characters that are *restricted* by this option are escaped, i.e. replaced with %HH, where HH is the hexadecimal number that corresponds to the restricted character. This option may also be used to force all alphabetical cases to be either lower- or uppercase.

By default, Wget escapes the characters that are not valid or safe as part of file names on your operating system, as well as control characters that are typically unprintable. This option is useful for changing these defaults, perhaps because you are downloading to a non-native partition, or because you want to disable escaping of the control characters, or you want to further restrict characters to only those in the ASCII range of values.

The *modes* are a comma-separated set of text values. The acceptable values are **unix**, **windows**, **nocontrol**, **ascii**, **lowercase**, and **uppercase**. The values **unix** and **windows** are mutually exclusive (one will override the other), as are **lowercase** and **uppercase**. Those last are special cases, as they do not change the set of characters that would be escaped, but rather force local file paths to be converted either to lower- or uppercase.

When "unix" is specified, Wget escapes the character / and the control characters in the ranges 0--31 and 128--159. This is the default on Unix-like operating systems.

When "windows" is given, Wget escapes the characters \, |, /,



`;`, `?`, `"`, `*`, `<`, `>`, and the control characters in the ranges 0--31 and 128--159. In addition to this, Wget in Windows mode uses `+` instead of `:` to separate host and port in local file names, and uses `@` instead of `?` to separate the query portion of the file name from the rest. Therefore, a URL that would be saved as

`www.xemacs.org:4300/search.pl?input=blah` in Unix mode would be saved as `www.xemacs.org+4300/search.pl@input=blah` in Windows mode. This mode is the default on Windows.

If you specify `nocontrol`, then the escaping of the control characters is also switched off. This option may make sense when you are downloading URLs whose names contain UTF-8 characters, on a system which can save and display filenames in UTF-8 (some possible byte values used in UTF-8 byte sequences fall in the range of values designated by Wget as "controls").

The `ascii` mode is used to specify that any bytes whose values are outside the range of ASCII characters (that is, greater than 127) shall be escaped. This can be useful when saving filenames whose encoding does not match the one used locally.

-4

--inet4-only

-6

--inet6-only

Force connecting to IPv4 or IPv6 addresses. With **--inet4-only** or **-4**, Wget will only connect to IPv4 hosts, ignoring AAAA records in DNS, and refusing to connect to IPv6 addresses specified in URLs. Conversely, with **--inet6-only** or **-6**, Wget will only connect to IPv6 hosts and ignore A records and IPv4 addresses.

Neither options should be needed normally. By default, an IPv6-aware Wget will use the address family specified by the host's DNS record. If the DNS responds with both IPv4 and IPv6 addresses, Wget will try them in sequence until it finds one it can connect to. (Also see "--prefer-family" option described below.)

These options can be used to deliberately force the use of IPv4 or IPv6 address families on dual family systems, usually to aid debugging or to deal with broken network configuration. Only one of **--inet6-only** and **--inet4-only** may be specified at the same time. Neither option is available in Wget compiled without IPv6 support.

--prefer-family=none/IPv4/IPv6

When given a choice of several addresses, connect to the



addresses with specified address family first. The address order returned by DNS is used without change by default.

This avoids spurious errors and connect attempts when accessing hosts that resolve to both IPv6 and IPv4 addresses from IPv4 networks. For example, **www.kame.net** resolves to **2001:200:0:8002:203:47ff:fea5:3085** and to **203.178.141.194**. When the preferred family is "IPv4", the IPv4 address is used first; when the preferred family is "IPv6", the IPv6 address is used first; if the specified value is "none", the address order returned by DNS is used without change.

Unlike **-4** and **-6**, this option doesn't inhibit access to any address family, it only changes the *order* in which the addresses are accessed. Also note that the reordering performed by this option is *stable*--it doesn't affect order of addresses of the same family. That is, the relative order of all IPv4 addresses and of all IPv6 addresses remains intact in all cases.

--retry-connrefused

Consider "connection refused" a transient error and try again. Normally Wget gives up on a URL when it is unable to connect to the site because failure to connect is taken as a sign that the server is not running at all and that retries would not help. This option is for mirroring unreliable sites whose servers tend to disappear for short periods of time.

--user=*user*

--password=*password*

Specify the username *user* and password *password* for both FTP and HTTP file retrieval. These parameters can be overridden using the **--ftp-user** and **--ftp-password** options for FTP connections and the **--http-user** and **--http-password** options for HTTP connections.

--ask-password

Prompt for a password for each connection established. Cannot be specified when **--password** is being used, because they are mutually exclusive.

--use-askpass=*command*

Prompt for a user and password using the specified command. If no command is specified then the command in the environment variable WGET_ASKPASS is used. If WGET_ASKPASS is not set then the command in the environment variable SSH_ASKPASS is used.

You can set the default command for use-askpass in the



`.wgetrc`. That setting may be overridden from the command line.

--no-iri

Turn off internationalized URI (IRI) support. Use `--iri` to turn it on. IRI support is activated by default.

You can set the default state of IRI support using the "iri" command in `.wgetrc`. That setting may be overridden from the command line.

--local-encoding=*encoding*

Force Wget to use *encoding* as the default system encoding. That affects how Wget converts URLs specified as arguments from locale to UTF-8 for IRI support.

Wget use the function `nl_langinfo()` and then the "CHARSET" environment variable to get the locale. If it fails, ASCII is used.

You can set the default local encoding using the "local_encoding" command in `.wgetrc`. That setting may be overridden from the command line.

--remote-encoding=*encoding*

Force Wget to use *encoding* as the default remote server encoding. That affects how Wget converts URIs found in files from remote encoding to UTF-8 during a recursive fetch. This options is only useful for IRI support, for the interpretation of non-ASCII characters.

For HTTP, remote encoding can be found in HTTP "Content-Type" header and in HTML "Content-Type http-equiv" meta tag.

You can set the default encoding using the "remoteencoding" command in `.wgetrc`. That setting may be overridden from the command line.

--unlink

Force Wget to unlink file instead of clobbering existing file. This option is useful for downloading to the directory with hardlinks.

Directory Options

-nd

--no-directories

Do not create a hierarchy of directories when retrieving recursively. With this option turned on, all files will get saved to the current directory, without clobbering (if a name shows up more than once, the filenames will get extensions



.n).

-x

--force-directories

The opposite of **-nd**---create a hierarchy of directories, even if one would not have been created otherwise. E.g. **wget -x http://fly.srk.fer.hr/robots.txt** will save the downloaded file to *fly.srk.fer.hr/robots.txt*.

-nH

--no-host-directories

Disable generation of host-prefixed directories. By default, invoking Wget with **-r http://fly.srk.fer.hr/** will create a structure of directories beginning with *fly.srk.fer.hr/*. This option disables such behavior.

--protocol-directories

Use the protocol name as a directory component of local file names. For example, with this option, **wget -r http://host** will save to **http/host/...** rather than just to *host/....*

--cut-dirs=number

Ignore *number* directory components. This is useful for getting a fine-grained control over the directory where recursive retrieval will be saved.

Take, for example, the directory at **ftp://ftp.xemacs.org/pub/xemacs/**. If you retrieve it with **-r**, it will be saved locally under *ftp.xemacs.org/pub/xemacs/*. While the **-nH** option can remove the *ftp.xemacs.org/* part, you are still stuck with *pub/xemacs*. This is where **--cut-dirs** comes in handy; it makes Wget not "see" *number* remote directory components. Here are several examples of how **--cut-dirs** option works.

```

No options          -> ftp.xemacs.org/pub/xemacs/
-nH                 -> pub/xemacs/
-nH --cut-dirs=1    -> xemacs/
-nH --cut-dirs=2    -> .

--cut-dirs=1        -> ftp.xemacs.org/xemacs/
...

```

If you just want to get rid of the directory structure, this option is similar to a combination of **-nd** and **-P**. However, unlike **-nd**, **--cut-dirs** does not lose with subdirectories---for instance, with **-nH --cut-dirs=1**, a *beta/* subdirectory will be placed to *xemacs/beta*, as one would expect.



-P *prefix*

--directory-prefix=*prefix*

Set directory prefix to *prefix*. The *directory prefix* is the directory where all other files and subdirectories will be saved to, i.e. the top of the retrieval tree. The default is `.` (the current directory).

HTTP Options

--default-page=*name*

Use *name* as the default file name when it isn't known (i.e., for URLs that end in a slash), instead of *index.html*.

-E

--adjust-extension

If a file of type **application/xhtml+xml** or **text/html** is downloaded and the URL does not end with the regexp `\.[Hh][Tt][Mm][Ll]?`, this option will cause the suffix **.html** to be appended to the local filename. This is useful, for instance, when you're mirroring a remote site that uses **.asp** pages, but you want the mirrored pages to be viewable on your stock Apache server. Another good use for this is when you're downloading CGI-generated materials. A URL like **http://site.com/article.cgi?25** will be saved as *article.cgi?25.html*.

Note that filenames changed in this way will be re-downloaded every time you re-mirror a site, because Wget can't tell that the local *X.html* file corresponds to remote URL *X* (since it doesn't yet know that the URL produces output of type **text/html** or **application/xhtml+xml**).

As of version 1.12, Wget will also ensure that any downloaded files of type **text/css** end in the suffix **.css**, and the option was renamed from **--html-extension**, to better reflect its new behavior. The old option name is still acceptable, but should now be considered deprecated.

As of version 1.19.2, Wget will also ensure that any downloaded files with a "Content-Encoding" of **br**, **compress**, **deflate** or **gzip** end in the suffix **.br**, **.Z**, **.zlib** and **.gz** respectively.

At some point in the future, this option may well be expanded to include suffixes for other types of content, including content types that are not parsed by Wget.

--http-user=*user*

--http-password=*password*

Specify the username *user* and password *password* on an HTTP server. According to the type of the challenge, Wget will



encode them using either the "basic" (insecure), the "digest", or the Windows "NTLM" authentication scheme.

Another way to specify username and password is in the URL itself. Either method reveals your password to anyone who bothers to run "ps". To prevent the passwords from being seen, use the **--use-askpass** or store them in *.wgetrc* or *.netrc*, and make sure to protect those files from other users with "chmod". If the passwords are really important, do not leave them lying in those files either---edit the files and delete them after Wget has started the download.

--no-http-keep-alive

Turn off the "keep-alive" feature for HTTP downloads. Normally, Wget asks the server to keep the connection open so that, when you download more than one document from the same server, they get transferred over the same TCP connection. This saves time and at the same time reduces the load on the server.

This option is useful when, for some reason, persistent (keep-alive) connections don't work for you, for example due to a server bug or due to the inability of server-side scripts to cope with the connections.

--no-cache

Disable server-side cache. In this case, Wget will send the remote server appropriate directives (**Cache-Control: no-cache** and **Pragma: no-cache**) to get the file from the remote service, rather than returning the cached version. This is especially useful for retrieving and flushing out-of-date documents on proxy servers.

Caching is allowed by default.

--no-cookies

Disable the use of cookies. Cookies are a mechanism for maintaining server-side state. The server sends the client a cookie using the "Set-Cookie" header, and the client responds with the same cookie upon further requests. Since cookies allow the server owners to keep track of visitors and for sites to exchange this information, some consider them a breach of privacy. The default is to use cookies; however, *storing* cookies is not on by default.

--load-cookies file

Load cookies from *file* before the first HTTP retrieval. *file* is a textual file in the format originally used by Netscape's *cookies.txt* file.



You will typically use this option when mirroring sites that require that you be logged in to access some or all of their content. The login process typically works by the web server issuing an HTTP cookie upon receiving and verifying your credentials. The cookie is then resent by the browser when accessing that part of the site, and so proves your identity.

Mirroring such a site requires Wget to send the same cookies your browser sends when communicating with the site. This is achieved by **--load-cookies**---simply point Wget to the location of the *cookies.txt* file, and it will send the same cookies your browser would send in the same situation. Different browsers keep textual cookie files in different locations:

"Netscape 4.x."

The cookies are in *~/.netscape/cookies.txt*.

"Mozilla and Netscape 6.x."

Mozilla's cookie file is also named *cookies.txt*, located somewhere under *~/.mozilla*, in the directory of your profile. The full path usually ends up looking somewhat like *~/.mozilla/default/some-weird-string/cookies.txt*.

"Internet Explorer."

You can produce a cookie file Wget can use by using the File menu, Import and Export, Export Cookies. This has been tested with Internet Explorer 5; it is not guaranteed to work with earlier versions.

"Other browsers."

If you are using a different browser to create your cookies, **--load-cookies** will only work if you can locate or produce a cookie file in the Netscape format that Wget expects.

If you cannot use **--load-cookies**, there might still be an alternative. If your browser supports a "cookie manager", you can use it to view the cookies used when accessing the site you're mirroring. Write down the name and value of the cookie, and manually instruct Wget to send those cookies, bypassing the "official" cookie support:

```
wget --no-cookies --header "Cookie: <name>=<value>"
```

--save-cookies *file*

Save cookies to *file* before exiting. This will not save cookies that have expired or that have no expiry time (so-called "session cookies"), but also see

--keep-session-cookies.



--keep-session-cookies

When specified, causes **--save-cookies** to also save session cookies. Session cookies are normally not saved because they are meant to be kept in memory and forgotten when you exit the browser. Saving them is useful on sites that require you to log in or to visit the home page before you can access some pages. With this option, multiple Wget runs are considered a single browser session as far as the site is concerned.

Since the cookie file format does not normally carry session cookies, Wget marks them with an expiry timestamp of 0. Wget's **--load-cookies** recognizes those as session cookies, but it might confuse other browsers. Also note that cookies so loaded will be treated as other session cookies, which means that if you want **--save-cookies** to preserve them again, you must use **--keep-session-cookies** again.

--ignore-length

Unfortunately, some HTTP servers (CGI programs, to be more precise) send out bogus "Content-Length" headers, which makes Wget go wild, as it thinks not all the document was retrieved. You can spot this syndrome if Wget retries getting the same document again and again, each time claiming that the (otherwise normal) connection has closed on the very same byte.

With this option, Wget will ignore the "Content-Length" header--as if it never existed.

--header=header-Line

Send *header-Line* along with the rest of the headers in each HTTP request. The supplied header is sent as-is, which means it must contain name and value separated by colon, and must not contain newlines.

You may define more than one additional header by specifying **--header** more than once.

```
wget --header='Accept-Charset: iso-8859-2' \  
      --header='Accept-Language: hr' \  
      http://fly.srk.fer.hr/
```

Specification of an empty string as the header value will clear all previous user-defined headers.

As of Wget 1.10, this option can be used to override headers otherwise generated automatically. This example instructs Wget to connect to localhost, but to specify **foo.bar** in the



"Host" header:

```
wget --header="Host: foo.bar" http://localhost/
```

In versions of Wget prior to 1.10 such use of **--header** caused sending of duplicate headers.

--compression=type

Choose the type of compression to be used. Legal values are **auto**, **gzip** and **none**.

If **auto** or **gzip** are specified, Wget asks the server to compress the file using the gzip compression format. If the server compresses the file and responds with the "Content-Encoding" header field set appropriately, the file will be decompressed automatically.

If **none** is specified, wget will not ask the server to compress the file and will not decompress any server responses. This is the default.

Compression support is currently experimental. In case it is turned on, please report any bugs to "bug-wget@gnu.org".

--max-redirect=number

Specifies the maximum number of redirections to follow for a resource. The default is 20, which is usually far more than necessary. However, on those occasions where you want to allow more (or fewer), this is the option to use.

--proxy-user=user

--proxy-password=password

Specify the username *user* and password *password* for authentication on a proxy server. Wget will encode them using the "basic" authentication scheme.

Security considerations similar to those with **--http-password** pertain here as well.

--referer=url

Include `Referer: *url*` header in HTTP request. Useful for retrieving documents with server-side processing that assume they are always being retrieved by interactive web browsers and only come out properly when Referer is set to one of the pages that point to them.

--save-headers

Save the headers sent by the HTTP server to the file, preceding the actual contents, with an empty line as the separator.



-U *agent-string*

--user-agent=*agent-string*

Identify as *agent-string* to the HTTP server.

The HTTP protocol allows the clients to identify themselves using a "User-Agent" header field. This enables distinguishing the WWW software, usually for statistical purposes or for tracing of protocol violations. Wget normally identifies as **Wget/version**, *version* being the current version number of Wget.

However, some sites have been known to impose the policy of tailoring the output according to the "User-Agent"-supplied information. While this is not such a bad idea in theory, it has been abused by servers denying information to clients other than (historically) Netscape or, more frequently, Microsoft Internet Explorer. This option allows you to change the "User-Agent" line issued by Wget. Use of this option is discouraged, unless you really know what you are doing.

Specifying empty user agent with **--user-agent=""** instructs Wget not to send the "User-Agent" header in HTTP requests.

--post-data=*string*

--post-file=*file*

Use POST as the method for all HTTP requests and send the specified data in the request body. **--post-data** sends *string* as data, whereas **--post-file** sends the contents of *file*. Other than that, they work in exactly the same way. In particular, they *both* expect content of the form "key1=value1&key2=value2", with percent-encoding for special characters; the only difference is that one expects its content as a command-line parameter and the other accepts its content from a file. In particular, **--post-file** is *not* for transmitting files as form attachments: those must appear as "key=value" data (with appropriate percent-coding) just like everything else. Wget does not currently support "multipart/form-data" for transmitting POST data; only "application/x-www-form-urlencoded". Only one of **--post-data** and **--post-file** should be specified.

Please note that wget does not require the content to be of the form "key1=value1&key2=value2", and neither does it test for it. Wget will simply transmit whatever data is provided to it. Most servers however expect the POST data to be in the above format when processing HTML Forms.

When sending a POST request using the **--post-file** option,



Wget treats the file as a binary file and will send every character in the POST request without stripping trailing newline or formfeed characters. Any other control characters in the text will also be sent as-is in the POST request.

Please be aware that Wget needs to know the size of the POST data in advance. Therefore the argument to "--post-file" must be a regular file; specifying a FIFO or something like */dev/stdin* won't work. It's not quite clear how to work around this limitation inherent in HTTP/1.0. Although HTTP/1.1 introduces *chunked* transfer that doesn't require knowing the request length in advance, a client can't use chunked unless it knows it's talking to an HTTP/1.1 server. And it can't know that until it receives a response, which in turn requires the request to have been completed -- a chicken-and-egg problem.

Note: As of version 1.15 if Wget is redirected after the POST request is completed, its behaviour will depend on the response code returned by the server. In case of a 301 Moved Permanently, 302 Moved Temporarily or 307 Temporary Redirect, Wget will, in accordance with RFC2616, continue to send a POST request. In case a server wants the client to change the Request method upon redirection, it should send a 303 See Other response code.

This example shows how to log in to a server using POST and then proceed to download the desired pages, presumably only accessible to authorized users:

```
# Log in to the server. This can be done only once.
wget --save-cookies cookies.txt \
      --post-data 'user=foo&password=bar' \
      http://example.com/auth.php

# Now grab the page or pages we care about.
wget --load-cookies cookies.txt \
      -p http://example.com/interesting/article.php
```

If the server is using session cookies to track user authentication, the above will not work because **--save-cookies** will not save them (and neither will browsers) and the *cookies.txt* file will be empty. In that case use **--keep-session-cookies** along with **--save-cookies** to force saving of session cookies.

--method=HTTP-Method

For the purpose of RESTful scripting, Wget allows sending of other HTTP Methods without the need to explicitly set them using **--header=Header-Line**. Wget will use whatever string is



passed to it after **--method** as the HTTP Method to the server.

--body-data=*Data-String*

--body-file=*Data-File*

Must be set when additional data needs to be sent to the server along with the Method specified using **--method**.

--body-data sends *string* as data, whereas **--body-file** sends the contents of *file*. Other than that, they work in exactly the same way.

Currently, **--body-file** is *not* for transmitting files as a whole. Wget does not currently support "multipart/form-data" for transmitting data; only "application/x-www-form-urlencoded". In the future, this may be changed so that wget sends the **--body-file** as a complete file instead of sending its contents to the server. Please be aware that Wget needs to know the contents of BODY Data in advance, and hence the argument to **--body-file** should be a regular file. See **--post-file** for a more detailed explanation. Only one of **--body-data** and **--body-file** should be specified.

If Wget is redirected after the request is completed, Wget will suspend the current method and send a GET request till the redirection is completed. This is true for all redirection response codes except 307 Temporary Redirect which is used to explicitly specify that the request method should *not* change. Another exception is when the method is set to "POST", in which case the redirection rules specified under **--post-data** are followed.

--content-disposition

If this is set to on, experimental (not fully-functional) support for "Content-Disposition" headers is enabled. This can currently result in extra round-trips to the server for a "HEAD" request, and is known to suffer from a few bugs, which is why it is not currently enabled by default.

This option is useful for some file-downloading CGI programs that use "Content-Disposition" headers to describe what the name of a downloaded file should be.

When combined with **--metalink-over-http** and **--trust-server-names**, a **Content-Type: application/metalink4+xml** file is named using the "Content-Disposition" filename field, if available.

--content-on-error

If this is set to on, wget will not skip the content when the server responds with a http status code that indicates error.



--trust-server-names

If this is set, on a redirect, the local file name will be based on the redirection URL. By default the local file name is based on the original URL. When doing recursive retrieving this can be helpful because in many web sites redirected URLs correspond to an underlying file structure, while link URLs do not.

--auth-no-challenge

If this option is given, Wget will send Basic HTTP authentication information (plaintext username and password) for all requests, just like Wget 1.10.2 and prior did by default.

Use of this option is not recommended, and is intended only to support some few obscure servers, which never send HTTP authentication challenges, but accept unsolicited auth info, say, in addition to form-based authentication.

--retry-on-host-error

Consider host errors, such as "Temporary failure in name resolution", as non-fatal, transient errors.

--retry-on-http-error=*code[,code,...]*

Consider given HTTP response codes as non-fatal, transient errors. Supply a comma-separated list of 3-digit HTTP response codes as argument. Useful to work around special circumstances where retries are required, but the server responds with an error code normally not retried by Wget. Such errors might be 503 (Service Unavailable) and 429 (Too Many Requests). Retries enabled by this option are performed subject to the normal retry timing and retry count limitations of Wget.

Using this option is intended to support special use cases only and is generally not recommended, as it can force retries even in cases where the server is actually trying to decrease its load. Please use wisely and only if you know what you are doing.

HTTPS (SSL/TLS) Options

To support encrypted HTTP (HTTPS) downloads, Wget must be compiled with an external SSL library. The current default is GnuTLS. In addition, Wget also supports HSTS (HTTP Strict Transport Security). If Wget is compiled without SSL support, none of these options are available.

--secure-protocol=*protocol*

Choose the secure protocol to be used. Legal values are



auto, **SSLv2**, **SSLv3**, **TLSv1**, **TLSv1_1**, **TLSv1_2**, **TLSv1_3** and **PFS**. If **auto** is used, the SSL library is given the liberty of choosing the appropriate protocol automatically, which is achieved by sending a TLSv1 greeting. This is the default.

Specifying **SSLv2**, **SSLv3**, **TLSv1**, **TLSv1_1**, **TLSv1_2** or **TLSv1_3** forces the use of the corresponding protocol. This is useful when talking to old and buggy SSL server implementations that make it hard for the underlying SSL library to choose the correct protocol version. Fortunately, such servers are quite rare.

Specifying **PFS** enforces the use of the so-called Perfect Forward Security cipher suites. In short, PFS adds security by creating a one-time key for each SSL connection. It has a bit more CPU impact on client and server. We use known to be secure ciphers (e.g. no MD4) and the TLS protocol. This mode also explicitly excludes non-PFS key exchange methods, such as RSA.

--https-only

When in recursive mode, only HTTPS links are followed.

--ciphers

Set the cipher list string. Typically this string sets the cipher suites and other SSL/TLS options that the user wish should be used, in a set order of preference (GnuTLS calls it 'priority string'). This string will be fed verbatim to the SSL/TLS engine (OpenSSL or GnuTLS) and hence its format and syntax is dependent on that. Wget will not process or manipulate it in any way. Refer to the OpenSSL or GnuTLS documentation for more information.

--no-check-certificate

Don't check the server certificate against the available certificate authorities. Also don't require the URL host name to match the common name presented by the certificate.

As of Wget 1.10, the default is to verify the server's certificate against the recognized certificate authorities, breaking the SSL handshake and aborting the download if the verification fails. Although this provides more secure downloads, it does break interoperability with some sites that worked with previous Wget versions, particularly those using self-signed, expired, or otherwise invalid certificates. This option forces an "insecure" mode of operation that turns the certificate verification errors into warnings and allows you to proceed.

If you encounter "certificate verification" errors or ones



saying that "common name doesn't match requested host name", you can use this option to bypass the verification and proceed with the download. *Only use this option if you are otherwise convinced of the site's authenticity, or if you really don't care about the validity of its certificate.* It is almost always a bad idea not to check the certificates when transmitting confidential or important data. For self-signed/internal certificates, you should download the certificate and verify against that instead of forcing this insecure mode. If you are really sure of not desiring any certificate verification, you can specify `--check-certificate=quiet` to tell wget to not print any warning about invalid certificates, albeit in most cases this is the wrong thing to do.

--certificate=*file*

Use the client certificate stored in *file*. This is needed for servers that are configured to require certificates from the clients that connect to them. Normally a certificate is not required and this switch is optional.

--certificate-type=*type*

Specify the type of the client certificate. Legal values are **PEM** (assumed by default) and **DER**, also known as **ASN1**.

--private-key=*file*

Read the private key from *file*. This allows you to provide the private key in a file separate from the certificate.

--private-key-type=*type*

Specify the type of the private key. Accepted values are **PEM** (the default) and **DER**.

--ca-certificate=*file*

Use *file* as the file with the bundle of certificate authorities ("CA") to verify the peers. The certificates must be in PEM format.

Without this option Wget looks for CA certificates at the system-specified locations, chosen at OpenSSL installation time.

--ca-directory=*directory*

Specifies directory containing CA certificates in PEM format. Each file contains one CA certificate, and the file name is based on a hash value derived from the certificate. This is achieved by processing a certificate directory with the "c_rehash" utility supplied with OpenSSL. Using **--ca-directory** is more efficient than **--ca-certificate** when many certificates are installed because it allows Wget to



fetch certificates on demand.

Without this option Wget looks for CA certificates at the system-specified locations, chosen at OpenSSL installation time.

--crl-file=file

Specifies a CRL file in *file*. This is needed for certificates that have been revoked by the CAs.

--pinnedpubkey=file/hashes

Tells wget to use the specified public key file (or hashes) to verify the peer. This can be a path to a file which contains a single public key in PEM or DER format, or any number of base64 encoded sha256 hashes preceded by "sha256//" and separated by ";"

When negotiating a TLS or SSL connection, the server sends a certificate indicating its identity. A public key is extracted from this certificate and if it does not exactly match the public key(s) provided to this option, wget will abort the connection before sending or receiving any data.

--random-file=file

[OpenSSL and LibreSSL only] Use *file* as the source of random data for seeding the pseudo-random number generator on systems without */dev/urandom*.

On such systems the SSL library needs an external source of randomness to initialize. Randomness may be provided by EGD (see **--egd-file** below) or read from an external source specified by the user. If this option is not specified, Wget looks for random data in \$RANDFILE or, if that is unset, in *\$HOME/.rnd*.

If you're getting the "Could not seed OpenSSL PRNG; disabling SSL." error, you should provide random data using some of the methods described above.

--egd-file=file

[OpenSSL only] Use *file* as the EGD socket. EGD stands for *Entropy Gathering Daemon*, a user-space program that collects data from various unpredictable system sources and makes it available to other programs that might need it. Encryption software, such as the SSL library, needs sources of non-repeating randomness to seed the random number generator used to produce cryptographically strong keys.

OpenSSL allows the user to specify his own source of entropy using the "RAND_FILE" environment variable. If this variable



is unset, or if the specified file does not produce enough randomness, OpenSSL will read random data from EGD socket specified using this option.

If this option is not specified (and the equivalent startup command is not used), EGD is never contacted. EGD is not needed on modern Unix systems that support */dev/urandom*.

--no-hsts

Wget supports HSTS (HTTP Strict Transport Security, RFC 6797) by default. Use **--no-hsts** to make Wget act as a non-HSTS-compliant UA. As a consequence, Wget would ignore all the "Strict-Transport-Security" headers, and would not enforce any existing HSTS policy.

--hsts-file=*file*

By default, Wget stores its HSTS database in *~/.wget-hsts*. You can use **--hsts-file** to override this. Wget will use the supplied file as the HSTS database. Such file must conform to the correct HSTS database format used by Wget. If Wget cannot parse the provided file, the behaviour is unspecified.

The Wget's HSTS database is a plain text file. Each line contains an HSTS entry (ie. a site that has issued a "Strict-Transport-Security" header and that therefore has specified a concrete HSTS policy to be applied). Lines starting with a dash ("#") are ignored by Wget. Please note that in spite of this convenient human-readability hand-hacking the HSTS database is generally not a good idea.

An HSTS entry line consists of several fields separated by one or more whitespace:

```
"<hostname> SP [<port>] SP <include subdomains> SP <created> SP <max-age>"
```

The *hostname* and *port* fields indicate the hostname and port to which the given HSTS policy applies. The *port* field may be zero, and it will, in most of the cases. That means that the port number will not be taken into account when deciding whether such HSTS policy should be applied on a given request (only the hostname will be evaluated). When *port* is different to zero, both the target hostname and the port will be evaluated and the HSTS policy will only be applied if both of them match. This feature has been included for testing/development purposes only. The Wget testsuite (in *testenv/*) creates HSTS databases with explicit ports with the purpose of ensuring Wget's correct behaviour. Applying HSTS policies to ports other than the default ones is discouraged by RFC 6797 (see Appendix B "Differences between HSTS Policy



and Same-Origin Policy"). Thus, this functionality should not be used in production environments and *port* will typically be zero. The last three fields do what they are expected to. The field *include_subdomains* can either be 1 or 0 and it signals whether the subdomains of the target domain should be part of the given HSTS policy as well. The *created* and *max-age* fields hold the timestamp values of when such entry was created (first seen by Wget) and the HSTS-defined value 'max-age', which states how long should that HSTS policy remain active, measured in seconds elapsed since the timestamp stored in *created*. Once that time has passed, that HSTS policy will no longer be valid and will eventually be removed from the database.

If you supply your own HSTS database via **--hsts-file**, be aware that Wget may modify the provided file if any change occurs between the HSTS policies requested by the remote servers and those in the file. When Wget exits, it effectively updates the HSTS database by rewriting the database file with the new entries.

If the supplied file does not exist, Wget will create one. This file will contain the new HSTS entries. If no HSTS entries were generated (no "Strict-Transport-Security" headers were sent by any of the servers) then no file will be created, not even an empty one. This behaviour applies to the default database file (*~/.wget-hsts*) as well: it will not be created until some server enforces an HSTS policy.

Care is taken not to override possible changes made by other Wget processes at the same time over the HSTS database. Before dumping the updated HSTS entries on the file, Wget will re-read it and merge the changes.

Using a custom HSTS database and/or modifying an existing one is discouraged. For more information about the potential security threats arose from such practice, see section 14 "Security Considerations" of RFC 6797, specially section 14.9 "Creative Manipulation of HSTS Policy Store".

--warc-file=file

Use *file* as the destination WARC file.

--warc-header=string

Use *string* into as the warcinfo record.

--warc-max-size=size

Set the maximum size of the WARC files to *size*.

--warc-cdx



Write CDX index files.

--warc-dedup=*file*

Do not store records listed in this CDX file.

--no-warc-compression

Do not compress WARC files with GZIP.

--no-warc-digests

Do not calculate SHA1 digests.

--no-warc-keep-log

Do not store the log file in a WARC record.

--warc-tempdir=*dir*

Specify the location for temporary files created by the WARC writer.

FTP Options

--ftp-user=*user*

--ftp-password=*password*

Specify the username *user* and password *password* on an FTP server. Without this, or the corresponding startup option, the password defaults to **-wget@**, normally used for anonymous FTP.

Another way to specify username and password is in the URL itself. Either method reveals your password to anyone who bothers to run "ps". To prevent the passwords from being seen, store them in *.wgetrc* or *.netrc*, and make sure to protect those files from other users with "chmod". If the passwords are really important, do not leave them lying in those files either---edit the files and delete them after Wget has started the download.

--no-remove-listing

Don't remove the temporary *.listing* files generated by FTP retrievals. Normally, these files contain the raw directory listings received from FTP servers. Not removing them can be useful for debugging purposes, or when you want to be able to easily check on the contents of remote server directories (e.g. to verify that a mirror you're running is complete).

Note that even though Wget writes to a known filename for this file, this is not a security hole in the scenario of a user making *.listing* a symbolic link to */etc/passwd* or something and asking "root" to run Wget in his or her directory. Depending on the options used, either Wget will refuse to write to *.listing*, making the globbing/recursion/time-stamping operation fail, or the



symbolic link will be deleted and replaced with the actual `.listing` file, or the listing will be written to a `.listing.number` file.

Even though this situation isn't a problem, though, "root" should never run `Wget` in a non-trusted user's directory. A user could do something as simple as linking `index.html` to `/etc/passwd` and asking "root" to run `Wget` with `-N` or `-r` so the file will be overwritten.

--no-glob

Turn off FTP globbing. Globbing refers to the use of shell-like special characters (*wildcards*), like `*`, `?`, `[` and `]` to retrieve more than one file from the same directory at once, like:

```
wget ftp://gnjilux.srk.fer.hr/*.msg
```

By default, globbing will be turned on if the URL contains a globbing character. This option may be used to turn globbing on or off permanently.

You may have to quote the URL to protect it from being expanded by your shell. Globbing makes `Wget` look for a directory listing, which is system-specific. This is why it currently works only with Unix FTP servers (and the ones emulating Unix "ls" output).

--no-passive-ftp

Disable the use of the *passive* FTP transfer mode. Passive FTP mandates that the client connect to the server to establish the data connection rather than the other way around.

If the machine is connected to the Internet directly, both passive and active FTP should work equally well. Behind most firewall and NAT configurations passive FTP has a better chance of working. However, in some rare firewall configurations, active FTP actually works when passive FTP doesn't. If you suspect this to be the case, use this option, or set `"passive_ftp=off"` in your init file.

--preserve-permissions

Preserve remote file permissions instead of permissions set by `umask`.

--retr-symlinks

By default, when retrieving FTP directories recursively and a symbolic link is encountered, the symbolic link is traversed and the pointed-to files are retrieved. Currently, `Wget` does



not traverse symbolic links to directories to download them recursively, though this feature may be added in the future.

When **--retr-symlinks=no** is specified, the linked-to file is not downloaded. Instead, a matching symbolic link is created on the local file system. The pointed-to file will not be retrieved unless this recursive retrieval would have encountered it separately and downloaded it anyway. This option poses a security risk where a malicious FTP Server may cause Wget to write to files outside of the intended directories through a specially crafted `.LISTING` file.

Note that when retrieving a file (not a directory) because it was specified on the command-line, rather than because it was recursed to, this option has no effect. Symbolic links are always traversed in this case.

FTPS Options

--ftps-implicit

This option tells Wget to use FTPS implicitly. Implicit FTPS consists of initializing SSL/TLS from the very beginning of the control connection. This option does not send an "AUTH TLS" command: it assumes the server speaks FTPS and directly starts an SSL/TLS connection. If the attempt is successful, the session continues just like regular FTPS ("PBSZ" and "PROT" are sent, etc.). Implicit FTPS is no longer a requirement for FTPS implementations, and thus many servers may not support it. If **--ftps-implicit** is passed and no explicit port number specified, the default port for implicit FTPS, 990, will be used, instead of the default port for the "normal" (explicit) FTPS which is the same as that of FTP, 21.

--no-ftps-resume-ssl

Do not resume the SSL/TLS session in the data channel. When starting a data connection, Wget tries to resume the SSL/TLS session previously started in the control connection. SSL/TLS session resumption avoids performing an entirely new handshake by reusing the SSL/TLS parameters of a previous session. Typically, the FTPS servers want it that way, so Wget does this by default. Under rare circumstances however, one might want to start an entirely new SSL/TLS session in every data connection. This is what **--no-ftps-resume-ssl** is for.

--ftps-clear-data-connection

All the data connections will be in plain text. Only the control connection will be under SSL/TLS. Wget will send a "PROT C" command to achieve this, which must be approved by the server.



--ftps-fallback-to-ftp

Fall back to FTP if FTPS is not supported by the target server. For security reasons, this option is not asserted by default. The default behaviour is to exit with an error. If a server does not successfully reply to the initial "AUTH TLS" command, or in the case of implicit FTPS, if the initial SSL/TLS connection attempt is rejected, it is considered that such server does not support FTPS.

Recursive Retrieval Options**-r****--recursive**

Turn on recursive retrieving. The default maximum depth is 5.

-l *depth***--level=*depth***

Set the maximum number of subdirectories that Wget will recurse into to *depth*. In order to prevent one from accidentally downloading very large websites when using recursion this is limited to a depth of 5 by default, i.e., it will traverse at most 5 directories deep starting from the provided URL. Set **-l 0** or **-l inf** for infinite recursion depth.

```
wget -r -l 0 http://<site>/1.html
```

Ideally, one would expect this to download just *1.html*. but unfortunately this is not the case, because **-l 0** is equivalent to **-l inf**---that is, infinite recursion. To download a single HTML page (or a handful of them), specify them all on the command line and leave away **-r** and **-l**. To download the essential items to view a single HTML page, see **page requisites**.

--delete-after

This option tells Wget to delete every single file it downloads, *after* having done so. It is useful for pre-fetching popular pages through a proxy, e.g.:

```
wget -r -nd --delete-after http://whatever.com/~popular/page/
```

The **-r** option is to retrieve recursively, and **-nd** to not create directories.

Note that **--delete-after** deletes files on the local machine. It does not issue the **DELE** command to remote FTP sites, for instance. Also note that when **--delete-after** is specified, **--convert-links** is ignored, so **.orig** files are simply not



created in the first place.

-k

--convert-links

After the download is complete, convert the links in the document to make them suitable for local viewing. This affects not only the visible hyperlinks, but any part of the document that links to external content, such as embedded images, links to style sheets, hyperlinks to non-HTML content, etc.

Each link will be changed in one of the two ways:

- The links to files that have been downloaded by Wget will be changed to refer to the file they point to as a relative link.

Example: if the downloaded file `/foo/doc.html` links to `/bar/img.gif`, also downloaded, then the link in `doc.html` will be modified to point to `../bar/img.gif`. This kind of transformation works reliably for arbitrary combinations of directories.

- The links to files that have not been downloaded by Wget will be changed to include host name and absolute path of the location they point to.

Example: if the downloaded file `/foo/doc.html` links to `/bar/img.gif` (or to `../bar/img.gif`), then the link in `doc.html` will be modified to point to `http://hostname/bar/img.gif`.

Because of this, local browsing works reliably: if a linked file was downloaded, the link will refer to its local name; if it was not downloaded, the link will refer to its full Internet address rather than presenting a broken link. The fact that the former links are converted to relative links ensures that you can move the downloaded hierarchy to another directory.

Note that only at the end of the download can Wget know which links have been downloaded. Because of that, the work done by **-k** will be performed at the end of all the downloads.

--convert-file-only

This option converts only the filename part of the URLs, leaving the rest of the URLs untouched. This filename part is sometimes referred to as the "basename", although we avoid that term here in order not to cause confusion.



It works particularly well in conjunction with **--adjust-extension**, although this coupling is not enforced. It proves useful to populate Internet caches with files downloaded from different hosts.

Example: if some link points to `//foo.com/bar.cgi?xyz` with **--adjust-extension** asserted and its local destination is intended to be `./foo.com/bar.cgi?xyz.css`, then the link would be converted to `//foo.com/bar.cgi?xyz.css`. Note that only the filename part has been modified. The rest of the URL has been left untouched, including the net path ("`//`") which would otherwise be processed by Wget and converted to the effective scheme (ie. "`http://`").

-K

--backup-converted

When converting a file, back up the original version with a **.orig** suffix. Affects the behavior of **-N**.

-m

--mirror

Turn on options suitable for mirroring. This option turns on recursion and time-stamping, sets infinite recursion depth and keeps FTP directory listings. It is currently equivalent to **-r -N -l inf --no-remove-listing**.

-p

--page-requisites

This option causes Wget to download all the files that are necessary to properly display a given HTML page. This includes such things as inlined images, sounds, and referenced stylesheets.

Ordinarily, when downloading a single HTML page, any requisite documents that may be needed to display it properly are not downloaded. Using **-r** together with **-l** can help, but since Wget does not ordinarily distinguish between external and inlined documents, one is generally left with "leaf documents" that are missing their requisites.

For instance, say document `1.html` contains an "``" tag referencing `1.gif` and an "`<A>`" tag pointing to external document `2.html`. Say that `2.html` is similar but that its image is `2.gif` and it links to `3.html`. Say this continues up to some arbitrarily high number.

If one executes the command:

```
wget -r -l 2 http://<site>/1.html
```



then *1.html*, *1.gif*, *2.html*, *2.gif*, and *3.html* will be downloaded. As you can see, *3.html* is without its requisite *3.gif* because Wget is simply counting the number of hops (up to 2) away from *1.html* in order to determine where to stop the recursion. However, with this command:

```
wget -r -l 2 -p http://<site>/1.html
```

all the above files *and 3.html's* requisite *3.gif* will be downloaded. Similarly,

```
wget -r -l 1 -p http://<site>/1.html
```

will cause *1.html*, *1.gif*, *2.html*, and *2.gif* to be downloaded. One might think that:

```
wget -r -l 0 -p http://<site>/1.html
```

would download just *1.html* and *1.gif*, but unfortunately this is not the case, because *-l 0* is equivalent to *-l inf*---that is, infinite recursion. To download a single HTML page (or a handful of them, all specified on the command-line or in a *-i* URL input file) and its (or their) requisites, simply leave off *-r* and *-l*:

```
wget -p http://<site>/1.html
```

Note that Wget will behave as if *-r* had been specified, but only that single page and its requisites will be downloaded. Links from that page to external documents will not be followed. Actually, to download a single page and all its requisites (even if they exist on separate websites), and make sure the lot displays properly locally, this author likes to use a few options in addition to *-p*:

```
wget -E -H -k -K -p http://<site>/<document>
```

To finish off this topic, it's worth knowing that Wget's idea of an external document link is any URL specified in an "<A>" tag, an "<AREA>" tag, or a "<LINK>" tag other than "<LINK REL='stylesheet'>".

--strict-comments

Turn on strict parsing of HTML comments. The default is to terminate comments at the first occurrence of *-->*.

According to specifications, HTML comments are expressed as SGML *declarations*. Declaration is special markup that begins with *<!* and ends with *>*, such as *<!DOCTYPE ...>*, that may contain comments between a pair of *--* delimiters. HTML



comments are "empty declarations", SGML declarations without any non-comment text. Therefore, `<!--foo-->` is a valid comment, and so is `<!--one-- --two-->`, but `<!--1--2-->` is not.

On the other hand, most HTML writers don't perceive comments as anything other than text delimited with `<!--` and `-->`, which is not quite the same. For example, something like `<!------->` works as a valid comment as long as the number of dashes is a multiple of four (!). If not, the comment technically lasts until the next `--`, which may be at the other end of the document. Because of this, many popular browsers completely ignore the specification and implement what users have come to expect: comments delimited with `<!--` and `-->`.

Until version 1.9, Wget interpreted comments strictly, which resulted in missing links in many web pages that displayed fine in browsers, but had the misfortune of containing non-compliant comments. Beginning with version 1.9, Wget has joined the ranks of clients that implements "naive" comments, terminating each comment at the first occurrence of `-->`.

If, for whatever reason, you want strict comment parsing, use this option to turn it on.

Recursive Accept/Reject Options

-A *acclist* **--accept** *acclist*

-R *rejlist* **--reject** *rejlist*

Specify comma-separated lists of file name suffixes or patterns to accept or reject. Note that if any of the wildcard characters, `*`, `?`, `[` or `]`, appear in an element of *acclist* or *rejlist*, it will be treated as a pattern, rather than a suffix. In this case, you have to enclose the pattern into quotes to prevent your shell from expanding it, like in **-A** `"*.mp3"` or **-A** `'*.mp3'`.

--accept-regex *urlregex*

--reject-regex *urlregex*

Specify a regular expression to accept or reject the complete URL.

--regex-type *regextype*

Specify the regular expression type. Possible types are **posix** or **pcre**. Note that to be able to use **pcre** type, wget has to be compiled with libpcre support.

-D *domain-list*

--domains=*domain-list*

Set domains to be followed. *domain-list* is a comma-separated



list of domains. Note that it does *not* turn on **-H**.

--exclude-domains *domain-list*

Specify the domains that are *not* to be followed.

--follow-ftp

Follow FTP links from HTML documents. Without this option, Wget will ignore all the FTP links.

--follow-tags=*list*

Wget has an internal table of HTML tag / attribute pairs that it considers when looking for linked documents during a recursive retrieval. If a user wants only a subset of those tags to be considered, however, he or she should specify such tags in a comma-separated *list* with this option.

--ignore-tags=*list*

This is the opposite of the **--follow-tags** option. To skip certain HTML tags when recursively looking for documents to download, specify them in a comma-separated *list*.

In the past, this option was the best bet for downloading a single page and its requisites, using a command-line like:

```
wget --ignore-tags=a,area -H -k -K -r http://<site>/<document>
```

However, the author of this option came across a page with tags like "`<LINK REL="home" HREF="/">`" and came to the realization that specifying tags to ignore was not enough. One can't just tell Wget to ignore "`<LINK>`", because then stylesheets will not be downloaded. Now the best bet for downloading a single page and its requisites is the dedicated **--page-requisites** option.

--ignore-case

Ignore case when matching files and directories. This influences the behavior of **-R**, **-A**, **-I**, and **-X** options, as well as globbing implemented when downloading from FTP sites. For example, with this option, **-A "*.txt"** will match **file1.txt**, but also **file2.TXT**, **file3.TxT**, and so on. The quotes in the example are to prevent the shell from expanding the pattern.

-H

--span-hosts

Enable spanning across hosts when doing recursive retrieving.

-L

--relative

Follow relative links only. Useful for retrieving a specific



home page without any distractions, not even those from the same hosts.

-I *list*

--include-directories=*list*

Specify a comma-separated list of directories you wish to follow when downloading. Elements of *list* may contain wildcards.

-X *list*

--exclude-directories=*list*

Specify a comma-separated list of directories you wish to exclude from download. Elements of *list* may contain wildcards.

-np

--no-parent

Do not ever ascend to the parent directory when retrieving recursively. This is a useful option, since it guarantees that only the files *below* a certain hierarchy will be downloaded.

ENVIRONMENT [top](#)

Wget supports proxies for both HTTP and FTP retrievals. The standard way to specify proxy location, which Wget recognizes, is using the following environment variables:

http_proxy

https_proxy

If set, the **http_proxy** and **https_proxy** variables should contain the URLs of the proxies for HTTP and HTTPS connections respectively.

ftp_proxy

This variable should contain the URL of the proxy for FTP connections. It is quite common that **http_proxy** and **ftp_proxy** are set to the same URL.

no_proxy

This variable should contain a comma-separated list of domain extensions proxy should *not* be used for. For instance, if the value of **no_proxy** is **.mit.edu**, proxy will not be used to retrieve documents from MIT.

EXIT STATUS [top](#)



Wget may return one of several error codes if it encounters problems.

- 0 No problems occurred.
- 1 Generic error code.
- 2 Parse error---for instance, when parsing command-line options, the **.wgetrc** or **.netrc**...
- 3 File I/O error.
- 4 Network failure.
- 5 SSL verification failure.
- 6 Username/password authentication failure.
- 7 Protocol errors.
- 8 Server issued an error response.

With the exceptions of 0 and 1, the lower-numbered exit codes take precedence over higher-numbered ones, when multiple types of errors are encountered.

In versions of Wget prior to 1.12, Wget's exit status tended to be unhelpful and inconsistent. Recursive downloads would virtually always return 0 (success), regardless of any issues encountered, and non-recursive fetches only returned the status corresponding to the most recently-attempted download.

FILES [top](#)

/usr/local/etc/wgetrc
Default location of the *global* startup file.

.wgetrc
User startup file.

BUGS [top](#)

You are welcome to submit bug reports via the GNU Wget bug tracker (see <https://savannah.gnu.org/bugs/?func=additem&group=wget> >) or to our mailing list bug-wget@gnu.org.

Visit <https://lists.gnu.org/mailman/listinfo/bug-wget> > to get

more info (how to subscribe, list archives, ...).

Before actually submitting a bug report, please try to follow a few simple guidelines.

1. Please try to ascertain that the behavior you see really is a bug. If Wget crashes, it's a bug. If Wget does not behave as documented, it's a bug. If things work strange, but you are not sure about the way they are supposed to work, it might well be a bug, but you might want to double-check the documentation and the mailing lists.
2. Try to repeat the bug in as simple circumstances as possible. E.g. if Wget crashes while downloading `wget -r10 -kKE -t5 --no-proxy http://example.com -o /tmp/log`, you should try to see if the crash is repeatable, and if will occur with a simpler set of options. You might even try to start the download at the page where the crash occurred to see if that page somehow triggered the crash.

Also, while I will probably be interested to know the contents of your `.wgetrc` file, just dumping it into the debug message is probably a bad idea. Instead, you should first try to see if the bug repeats with `.wgetrc` moved out of the way. Only if it turns out that `.wgetrc` settings affect the bug, mail me the relevant parts of the file.

3. Please start Wget with `-d` option and send us the resulting output (or relevant parts thereof). If Wget was compiled without debug support, recompile it---it is *much* easier to trace bugs with debug support on.

Note: please make sure to remove any potentially sensitive information from the debug log before sending it to the bug address. The `-d` won't go out of its way to collect sensitive information, but the log *will* contain a fairly complete transcript of Wget's communication with the server, which may include passwords and pieces of downloaded data. Since the bug address is publicly archived, you may assume that all bug reports are visible to the public.

4. If Wget has crashed, try to run it in a debugger, e.g. "gdb `which wget` core" and type "where" to get the backtrace. This may not work if the system administrator has disabled core files, but it is safe to try.

SEE ALSO [top](#)



This is **not** the complete manual for GNU Wget. For more complete information, including more detailed explanations of some of the options, and a number of commands available for use with `.wgetrc` files and the `-e` option, see the GNU Info entry for `wget`.

Also see `wget2(1)`, the updated version of GNU Wget with even better support for recursive downloading and modern protocols like HTTP/2.

AUTHOR [top](#)

Originally written by Hrvoje Nikšić <hniksic@xemacs.org>. Currently maintained by Darshit Shah <darnir@gnu.org> and Tim Rühsen <tim.ruehsen@gmx.de>.

COPYRIGHT [top](#)

Copyright (c) 1996--2011, 2015, 2018--2023 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

COLOPHON [top](#)

This page is part of the `wget` (interactive network downloader) project. Information about the project can be found at <http://www.gnu.org/software/wget/>. If you have a bug report for this manual page, send it to bug-sed@gnu.org. This page was obtained from the tarball `wget-1.21.4.tar.gz` fetched from <https://www.gnu.org/software/wget/> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

GNU Wget 1.21.4

2023-12-22

WGET(1)

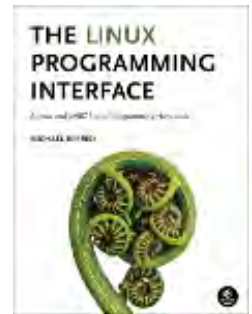
Pages that refer to this page: [curl\(1\)](#), [update-pciids\(8\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



grep(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [REGULAR EXPRESSIONS](#) | [EXIT STATUS](#) | [ENVIRONMENT](#) | [NOTES](#) | [COPYRIGHT](#) | [BUGS](#) | [EXAMPLE](#) | [SEE ALSO](#) | [COLOPHON](#)

 GREP(1)

User Commands

GREP(1)**NAME** [top](#)

grep - print lines that match patterns

SYNOPSIS [top](#)

```
grep [OPTION...] PATTERNS [FILE...]  
grep [OPTION...] -e PATTERNS ... [FILE...]  
grep [OPTION...] -f PATTERN_FILE ... [FILE...]
```

DESCRIPTION [top](#)

grep searches for *PATTERNS* in each *FILE*. *PATTERNS* is one or more patterns separated by newline characters, and **grep** prints each line that matches a pattern. Typically *PATTERNS* should be quoted when **grep** is used in a shell command.

A *FILE* of “-” stands for standard input. If no *FILE* is given, recursive searches examine the working directory, and nonrecursive searches read standard input.

OPTIONS [top](#)**Generic Program Information**

--help Output a usage message and exit.



-V, --version

Output the version number of **grep** and exit.

Pattern Syntax**-E, --extended-regexp**

Interpret *PATTERNS* as extended regular expressions (EREs, see below).

-F, --fixed-strings

Interpret *PATTERNS* as fixed strings, not regular expressions.

-G, --basic-regexp

Interpret *PATTERNS* as basic regular expressions (BREs, see below). This is the default.

-P, --perl-regexp

Interpret *PATTERNS* as Perl-compatible regular expressions (PCREs). This option is experimental when combined with the **-z** (**--null-data**) option, and **grep -P** may warn of unimplemented features.

Matching Control**-e PATTERNS, --regexp=PATTERNS**

Use *PATTERNS* as the patterns. If this option is used multiple times or is combined with the **-f** (**--file**) option, search for all patterns given. This option can be used to protect a pattern beginning with “-”.

-f FILE, --file=FILE

Obtain patterns from *FILE*, one per line. If this option is used multiple times or is combined with the **-e** (**--regexp**) option, search for all patterns given. The empty file contains zero patterns, and therefore matches nothing. If *FILE* is **-**, read patterns from standard input.

-i, --ignore-case

Ignore case distinctions in patterns and input data, so that characters that differ only in case match each other.

--no-ignore-case

Do not ignore case distinctions in patterns and input data. This is the default. This option is useful for



passing to shell scripts that already use **-i**, to cancel its effects because the two options override each other.

-v, --invert-match

Invert the sense of matching, to select non-matching lines.

-w, --word-regexp

Select only those lines containing matches that form whole words. The test is that the matching substring must either be at the beginning of the line, or preceded by a non-word constituent character. Similarly, it must be either at the end of the line or followed by a non-word constituent character. Word-constituent characters are letters, digits, and the underscore. This option has no effect if **-x** is also specified.

-x, --line-regexp

Select only those matches that exactly match the whole line. For a regular expression pattern, this is like parenthesizing the pattern and then surrounding it with **^** and **\$**.

General Output Control

-c, --count

Suppress normal output; instead print a count of matching lines for each input file. With the **-v, --invert-match** option (see above), count non-matching lines.

--color[=*WHEN*], --colour[=*WHEN*]

Surround the matched (non-empty) strings, matching lines, context lines, file names, line numbers, byte offsets, and separators (for fields and groups of context lines) with escape sequences to display them in color on the terminal. The colors are defined by the environment variable **GREP_COLORS**. *WHEN* is **never**, **always**, or **auto**.

-L, --files-without-match

Suppress normal output; instead print the name of each input file from which no output would normally have been printed.

-l, --files-with-matches

Suppress normal output; instead print the name of each



input file from which output would normally have been printed. Scanning each input file stops upon first match.

-m NUM, --max-count=NUM

Stop reading a file after *NUM* matching lines. If *NUM* is zero, **grep** stops right away without reading input. A *NUM* of -1 is treated as infinity and **grep** does not stop; this is the default. If the input is standard input from a regular file, and *NUM* matching lines are output, **grep** ensures that the standard input is positioned to just after the last matching line before exiting, regardless of the presence of trailing context lines. This enables a calling process to resume a search. When **grep** stops after *NUM* matching lines, it outputs any trailing context lines. When the **-c** or **--count** option is also used, **grep** does not output a count greater than *NUM*. When the **-v** or **--invert-match** option is also used, **grep** stops after outputting *NUM* non-matching lines.

-o, --only-matching

Print only the matched (non-empty) parts of a matching line, with each such part on a separate output line.

-q, --quiet, --silent

Quiet; do not write anything to standard output. Exit immediately with zero status if any match is found, even if an error was detected. Also see the **-s** or **--no-messages** option.

-s, --no-messages

Suppress error messages about nonexistent or unreadable files.

Output Line Prefix Control

-b, --byte-offset

Print the 0-based byte offset within the input file before each line of output. If **-o** (**--only-matching**) is specified, print the offset of the matching part itself.

-H, --with-filename

Print the file name for each match. This is the default when there is more than one file to search. This is a GNU extension.



-h, --no-filename

Suppress the prefixing of file names on output. This is the default when there is only one file (or only standard input) to search.

--label=LABEL

Display input actually coming from standard input as input coming from file *LABEL*. This can be useful for commands that transform a file's contents before searching, e.g., **gzip -cd foo.gz | grep --label=foo -H 'some pattern'**. See also the **-H** option.

-n, --line-number

Prefix each line of output with the 1-based line number within its input file.

-T, --initial-tab

Make sure that the first character of actual line content lies on a tab stop, so that the alignment of tabs looks normal. This is useful with options that prefix their output to the actual content: **-H**, **-n**, and **-b**. In order to improve the probability that lines from a single file will all start at the same column, this also causes the line number and byte offset (if present) to be printed in a minimum size field width.

-Z, --null

Output a zero byte (the ASCII **NUL** character) instead of the character that normally follows a file name. For example, **grep -lZ** outputs a zero byte after each file name instead of the usual newline. This option makes the output unambiguous, even in the presence of file names containing unusual characters like newlines. This option can be used with commands like **find -print0**, **perl -0**, **sort -z**, and **xargs -0** to process arbitrary file names, even those that contain newline characters.

Context Line Control**-A NUM, --after-context=NUM**

Print *NUM* lines of trailing context after matching lines. Places a line containing a group separator (**--**) between contiguous groups of matches. With the **-o** or **--only-matching** option, this has no effect and a warning is given.



-B *NUM*, --before-context=*NUM*

Print *NUM* lines of leading context before matching lines. Places a line containing a group separator (--) between contiguous groups of matches. With the **-o** or **--only-matching** option, this has no effect and a warning is given.

-C *NUM*, -*NUM*, --context=*NUM*

Print *NUM* lines of output context. Places a line containing a group separator (--) between contiguous groups of matches. With the **-o** or **--only-matching** option, this has no effect and a warning is given.

--group-separator=*SEP*

When **-A**, **-B**, or **-C** are in use, print *SEP* instead of -- between groups of lines.

--no-group-separator

When **-A**, **-B**, or **-C** are in use, do not print a separator between groups of lines.

File and Directory Selection**-a, --text**

Process a binary file as if it were text; this is equivalent to the **--binary-files=text** option.

--binary-files=*TYPE*

If a file's data or metadata indicate that the file contains binary data, assume that the file is of type *TYPE*. Non-text bytes indicate binary data; these are either output bytes that are improperly encoded for the current locale, or null input bytes when the **-z** option is not given.

By default, *TYPE* is **binary**, and **grep** suppresses output after null input binary data is discovered, and suppresses output lines that contain improperly encoded data. When some output is suppressed, **grep** follows any output with a message to standard error saying that a binary file matches.

If *TYPE* is **without-match**, when **grep** discovers null input binary data it assumes that the rest of the file does not



match; this is equivalent to the **-I** option.

If *TYPE* is **text**, **grep** processes a binary file as if it were text; this is equivalent to the **-a** option.

When *type* is **binary**, **grep** may treat non-text bytes as line terminators even without the **-z** option. This means choosing **binary** versus **text** can affect whether a pattern matches a file. For example, when *type* is **binary** the pattern **q\$ might** match **q** immediately followed by a null byte, even though this is not matched when *type* is **text**. Conversely, when *type* is **binary** the pattern **.** (period) might not match a null byte.

Warning: The **-a** option might output binary garbage, which can have nasty side effects if the output is a terminal and if the terminal driver interprets some of it as commands. On the other hand, when reading files whose text encodings are unknown, it can be helpful to use **-a** or to set **LC_ALL='C'** in the environment, in order to find more matches even if the matches are unsafe for direct display.

-D ACTION, --devices=ACTION

If an input file is a device, FIFO or socket, use *ACTION* to process it. By default, *ACTION* is **read**, which means that devices are read just as if they were ordinary files. If *ACTION* is **skip**, devices are silently skipped.

-d ACTION, --directories=ACTION

If an input file is a directory, use *ACTION* to process it. By default, *ACTION* is **read**, i.e., read directories just as if they were ordinary files. If *ACTION* is **skip**, silently skip directories. If *ACTION* is **recurse**, read all files under each directory, recursively, following symbolic links only if they are on the command line. This is equivalent to the **-r** option.

--exclude=GLOB

Skip any command-line file with a name suffix that matches the pattern *GLOB*, using wildcard matching; a name suffix is either the whole name, or a trailing part that starts with a non-slash character immediately after a slash (/) in the name. When searching recursively, skip any subfile



whose base name matches *GLOB*; the base name is the part after the last slash. A pattern can use ***, *?*, and *[...]* as wildcards, and ** to quote a wildcard or backslash character literally.

--exclude-from=FILE

Skip files whose base name matches any of the file-name globs read from *FILE* (using wildcard matching as described under **--exclude**).

--exclude-dir=GLOB

Skip any command-line directory with a name suffix that matches the pattern *GLOB*. When searching recursively, skip any subdirectory whose base name matches *GLOB*. Ignore any redundant trailing slashes in *GLOB*.

-I Process a binary file as if it did not contain matching data; this is equivalent to the **--binary-files=without-match** option.

--include=GLOB

Search only files whose base name matches *GLOB* (using wildcard matching as described under **--exclude**). If contradictory **--include** and **--exclude** options are given, the last matching one wins. If no **--include** or **--exclude** options match, a file is included unless the first such option is **--include**.

-r, --recursive

Read all files under each directory, recursively, following symbolic links only if they are on the command line. Note that if no file operand is given, **grep** searches the working directory. This is equivalent to the **-d recurse** option.

-R, --dereference-recursive

Read all files under each directory, recursively. Follow all symbolic links, unlike **-r**.

Other Options

--line-buffered

Use line buffering on output. This can cause a performance penalty.



-U, --binary

Treat the file(s) as binary. By default, under MS-DOS and MS-Windows, **grep** guesses whether a file is text or binary as described for the **--binary-files** option. If **grep** decides the file is a text file, it strips the CR characters from the original file contents (to make regular expressions with **^** and **\$** work correctly). Specifying **-U** overrides this guesswork, causing all files to be read and passed to the matching mechanism verbatim; if the file is a text file with CR/LF pairs at the end of each line, this will cause some regular expressions to fail. This option has no effect on platforms other than MS-DOS and MS-Windows.

-z, --null-data

Treat input and output data as sequences of lines, each terminated by a zero byte (the ASCII NUL character) instead of a newline. Like the **-Z** or **--null** option, this option can be used with commands like **sort -z** to process arbitrary file names.

REGULAR EXPRESSIONS [top](#)

A regular expression is a pattern that describes a set of strings. Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions.

grep understands three different versions of regular expression syntax: “basic” (BRE), “extended” (ERE) and “perl” (PCRE). In GNU **grep**, basic and extended regular expressions are merely different notations for the same pattern-matching functionality. In other implementations, basic regular expressions are ordinarily less powerful than extended, though occasionally it is the other way around. The following description applies to extended regular expressions; differences for basic regular expressions are summarized afterwards. Perl-compatible regular expressions have different functionality, and are documented in [pcre2syntax\(3\)](#) and [pcre2pattern\(3\)](#), but work only if PCRE support is enabled.

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters



and digits, are regular expressions that match themselves. Any meta-character with special meaning may be quoted by preceding it with a backslash.

The period `.` matches any single character. It is unspecified whether it matches an encoding error.

Character Classes and Bracket Expressions

A *bracket expression* is a list of characters enclosed by `[` and `]`. It matches any single character in that list. If the first character of the list is the caret `^` then it matches any character *not* in the list; it is unspecified whether it matches an encoding error. For example, the regular expression `[0123456789]` matches any single digit.

Within a bracket expression, a *range expression* consists of two characters separated by a hyphen. It matches any single character that sorts between the two characters, inclusive, using the locale's collating sequence and character set. For example, in the default C locale, `[a-d]` is equivalent to `[abcd]`. Many locales sort characters in dictionary order, and in these locales `[a-d]` is typically not equivalent to `[abcd]`; it might be equivalent to `[aBbCcDd]`, for example. To obtain the traditional interpretation of bracket expressions, you can use the C locale by setting the `LC_ALL` environment variable to the value `C`.

Finally, certain named classes of characters are predefined within bracket expressions, as follows. Their names are self explanatory, and they are `[:alnum:]`, `[:alpha:]`, `[:blank:]`, `[:cntrl:]`, `[:digit:]`, `[:graph:]`, `[:lower:]`, `[:print:]`, `[:punct:]`, `[:space:]`, `[:upper:]`, and `[:xdigit:]`. For example, `[:alnum:]` means the character class of numbers and letters in the current locale. In the C locale and ASCII character set encoding, this is the same as `[0-9A-Za-z]`. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket expression.) Most meta-characters lose their special meaning inside bracket expressions. To include a literal `]` place it first in the list. Similarly, to include a literal `^` place it anywhere but first. Finally, to include a literal `-` place it last.

Anchoring

The caret `^` and the dollar sign `$` are meta-characters that respectively match the empty string at the beginning and end of a



line.

The Backslash Character and Special Expressions

The symbols `\<` and `\>` respectively match the empty string at the beginning and end of a word. The symbol `\b` matches the empty string at the edge of a word, and `\B` matches the empty string provided it's *not* at the edge of a word. The symbol `\w` is a synonym for `[_[:alnum:]]` and `\W` is a synonym for `[^_[:alnum:]]`.

Repetition

A regular expression may be followed by one of several repetition operators:

- `?` The preceding item is optional and matched at most once.
- `*` The preceding item will be matched zero or more times.
- `+` The preceding item will be matched one or more times.
- `{n}` The preceding item is matched exactly *n* times.
- `{n,}` The preceding item is matched *n* or more times.
- `{,m}` The preceding item is matched at most *m* times. This is a GNU extension.
- `{n,m}` The preceding item is matched at least *n* times, but not more than *m* times.

Concatenation

Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated expressions.

Alternation

Two regular expressions may be joined by the infix operator `|`; the resulting regular expression matches any string matching either alternate expression.

Precedence

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole expression may be enclosed in parentheses to override these precedence rules and form a subexpression.

Back-references and Subexpressions

The back-reference `\n`, where *n* is a single digit, matches the substring previously matched by the *n*th parenthesized subexpression of the regular expression.

Basic vs Extended Regular Expressions



In basic regular expressions the meta-characters `?`, `+`, `{`, `|`, `(`, and `)` lose their special meaning; instead use the backslashed versions `\?`, `\+`, `\{`, `\|`, `\(`, and `\)`.

EXIT STATUS [top](#)

Normally the exit status is 0 if a line is selected, 1 if no lines were selected, and 2 if an error occurred. However, if the `-q` or `--quiet` or `--silent` is used and a line is selected, the exit status is 0 even if an error occurred.

ENVIRONMENT [top](#)

The behavior of `grep` is affected by the following environment variables.

The locale for category `LC_foo` is specified by examining the three environment variables `LC_ALL`, `LC_foo`, `LANG`, in that order. The first of these variables that is set specifies the locale. For example, if `LC_ALL` is not set, but `LC_MESSAGES` is set to `pt_BR`, then the Brazilian Portuguese locale is used for the `LC_MESSAGES` category. The C locale is used if none of these environment variables are set, if the locale catalog is not installed, or if `grep` was not compiled with national language support (NLS). The shell command `locale -a` lists locales that are currently available.

GREP_COLORS

Controls how the `--color` option highlights output. Its value is a colon-separated list of capabilities that defaults to `ms=01;31:mc=01;31:sl=:cx=:fn=35:ln=32:bn=32:se=36` with the `rv` and `ne` boolean capabilities omitted (i.e., false). Supported capabilities are as follows.

sl= SGR substring for whole selected lines (i.e., matching lines when the `-v` command-line option is omitted, or non-matching lines when `-v` is specified). If however the boolean `rv` capability and the `-v` command-line option are both specified, it applies to context matching lines instead. The default is empty (i.e., the terminal's default



color pair).

- cx=** SGR substring for whole context lines (i.e., non-matching lines when the **-v** command-line option is omitted, or matching lines when **-v** is specified). If however the boolean **rv** capability and the **-v** command-line option are both specified, it applies to selected non-matching lines instead. The default is empty (i.e., the terminal's default color pair).
- rv** Boolean value that reverses (swaps) the meanings of the **sl=** and **cx=** capabilities when the **-v** command-line option is specified. The default is false (i.e., the capability is omitted).

mt=01;31

SGR substring for matching non-empty text in any matching line (i.e., a selected line when the **-v** command-line option is omitted, or a context line when **-v** is specified). Setting this is equivalent to setting both **ms=** and **mc=** at once to the same value. The default is a bold red text foreground over the current line background.

ms=01;31

SGR substring for matching non-empty text in a selected line. (This is only used when the **-v** command-line option is omitted.) The effect of the **sl=** (or **cx=** if **rv**) capability remains active when this kicks in. The default is a bold red text foreground over the current line background.

mc=01;31

SGR substring for matching non-empty text in a context line. (This is only used when the **-v** command-line option is specified.) The effect of the **cx=** (or **sl=** if **rv**) capability remains active when this kicks in. The default is a bold red text foreground over the current line background.

- fn=35** SGR substring for file names prefixing any content line. The default is a magenta text foreground over the terminal's default background.



- ln=32** SGR substring for line numbers prefixing any content line. The default is a green text foreground over the terminal's default background.
- bn=32** SGR substring for byte offsets prefixing any content line. The default is a green text foreground over the terminal's default background.
- se=36** SGR substring for separators that are inserted between selected line fields (:), between context line fields, (-), and between groups of adjacent lines when nonzero context is specified (--). The default is a cyan text foreground over the terminal's default background.
- ne** Boolean value that prevents clearing to the end of line using Erase in Line (EL) to Right (**\33[K**) each time a colored item ends. This is needed on terminals on which EL is not supported. It is otherwise useful on terminals for which the **back_color_erase (bce)** boolean terminfo capability does not apply, when the chosen highlight colors do not affect the background, or when EL is too slow or causes too much flicker. The default is false (i.e., the capability is omitted).

Note that boolean capabilities have no =... part. They are omitted (i.e., false) by default and become true when specified.

See the Select Graphic Rendition (SGR) section in the documentation of the text terminal that is used for permitted values and their meaning as character attributes. These substring values are integers in decimal representation and can be concatenated with semicolons. **grep** takes care of assembling the result into a complete SGR sequence (**\33[...m**). Common values to concatenate include **1** for bold, **4** for underline, **5** for blink, **7** for inverse, **39** for default foreground color, **30** to **37** for foreground colors, **90** to **97** for 16-color mode foreground colors, **38;5;0** to **38;5;255** for 88-color and 256-color modes foreground colors, **49** for default background color, **40** to **47** for background colors, **100** to



107 for 16-color mode background colors, and **48;5;0** to **48;5;255** for 88-color and 256-color modes background colors.

LC_ALL, LC_COLLATE, LANG

These variables specify the locale for the **LC_COLLATE** category, which determines the collating sequence used to interpret range expressions like **[a-z]**.

LC_ALL, LC_CTYPE, LANG

These variables specify the locale for the **LC_CTYPE** category, which determines the type of characters, e.g., which characters are whitespace. This category also determines the character encoding, that is, whether text is encoded in UTF-8, ASCII, or some other encoding. In the C or POSIX locale, all characters are encoded as a single byte and every byte is a valid character.

LC_ALL, LC_MESSAGES, LANG

These variables specify the locale for the **LC_MESSAGES** category, which determines the language that **grep** uses for messages. The default C locale uses American English messages.

POSIXLY_CORRECT

If set, **grep** behaves as POSIX requires; otherwise, **grep** behaves more like other GNU programs. POSIX requires that options that follow file names must be treated as file names; by default, such options are permuted to the front of the operand list and are treated as options. Also, POSIX requires that unrecognized options be diagnosed as “illegal”, but since they are not really against the law the default is to diagnose them as “invalid”.

NOTES [top](#)

This man page is maintained only fitfully; the full documentation is often more up-to-date.

COPYRIGHT [top](#)



Copyright 1998-2000, 2002, 2005-2023 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

BUGS [top](#)

Reporting Bugs

Email bug reports to the bug-reporting address `<bug-grep@gnu.org>`. An email archive [⟨https://lists.gnu.org/mailman/listinfo/bug-grep⟩](https://lists.gnu.org/mailman/listinfo/bug-grep) and a bug tracker [⟨https://debbugs.gnu.org/cgi/pkgreport.cgi?package=grep⟩](https://debbugs.gnu.org/cgi/pkgreport.cgi?package=grep) are available.

Known Bugs

Large repetition counts in the `{n,m}` construct may cause **grep** to use lots of memory. In addition, certain other obscure regular expressions require exponential time and space, and may cause **grep** to run out of memory.

Back-references are very slow, and may require exponential time.

EXAMPLE [top](#)

The following example outputs the location and contents of any line containing “f” and ending in “.c”, within all files in the current directory whose names contain “g” and end in “.h”. The `-n` option outputs line numbers, the `--` argument treats expansions of “*g*.h” starting with “-” as file names not options, and the empty file `/dev/null` causes file names to be output even if only one file name happens to be of the form “*g*.h”.

```
$ grep -n -- 'f.*\.c$' *g*.h /dev/null
argmatch.h:1:/* definitions and prototypes for argmatch.c
```

The only line that matches is line 1 of `argmatch.h`. Note that the regular expression syntax used in the pattern differs from the globbing syntax that the shell uses to match file names.



SEE ALSO [top](#)

Regular Manual Pages

[awk\(1\)](#), [cmp\(1\)](#), [diff\(1\)](#), [find\(1\)](#), [perl\(1\)](#), [sed\(1\)](#), [sort\(1\)](#), [xargs\(1\)](#), [read\(2\)](#), [pcre2\(3\)](#), [pcre2syntax\(3\)](#), [pcre2pattern\(3\)](#), [terminfo\(5\)](#), [glob\(7\)](#), [regex\(7\)](#)

Full Documentation

A complete manual (<https://www.gnu.org/software/grep/manual/>) is available. If the **info** and **grep** programs are properly installed at your site, the command

info grep

should give you access to the complete manual.

COLOPHON [top](#)

This page is part of the *GNU grep* (regular expression file search tool) project. Information about the project can be found at (<https://www.gnu.org/software/grep/>). If you have a bug report for this manual page, send it to bug-grep@gnu.org. This page was obtained from the project's upstream Git repository (<git://git.savannah.gnu.org/grep.git>) on 2023-12-22. (At that time, the date of the most recent commit that was found in the repository was 2023-09-14.) If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

GNU grep 3.11.21-102b-dirty 2019-12-29

GREP(1)

Pages that refer to this page: [look\(1\)](#), [pmrep\(1\)](#), [sed\(1\)](#), [regex\(3\)](#), [regex\(7\)](#), [bridge\(8\)](#), [ip\(8\)](#), [tc\(8\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



gcc(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [ENVIRONMENT](#) | [BUGS](#) | [FOOTNOTES](#) | [SEE ALSO](#) | [AUTHOR](#) | [COPYRIGHT](#) | [COLOPHON](#)

 GCC(1)

GNU

GCC(1)**NAME** [top](#)

gcc - GNU project C and C++ compiler

SYNOPSIS [top](#)

```
gcc [-c|-S|-E] [-std=standard]  
    [-g] [-pg] [-Olevel]  
    [-Wwarn...] [-Wpedantic]  
    [-Idir...] [-Ldir...]  
    [-Dmacro[=defn]...] [-Umacro]  
    [-foption...] [-mmachine-option...]  
    [-o outfile] [@file] infile...
```

Only the most useful options are listed here; see below for the remainder. **g++** accepts mostly the same options as **gcc**.

DESCRIPTION [top](#)

When you invoke GCC, it normally does preprocessing, compilation, assembly and linking. The "overall options" allow you to stop this process at an intermediate stage. For example, the **-c** option says not to run the linker. Then the output consists of object files output by the assembler.

Other options are passed on to one or more stages of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; most of these are not documented here, since you rarely need to use any of them.

Most of the command-line options that you can use with GCC are useful for C programs; when an option is only useful with another language (usually C++), the explanation says so explicitly. If the description for a particular option does not mention a source language, you can use that option with all supported languages.

The usual way to run GCC is to run the executable called **gcc**, or **machine-gcc** when cross-compiling, or **machine-gcc-version** to run a specific version of GCC. When you compile C++ programs, you should invoke GCC as **g++** instead.



The **gcc** program accepts options and file names as operands. Many options have multi-letter names; therefore multiple single-letter options may *not* be grouped: **-dv** is very different from **-d -v**.

You can mix options and other arguments. For the most part, the order you use doesn't matter. Order does matter when you use several options of the same kind; for example, if you specify **-L** more than once, the directories are searched in the order specified. Also, the placement of the **-l** option is significant.

Many options have long names starting with **-f** or with **-W---**for example, **-fmove-loop-invariants**, **-Wformat** and so on. Most of these have both positive and negative forms; the negative form of **-ffoo** is **-fno-foo**. This manual documents only one of these two forms, whichever one is not the default.

Some options take one or more arguments typically separated either by a space or by the equals sign (=) from the option name. Unless documented otherwise, an argument can be either numeric or a string. Numeric arguments must typically be small unsigned decimal or hexadecimal integers. Hexadecimal arguments must begin with the **0x** prefix. Arguments to options that specify a size threshold of some sort may be arbitrarily large decimal or hexadecimal integers followed by a byte size suffix designating a multiple of bytes such as "kB" and "KiB" for kilobyte and kibibyte, respectively, "MB" and "MiB" for megabyte and mebibyte, "GB" and "GiB" for gigabyte and gibibyte, and so on. Such arguments are designated by *byte-size* in the following text. Refer to the NIST, IEC, and other relevant national and international standards for the full listing and explanation of the binary and decimal byte size prefixes.

OPTIONS [top](#)

Option Summary

Here is a summary of all the options, grouped by type. Explanations are in the following sections.

Overall Options

```
-c -S -E -o file -x language -v -###
--help[=class[,...]] --target-help --version
-pass-exit-codes -pipe -specs=file -wrapper @file
-ffile-prefix-map=old=new -fplugin=file
-fplugin-arg-name=arg -fdump-ada-spec[-slim]
-fada-spec-parent=unit -fdump-go-spec=file
```

C Language Options

```
-ansi -std=standard -fgnu89-inline
-fpermitted-flt-eval-methods=standard -aux-info filename
-fallow-parameterless-variadic-functions -fno-asm
-fno-builtin -fno-builtin-function -fgimple -fhosted
-ffreestanding -fopenacc -fopenacc-dim=geom -fopenmp
-fopenmp-simd -fms-extensions -fplan9-extensions
-fsso-struct=endianness -fallow-single-precision
-fcond-mismatch -flax-vector-conversions -fsigned-bitfields
-fsigned-char -funsigned-bitfields -funsigned-char
```



SOURCE_DATE_EPOCH

If this variable is set, its value specifies a UNIX timestamp to be used in replacement of the current date and time in the `"__DATE__"` and `"__TIME__"` macros, so that the embedded timestamps become reproducible.

The value of `SOURCE_DATE_EPOCH` must be a UNIX timestamp, defined as the number of seconds (excluding leap seconds) since 01 Jan 1970 00:00:00 represented in ASCII; identical to the output of `@command{date +%s}` on GNU/Linux and other systems that support the `%s` extension in the `"date"` command.

The value should be a known timestamp such as the last modification time of the source or package and it should be set by the build process.

BUGS

[top](#)

For instructions on reporting bugs, see [<https://gcc.gnu.org/bugs/>](https://gcc.gnu.org/bugs/).

FOOTNOTES

[top](#)

1. On some systems, `gcc -shared` needs to build supplementary stub code for constructors to work. On multi-libbed systems, `gcc -shared` must select the correct support libraries to link against. Failing to supply the correct flags may lead to subtle defects. Supplying them in cases where they are not necessary is innocuous.

SEE ALSO

[top](#)

[gpl\(7\)](#), [gfdl\(7\)](#), [fsf-funding\(7\)](#), [cpp\(1\)](#), [gcov\(1\)](#), [as\(1\)](#), [ld\(1\)](#), [gdb\(1\)](#), [dbx\(1\)](#) and the Info entries for [gcc](#), [cpp](#), [as](#), [ld](#), [binutils](#) and [gdb](#).

AUTHOR

[top](#)

See the Info entry for `gcc`, or <http://gcc.gnu.org/onlinedocs/gcc/Contributors.html> >, for contributors to GCC.

COPYRIGHT

[top](#)

Copyright (c) 1988-2019 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with the Invariant Sections being "GNU General Public License" and "Funding Free Software", the Front-Cover texts being (a) (see below), and with the Back-Cover Texts being (b) (see below). A copy of the license is included in the [gfdl\(7\)](#) man



page.

(a) The FSF's Front-Cover Text is:

A GNU Manual

(b) The FSF's Back-Cover Text is:

You have freedom to copy and modify this GNU Manual, like GNU software. Copies published by the Free Software Foundation raise funds for GNU development.

COLOPHON [top](#)

This page is part of the *gcc* (GNU Compiler Collection) project. Information about the project can be found at <http://gcc.gnu.org/>. If you have a bug report for this manual page, see <http://gcc.gnu.org/bugs/>. This page was obtained from the tarball `gcc-9.5.0.tar.xz` fetched from <ftp://ftp.gwdg.de/pub/misc/gcc/releases/> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

gcc-9.5.0

2022-05-27

GCC(1)

Pages that refer to this page: [as\(1\)](#), [dpkg-architecture\(1\)](#), [uselib\(2\)](#), [backtrace\(3\)](#), [dladdr\(3\)](#), [dlopen\(3\)](#), [lttng-ust-cyg-profile\(3\)](#), [offsetof\(3\)](#), [printf\(3\)](#), [sincos\(3\)](#), [strftime\(3\)](#), [feature_test_macros\(7\)](#), [hier\(7\)](#), [math_error\(7\)](#), [warning::debuginfo\(7stap\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



gdb(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#) | [COLOPHON](#)

 GDB(1)

GNU Development Tools

GDB(1)**NAME** [top](#)

gdb - The GNU Debugger

SYNOPSIS [top](#)

gdb [OPTIONS] [*prog*|*prog procID*|*prog core*]

DESCRIPTION [top](#)

The purpose of a debugger such as GDB is to allow you to see what is going on "inside" another program while it executes -- or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

You can use GDB to debug programs written in C, C++, Fortran and Modula-2.

GDB is invoked with the shell command "gdb". Once started, it reads commands from the terminal until you tell it to exit with the GDB command "quit" or "exit". You can get online help from GDB itself by using the command "help".

You can run "gdb" with no arguments or options; but the most usual way to start GDB is with one argument or two, specifying an executable program as the argument:

```
gdb program
```

You can also start with both an executable program and a core file specified:

```
gdb program core
```

You can, instead, specify a process ID as a second argument or use option "-p", if you want to debug a running process:

```
gdb program 1234
gdb -p 1234
```

would attach GDB to process 1234. With option **-p** you can omit the *program* filename.

Here are some of the most frequently needed GDB commands:

break [*file:*][*function*|*line*]

Set a breakpoint at *function* or *line* (in *file*).

run [*arglist*]

Start your program (with *arglist*, if specified).

bt Backtrace: display the program stack.

print *expr*

Display the value of an expression.

c Continue running your program (after stopping, e.g. at a breakpoint).



next

Execute next program line (after stopping); step *over* any function calls in the line.

edit [*file:*]*function*

look at the program line where it is presently stopped.

list [*file:*]*function*

type the text of the program in the vicinity of where it is presently stopped.

step

Execute next program line (after stopping); step *into* any function calls in the line.

help [*name*]

Show information about GDB command *name*, or general information about using GDB.

quit**exit**

Exit from GDB.

For full details on GDB, see *Using GDB: A Guide to the GNU Source-Level Debugger*, by Richard M. Stallman and Roland H. Pesch. The same text is available online as the "gdb" entry in the "info" program.

OPTIONS[top](#)

Any arguments other than options specify an executable file and core file (or process ID); that is, the first argument encountered with no associated option flag is equivalent to a **--se** option, and the second, if any, is equivalent to a **-c** option if it's the name of a file. Many options have both long and abbreviated forms; both are shown here. The long forms are also recognized if you truncate them, so long as enough of the option is present to be unambiguous.

The abbreviated forms are shown here with **-** and long forms are shown with **--** to reflect how they are shown in **--help**. However, GDB recognizes all of the following conventions for most options:



```
"--option=value"  
"--option value"  
"-option=value"  
"-option value"  
"--o=value"  
"--o value"  
"-o=value"  
"-o value"
```

All the options and command line arguments you give are processed in sequential order. The order makes a difference when the **-x** option is used.

--help

-h List all options, with brief explanations.

--symbols=*file*

-s *file*

Read symbol table from *file*.

--write

Enable writing into executable and core files.

--exec=*file*

-e *file*

Use *file* as the executable file to execute when appropriate, and for examining pure data in conjunction with a core dump.

--se=*file*

Read symbol table from *file* and use it as the executable file.

--core=*file*

-c *file*

Use *file* as a core dump to examine.

--command=*file*

-x *file*

Execute GDB commands from *file*.

--eval-command=*command*

-ex *command*

Execute given GDB *command*.



--init-eval-command=command

-iex

Execute GDB *command* before loading the inferior.

--directory=directory

-d directory

Add *directory* to the path to search for source files.

--nh

Do not execute commands from *~/.config/gdb/gdbinit*, *~/.gdbinit*, *~/.config/gdb/gdbearlyinit*, or *~/.gdbearlyinit*

--nx

-n Do not execute commands from any *.gdbinit* or *.gdbearlyinit* initialization files.

--quiet

--silent

-q "Quiet". Do not print the introductory and copyright messages. These messages are also suppressed in batch mode.

--batch

Run in batch mode. Exit with status 0 after processing all the command files specified with **-x** (and *.gdbinit*, if not inhibited). Exit with nonzero status if an error occurs in executing the GDB commands in the command files.

Batch mode may be useful for running GDB as a filter, for example to download and run a program on another computer; in order to make this more useful, the message

Program exited normally.

(which is ordinarily issued whenever a program running under GDB control terminates) is not issued when running in batch mode.

--batch-silent

Run in batch mode, just like **--batch**, but totally silent. All GDB output is suppressed (stderr is unaffected). This is much quieter than **--silent** and would be useless for an interactive session.



This is particularly useful when using targets that give **Loading section** messages, for example.

Note that targets that give their output via GDB, as opposed to writing directly to "stdout", will also be made silent.

--args *prog* [*arglist*]

Change interpretation of command line so that arguments following this option are passed as arguments to the inferior. As an example, take the following command:

```
gdb ./a.out -q
```

It would start GDB with **-q**, not printing the introductory message. On the other hand, using:

```
gdb --args ./a.out -q
```

starts GDB with the introductory message, and passes the option to the inferior.

--pid=pid

Attach GDB to an already running program, with the PID *pid*.

--tui

Open the terminal user interface.

--readnow

Read all symbols from the given symfile on the first access.

--readnever

Do not read symbol files.

--return-child-result

GDB's exit code will be the same as the child's exit code.

--configuration

Print details about GDB configuration and then exit.

--version

Print version information and then exit.

--cd=directory

Run GDB using *directory* as its working directory, instead of



the current directory.

--data-directory=*directory*

-D Run GDB using *directory* as its data directory. The data directory is where GDB searches for its auxiliary files.

--fullname

-f Emacs sets this option when it runs GDB as a subprocess. It tells GDB to output the full file name and line number in a standard, recognizable fashion each time a stack frame is displayed (which includes each time the program stops). This recognizable format looks like two `\032` characters, followed by the file name, line number and character position separated by colons, and a newline. The Emacs-to-GDB interface program uses the two `\032` characters as a signal to display the source code for the frame.

-b *baudrate*

Set the line speed (baud rate or bits per second) of any serial interface used by GDB for remote debugging.

-l *timeout*

Set timeout, in seconds, for remote debugging.

--tty=*device*

Run using *device* for your program's standard input and output.

SEE ALSO [top](#)

The full documentation for GDB is maintained as a Texinfo manual. If the "info" and "gdb" programs and GDB's Texinfo documentation are properly installed at your site, the command

```
info gdb
```

should give you access to the complete manual.

Using GDB: A Guide to the GNU Source-Level Debugger, Richard M. Stallman and Roland H. Pesch, July 1991.

COPYRIGHT [top](#)

Copyright (c) 1988-2023 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with the Invariant Sections being "Free Software" and "Free Software Needs Free Documentation", with the Front-Cover Texts being "A GNU Manual," and with the Back-Cover Texts as in (a) below.

(a) The FSF's Back-Cover Text is: "You are free to copy and modify this GNU Manual. Buying copies from GNU Press supports the FSF in developing GNU and promoting software freedom."

COLOPHON [top](#)

This page is part of the *gdb* (GNU debugger) project. Information about the project can be found at <http://www.gnu.org/software/gdb/>. If you have a bug report for this manual page, see <http://www.gnu.org/software/gdb/bugs/>. This page was obtained from the tarball `gdb-14.1.tar.gz` fetched from <https://ftp.gnu.org/gnu/gdb/> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

gdb-14.1

2023-12-03

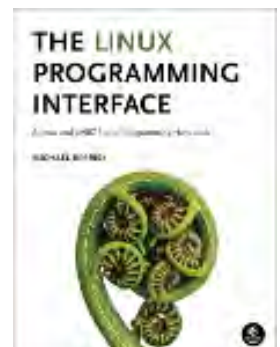
GDB(1)

Pages that refer to this page: [coredumpctl\(1\)](#), [dbpmda\(1\)](#), [pldd\(1\)](#), [pmdbg\(1\)](#), [stap\(1\)](#), [stap-merge\(1\)](#), [ptrace\(2\)](#), [abort\(3\)](#), [backtrace\(3\)](#), [core\(5\)](#), [elf\(5\)](#), [gdbinit\(5\)](#), [proc\(5\)](#), [stappaths\(7\)](#), [crash\(8\)](#), [systemd-coredump\(8\)](#), [systemd-sysext\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



exit(1p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [OPERANDS](#) | [STDIN](#) | [INPUT FILES](#) | [ENVIRONMENT VARIABLES](#) | [ASYNCHRONOUS EVENTS](#) | [STDOUT](#) | [STDERR](#) | [OUTPUT FILES](#) | [EXTENDED DESCRIPTION](#) | [EXIT STATUS](#) | [CONSEQUENCES OF ERRORS](#) | [APPLICATION USAGE](#) | [EXAMPLES](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 EXIT(1P)

POSIX Programmer's Manual

EXIT(1P)**PROLOG** [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

exit – cause the shell to exit

SYNOPSIS [top](#)

exit [*n*]

DESCRIPTION [top](#)

The *exit* utility shall cause the shell to exit from its current execution environment with the exit status specified by the unsigned decimal integer *n*. If the current execution environment is a subshell environment, the shell shall exit from the subshell environment with the specified exit status and continue in the environment from which that subshell environment was invoked; otherwise, the shell utility shall terminate with the specified

exit status. If *n* is specified, but its value is not between 0 and 255 inclusively, the exit status is undefined.

A *trap* on **EXIT** shall be executed before the shell terminates, except when the *exit* utility is invoked in that *trap* itself, in which case the shell shall exit immediately.

OPTIONS [top](#)

None.

OPERANDS [top](#)

See the DESCRIPTION.

STDIN [top](#)

Not used.

INPUT FILES [top](#)

None.

ENVIRONMENT VARIABLES [top](#)

None.

ASYNCHRONOUS EVENTS [top](#)

Default.

STDOUT [top](#)

Not used.

STDERR [top](#)

The standard error shall be used only for diagnostic messages.

OUTPUT FILES [top](#)

None.

EXTENDED DESCRIPTION [top](#)

None.

EXIT STATUS [top](#)

The exit status shall be *n*, if specified, except that the behavior is unspecified if *n* is not an unsigned decimal integer or is greater than 255. Otherwise, the value shall be the exit value of the last command executed, or zero if no command was executed. When *exit* is executed in a *trap* action, the last command is considered to be the command that executed immediately preceding the *trap* action.

CONSEQUENCES OF ERRORS [top](#)

Default.

The following sections are informative.

APPLICATION USAGE [top](#)

None.

EXAMPLES [top](#)

Exit with a *true* value:

```
exit 0
```

Exit with a *false* value:

```
exit 1
```



Propagate error handling from within a subshell:

```
(
    command1 || exit 1
    command2 || exit 1
    exec command3
) > outputfile || exit 1
echo "outputfile created successfully"
```

RATIONALE [top](#)

As explained in other sections, certain exit status values have been reserved for special uses and should be used by applications only for those purposes:

- 126 A file to be executed was found, but it was not an executable utility.
- 127 A utility to be executed was not found.
- >128 A command was interrupted by a signal.

The behavior of *exit* when given an invalid argument or unknown option is unspecified, because of differing practices in the various historical implementations. A value larger than 255 might be truncated by the shell, and be unavailable even to a parent process that uses *waitid()* to get the full exit value. It is recommended that implementations that detect any usage error should cause a non-zero exit status (or, if the shell is interactive and the error does not cause the shell to abort, store a non-zero value in "\$?"), but even this was not done historically in all shells.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

Section 2.14, Special Built-In Utilities

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

IEEE/The Open Group

2017

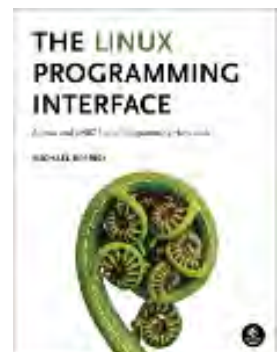
EXIT(1P)

Pages that refer to this page: [return\(1p\)](#), [sh\(1p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



Materials for Topic 2: Useful Linux Commands and Text Editors

Full C Programs

- [prog1.c](#) - a C program that prints a short message about daemons to the terminal using some Linux system calls.
- [heap_fault.c](#) - a C program that attempts to access an address on the **heap** that it hasn't allocated, expectedly causing a segmentation fault.
- [stack_fault.c](#) - a C program that attempts to access an address on the **stack** that it hasn't allocated, expectedly causing a segmentation fault.

Runnable Linux Commands

Quick Links:

- [./short_prompt](#)
- [./long_prompt](#)
- [pwd](#)
- [whoami](#)
- [ls](#)
- [ls -a](#)
- [ls -l](#)
- [ls -al](#)
- [ls dir](#)
- [cd ..](#)
- [cd .](#)
- [cd ~](#)
- [cd /](#)
- [cd dir](#)
- [mkdir](#)
- [touch](#)
- [rmdir](#)
- [rm](#)
- [clear](#)
- [cp](#)
- [mv](#)
- [sudo](#)
- [su](#)
- [sudo -i](#)
- [man](#)
- [info](#)
- [ping](#)
- [grep](#)
- [gcc](#)
- [./executableFile](#)
- [gdb](#)
- [exit](#)
- [vim](#)
- [emacs](#)
- [nano](#)



- [cd](#)
 - [cat](#)
 - [wget](#)
-

- The command:

```
./short_prompt
```

executes code inside a file named `short_prompt` and sources it (applies all the changes to the current session.)

See more details [here](#).

- The command:

```
./long_prompt
```

executes code inside a file named `long_prompt` and sources it (applies all the changes to the current session.)

See more details [here](#).

- The command:

```
pwd
```

stands for 'print working directory' and shows the current folder in which the terminal is positioned. An example of output from this command is `/home/cisc3350/Desktop`, which means that the terminal is currently located in the Desktop folder.

- The command:

```
whoami
```

outputs the username of the user that is currently logged into the device/account and are using this terminal.

Examples of outputs from this command are `miriam`, `student`, `admin`, or `root`.



- The command:

```
ls
```

without any additional command-line arguments, stands for 'list' (verb) and shows the contents of the current directory. This includes only the name of the various non-hidden files and folders included in the current folder.

- The command:

```
ls -a
```

(note that the `-a` is a command-line argument) shows the name of the various files and folders of the current directory, including all the hidden file. A hidden file (or folder) on Linux is one whose name starts with a dot symbol, and examples of hidden files are: `.` (the link to the current directory [itself!],) `..` (the link to the parent directory of the current directory), `.bashrc`, and the `.git` folder that might be found in your directory.

- The command:

```
ls -l
```

(note that the `-l` like 'Larry' is a command-line argument) shows a long printout of the various files and folders of the current directory. In this printout, the information of each file or folder is displayed on a separate line, and includes metadata such as: permissions on using the file, the number of files inside the folder (in case it is a folder,) the username of the user who owns the file, the group of the user who owns the file, the file size in bytes, the recent



modification date of the file, and, of course, the name of the file.

- The command:

```
ls -al
```

(or `ls -la` or `ls -a -l` or `ls -l -a`) is a combination of `ls -a` and `ls -l`: it shows a long printout of the various files and folders of the current directory, including of all the hidden files.

- The command:

```
ls existingDirectory
```

lists the contents of a directory whose name (or path to) is `existingDirectory`. You can also add the `-a` or `-l` command-line arguments if wanted before or after `existingDirectory`. Here, `existingDirectory` could be either a relative or an absolute path to a directory. An absolute path is one that starts with the root directory, like `/home/cisc3350/Desktop/`, and a relative path is one without the forward slash to its left, like `Desktop/games/`.

- The command:

```
cd
```

(without any additional command-line arguments) stands for 'change directory'. When typed into the terminal, it will change the current directory of the terminal to your user's home directory. The Linux computer saves the home directories of all the users within the file `/etc/passwd`, which you can view at any time. To change the home directory to



a different one, you should contact the system administrator of your Linux device/account to change it, or, if the machine belongs to you, you can use of the suggested commands at [this StackOverflow page](#) to change your home directory to a different one.

- The command:

```
cd ..
```

(note that `..` here is a command-line argument) will change the current directory of the terminal to the parent directory of the current folder. For example, if `pwd` says that the current folder is `/home/cisc3350/Desktop/`, typing `cd ..` will change the current directory to `/home/cisc3350/`. Note that the root directory, `/` is special. Theoretically, it doesn't have a parent directory, but if you try doing `cd ..` when inside the root, you will return to ... the root directory itself! So it is like a self-loop.

- The command:

```
cd .
```

will change the current directory of the terminal to the current directory itself! That is, you won't leave the current directory, and `cd .` serves as a self-loop.

- The command:

```
cd ~
```

just as `cd` does, will change the current directory of the terminal to your user's home directory.



- The command:

```
cd /
```

will change the current directory of the terminal to the computer's root directory, which is the grandparent of all directories in Linux. Note that we might mention the term 'root' in several contexts: not only do we have a directory called 'root', but we also most likely have a user whose name is `root`, which is the ultimately-privileged user of the computer who can perform almost any action, including installing new applications, deleting the disk, and accessing files of other users.

- The command:

```
cd existingDirectory
```

will change the current directory of the terminal to the directory indicated by `existingDirectory`. Here, `existingDirectory` could be either a relative or an absolute path to a directory. An absolute path is one that starts with the root directory, like `/home/cisc3350/Desktop/`, and a relative path is one without the forward slash to its left, like `Desktop/games/`.

- The command:

```
mkdir newDirectory
```

stands for 'make directory' and creates a new directory whose name is `newDirectory` inside the current directory. For example, if the terminal is currently situated inside `/home/cisc3350/Desktop/`, and you type `mkdir games`, a new



folder called `games` will be created inside the `Desktop/` directory (in case it is not present there already; if so, an error message that tells that the directory already exists will be displayed to the terminal.)

- The command:

```
touch newFile
```

does the following: if the file named `newFile` already exists in the current directory, its recently modified date will be updated to now. Otherwise, if a file with such a name does not exist, the `touch` will create an empty file called `newFile` in the current directory. For example, typing `touch notes.txt` will create an empty text file called `notes` inside the current directory (in case this file doesn't already exist there.)

- The command:

```
rmdir existingDirectory
```

stands for 'remove directory' and will delete the folder named `ExistingFolder` located inside the current directory. If the `existingDirectory` directory contains any files, or if the directory doesn't exist, an error message will be printed to the terminal. In case you wish to erase a non-empty directory, first delete the files inside the directory, and then `rmdir` the directory itself.

- The command:

```
rm existingFile
```

stands for 'remove' and will delete the file named



`existingFile` located inside the current directory. If the `existingFile` doesn't exist, an error message will be printed to the terminal. An example would be `rm myfile.txt`, using which we can delete a file called `myfile.txt` inside the current directory. Note that, after a file is deleted, it is permanently erased from the computer, unlike files that are deleted on a Windows or a Mac. A GUI version of Linux might feature a 'garbage bin' that temporarily stores deleted files, but the command-line version doesn't have such a feature by default.

- The command:

```
cat existingFile
```

stands for 'concatenate' and will output the contents of the file named `existingFile` located inside the current directory to the terminal's window. You can't edit the contents of `existingFile` using `cat`, but you can see the file full content, regardless of how large the file is. Once the content of `existingFile` is output, the command prompt (e.g., `student@student:~/Desktop$`, which is the line in Linux that awaits your commands) will return back to you to get further commands from you. The reason that the command is called 'concatenate' is that it can actually take the names of several files and print them to the screen, one after the other, which resembles the concatenation of files. An example of using `cat` with a single file is by typing `cat`



`myfile.txt`, which will show the contents of `myfile.txt` on the screen.

- The command:

`clear`

brings the command prompt (e.g., `student@student:~/Desktop$`, which is the line in Linux that awaits your commands) to the top of the terminal in a way that is similar to 'clearing' the commands and data that were previously output to the screen. After typing entering this command, there is going to be nothing on the screen besides the command prompt.

- The command:

`cp path/To/ExistingFile path/To/The/Newly/CreatedCopyFile`

stands for 'copy' and creates a copy of the file that is found in the directory `path/To/` called `ExistingFile` in the destination directory `path/To/The/Newly/` under the name `CreatedCopyFile`. That is, the original file `ExistingFile` remains unchanged, and we just create a 'clone' of this file inside a different folder (and, possibly, with a new name.)

- The command:

`mv path/To/ExistingFile path/To/The/Newly/CreatedCopyFile`

stands for 'move' and performs two actions: (1) it creates a copy of the file `ExistingFile` that is found in the directory `path/To/` and pastes it in the directory `path/To/The/Newly/` under the name `CreatedCopyFile` (2) afterwards, it deletes the original file `ExistingFile` from the `path/To/` directory.



This is essentially the action of 'moving' a file from one place to another, especially if you name the new file with the name of the original one.

- The command:

```
sudo someCommand
```

stands for 'super user do' and lets you execute the command `someCommand` with elevated privileges. The reason that the execution of some commands, such as those that install a new program, require the call to `sudo` is that these actions are considered sensitive and might bring some danger to the computer if done incorrectly. As such, not all users are permitted to perform such actions. One way to perform such an action is by calling `sudo`, which temporarily elevates the user's privileges to those of the 'super user'. However, not all users can use the `sudo` command: for some of you, typing `sudo` will result in an error message like: "user x is not on the sudoers list", which means that you can't request to elevate your privileges to perform that action. In such a case, if the action is necessary, the user should contact the system administrator and request that they add the user to the sudoers list, if the organization/institution allows. An example of using `sudo` is typing `sudo apt-get update`, which updates all the installed software on the Linux device to the latest versions.

Important Note: If you are a regular/unprivileged Linux



user (for example, if you are using your Brooklyn College Linux Account,) you might have no permission to use the `sudo` command, as explained in the paragraph above. No homework assignment or exam will ask you to use the `sudo` command, so don't worry about this current inability to use `sudo`. If you have access to a Linux virtual machine in which you are an administrator user, you should be able to use `sudo`.

- The command:

`su`

stands for 'super user' and elevates your user privileges until you exit the super user mode with the Linux `exit` command. If, when typing this command, you receive an error message like:

```
su: Authentication failure
```

but you know that your user is on the sudoers list, you might instead type the command `sudo -i` below. If you aren't on the sudoers list, you can't use `su`.

- The command:

`sudo -i`

is an alternative to typing `su` and elevates your user privileges until you exit the super user mode with the Linux `exit` command. In many cases, the computer's super user is the user whose name is `root`, so you actually switch to the `root`'s account by entering `sudo -i`. One quick way to recognize that you are in 'super user mode' is to note that



the symbol at the end of the command prompt is the number sign, #, and not the dollar sign, \$.

- The command:

```
man someCommand
```

stands for 'manual' and opens the manual page for the Linux command `someCommand`. For example, entering `man cp` will show the manual page for the 'copy' command. A manual page includes the description of the command, the list of arguments that the command accepts, and any other essential details for the use of the command. If the manual page is longer than the screen, you can scroll down by pressing and holding the down arrow of your keyboard. To exit the manual page and return to the terminal, press the key 'Q' (stands for 'quit') at any time.

- The command:

```
info someCommand
```

opens an information page for the Linux command `someCommand`, which usually contains more information, usage recommendations, and examples about the command than the manual page for this command does. For example, entering `info cp` will show the information page for the 'copy' command. You can navigate an information page in the same way as you do a manual page, by using the arrow keys (to move up and down the page) and the 'Q' key to quit the page.

- The command:

```
ping serverName
```



stands for 'Packet Internet Groper' and checks if the server (essentially, a remote device or webpage) whose name, link, or IP address is `serverName` is responsive over a network, like the Internet. For instance, typing `ping www.google.com` will initiate a connection with the Google website; if the connection is successful, your Linux device and Google will start exchanging data. Otherwise, the data that your Linux device sends won't be 'answered' by Google. This exchange of data will continue endlessly, so you need to press Ctrl + C to finish the exchange and return the command prompt back to you. After you press Ctrl + C, the overall data exchange statistics will be displayed on the terminal before the prompt is returned to you.

A different purpose for using the `ping` command is to check if your device is connected to the Internet. For example, since Google is 'up and running' at all times, getting an error message after entering `ping www.google.com` means that the Linux device is not connected to the Internet (since the connection error is not Google's fault.)

- The command:

```
wget someRemoteFile
```

stands for 'web get' and copies over a file whose URL is `someRemoteFile` from a remote server (e.g., a website). This file could be anything, including text files, Word files, C source files, and even HTML pages (webpages). After this command runs, if the output says that "file such and such



saved", it means that the file was successfully copied to the current directory where the terminal is located. Otherwise, either the file doesn't exist, a bad URL was typed, or there is no Internet connection. Another possibility is that the `wget` program is not installed on your device, in which case you could install it on your device (if you know you have permissions to use the `sudo` command) by typing `sudo apt-get update` followed by `sudo apt-get install wget` and then following the installation commands on the screen.

- The command:

```
grep "pattern" existingDirectory
```

stands for 'global regular expression print' and output the names of the file where `pattern`, which is either a simple search string or some regular expression, is present within the `existingDirectory` folder. For example, entering `grep "Bla Bla!" *` will show all the files in the current directory where the string `Bla Bla!` is present. This is because the `*` symbol is a 'wild-card' that selects all the files in the current directory for this search. The `grep` command has many useful command-line options, which you can learn more about in the `man grep` or `info grep` manual pages.

- The command:

```
gcc -Wall -Wextra -O2 -g -o program program.c
```

compiles the C source code located inside the file `program.c`. See more details [here](#).



- The command:

```
./executableFile
```

executes (runs) a program whose name is `executableFile` and that is located in the current directory using the terminal. The output that this program produces will be displayed in the terminal. In our class, we will be creating and running C programs after we compile them. For example, entering `./prog1` will run the program/application called `prog1` that is located within the current directory, and entering `./a.out` will run the program/application called `a.out` that is located within the current directory. You may, in a similar way, run programs that are located in other directories on your device/account by replacing the dot `.` with the path to the folder where that program that you are trying to run resides. For example, if a program called `my_app` is located in the folder `/home/cisc3350/Desktop`, you can execute it by typing:

```
/home/cisc3350/Desktop/my_app
```

in the terminal and then pressing ENTER/RETURN.

- The command:

```
gdb ./executableFile
```

runs the [GNU C Debugger](#) software whose purpose is to run the program located in the current directory whose name is `executableFile` and trace/debug this program. Once you enter this command, you will enter the debugger's menu. To start running the program, type `run`. If



an error occurs during the execution, the debugger will pause the execution and show what the error is and where (inside which function and on which line) it happened. You can then type the `backtrace` (or `bt`, in short) command to show the sequence of function calls starting from the `main` function that describes how this faulty statement was reached. To exit the debugger app, press 'Q'. The manual page of `gdb` features the full list of commands that you can use within the debugger, such as printing the values stored in variable, creating breakpoints, and other useful commands that you can use to find bugs faster!

- The command:

```
exit
```

closes the terminal window. This is an alternative to clicking the 'X' button at the top corner of the terminal application window.

- The command:

```
vim someFile
```

either creates an empty file named `someFile` within the current directory (if a file under this name doesn't exist yet) using the `vim` command-line text editor, or opens an existing file named `someFile`. Within the `vim` editor, you can change, delete, and add content. You can find several internet guides and tutorials on `vim` on our Blackboard page under [Textbook & Guides > **Guides** > "Text Editors -- Tutorials + YouTube + Cheat Sheets"](#).



- The command:

```
emacs someFile
```

either creates an empty file named `someFile` within the current directory (if a file under this name doesn't exist yet) using the `emacs` command-line text editor, or opens an existing file named `someFile`. Within the `emacs` editor, you can change, delete, and add content. You can find several internet guides and tutorials on `emacs` on our Blackboard page under [Textbook & Guides > **Guides** > "Text Editors -- Tutorials + YouTube + Cheat Sheets"](#).

- The command:

```
nano someFile
```

either creates an empty file named `someFile` within the current directory (if a file under this name doesn't exist yet) using the `nano` command-line text editor, or opens an existing file named `someFile`. Within the `nano` editor, you can change, delete, and add content. You can find several internet guides and tutorials on `nano` on our Blackboard page under [Textbook & Guides > **Guides** > "Text Editors -- Tutorials + YouTube + Cheat Sheets"](#).



This website by [Miriam Briskman](#) is licensed under [CC BY-NC 4.0](#).



Topic 3: Review of the C Language

Note: A copy of each one of the sources listed below is included in this packet. To jump to the location of a source copy, click the page link under “Page”.

In case a source isn’t included (e.g., a YouTube video,) click the ↗ (external link) symbol under “Page” to open the external source in a browser window.

| # | Citation & Source Link | Page |
|----|---|------|
| 1 | Tychonievich, Luther. “C (Guide and Reference).” CS 2130, University of Virginia, 2023. URL: https://www.cs.virginia.edu/~jh2jf/courses/cs2130/spring2023/readings/c.html | 385 |
| 2 | Silverman, Joseph H. “C Reference Card (ANSI).” Brown University, 2007. URL: https://www.cheat-sheets.org/saved-copy/CRefCard.v2.2.pdf | 407 |
| 3 | Black, Ashlyn. “C Reference Cheat Sheet.” Cheatography, 12 May 2016. URL: https://cheatography.com/ashlyn-black/cheat-sheets/c-reference/pdf/ | ↗ |
| 4 | Briskman, Miriam. “Materials for Topic 3: Review of the C Language.” <i>Topic 3: Review of the C Language — CISC 3350 Materials</i> , 2023. URL: www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_03/ | 409 |
| 5 | Briskman, Miriam. “helloworld.c .” (C source code) 6 Jan. 2024. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_03/helloworld.c | 158 |
| 6 | Briskman, Miriam. “variables.c .” (C source code) 15 Feb. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_03/variables.c | 411 |
| 7 | “EXIT_SUCCESS(3const) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/EXIT_SUCCESS.3const.html | 416 |
| 8 | Briskman, Miriam. “conditions.c .” (C source code) 8 Feb. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_03/conditions.c | 418 |
| 9 | Briskman, Miriam. “loops.c .” (C source code) 8 Feb. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_03/loops.c | 420 |
| 10 | Briskman, Miriam. “char_arrays_and_pointers.c .” (C source code) 15 Feb. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_03/char_arrays_and_pointers.c | 423 |
| 11 | “NULL(3const) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/NULL.3const.html | 432 |

| # | Citation & Source Link | Page |
|----|--|------|
| 12 | Briskman, Miriam. “files.c .” (C source code) 2 Feb. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_03/files.c | 435 |
| 13 | “EOF(3const) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/EOF.3const.html | 441 |
| 14 | Briskman, Miriam. “functions.c .” (C source code) 22 Feb. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_03/functions.c | 443 |
| 15 | Briskman, Miriam. “structs.c .” (C source code) 8 Feb. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_03/structs.c | 448 |
| 16 | Briskman, Miriam. “error_checking.c .” (C source code) 8 Feb. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_03/error_checking.c | 451 |
| 17 | Briskman, Miriam. “command_line_arguments.c .” (C source code) 8 Feb. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_03/command_line_arguments.c | 453 |

CS 2130

[Readings](#) / C Reference

C (guide and reference)



This is intended to be a practical guide (rather than an authoritative guide) to C, as implemented by clang and gcc for the x86-64 processor family.

Data Types

The `sizeof(...)` operator returns the size of a type in bytes. Thus, `sizeof(int)` is `4`, not `32`.

Primitive

INTEGER

The integer data types are

| name | bits | representation | notes |
|----------------------------|-----------|-------------------------|---|
| <code>_Bool</code> | 1 or more | undefined | rarely used; for all types, <code>0</code> is false, anything else is true |
| <code>char</code> | 8 | signedness undefined | usually used for characters, sometimes for bytes |
| <code>signed char</code> | 8 | 2's complement | |
| <code>unsigned char</code> | 8 | unsigned integer | |
| <code>short</code> | 16 | 2's complement | |
| <code>int</code> | 32 | 2's complement | |
| <code>long</code> | 32 or 64 | 2's complement | 32 bits if compiled with <code>-m32</code> , 64-bit if compiled with <code>-m64</code> , compiler's choice if neither |
| <code>long long</code> | 64 | 2's complement | |

Each has an `unsigned` version (e.g., `unsigned short`, etc). If `unsigned` is used as a type by itself, it means `unsigned int`.



Integer literals will be implicitly cast to the correct type upon assignment; thus `char x = -3` will turn `-3` into an 8-bit value automatically, and `int x = 'x'` will turn `'x'` into a 32-bit value. This only works up to int-sized literals.

To force a literal to be long add a `l` or `L` to the end; to force it to be unsigned add a `u` or `U`. This is generally only needed for very large constants, like `unsigned long very_big = 9223372036854775808uL`.

Character literals are integer literals written with a different syntax. There is no significant difference between `'0'` and `48` other than legibility.

FLOATING-POINT

The floating-point datatypes are

| name | exponent bits | fraction bits | total size | literal syntax |
|--------------------------|---------------|---------------|--------------------|--|
| <code>float</code> | 8 | 23 | 32 bits (4 bytes) | <code>3.1415f</code> (<code>f</code> or <code>F</code> for <code>float</code>) |
| <code>double</code> | 11 | 52 | 64 bits (8 bytes) | <code>3.1415</code> (no suffix) |
| <code>long double</code> | 15 | 64 | 80 bits (10 bytes) | <code>3.1415L</code> (<code>l</code> or <code>L</code> for <code>long</code>) |

Note that `long double` has traditionally only differed from `double` on x86 architectures.

ENUMERATIONS

The `enum` keyword is a special way of defining named integer constants, in ascending order unless otherwise specified.

```
enum { a, b, c, d=100, e };
/* a is 0, b is 1, c is 2, d is 100, and e is 101 */

int f = e; /* equivalent to f = 101 */
```

VOID AND CASTING

There is also a special `void` type that means either "a byte with no known meaning" (if used as part of a pointer type) or "nothing at all" (if used as a return type or parameter list).

Casting between integer types truncates (if going smaller) or zero- or sign-extends (if going larger, depending on the signedness of the value) to fit the available space. Casting to or from floating-point types converts to a nearby representable value (which may be infinity), with the exception that casting from float to int truncates the remainder instead of rounding.



Pointers

For every type, there is a type for a pointer to a value of that type. These are written with a `*` after the type:

```
int *x;    /* points to an int */
char *s;   /* points to a char */
float **w; /* points to a pointer that points to a float */
float ***a; /* points to a pointer that points to a pointer that points to a float */
```

A pointer to any value stored in memory can be taken by using the address-of operator `&`. Thus `&x` is the address of the value stored in `x`, but `&3` is an error because 3 is a literal and does not have an address. You also can't take the address of the result of an expression: `&(x + y)` or `&&x` are both errors as well.

You de-reference pointers with the same syntax used to declare them: a `*` before the variable.

```
int *x = &z;    /* x = pointer to z */
int x1 = *x;    /* x1 == z */
```

You can also de-reference pointers with subscript notation; `*x` and `x[0]` are entirely equivalent, as are `*(x + n)` and `x[n]`.

There is syntactic ambiguity when combining `*` and `[1]`. Is `*a[1]` the same as `*(a[1])` or `(*a)[1]`? This is solved by operator precedence (`[]` before `*`), but is not intuitive to most programmers so you *should always* use parentheses in these cases.

All pointers are the same size (the size of an address in the underlying ISA) regardless of the size of what they are pointing to; thus `sizeof(char *) == sizeof(long double *)`. Two special int types¹ are used to be "an integer the size of a pointer": `size_t` is an `unsigned` integer of this size, and `ssize_t` is a `signed` integer of this size. With the compilers and ISAs we are using this semester `size_t` is the same as `unsigned long` and `ssize_t` is the same as `long`.

When you add an integer to a pointer, the address stored in the pointer increases by a multiple of the `sizeof` the pointed-to type.

```
int x = 10;
int *y = &x;           // y points to x
```



```
int *z = y + 2;           // z points 2 ints after x
long w = ((long)z) - ((long)y); // w is 8, not 2.
```

Compound Types

There are two basic compound types in C: the `struct` and the array.

ARRAY

An array is zero or more values of the same type stored contiguously in memory.

```
int array[1000];           /* an array of 1000 int values */
```

Except when used with `sizeof` and `&`, arrays act exactly like pointers to their first element; notably, this means that `array[23]` does what you expect it to do: access the 24th element of the array.

The `sizeof` an array is the total bytes used by all elements of the array:

```
unsigned x = sizeof(array); /* 4000: sizeof(int) * 1000 */
```

The `&` of an array is the `&` of its first element (i.e., `&array == &(array[0])`).

Parentheses are allowed when declaring types, although their meaning is counter-intuitive to many students:

```
char *pc[10];           /* an array of 10 (char *)s */
```

```
char *(pc[10]);        /* an array of 10 (char *)s */
```

```
char (*pc)[10];        /* a pointer to an array of 10 (char)s */
```

The rule here is that we declare variables *exactly* as we would use them: a pointer to an array would first be dereferenced (`(*pc)`) and then indexed (`(*pc)[i]`) to get a `char` so we declare it as `char (*pc)[10]`.

Arrays literals use curly braces and commas.

```
int x[10] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
```



Unless initialized with a literal like this, the contents of an array are *undefined* (i.e., may be any random values the compiler thinks is most efficient) when created.

Arrays cannot be resized after being created.

STRUCT

A `struct` also stores values contiguously in memory, but the values may have different types and are accessed by name, not index.

```
struct foo {
    long a;
    int b;
    short c;
    char d;
};          /* note the ; at the end; it is REQUIRED! */
```

The name of the resulting type includes the word `struct`

```
struct foo x;
unsigned long y = sizeof(struct foo);
x.b = 1234;
x.a = x.b - 5;
```

Compilers are free to lay out the data elements of a structure with padding between elements if they wish; this is often done in practice because memory tends to be faster when the address of a 4-byte value is a multiple of 4, so in the above example we expect `y` to have a value larger than the minimal 15 bytes needed to store the fields of `struct foo`.

Structures are passed by value; that is, using them as arguments, return types, or with `=` means that all of their fields are copied. This is inefficient for all by the smallest `structs`, so often pointers to structures are passed, not the structures themselves.

Because all pointers are the same size, you can have code use a pointer to a `struct` without knowing what is inside the `struct`; the fields only need to be known for `sizeof` and the `.` operator to work, not for parameter passing.

```
struct baz;          /* just says "a struct of this name exists" */
void swizzle(struct baz *p); /* just says "a function of this name exists" */
```



```
/* Swizzles an array of struct bazu */
* This code does not need to understand what a struct baz is */
void swizzle(struct baz **x, int n) {
    for(int i=0; i<n; i+=1) swizzle(x[i]);
}
```

Structure literals are written using curly braces and commas, optionally with `.fieldname =` prefixes

```
struct a {
    int b;
    double c;
};

/* Both of the following initialize b to 0 and c to 1.0 */
struct a x = { 0, 1.0 };
struct a y = { .b = 0, .c = 1.0 };
```

Unless initialized with a literal like this, the values of fields of a struct are *undefined* (i.e., may be any random values the compiler thinks is most efficient) when created.

typedef

You can give new names to any type by using the `typedef` statement:

```
typedef int Integer;
Integer x = 23;

typedef double ** dpp;
double y0 = 12.34;
double *y1 = &y0;
dpp y = &y1;

struct foo { int x; double y; };
typedef struct foo foo;
foo z;
```



```
z.x = x;  
z.y = **y;
```

`typedef` type names are aliases to the old names; the compiler will treat both the original and new name as equivalent in all type checking.

Sometimes `typedef` is used with *anonymous* `struct`s:

```
typedef struct { int x; double y; } foo;  
foo z;  
z.x = 3;
```

Anonymous `struct`s can also be used directly as type names, though that's quite uncommon

```
struct { int x; double y; } z;  
z.x = 3;
```

Union

A union is like a struct, except that all of the fields are stored in the same memory address. In practice, this means only one of them has a meaningful value at a time.

```
union odd {  
    long long i;  
    double d;  
};  
  
union odd x;  
x.i = 0x1234; /* x's memory now contains 34 12 00 00 00 00 00 00 */  
double y = x.d; /* y is now 2.30235e-320 (those same bytes) */  
  
x.d = 0x1234; /* x's memory now contains 00 00 00 00 00 34 b2 40 */  
long long z = x.i; /* z is now 0x40b2340000000000 (those same bytes) */
```

You can do bad things

C does not try to prevent you from doing bad things.



```

float x = 123.567f; /* A floating-point number */
int y = *((int *)&x); /* An integer made from the same bytes as the floating-point number */

int z[4]; /* An array of 4 integers */
int w = z[254]; /* An integer made from the contents of memory 1000 bytes after the end of z */

const char *s = "hi"; /* compiler makes the string in memory the OS won't allow us to change */
char *t = (char *)s; /* we get a pointer to that memory that C will allow us to change */
t[0] = 'H'; /* we try to change that memory (the OS will crash our program) */

```

C's general attitude is "every rule has an exception" and "the programmer knows best". It might make you do some complicated casting to do things, but it won't stop you if you are determined.

Constant

If a type is preceded by `const`, the compiler is free to perform optimizations that assume that no code will ever change the values of this type after they are first initialized.

As a special syntax, a string literal like `"hello"` does two things:

- 1 It ensures there exists somewhere an array of characters `{'h', 'e', 'l', 'l', 'o', 0}`, typically in read-only memory.
 - Note the `0` at the end (that's byte-0 not character-0). This is how C knows the string is over.
- 2 It returns a `const char *` pointing to the `h`.

Storage Classes

Variable declarations can optionally have one or more **storage class**. The most important of these are:

`const`: Important enough to have it's own section: [Constant](#)

`static` **inside a function**: Declaring a local variable as `static` makes it a global variable that only that one function can see.

Example:

```

int prev(int y) {
    static int q = 0; // only initialized once
    int ans = q;
    q = y;
}

```




```

    return ans;
}
int main() {
    printf("%d\n", prev(3));
    printf("%d\n", prev(1));
    // q = 5; // <- error, only prev can access q
    printf("%d\n", prev(4));
    printf("%d\n", prev(1));
}

prints

0
3
1
4

```

If you need a function to have persistent state, a `static` local is the preferred way to do that.

`static` **for a global:** Declaring a global variable as `static` makes it visible only inside that `.c` file.

If you need a few related functions to share some persistent state, a `static` global is the preferred way to do that. If just one function needs the state, use a `static` local instead.

`extern` **for a global:** Declaring a global variable as `extern` tells the compiler “assume it exists, don’t create it. It will be supplied by a different `.c` file. Connect the two during linking.”

In general, globals are declared as `extern` in `.h` files and re-declared without `extern` in exactly one `.c` file.

If you need a many functions to all access the same persistent state, use a non-`static` global declared as `extern` everywhere it is used except in one `.c` file. Non-`static` globals are associated with various bugs, so you should prefer a `static` global or `static` local instead where possible.

`register`, `volatile`, and `restrict`: These are hints to the compiler. They have no direct impact on code operation, but can make some optimizations work better.

- `register` suggests that this variable be stored in a register, not in memory, and is ignored by many compilers.
- `volatile` tells the optimizer to assume the variable is being changed by something other than the code itself and blocks certain kinds of optimizations that might otherwise assume differently.
- `restrict` can only be used with pointers, and tells the compiler it is safe to assume that no other pointer describes the same memory as this pointer, enabling certain optimizations that might otherwise not be permitted.

Control constructs

Braces and scope

Any statement may be replaced with a sequence of statements inside braces. Variables declared inside a set of braces vanish at the end of those braces.

```
int x;
{
    int y;
    x = y; /* OK, both x and y in scope */
}
y = x; /* ERROR: y is no longer in scope */
```

Flow of control

NICE AND COMMON ONES

if

Any statement may be preceded by `if (...)`; the statement will only be executed if the expression inside the parentheses yields a non-zero value.

Any statement following a statement preceded by `if (...)` may be preceded by `else`; the statement will only be executed if the expression inside the `if`'s parentheses yields a zero value.

while



Any statement may be preceded by `while (...)`; the statement will only be executed if the expression inside the parentheses yields a non-zero value, and will continue to be executed until that condition stops being true.

for

The special construct `for (e1; e2; e3) s;` is equivalent to the following:

```
{
    e1;
    while (e2) {
        s;
        e3;
    }
}
```

with a slight twist: if `s` contains a `continue`, it jumps to `e3` instead of to `while (e2)`.

If `e2` is omitted, it is assumed to be `1`, so `for(;;) s;` repeats `s` forever.

UGLY AND UNCOMMON ONES

do-while

The syntax `do s; while (e);` means the same as `s; while (e) s;`: that is, it always does `s` once before first checking `e`. In my experience, this is used for less than 1% of loops.

label and goto

Any line of code may be preceded by a label: which is an identifier followed by a colon (e.g. `some_label:`).

The `goto some_label;` statement unconditionally jumps to the code identified by that label.

In 1968 Edgar Dijkstra write an article "[Go To Statement Considered Harmful](#)". Since then, the use of `goto` in code has dropped significantly; it's now usually a sign either of over-emphasis on optimization or a shim to avoid having to redesign poorly-organized code. However, there are a few situations where it can be handy, so it does sometimes show up in high-quality code.

switch

The `switch` statement in C may be implemented in several ways by the compiler, but it is designed to be a good match for the "jump table" approach.

The syntax of the `switch` is as follows:



```
switch(i) {
    case 0:
        statements;
        break;
    case 1:
        statements;
        break;
    case 3:
        statements;
        break;
    case 4:
        statements;
        break;
    default:
        statements;
}
```

Conceptually, this is

- a block of code
- with multiple labels
- where the labels are numbered, not named

and it operates like the (invalid) code

```
c_code *targets[5] = { (case 0), (case 1), (default), (case 3), (case 4) };
if (0 <= i && i < 5) goto targets[i];
else goto default;
```

The `break` (as with a `break` in a loop) stops running the code block and goes to the first statement after it.

Many people think of a `switch` as being a nice way to write a long `if/else if` sequence, and are then annoyed by its limitations and quirks: it has to have an integer selector (as this is really an index), and it “falls through” to the next case if there is no `break`. Hence the following example, taken from [wikipedia](https://en.cppreference.com/w/cpp/switch):



```
switch (age) {
    case 1: printf("You're one.\n");           break;
    case 2: printf("You're two.\n");           break;
    case 4: printf("You're four.\n");
    case 5: printf("You're four or five.\n");   break;
    default: printf("You're not 1, 2, 4 or 5!\n");
}
```

Because many programmers make mistakes with `switch`, it is common to see them banned by style, or augmented with a special style, or later languages to use a similar syntax in ways a jump table cannot handle, or mostly C-compatible languages augmenting them with rules like “each case must either end with `break` or with an explicit `fallthrough`/`goto case`”.

Most compilers have several different implementations of `switch` they can pick between; they might use a jump table, a sequence of `if/else if`s, a binary search, etc.

Example: We can change a switch into gotos as follows

```
switch (age) {
    case 1: printf("You're one.\n");           break;
    case 2: printf("You're two.\n");           break;
    case 4: printf("You're four.\n");
    case 5: printf("You're four or five.\n");   break;
    default: printf("You're not 1, 2, 4 or 5!\n");
}

void *locations[] = {&&L1, &&L2, &&L6, &&L3, &&L4};
if (age < 1 || age > 4) goto L5;
else goto *locations[age-1];
if (0) {
    L1: printf("You're one.\n");
        goto L6;
    L2: printf("You're two.\n");
        goto L6;
    L3: printf("You're four.\n");
    L4: printf("You're four or five.\n");
}
```



```
    goto L6;
L5: printf("You're not 1, 2, 4 or 5!\n");
L6: ;
}
```

Note that the `&&label_name` syntax is a GCC and Clang extension and not part of the official C language.

Functions

Most common use

The most common use of functions in C looks much like you are used to from other languages: a return type, a name, a list of typed parameters in parentheses, and a body in braces.

```
void baz(int i, char *b, float c) {
    b[i] = (char)c;
    return;
}
```

It is also common to declare functions before defining them, in part because C requires functions to be declared before use.

```
int is_even(unsigned n);
int is_odd(unsigned n);

int is_even(unsigned n) {
    if (n == 0) return 1;
    else      return is_odd(n - 1);
}

int is_odd(unsigned n) {
    if (n == 0) return 0;
    else      return is_even(n - 1);
}
```



Often the declarations or **function headers** are put in a separate file, called a **header file** and traditionally named with the suffix `.h`. The `#include` directive can thus grab all of these at once, simplifying coding without increasing the size of the resulting `.c` file or the compiled binary.

Syntax variations

C allows several variations to standard function syntax. Most of these are consider bad programming style.

- Function return types can be omitted, defaulting to `int`:

```
min(int a, int b) { return a < b ? a : b; }
```

- Function parameter types can be omitted, defaulting to `int`:

```
min(a, b) { return a < b ? a : b; }
```

- Function parameter types can be specified between the `)` and the `{`:

```
void baz(i, b, c)
int i;
char *b;
float c;
{
    b[i] = (char)c;
    return;
}
```

Technically, this does something called “promotion” and has various quirks; for this and other reasons it is often called “old-style” and generally discouraged.

- A zero-argument function can be written as either

```
int three() {
    return 3;
}
```

or



```
int three(void) {
    return 3;
}
```

- The `main` function (only) will return `0` if it is missing a `return`, and may omit its arguments upon definition.

It's all convention

C passes arguments using a calling convention. This is obeyed blindly by both the caller and the callee; so if the caller thought the callee had different argument types than it did, neither will notice they have a problem; they'll just silently do the wrong thing.

Example: Consider the following pair of files:

baz.c

```
long bar(char *);

/***** adds bar("hello") to its argument *****/
long baz(long x) {
    return bar("hello") + x;
}
```

bar.c

```
/** returns the requested suffix of "ten letter" */
char *bar(long x) {
    char *c = "ten letter";
    return c + (x%10);
}
```

When executed,

- `baz` will put the address of the first character of `"hello"` into the `%rdi` register and then `callq bar`.
- `bar` will look in `%rdi` for an integer, modulo it by 10, and use it to put an address of a character in the string `"ten letter"` into `%rax`



3 `baz` will look in `%rax` for an integer, add `x` to it, and return

This is almost certainly not what was wanted, but no part of it violates the rules.

Variadic functions

The number of arguments in a function is known as the function's **arity**. Many functions have fixed arity, requiring the same number of arguments each time they are invoked, but sometimes it is nice to have a function that has variable arity, or a **variadic** function.

In C, when invoking a function of variable arity the invoking code simply follows the calling convention, putting some arguments in registers and others on the stack. The invoked function then needs to know how many arguments it received. Since it can't tell anything without consulting at least one argument, all variadic functions in C require at least one argument, and almost all use that argument to decide how many (and what type) the other arguments are.

By far the most famous variadic function in C is `printf`, which is defined as

```
int printf(const char *format, ...);
```

Note the trailing `...` means "this is a variadic function." Thus, `printf` may be invoked with any arguments you want, as long as the first is a `const char *` (that is, a string):

```
printf("%s, %s %d, %.2d:%.2d\n", weekday, month, day, hour, min);
```

The `printf` function uses fairly involved rules about `%s` in its first argument to determine how many and what type the other arguments should be.

Writing a variadic function is somewhat complicated by the fact that the extra arguments do not have names. C provides (declared in `stdarg.h`) a special data type `va_list` and a set of special macros to use in accessing variadic arguments.

```
void va_start(va_list ap, argN);  
type va_arg(va_list ap, type);  
void va_end(va_list ap);
```

To use these, you might do something like



```
int sign_swaps(int num0, ...) {
    va_list ap;
    int last = num0;
    int ans = 0;

    va_start(ap, num0);
    while(last != 0) {
        int next = va_arg(ap, int);
        if ((last < 0) != (next < 0)) ans += 1;
        last = next;
    }
    va_end(ap);

    return ans;
}
```

If you want to write variadic functions, you should

- 1 Read all of `man stdarg.h`
- 2 Read all of `man stdarg.h` again, because you almost certainly missed something important
- 3 Look up variadic security vulnerabilities like the [format string attack](#)
- 4 Write good tests, including too-few- and too-many- and wrong-type-argument invocations.

Preprocessor

Before compilers compile code, they run the C Preprocessor. This does several uninteresting tasks like removing comments, but also processes various *macros* and *directives*.

```
#include <somefile.h>
```

Looks for `somefile.h` in the *include path*, a set of directories typically including `/usr/include` and sometimes a few others.

Upon finding the file, it dumps its entire contents into this part of the file, as if you had copy-pasted it here.

```
#include "somefile.h"
```

Looks for `somefile.h` in the current source directory and, if not found there, in the *include path*



Upon finding the file, it dumps its entire contents into this part of the file, as if you had copy-pasted it here.

```
#if expression, #else, #elif expression, and #endif
```

Upon encountering an `#if`, the preprocessor evaluates the truth of the expression, which must contain only literals (and operators) because the preprocessor is not running code. If it is false, all code from that `#if` to the matching `#else`, `#elif`, or `#endif` is removed from the source code as if you had deleted it.

```
#else and #elif expression behave like else and else if (expression) would in C.
```

```
#define NAME anything at all
```

Defines an *object-like macro*. Anywhere `NAME` appears in the source code, this tells the preprocessor to replace it with `anything at all`—literally those exact tokens, as if you had done a global find-and-replace in your source file.

```
#define NAME(a,b,c) anything including a and b and c
```

Defines a *function-like macro*. Anywhere `NAME(x,y,z)` appears in the source code, this tells the preprocessor to replace it with `anything including x and y and z`—that is, it does a find-and-replace with some parameterization.

This is a lexical replacement, not a syntactic one, so you should almost always add parentheses around each argument and around the full expression:

```
#define TIMES2(x) x * 2          /* bad practice */
#define TIMES2b(x) ((x) * 2)   /* good practice */

int x = ! TIMES2(2 + 3);      /* int x = ! 2 + 3 * 2;      (i.e., !2 + 6 == 6) */
int y = ! TIMES2b(2 + 3);    /* int x = ! ((2 + 3) * 2); (i.e., !10 == 0) */
```

If you decide to become a C expert, there is more to know about macros; see https://en.wikipedia.org/wiki/C_preprocessor#Special_macros_and_directives for a reasonable overview.

```
#ifdef NAME, #ifndef NAME
```

These act like `#if`, except instead of checking if something is true they check if a name has been `#defined` (`#ifdef`) or not (`#ifndef`).

A very common use of these macros is to ensure only one copy of an `.h` file is included. For example, `my_file.h` might look like



```
#ifndef __MY_FILE_HAS_BEEN_INCLUDED__
#define __MY_FILE_HAS_BEEN_INCLUDED__

/* file contents here */

#endif
```

This way if a file `#include "my_file.h"` twice (as, for example, because it includes two other `.h` files that each `#include my_file.h`) then the first one will define `__MY_FILE_HAS_BEEN_INCLUDED__` and the second one, seeing `__MY_FILE_HAS_BEEN_INCLUDED__` is already defined, will have all its contents removed by the `#ifndef`.

Example: If we have something like

foo.h

```
#ifndef __FOO_H
#define __FOO_H
int foo;
#endif
```

foo.c

```
int x;
#include "foo.h"
int y;
#include "foo.h"
int z;
```

the `#include` processing will create

```
int x;
#ifndef __FOO_H
#define __FOO_H
int foo;
```



```
#endif
int y;
#ifndef __FOO_H
#define __FOO_H
int foo;

#endif
int z;
```

The first `#ifndef` is true, since `__FOO_H` had not been defined before that line

```
int x;
#define __FOO_H
int foo;

int y;
#ifndef __FOO_H
#define __FOO_H
int foo;

#endif
int z;
```

That means that the second `#ifndef` is false, since the first defined `__FOO_H`

```
int x;
#define __FOO_H
int foo;

int y;
int z;
```

`__FILE__` and `__LINE__`

The preprocessor is guaranteed to define `__FILE__` as an object-like macro expanding to the name of the current file, in quotes, like `"my_file.c"`. The preprocessor is also guaranteed to define `__LINE__` as an object-like macro expanding to the line number on which `__LINE__` appears, like `23`.



These are often used in debugging messages, as e.g. `printf("Error in %s on line %d\n", __FILE__, __LINE__);`.

Because the preprocessor redefines these on its own on each new line of code, they have a special `#line` directive to change them if you need to do that (not the usual `#define`). The `#include` processing and comment removing adds such `#line` directives so that it does not change source line numbers.

```
#error "error message"
```

Shows `error message` as an error message during compilation.

Most C compilers add several other compiler-specific preprocessor directives, like `#warning`, `#pragma message`, `#pragma once`, `#include_next`, `#import`, etc. Each is added to simplify some common task, but also makes code harder to port to other platforms.

1 Defined using `typedef` in `<types.h>`. ↩

Copyright © 2023 John Hott, portions Luther Tychonievich.
Released under the [CC-BY-NC-SA 4.0](https://creativecommons.org/licenses/by-nc-sa/4.0/) license.



C Reference Card (ANSI)

Program Structure/Functions

| | |
|---|-----------------------------|
| <code>type fnc(type₁, ...);</code> | function prototype |
| <code>type name;</code> | variable declaration |
| <code>int main(void) {</code> | main routine |
| <code>declarations</code> | local variable declarations |
| <code>statements</code> | |
| <code>}</code> | |
| <code>type fnc(arg₁, ...) {</code> | function definition |
| <code>declarations</code> | local variable declarations |
| <code>statements</code> | |
| <code>return value;</code> | |
| <code>}</code> | |
| <code>/* */</code> | comments |
| <code>int main(int argc, char *argv[])</code> | main with args |
| <code>exit(argv);</code> | terminate execution |

C Preprocessor

| | |
|---|--|
| include library file | <code>#include <filename></code> |
| include user file | <code>#include "filename"</code> |
| replacement text | <code>#define name text</code> |
| replacement macro | <code>#define name(var) text</code> |
| Example. <code>#define max(A,B) ((A)>(B) ? (A) : (B))</code> | |
| undefine | <code>#undef name</code> |
| quoted string in replace | <code>#</code> |
| Example. <code>#define msg(A) printf("%s = %d", #A, (A))</code> | |
| concatenate args and rescan | <code>##</code> |
| conditional execution | <code>#if, #else, #elif, #endif</code> |
| is <i>name</i> defined, not defined? | <code>#ifdef, #ifndef</code> |
| <i>name</i> defined? | <code>defined(name)</code> |
| line continuation char | <code>\</code> |

Data Types/Declarations

| | |
|--|---|
| character (1 byte) | <code>char</code> |
| integer | <code>int</code> |
| real number (single, double precision) | <code>float, double</code> |
| short (16 bit integer) | <code>short</code> |
| long (32 bit integer) | <code>long</code> |
| double long (64 bit integer) | <code>long long</code> |
| positive or negative | <code>signed</code> |
| non-negative modulo 2 ^m | <code>unsigned</code> |
| pointer to int, float,... | <code>int*, float*,...</code> |
| enumeration constant | <code>enum tag {name₁=value₁,...};</code> |
| constant (read-only) value | <code>type const name;</code> |
| declare external variable | <code>extern</code> |
| internal to source file | <code>static</code> |
| local persistent between calls | <code>static</code> |
| no value | <code>void</code> |
| structure | <code>struct tag {...};</code> |
| create new name for data type | <code>typedef type name;</code> |
| size of an object (type is <code>size_t</code>) | <code>sizeof object</code> |
| size of a data type (type is <code>size_t</code>) | <code>sizeof(type)</code> |

Initialization

| | |
|------------------------|---|
| initialize variable | <code>type name=value;</code> |
| initialize array | <code>type name[]={value₁,...};</code> |
| initialize char string | <code>char name[]="string";</code> |

© 2007 Joseph H. Silverman Permissions on back. v2.2

Constants

| | |
|--|---------------------|
| suffix: long, unsigned, float | 65536L, -1U, 3.0F |
| exponential form | 4.2e1 |
| prefix: octal, hexadecimal | 0, 0x or 0X |
| Example. 031 is 25, 0x31 is 49 decimal | |
| character constant (char, octal, hex) | 'a', '\ooo', '\xhh' |
| newline, cr, tab, backspace | \n, \r, \t, \b |
| special characters | \\, \?, \', \" |
| string constant (ends with '\0') | "abc...de" |

Pointers, Arrays & Structures

| | |
|---|--|
| declare pointer to <i>type</i> | <code>type *name;</code> |
| declare function returning pointer to <i>type</i> | <code>type *f();</code> |
| declare pointer to function returning <i>type</i> | <code>type (*pf)();</code> |
| generic pointer type | <code>void *</code> |
| null pointer constant | <code>NULL</code> |
| object pointed to by <i>pointer</i> | <code>*pointer</code> |
| address of object <i>name</i> | <code>&name</code> |
| array | <code>name[dim]</code> |
| multi-dim array | <code>name[dim₁][dim₂]...</code> |

Structures

| | |
|---------------------------|------------------------|
| <code>struct tag {</code> | structure template |
| <code>declarations</code> | declaration of members |
| <code>};</code> | |

| | |
|--|-----------------------------------|
| create structure | <code>struct tag name</code> |
| member of structure from template | <code>name.member</code> |
| member of pointed-to structure | <code>pointer -> member</code> |
| Example. <code>(*p).x</code> and <code>p->x</code> are the same | |
| single object, multiple possible types | <code>union</code> |
| bit field with <i>b</i> bits | <code>unsigned member: b;</code> |

Operators (grouped by precedence)

| | |
|--|---|
| struct member operator | <code>name.member</code> |
| struct member through pointer | <code>pointer->member</code> |
| increment, decrement | <code>++, --</code> |
| plus, minus, logical not, bitwise not | <code>+, -, !, ~</code> |
| indirection via pointer, address of object | <code>*pointer, &name</code> |
| cast expression to type | <code>(type) expr</code> |
| size of an object | <code>sizeof</code> |
| multiply, divide, modulus (remainder) | <code>*, /, %</code> |
| add, subtract | <code>+, -</code> |
| left, right shift [bit ops] | <code><<, >></code> |
| relational comparisons | <code>>, >=, <, <=</code> |
| equality comparisons | <code>==, !=</code> |
| and [bit op] | <code>&</code> |
| exclusive or [bit op] | <code>^</code> |
| or (inclusive) [bit op] | <code> </code> |
| logical and | <code>&&</code> |
| logical or | <code> </code> |
| conditional expression | <code>expr₁ ? expr₂ : expr₃</code> |
| assignment operators | <code>+=, -=, *=, ...</code> |
| expression evaluation separator | <code>,</code> |

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

Flow of Control

| | |
|----------------------------------|-------------------------------|
| statement terminator | <code>;</code> |
| block delimiters | <code>{ }</code> |
| exit from switch, while, do, for | <code>break;</code> |
| next iteration of while, do, for | <code>continue;</code> |
| go to | <code>goto label;</code> |
| label | <code>label: statement</code> |
| return value from function | <code>return expr</code> |

Flow Constructions

| | |
|------------------|---|
| if statement | <code>if (expr₁) statement₁</code> <code>else if (expr₂) statement₂</code> <code>else statement₃</code> |
| while statement | <code>while (expr)</code> <code>statement</code> |
| for statement | <code>for (expr₁; expr₂; expr₃)</code> <code>statement</code> |
| do statement | <code>do statement</code> <code>while(expr);</code> |
| switch statement | <code>switch (expr) {</code> <code>case const₁: statement₁ break;</code> <code>case const₂: statement₂ break;</code> <code>default: statement</code> <code>}</code> |

ANSI Standard Libraries

| | | | | |
|-------------------------------|------------------------------|-------------------------------|-------------------------------|-------------------------------|
| <code><assert.h></code> | <code><ctype.h></code> | <code><errno.h></code> | <code><float.h></code> | <code><limits.h></code> |
| <code><locale.h></code> | <code><math.h></code> | <code><setjmp.h></code> | <code><signal.h></code> | <code><stdarg.h></code> |
| <code><stddef.h></code> | <code><stdio.h></code> | <code><stdlib.h></code> | <code><string.h></code> | <code><time.h></code> |

Character Class Tests <ctype.h>

| | |
|--|--------------------------|
| alphanumeric? | <code>isalnum(c)</code> |
| alphabetic? | <code>isalpha(c)</code> |
| control character? | <code>isctrl(c)</code> |
| decimal digit? | <code>isdigit(c)</code> |
| printing character (not incl space)? | <code>isgraph(c)</code> |
| lower case letter? | <code>islower(c)</code> |
| printing character (incl space)? | <code>isprint(c)</code> |
| printing char except space, letter, digit? | <code>ispunct(c)</code> |
| space, formfeed, newline, cr, tab, vtab? | <code>isspace(c)</code> |
| upper case letter? | <code>isupper(c)</code> |
| hexadecimal digit? | <code>isxdigit(c)</code> |
| convert to lower case | <code>tolower(c)</code> |
| convert to upper case | <code>toupper(c)</code> |

String Operations <string.h>

| | |
|--|-------------------------------|
| <i>s</i> is a string; <i>cs</i> , <i>ct</i> are constant strings | |
| length of <i>s</i> | <code>strlen(s)</code> |
| copy <i>ct</i> to <i>s</i> | <code>strcpy(s,ct)</code> |
| concatenate <i>ct</i> after <i>s</i> | <code>strcat(s,ct)</code> |
| compare <i>cs</i> to <i>ct</i> | <code>strcmp(cs,ct)</code> |
| only first <i>n</i> chars | <code>strncmp(cs,ct,n)</code> |
| pointer to first <i>c</i> in <i>cs</i> | <code>strchr(cs,c)</code> |
| pointer to last <i>c</i> in <i>cs</i> | <code>strrchr(cs,c)</code> |
| copy <i>n</i> chars from <i>ct</i> to <i>s</i> | <code>memcpy(s,ct,n)</code> |
| copy <i>n</i> chars from <i>ct</i> to <i>s</i> (may overlap) | <code>memmove(s,ct,n)</code> |
| compare <i>n</i> chars of <i>cs</i> with <i>ct</i> | <code>memcmp(cs,ct,n)</code> |
| pointer to first <i>c</i> in first <i>n</i> chars of <i>cs</i> | <code>memchr(cs,c,n)</code> |
| put <i>c</i> into first <i>n</i> chars of <i>s</i> | <code>memset(s,c,n)</code> |

C Reference Card (ANSI)

Input/Output <stdio.h>

Standard I/O

| | |
|---------------------------|---|
| standard input stream | stdin |
| standard output stream | stdout |
| standard error stream | stderr |
| end of file (type is int) | EOF |
| get a character | getchar() |
| print a character | putchar(<i>chr</i>) |
| print formatted data | printf("format", <i>arg1</i> ,...) |
| print to string <i>s</i> | sprintf(<i>s</i> , "format", <i>arg1</i> ,...) |
| read formatted data | scanf("format", & <i>name1</i> ,...) |
| read from string <i>s</i> | sscanf(<i>s</i> , "format", & <i>name1</i> ,...) |
| print string <i>s</i> | puts(<i>s</i>) |

File I/O

| | |
|--|--|
| declare file pointer | FILE * <i>fp</i> ; |
| pointer to named file | fopen("name", "mode") |
| modes: r (read), w (write), a (append), b (binary) | |
| get a character | getc(<i>fp</i>) |
| write a character | putc(<i>chr</i> , <i>fp</i>) |
| write to file | fprintf(<i>fp</i> , "format", <i>arg1</i> ,...) |
| read from file | fscanf(<i>fp</i> , "format", <i>arg1</i> ,...) |
| read and store <i>n</i> elts to * <i>ptr</i> | fread(* <i>ptr</i> , <i>elsize</i> , <i>n</i> , <i>fp</i>) |
| write <i>n</i> elts from * <i>ptr</i> to file | fwrite(* <i>ptr</i> , <i>elsize</i> , <i>n</i> , <i>fp</i>) |
| close file | fclose(<i>fp</i>) |
| non-zero if error | ferror(<i>fp</i>) |
| non-zero if already reached EOF | feof(<i>fp</i>) |
| read line to string <i>s</i> (< max chars) | fgets(<i>s</i> , max, <i>fp</i>) |
| write string <i>s</i> | fputs(<i>s</i> , <i>fp</i>) |

Codes for Formatted I/O: "%-+ 0w.pmc"

| | |
|-------|--|
| - | left justify |
| + | print with sign |
| space | print space if no sign |
| 0 | pad with leading zeros |
| w | min field width |
| p | precision |
| m | conversion character: h short, l long, L long double |
| c | conversion character: d,i integer u unsigned c single char s char string f double (printf) e,E exponential f float (scanf) lf double (scanf) o octal x,X hexadecimal p pointer n number of chars written g,G same as f or e,E depending on exponent |

Variable Argument Lists <stdarg.h>

| | |
|--|---|
| declaration of pointer to arguments | va_list <i>ap</i> ; |
| initialization of argument pointer | va_start(<i>ap</i> , <i>lastarg</i>); |
| <i>lastarg</i> is last named parameter of the function | |
| access next unnamed arg, update pointer | va_arg(<i>ap</i> , <i>type</i>) |
| call before exiting function | va_end(<i>ap</i>); |

Standard Utility Functions <stdlib.h>

| | |
|---|-----------------------------|
| absolute value of int <i>n</i> | abs(<i>n</i>) |
| absolute value of long <i>n</i> | labs(<i>n</i>) |
| quotient and remainder of ints <i>n</i> , <i>d</i> | div(<i>n</i> , <i>d</i>) |
| returns structure with <i>div_t.quot</i> and <i>div_t.rem</i> | |
| quotient and remainder of longs <i>n</i> , <i>d</i> | ldiv(<i>n</i> , <i>d</i>) |
| returns structure with <i>ldiv_t.quot</i> and <i>ldiv_t.rem</i> | |
| pseudo-random integer [0,RAND_MAX] | rand() |
| set random seed to <i>n</i> | srand(<i>n</i>) |
| terminate program execution | exit(<i>status</i>) |
| pass string <i>s</i> to system for execution | system(<i>s</i>) |

Conversions

| | |
|---|--|
| convert string <i>s</i> to double | atof(<i>s</i>) |
| convert string <i>s</i> to integer | atoi(<i>s</i>) |
| convert string <i>s</i> to long | atol(<i>s</i>) |
| convert prefix of <i>s</i> to double | strtod(<i>s</i> , & <i>endp</i>) |
| convert prefix of <i>s</i> (base <i>b</i>) to long | strtoul(<i>s</i> , & <i>endp</i> , <i>b</i>) |
| same, but unsigned long | |

Storage Allocation

| | |
|------------------------|--|
| allocate storage | malloc(<i>size</i>), calloc(<i>nobj</i> , <i>size</i>) |
| change size of storage | newptr = realloc(<i>ptr</i> , <i>size</i>); |
| deallocate storage | free(<i>ptr</i>); |

Array Functions

| | |
|----------------------------|---|
| search array for key | bsearch(<i>key</i> , <i>array</i> , <i>n</i> , <i>size</i> , <i>cmpf</i>) |
| sort array ascending order | qsort(<i>array</i> , <i>n</i> , <i>size</i> , <i>cmpf</i>) |

Time and Date Functions <time.h>

| | |
|--|--|
| processor time used by program | clock() |
| Example. clock()/CLOCKS_PER_SEC is time in seconds | |
| current calendar time | time() |
| time ₂ -time ₁ in seconds (double) | difftime(time ₂ , time ₁) |
| arithmetic types representing times | clock_t, time_t |
| structure type for calendar time comps | struct tm |
| tm_sec seconds after minute | |
| tm_min minutes after hour | |
| tm_hour hours since midnight | |
| tm_mday day of month | |
| tm_mon months since January | |
| tm_year years since 1900 | |
| tm_wday days since Sunday | |
| tm_yday days since January 1 | |
| tm_isdst Daylight Savings Time flag | |

| | |
|--|---|
| convert local time to calendar time | mktime(<i>tp</i>) |
| convert time in <i>tp</i> to string | asctime(<i>tp</i>) |
| convert calendar time in <i>tp</i> to local time | ctime(<i>tp</i>) |
| convert calendar time to GMT | gmtime(<i>tp</i>) |
| convert calendar time to local time | localtime(<i>tp</i>) |
| format date and time info | strftime(<i>s</i> , <i>max</i> , "format", <i>tp</i>) |
| <i>tp</i> is a pointer to a structure of type tm | |

Mathematical Functions <math.h>

Arguments and returned values are double

| | |
|-------------------------------|--|
| trig functions | sin(x), cos(x), tan(x) |
| inverse trig functions | asin(x), acos(x), atan(x) |
| arctan(<i>y/x</i>) | atan2(<i>y</i> , <i>x</i>) |
| hyperbolic trig functions | sinh(x), cosh(x), tanh(x) |
| exponentials & logs | exp(x), log(x), log10(x) |
| exponentials & logs (2 power) | ldexp(x, <i>n</i>), frexp(x, & <i>e</i>) |
| division & remainder | modf(x, <i>ip</i>), fmod(x, <i>y</i>) |
| powers | pow(x, <i>y</i>), sqrt(x) |
| rounding | ceil(x), floor(x), fabs(x) |

Integer Type Limits <limits.h>

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system, followed by minimum required values (if significantly different).

| | | |
|-----------|--------------------|----------------------------|
| CHAR_BIT | bits in char | (8) |
| CHAR_MAX | max value of char | (SCHAR_MAX or UCHAR_MAX) |
| CHAR_MIN | min value of char | (SCHAR_MIN or 0) |
| SCHAR_MAX | max signed char | (+127) |
| SCHAR_MIN | min signed char | (-128) |
| SHRT_MAX | max value of short | (+32,767) |
| SHRT_MIN | min value of short | (-32,768) |
| INT_MAX | max value of int | (+2,147,483,647) (+32,767) |
| INT_MIN | min value of int | (-2,147,483,648) (-32,767) |
| LONG_MAX | max value of long | (+2,147,483,647) |
| LONG_MIN | min value of long | (-2,147,483,648) |
| UCHAR_MAX | max unsigned char | (255) |
| USHRT_MAX | max unsigned short | (65,535) |
| UINT_MAX | max unsigned int | (4,294,967,295) (65,535) |
| ULONG_MAX | max unsigned long | (4,294,967,295) |

Float Type Limits <float.h>

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system.

| | | |
|--------------|---|--------------|
| FLT_RADIX | radix of exponent rep | (2) |
| FLT_DIG | floating point rounding mode | |
| FLT_ROUND | decimal digits of precision | (6) |
| FLT_EPSILON | smallest <i>x</i> so 1.0f + <i>x</i> ≠ 1.0f | (1.1E - 7) |
| FLT_MANT_DIG | number of digits in mantissa | |
| FLT_MAX | maximum float number | (3.4E38) |
| FLT_MAX_EXP | maximum exponent | |
| FLT_MIN | minimum float number | (1.2E - 38) |
| FLT_MIN_EXP | minimum exponent | |
| DBL_DIG | decimal digits of precision | (15) |
| DBL_EPSILON | smallest <i>x</i> so 1.0 + <i>x</i> ≠ 1.0 | (2.2E - 16) |
| DBL_MANT_DIG | number of digits in mantissa | |
| DBL_MAX | max double number | (1.8E308) |
| DBL_MAX_EXP | maximum exponent | |
| DBL_MIN | min double number | (2.2E - 308) |
| DBL_MIN_EXP | minimum exponent | |

January 2007 v2.2. Copyright © 2007 Joseph H. Silverman

Permission is granted to make and distribute copies of this card provided the copyright notice and this permission notice are preserved on all copies.

Send comments and corrections to J.H. Silverman, Math. Dept., Brown Univ., Providence, RI 02912 USA. (jhs@math.brown.edu)

Materials for Topic 3: Review of the C Language

Full C Programs

- [helloworld.c](#) - a C program that prints a short message to the terminal and introduces C comments.
- [variables.c](#) - a C program showing how to work with variables, constants, assignments and basic math, and printf statements.
- [conditions.c](#) - a C program that works with if, if-else-if, and switch statements (conditions).
- [loops.c](#) - a C program that works with for, while, and do-while loops.
- [char_arrays_and_pointers.c](#) - In C, unlike Java, there are no string objects; instead, we have "C-strings", which are arrays of characters!
- [files.c](#) - a C program that reads and writes to files. We will learn more ways when covering File I/O.
- [functions.c](#) - a C program that declares and defines several functions and calls them inside `main()`.
- [structs.c](#) - a C program that introduces C structs (structures).
- [error_checking.c](#) - a C program that reminds us that errors must be checked even after a user entered some input (there are 1000s of ways things can go wrong during a program's execution!)

- [command_line_arguments.c](#) - since we will create programs that take command line arguments (just like gcc does,) we view here how to access the command line arguments that were passed to a C program.

Runnable Linux Commands

- The command:

```
gcc -Wall -Wextra -O2 -g -o program program.c
```

compiles the C source code located inside the file `program.c`. See more details [here](#).

- The command:

```
./short_prompt
```

executes code inside a file named `short_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).

- The command:

```
./long_prompt
```

executes code inside a file named `long_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).



This website by [Miriam Briskman](#) is licensed under [CC BY-NC 4.0](#).



```
1  /* C program showing how to work with variables,
2  *   constants, assignments and basic math,
3  *   and printf statements.
4  *
5  *   Miriam Briskman, 2/15/2023
6  *   CISC 3350, Brooklyn College
7  *   Licensed under CC BY-NC 4.0
8  */
9
10
11 // Include a reference to C's standard input and
12 //   output library:
13 #include <stdio.h>
14 // Library defining some frequently used macros
15 //   and functions. We include this library in
16 //   this program because we will be using one
17 //   of its macros, 'EXIT_SUCCESS', in this
18 //   program's code:
19 #include <stdlib.h>
20 // Library with string-manipulating functions:
21 #include <string.h>
22
23 /* Let's talk about constants in C:
24 1. A constant value is set only once and
25    CANNOT be changed throughout the program.
26 2. A constant definition in C begins with
27    the keyword "const".
28 3. The type of the constant must be specified
29    as well: int, double, char, etc.
30 4. The name of a constant should contain all
31    capital letters, digits, and/or underscores,
32    like PI or THIS_CONST_2.
33 5. As with non-constant variables, the value of
34    the constant is assigned using the = symbol.
35 6. A good programming practice is to place
36    constants after the #include statements and
37    before the main function.
38    --> This way, any piece of code defined
39         below the constant's definition has
40         access to the constant!
41 */
42
43 // First, we create a constant storing the ratio
44 //   of a circle's circumference to its diameter (pi).
```



```
45 // --> Since pi is a floating point number,
46 //     we store it as a 'double' constant:
47 const double PI = 3.1415926;
48 // Now, let's have a constant storing the number
49 //     of months in year.
50 // --> A 'short int' constant can store a value
51 //     between 0 to 65,535, so it does not consume
52 //     as much memory as a regular 'int'.
53 const unsigned short int MONTHS_PER_YEAR = 12;
54 // How about string constants? Let's store the name
55 //     of the 1st month of the year.
56 // --> char[] is an array of characters, also called
57 //     a string literal.
58 // --> Usually, the size of the char array should be
59 //     specified. However, since we define the
60 //     constant to be "January", a string literal,
61 //     the compiler is able to find the size needed
62 //     to store all the characters of "January".
63 // --> C does not have string objects, which is why
64 //     we use char arrays instead.
65 // --> Any array in C is stored with a '\0' null
66 //     character at its end to let the computer know
67 //     where the array ends.
68 // --> In our case, MONTH_1 appears in memory as:
69 //     +---+---+---+---+---+---+---+---+
70 //     | J | a | n | u | a | r | y | \0 |
71 //     +---+---+---+---+---+---+---+---+
72 //     in bytes (each char in C contains 8-bits.)
73 const char MONTH_1 [] = "January";
74
75 // Unlike const, which requires extra execution
76 //     time since it needs to access main memory,
77 //     a define statement makes the compiler
78 //     replace each instance of a constant,
79 //     MAX_CHARS in the example below, by 1024
80 //     before the program is compiled. The #define
81 //     statement, therefore, executes much faster
82 //     than a constant definition with 'const'.
83 //     A 'constant' that is defined with #define
84 //     is called a "macro".
85 #define MAX_CHARS 1024
86
87 // main function
88 int main()
```



```
89  {
90      /* Phew! We covered constants. Now about
91         variables in C:
92         1. Unlike constants, variables can have
93            their values changed throughout the
94            program.
95         2. Variables are declared very similarly
96            to constants: just omit the 'const'
97            keyword from the declaration.
98         3. Variables have scopes: the regions in
99            the program where the variable is
100           visible and accessible.
101           --> If a variable is defined inside
102              main(), it is accessible inside
103              main() and inside other functions
104              that main calls and passes the
105              variable as an argument to those
106              functions.
107      */
108      char    newline_character = '\n',
109             tab_character = '\t',
110             firstEnglishLetter = 'A';
111      unsigned short grade = 100;
112      short    NYC_boroughs = 5;
113      int      whowantsToBeAMillionaire = 1000000;
114      long     billion = 1000000000;
115      long long trillion = 1000000000000;
116      double   my_circle_diameter = 2.5;
117      long double one_ninth = 0.1111111111;
118      // C does not have a boolean data type!
119      // Use:
120      int true = 1,
121         false = 0;
122      // instead.
123
124      // What would the following printing
125      // statements print to the screen?
126      double circumference = my_circle_diameter * PI;
127      printf ("The circumference of my "
128             "circle is: %f\n", circumference);
129
130      // When the strings that we print are long,
131      // we can 'chop' them into smaller pieces
132      // and include them on separate lines,
```



```
133 // just as we did in the printf()
134 // statement above. The C compiler will
135 // then implicitly concatenate these
136 // strings.
137
138 printf ("I%cLove%cNew%cYork%c", newline_character,
139        newline_character, newline_character,
140        newline_character);
141
142 printf ("I%cLove%cNew%cYork%c", tab_character,
143        tab_character, tab_character,
144        newline_character);
145
146 printf ("This fancy restaurant, which has "
147        "locations in all %hd boroughs, received "
148        "the grade of %c.\n",
149        NYC_boroughs, firstEnglishLetter);
150
151 printf ("The 1st of the %d months, %s, "
152        "has %zu letters!\n",
153        MONTHS_PER_YEAR, MONTH_1, strlen(MONTH_1));
154
155 printf ("1/9, rounded to 2 decimal places, "
156        "is: %.2Lf\n", one_ninth);
157
158 printf ("Musk once had a %d, then a %ld, and soon "
159        "he'll have a %lld!\n",
160        whoWantsToBeAMillionaire, billion, trillion);
161
162 printf ("Since a numerical grade is non-negative "
163        "and is a small number, we use an unsigned "
164        "short variable, as with the grade: %hu.\n"
165        "An unsigned int can only store a zero or "
166        "a positive number.\n",
167        grade);
168
169 // Above, character sequences like:
170 // %f, %c, %d, %s, %u, ...
171 // are called format specifiers and tell
172 // printf() what kind of a variable is being
173 // printed and at what location within the
174 // printout the variable should show up. This
175 // is the way C prints messages to the
176 // terminal similar to Java's concatenation
```



```
177 // using the plus (+) sign.
178 // You can view the full list of specifiers of
179 // printf() at:
180 // https://www.tutorialspoint.com/format-specifiers-in-c
181 //
182 // P.S., our course ain't about memorization.
183 // Whenever you do your homework assignments
184 // or work on exams, please feel free to
185 // simply consult the link above to find
186 // what specifier should be used if, for
187 // example, you want to print the value of,
188 // say, an unsigned long integer.
189
190 return EXIT_SUCCESS;
191
192 // The macro 'EXIT_SUCCESS', on most Linux
193 // computers, is defined as 0. That is,
194 // somewhere in the library <stdlib.h>,
195 // we will find the following #define
196 // statement (or something similar:)
197 //
198 // #define EXIT_SUCCESS 0
199 //
200 // Because there is no guarantee that '0',
201 // (as in 'return 0;') will keep
202 // signifying a successful termination
203 // of a program (e.g., '0' might be
204 // replaced by '1' in some future Linux
205 // devices as representing a successful
206 // termination,) we should, from now on,
207 // exit programs with:
208 //
209 // return EXIT_SUCCESS;
210 //
211 // instead of:
212 //
213 // return 0;
214 //
215 // because 'EXIT_SUCCESS' is a more
216 // portable (reliable) way of exiting
217 // the program.
218 }
219
```



EXIT_SUCCESS(3const) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [CONFORMING TO](#) | [EXAMPLES](#) | [SEE ALSO](#)

EXIT_SUCCESS(3const)

EXIT_SUCCESS(3const)

NAME [top](#)

EXIT_SUCCESS, EXIT_FAILURE - termination status constants

LIBRARY [top](#)

Standard C library (*libc*)

SYNOPSIS [top](#)

```
#include <stdlib.h>

#define EXIT_SUCCESS 0
#define EXIT_FAILURE /* nonzero */
```

DESCRIPTION [top](#)

EXIT_SUCCESS and **EXIT_FAILURE** represent a successful and unsuccessful exit status respectively, and can be used as arguments to the [exit\(3\)](#) function.

CONFORMING TO [top](#)

C99 and later; POSIX.1-2001 and later.

EXAMPLES [top](#)

```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    FILE *fp;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }

    /* Other code omitted */

    fclose(fp);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO [top](#)

[exit\(3\)](#), [sysexits.h\(3head\)](#)

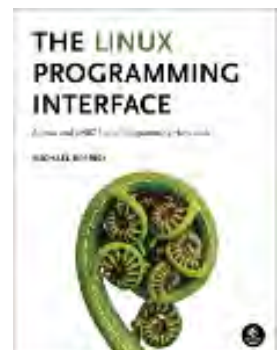
Linux man-pages (unreleased) (date)

EXIT_SUCCESS(3const)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



```
1  /* A C program that shows how condition
2  *   statements (if, if-else, if-else-if,
3  *   and switch) work.
4  *
5  *   Miriam Briskman, 2/8/2023
6  *   CISC 3350, Brooklyn College
7  *   Licensed under CC BY-NC 4.0
8  */
9
10 #include <stdio.h>
11 #include <stdlib.h>
12
13 int main ()
14 {
15     unsigned int salary = 58700;
16
17     // If statement (if the condition is true,
18     //   execute the statement that follows)
19     if (salary >= 50000)
20         printf ("This is a middle-class person.\n");
21
22     // If-else statement (if the 'if' condition
23     //   is false, execute the statement that
24     //   follows the 'else':)
25     if (salary < 32320)
26         printf ("Below NYC's average salary.\n");
27     else
28         printf ("Above NYC's average salary.\n");
29
30     // If-else-if statement:
31     if (salary < 40000)
32         printf ("Tax of 20%.\n");
33     else if (salary < 50000)
34         printf ("Tax of 25%.\n");
35     else if (salary < 60000)
36         printf ("Tax of 30%.\n");
37     else
38         printf ("Tax of 35%.\n");
39
40     // Follow-up question:
41     // Which salaries fall into the last
42     //   'else' statement above?
43
44     // [Note the double %% above. Since % is used
45     //   to denote format specifiers like %d, it
```




```
1  /* A C program that works with for, while,
2  *   and do-while loops.
3  *
4  *   Miriam Briskman, 2/8/2023
5  *   CISC 3350, Brooklyn College
6  *   Licensed under CC BY-NC 4.0
7  */
8
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 int main ()
13 {
14     unsigned int sum = 0,
15                 num = 5,
16                 moves = 3;
17
18     // Above, we must initialize 'sum' to
19     // zero before starting to use 'sum'
20     // because C doesn't automatically
21     // initialize your variables to zero
22     // when they are created. It will
23     // likely contain some random garbage
24     // value, e.g., 1049472647.
25
26     // for loop:
27
28     int i;
29
30     for (i = 1; i <= 10; i++)
31     {
32         sum = sum + i;
33     }
34
35     // We could've also written:
36     // sum += i;
37     // instead of
38     // sum = sum + i;
39
40     // Note: writing 'int i = 1'
41     // (i.e., for (int i = 1 ....))
42     // inside the for loop's header
```



```
43     // is unsupported in earlier
44     // versions of the C language.
45     // As such, make sure to declare
46     // the i variable outside of the
47     // for loop, just like we did
48     // above: int i;
49
50     printf ("The young Carl Gauss computed "
51            "the sum of the 1st 10 positive "
52            "integers to be %u in a few "
53            "moments of thought!\n", sum);
54
55     // while loop:
56
57     printf ("Upside-down pyramid:\n");
58
59     while (num > 0)
60     {
61         for (i = num; i > 0; i--)
62             printf ("* ");
63         printf("\n");
64         num--;
65     }
66
67     // do-while loop:
68
69     printf("Claw moves remaining: %d,\n", moves);
70
71     do
72     {
73         moves--;
74         printf("Claw moves remaining: %d,\n",
75              moves);
76     } while (moves > 0);
77
78     printf("Now, send the claw down and "
79            "catch a toy!\n");
80
81     // *****
82
83     /* Follow-up question:
84
85        How will the following while loop behave,
```

```
86     if we were to uncomment it?
87
88     int true = 1;
89     while (true)
90     {
91         printf ("Till the world ends!\n");
92     }
93     */
94
95     return EXIT_SUCCESS;
96 }
97
```



```

1  /* A C program demonstrating the use of arrays
2  *   of characters, a.k.a. C-strings!
3  *
4  *   Miriam Briskman, 2/15/2023
5  *   CISC 3350, Brooklyn College
6  *   Licensed under CC BY-NC 4.0
7  */
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 // A string library: strlen(), strcmp() ...
12 #include <string.h>
13
14 int main ()
15 {
16     // *****
17
18     /* String literals + <string.h> Functions */
19
20     /* In C, we can create a variable that holds a
21        single char, such as by writing:
22        char c = 'A';
23
24        However, C doesn't have a String 'variable'
25        (object) like Java does. Instead, we use
26        char arrays, which are also called
27        C-strings:
28    */
29     char arr [6] = "Hello";
30
31     /* The reason we wrote:
32        char arr [6]
33        and not
34        char arr [5]
35        is that, despite "Hello" having only 5 letters,
36        we must avail one more memory slot for the
37        null terminator '\0' at the end:
38        +---+---+---+---+---+---+
39        | 0 | 1 | 2 | 3 | 4 | 5 |
40        +---+---+---+---+---+---+
41        | H | e | l | l | o | \0 |
42        +---+---+---+---+---+---+
43        Thus, we need 5 + 1 = 6 items in the array.
44
45        C automatically appends the '\0' character
46        at the end of a string literal (but not at
47        the end of non-literal arrays, as we shall
48        see below.) We, therefore, don't need to
49        write:
50        arr [5] = '\0';
51        since in this case, it is done automatically.
52    */
53
54     // Let's print the array's content:
55     printf ("The string is: %s\n", arr);
56
57     // To access a specific char within the C
58     // string, use brackets []:
59     char fourth_char = arr[4];

```



```

60
61 // Follow-up question:
62 // What character will 'fourth_char' contain,
63 // given that 'arr' contains the string
64 // "Hello"?
65 // Hint:
66 // +---+---+---+---+---+---+
67 // | 0 | 1 | 2 | 3 | 4 | 5 |
68 // +---+---+---+---+---+---+
69 // | H | e | l | l | o | \0 |
70 // +---+---+---+---+---+---+
71
72 // Let's print the character out:
73 printf ("The 4th character of \"%s\" is: %c\n",
74         arr, fourth_char);
75
76 // Note that " (double quotes) is another
77 // escape character that we must escape
78 // for it to be properly printed on the
79 // screen, so we write: \" each time we
80 // need to include " inside a printout.
81
82 // We can find the length of 'arr' by using
83 // <string.h>'s strlen() function:
84 int length = strlen (arr);
85
86 // Fact: the null character '\0' isn't counted
87 // as part of a string's length. As such,
88 // the variable 'length' above will contain
89 // the number 5, not 6.
90
91 // Let's print the length out:
92 printf ("The length of \"%s\" is: "
93         "%d characters\n", arr, length);
94
95 // Here is another C string:
96 char arr2 [3] = "Hi";
97
98 // To check if two strings are equal (that is,
99 // to check if both strings contain the same
100 // characters,) use the strcmp() function
101 // of <string.h>:
102 int result = strcmp (arr, arr2);
103 // strcmp() returns 0 if the strings are equal,
104 // a negative integer if arr < arr2, and a
105 // positive integer if arr > arr2.
106 //
107 // When we say "arr < arr2", we mean that 'arr'
108 // would appear before 'arr2' in a dictionary.
109 //
110 // Examples: (1) strcmp ("No", "No") = 0
111 //           (2) strcmp ("cat", "dog") < 0
112
113 // Let's print the result!
114 if (result < 0)
115     printf ("\"%s\" appears before \"%s\" in "
116           "the dictionary.\n", arr, arr2);
117 else if (result > 0)
118     printf ("\"%s\" appears before \"%s\" in "

```




```

178  /* Note:
179  'sizeof' is an operator in C. It tells how
180  many bytes in memory are needed to store a
181  data type. For example, on most Linux
182  computers, 'char' takes up 1 byte. Because
183  the sizes of data types is computer-
184  dependent, it is advised to use 'sizeof'
185  when allocating arrays on the heap.
186
187  sizeof (array) finds how many bytes
188  were allocated for the array 'array'.
189  Note that, to find the length of a
190  string inside a char array (in case the
191  string occupies less than the entire
192  array's memory) you should use strlen()
193  and not sizeof. Example: If you create
194  the array
195      char name[100] = "Anna";
196  then sizeof (name) will return 100
197  while strlen (name) will return 4.
198  */
199
200  // Check that memory for the heap array was
201  // successfully allocated (i.e., check
202  // that no error happened in the execution
203  // of malloc():)
204  if (heap_array == NULL)
205  {
206      printf ("Error: malloc() failed.\n");
207      exit (EXIT_FAILURE);
208  }
209
210  /* Notes:
211  (1) After asking the computer to allocate
212  memory on the heap for an array or a
213  variable with malloc(), there is no
214  guarantee that memory would be allocated
215  due to various possible errors (e.g.,
216  insufficient memory space left.) In such
217  a case, malloc() would set the array or
218  pointer to NULL, which is a macro whose
219  value is a null pointer (that is, a
220  pointer to the 0th address in memory.)
221
222      +---+---+---+---+---+---+
223      | 0 | 1 | 2 | 3 | 4 | ... |
224      +---+---+---+---+---+---+
225  NULL -> |   |   |   |   |   | ... |
226      +---+---+---+---+---+---+
227
228  That is, NULL is defined in <stdlib.h>
229  as:
230
231  #define NULL ( (void *) 0)
232
233  (i.e., a conversion of 0 to a pointer.)
234
235  This special value is the way of C to
236  tell us that no memory was allocated for

```



```
237     the array/variable that we requested to
238     create, and that we should issue this
239     request again (perhaps, after first
240     resolving the underlying issue or error.)
241 (2) The difference between '\0' (the null
242     terminator) and NULL (the null pointer)
243     is that '\0' is of the 'char' type,
244     while NULL is of a void pointer type.
245     We use them for different purposes: '\0'
246     tells when a string ends, and NULL tells
247     that an array couldn't be created (or
248     that some array-returning function
249     failed.)
250 (3) The exit() function exits the entire
251     program. The only argument it accepts
252     is the "exit code", which tells the mode
253     in which we exit the program. For
254     example, 0 (or 'EXIT_SUCCESS') represents
255     a successful exit (no errors occurred,)
256     and 1 (or 'EXIT_FAILURE') represents a
257     failing program execution that lead to
258     an abrupt exit. For portability, we use:
259
260     exit (EXIT_FAILURE);
261
262     instead of:
263
264     exit (1);
265
266     just as we do with return EXIT_SUCCESS;
267 */
268
269 // Fill the array up with the alphabet:
270 for (i = 0; i < 27 - 1; i++)
271     heap_array[i] = 'A' + i; // "ABCDEFGH...."
272
273 // We put the '\0' character at the end of
274 // the 'heap_array' string, too:
275 heap_array[27 - 1] = '\0';
276
277 /* A pointer variable is a variable that
278     stores the memory address where the
279     value of the variable is located.
280
281     Example:
282     When we write:
283         int n = 5;
284     we create an integer variable whose
285     value is 5.
286
287     When we write:
288         int * addr = &n;
289     we create a pointer variable to an
290     integer (shortly called "integer
291     pointer",) and set 'addr' to contain
292     the address of the variable 'n'.
293
294     Here's how main memory looks:
```



```

296         addr
297         |
298         V
299 +-----+-----+-----+
300 | ... | 5 | ... |
301 +-----+-----+-----+

```

302
303 Summary so far:

- 304 (1) To create a pointer to an int,
305 char, double, etc., we need to
306 place a * sign right after the
307 data type.
- 308 (2) To find the address of a variable,
309 say 'var', write: '&var'.

310
311 Fact: An 'address' is an integer whose
312 size depends on the architecture
313 (features) of the operating system:
314 in a 32-bit OS, an address is an
315 integer of size 4 bytes (= 32 bits)
316 and in a 64-bit OS, an address is
317 of size 8 bytes (= 64 bits).

318
319 Fact: The name of an array, say 'arr',
320 is an address variable itself.
321 For example, to store the address
322 to the start of an integer array
323 'arr', we can do:
324 int * addr = arr;
325 [That is, we don't include the '&'
326 character to the left of 'arr'.]

```

327
328     arr addr
329     |   |
330     V   V
331 +-----+-----+-----+-----+
332 | 101 | 83 | -13 | 2 | 121 |
333 +-----+-----+-----+-----+

```

334
335 Now, let's create some char pointers and
336 see what we can do with pointers!

```

337 */
338
339 // Create a pointer to the stack array:
340 char * ptr_stack_array = stack_array;
341 // 'ptr_stack_array' above points at the
342 // beginning of the array, specifically
343 // at the letter 'A'. In other words,
344 // 'ptr_stack_array' contains the address
345 // of the spot in memory where the letter
346 // 'A' is stored:
347 //
348 // stack_array
349 // |
350 // V
351 // +---+---+---+---+---+---+---+---+
352 // | A | B | C | D | E | F | G | H | ... |
353 // +---+---+---+---+---+---+---+---+
354 // ^

```



```

355 // |
356 // ptr_stack_array
357
358
359 // We can move the pointer to other places!
360 ptr_stack_array += 3;
361 // Now 'ptr_stack_array' points at the letter 'D':
362 //
363 // stack_array
364 // |
365 // v
366 // +---+---+---+---+---+---+---+---+---+
367 // | A | B | C | D | E | F | G | H | ... |
368 // +---+---+---+---+---+---+---+---+---+
369 //
370 //           ^
371 //           |
372 //           ptr_stack_array
373
374 // Print the string beginning at the pointer:
375 printf ("String beginning at ptr_stack_array: %s\n",
376         ptr_stack_array);
377
378 // Follow-up question:
379 // What string will be printed to the screen
380 // as a result of the above printf()?
381
382 // Create a pointer to the heap array:
383 char * ptr_heap_array = heap_array;
384 // Here is how a part of the heap section
385 // of memory would look:
386 //
387 // heap_array
388 // |
389 // v
390 // +---+---+---+---+---+---+---+---+---+
391 // | A | B | C | D | E | F | G | H | ... |
392 // +---+---+---+---+---+---+---+---+---+
393 //
394 //   ^
395 //   |
396 //   ptr_heap_array
397
398 // Move the pointer forward:
399 ptr_heap_array += 4;
400
401 // Follow-up question:
402 // What letter does the pointer 'ptr_heap_array'
403 // point at after the instruction:
404 //   ptr_heap_array += 4;
405 // finished executing?
406 //
407 // heap_array
408 // |
409 // v
410 // +---+---+---+---+---+---+---+---+---+
411 // | A | B | C | D | E | F | G | H | ... |
412 // +---+---+---+---+---+---+---+---+---+
413 //
414 //           ?
415 //           ptr_heap_array

```



```

414
415 // Replace the character at which 'ptr_heap_array'
416 //   currently points by the letter 'Z':
417 ptr_heap_array[0] = 'Z';
418 // The above change affects the 'heap_array'
419 //   itself! That is, one of the letters in
420 //   'heap_array' will change to 'Z'.
421
422 // Print the entire updated heap array:
423 printf ("Heap array: %s\n", heap_array);
424
425 // Functions in <string.h> can be used not
426 //   only on string literals, but also on
427 //   stack and heap char arrays!
428 if (strcmp (stack_array, heap_array) != 0)
429     printf ("The strings %s and %s are NOT the same!\n",
430           stack_array, heap_array);
431 else
432     printf ("The strings %s and %s are the same!\n",
433           stack_array, heap_array);
434
435 // Moving our pointers a bit forward:
436 ptr_heap_array++;
437 ptr_stack_array += 2;
438
439 // Inside the stack:
440 //
441 //   stack_array
442 //   |
443 //   V
444 // +---+---+---+---+---+---+---+---+---+---+
445 // | A | B | C | D | E | F | G | H | ... |
446 // +---+---+---+---+---+---+---+---+---+---+
447 //                                     ^
448 //                                     |
449 //                               ptr_stack_array
450 //
451 // Inside the heap:
452 //
453 //   heap_array
454 //   |
455 //   V
456 // +---+---+---+---+---+---+---+---+---+---+
457 // | A | B | C | D | Z | F | G | H | ... |
458 // +---+---+---+---+---+---+---+---+---+---+
459 //                                     ^
460 //                                     |
461 //                               ptr_heap_array
462
463 // We will now compare the substrings that
464 //   begin with 'ptr_stack_array' and
465 //   with 'ptr_heap_array':
466 if (strcmp (ptr_stack_array, ptr_heap_array) != 0)
467     printf ("The strings %s and %s are NOT the same!\n",
468           ptr_stack_array, ptr_heap_array);
469 else
470     printf ("The strings %s and %s are the same!\n",
471           ptr_stack_array, ptr_heap_array);
472

```



```
473 // Follow-up question:
474 // Which printf() statement will be executed:
475 //   does the string that starts at 'ptr_stack_array'
476 //   have the same characters as the string that
477 //   starts at 'ptr_heap_array'?
478 // Hint: use the above picture of the stack and
479 //   heap sections of main memory.
480
481 // Finally, we need to free the memory for the heap
482 //   array because there is NO garbage collector in
483 //   the C language. (Stack arrays are discarded
484 //   when the program exits the function where the
485 //   array was created, such as the main() function
486 //   in our case.)
487 free (heap_array);
488 // Make sure to keep at least 1 pointer to the
489 //   beginning of the array to be able to free it
490 //   later. This is why we didn't do things like:
491 //   heap_array += 10;
492 //   to keep 'heap_array' pointing at the start.
493
494 // You must free a heap array only once, even if
495 //   several pointers to it were created! It's like
496 //   having several doors in a house: when you
497 //   decide to demolish the house, you demolish
498 //   it only once, not as many times as the number
499 //   of the doors it has :)
500
501 return EXIT_SUCCESS;
502 }
503
```



NULL(3const) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [CONFORMING TO](#) | [NOTES](#) | [CAVEATS](#) | [BUGS](#) | [SEE ALSO](#)

NULL(3const)

NULL(3const)

NAME [top](#)

NULL - null pointer constant

LIBRARY [top](#)

Standard C library (*libc*)

SYNOPSIS [top](#)

```
#include <stddef.h>

#define NULL ((void *) 0)
```

DESCRIPTION [top](#)

NULL represents a null pointer constant, that is, a pointer that does not point to anything.

CONFORMING TO [top](#)

C99 and later; POSIX.1-2001 and later.

NOTES [top](#)

The following headers also provide **NULL**: `<locale.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, `<time.h>`, `<unistd.h>`, and `<wchar.h>`.

CAVEATS [top](#)

It is undefined behavior to dereference a null pointer, and that usually causes a segmentation fault in practice.

It is also undefined behavior to perform pointer arithmetic on it.

NULL - **NULL** is undefined behavior, according to ISO C, but is defined to be `0` in C++.

To avoid confusing human readers of the code, do not compare pointer variables to `0`, and do not assign `0` to them. Instead, always use **NULL**.

NULL shouldn't be confused with **NUL**, which is an `ascii(7)` character, represented in C as `'\0'`.

BUGS [top](#)

When it is necessary to set a pointer variable to a null pointer, it is not enough to use `memset(3)` to zero the pointer (this is usually done when zeroing a struct that contains pointers), since ISO C and POSIX don't guarantee that a bit pattern of all `0`s represent a null pointer. See the EXAMPLES section in `getaddrinfo(3)` for an example program that does this correctly.

SEE ALSO [top](#)

[void\(3type\)](#)

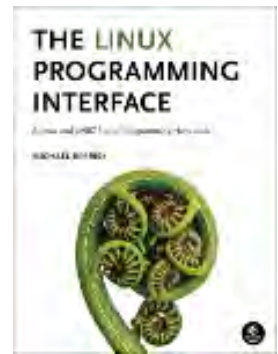
Linux man-pages (unreleased) (date)

NULL(3const)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *[The Linux Programming Interface](#)*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



```
1  /* C program that reads from and writes to files.
2  *
3  *   Miriam Briskman, 2/8/2023
4  *   CISC 3350, Brooklyn College
5  *   Licensed under CC BY-NC 4.0
6  */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10
11 // Constant storing array size:
12 const int ARR_SIZE = 400;
13
14 int main ()
15 {
16     // Declare a pointer to a file structure: (A
17     //   'structure', or shortly 'struct', is
18     //   like an 'object' in C. That is, it
19     //   is a data type that consists of a
20     //   mixture of variables. We will learn
21     //   more about structs later.)
22     FILE * my_file;
23
24     //////////////////////////////////////
25     // Writing into a file //
26     //////////////////////////////////////
27
28     // Open text.txt in a writing mode:
29     my_file = fopen ("text.txt", "w");
30     // The option 'w' creates an empty file for
31     //   writing. If a file with this name
32     //   already exists, its content is erased
33     //   and the file is considered as a new
34     //   empty file.
35
36     // If the file didn't open successfully,
37     //   the pointer will be NULL. We use
38     //   this if statement to check that no
39     //   file-opening errors happened.
40     if (my_file == NULL)
41     {
42         printf ("Error occured while opening text.txt");
```



```
43     exit (EXIT_FAILURE);
44 }
45 else // The file opened successfully.
46 {
47     // Let's add content into the file:
48     fprintf (my_file, "Bla!\nBlu!");
49
50     // Close the file:
51     fclose (my_file);
52 }
53
54 ////////////////////////////////////////////////////////////////////
55 // Reading a file character-by-character //
56 ////////////////////////////////////////////////////////////////////
57
58 // Open text.txt in a reading mode:
59 my_file = fopen ("text.txt", "r");
60 // Note that we attempt to open the
61 //     same file that we used above
62 //     for writing content!
63 // The option "r" allows us to only
64 //     read from the file.
65 // We will learn what other file
66 //     access options exist in Topic 5
67 //     when we cover fopen() more
68 //     closely.
69
70 // Check for errors in opening the file:
71 if (my_file == NULL)
72 {
73     printf ("Error occured while opening text.txt");
74     exit (EXIT_FAILURE);
75 }
76 else // The file opened successfully.
77 {
78     // Create an integer variable:
79     int character;
80
81     // Read a single character from the file:
82     while ((character = getc (my_file)) != EOF)
83     {
84         printf ("The character we read-in was: "
85                "%c, with the ASCII code of %d.\n",
```



```
86         (char) character, character);
87     }
88
89     printf ("We reached the bottom of the file!\n");
90
91     // Close the file:
92     fclose (my_file);
93 }
94
95 /* Notes:
96     (1) The 'while' statement asks whether:
97         (character = getc (my_file))
98         is not equal to EOF. Let's explain
99         this: The function getc() returns
100        the numeric (ASCII) value of the
101        next character in the file 'my_file'
102        and puts this value into 'character'.
103        Then, the while loop checks whether
104        'character' is anything other than the
105        end-of-file flag, which is represented
106        by the EOF constant (this constant is
107        defined in the <stdio.h> library.)
108
109        Bottom line: the 'while' loop reads one
110        character at a time from the file until
111        the end of the file is reached.
112
113     (2) In most Linux computers, the constant
114        EOF is equal to -1. However, since
115        this value depends on how the library
116        <stdio.h> is implemented on every
117        Linux computer type, we should avoid
118        comparing characters with -1 and
119        instead should use EOF (to be on the
120        safe side in case some computer
121        internally defines EOF to be a value
122        other than -1.)
123
124     (3) In the printf() function inside the
125        'while' loop, we print both the
126        ASCII value of the character and the
127        character itself. To convert the
128        numeric (int) value into a char value,
```



```

129     we use 'type casting':
130         (char) character
131     which performs an inline conversion of
132     an integer into its respective
133     character.
134     Examples:
135     (a) 32 -> [SPACE]
136     (b) 46 -> '.'
137     (c) 48 -> '0'
138     (d) 65 -> 'A'
139     (e) 97 -> 'a'
140
141     Here is a quick link to the full
142     ASCII table (256 entries) for your
143     reference:
144     https://www.asciitable.com/
145 */
146
147 ////////////////////////////////////////////////////////////////////
148 // Reading a file line-by-line //
149 ////////////////////////////////////////////////////////////////////
150
151 // Re-open the same file:
152 my_file = fopen ("text.txt", "r");
153
154 if (my_file == NULL)
155 {
156     printf ("Error occured while opening text.txt");
157     exit (EXIT_FAILURE);
158 }
159 else
160 {
161     // Allocate a char array on the heap with
162     //     (hopefully) enough space to store
163     //     a file's contents:
164     char * my_arr = malloc (ARR_SIZE * sizeof(char));
165
166     // Follow-up question:
167     // What is the size of the array 'my_arr'?
168     // (I.e., how many chars can 'my_arr'
169     // contain?)
170     // Hint: check what 'ARR_SIZE' equals to.
171

```



```
172
173 /* Note that we could've also used a stack
174    array instead:
175     char my_arr [ARR_SIZE];
176    The reasons many C programmers prefer to
177    store arrays on the heap are:
178    (1) The stack is smaller than the heap,
179        so larger arrays are advised to be
180        created on the heap.
181    (2) You can change the size of a heap
182        array to a larger (or smaller) size
183        just by calling one function:
184        realloc() [The sizes of stack arrays
185        remain fixed and can't be changed;
186        if you want a larger stack array,
187        a new one must be created.]
188    (3) Sometimes, you want to create a new
189        array inside a function (e.g., as
190        part of some algorithm,) and then
191        pass it to other functions to be used
192        after the function exits. This can
193        be done only with heap arrays because
194        a stack array that was created inside
195        a function will be erased once the
196        function exits (because a stack array
197        is a 'local' variable that is bound
198        only to the scope of the function.)
199        We'll see how to 'return' heap arrays
200        from functions when we review the
201        functions.c
202        program right after we finish talking
203        about files now.
204    */
205
206    int line_counter = 1;
207
208    // The following loop reads entire lines from
209    // the file and then prints them out.
210    // fgets() takes 3 arguments: the array where
211    // the line should be copied to, the size
212    // of this array, and the file variable.
213    // This function reads up to (ARR_SIZE - 1)
214    // characters into 'my_arr' (or until a
```



```
215 // newline character is encountered,) and
216 // puts a null character '\0' at the end
217 // of the copied line inside 'my_arr'.
218 // fgets() equals NULL when we reach the end
219 // of the file or if some other error in
220 // happens during the reading.
221 while (fgets (my_arr, ARR_SIZE, my_file) != NULL)
222 {
223     printf("Line %d: %s", line_counter, my_arr);
224     line_counter++;
225 }
226
227 printf ("\n");
228
229 // Free the heap memory (remember there is
230 // no garbage collection in C!)
231 free (my_arr);
232
233 // Close the file to release all its resources:
234 fclose (my_file);
235 }
236
237 return EXIT_SUCCESS;
238 }
239
```



EOF(3const) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [CONFORMING TO](#) | [CAVEATS](#) | [SEE ALSO](#)

EOF(3const)

EOF(3const)

NAME [top](#)

EOF - end of file or error indicator

LIBRARY [top](#)

Standard C library (*libc*)

SYNOPSIS [top](#)

```
#include <stdio.h>

#define EOF /* ... */
```

DESCRIPTION [top](#)

EOF represents the end of an input file, or an error indication. It is a negative value, of type *int*.

EOF is not a character (it can't be represented by *unsigned char*). It is instead a sentinel value outside of the valid range for valid characters.

CONFORMING TO [top](#)

C99 and later; POSIX.1-2001 and later.

CAVEATS [top](#)

Programs can't pass this value to an output function to "write" the end of a file. That would likely result in undefined behavior. Instead, closing the writing stream or file descriptor that refers to such file is the way to signal the end of that file.

SEE ALSO [top](#)

[feof\(3\)](#), [fgetc\(3\)](#)

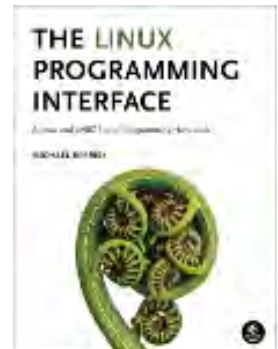
Linux man-pages (unreleased) (date)

[EOF\(3const\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



```
1  /* A C program that declares and defines functions
2  *    of various types and calls them inside main().
3  *
4  *
5  *    Miriam Briskman, 2/22/2023
6  *    CISC 3350, Brooklyn College
7  *    Licensed under CC BY-NC 4.0
8  */
9
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h> // The string library of C
13
14 /* Declarations of functions
15
16     In C and C++, the headers of functions that
17     are defined below the main function must
18     must be repeated above the main function.
19     These headers are called 'declarations'.
20     Otherwise, the compiler won't find them,
21     which will result in a compilation error.
22     Here are the declarations of the four
23     functions that we create in this program:
24 */
25 short sign (double num);
26 void increment (int * num);
27 void increment2 (int num);
28 char * allocate_and_populate (int length, char content);
29
30 int main ()
31 {
32     // Output separator (just for the beauty of the
33     //   printout:)
34     printf ("-----\n");
35
36     double x = -15.2;
37     // Here is our call to the sign() function
38     //   that we created:
39     short the_sign_of_x = sign(x);
40
41     printf ("The number %.3f is ", x);
42
43     if (the_sign_of_x < 0)
```



```
44     printf ("negative!\n");
45 else if (the_sign_of_x > 0)
46     printf ("positive!\n");
47 else
48     printf ("a zero!\n");
49
50 printf ("-----\n");
51
52 int age = 20;
53
54 // Happy Birthday! Let's increment the age
55 //   via a call to the increment()
56 //   function that we created:
57 increment (&age);
58
59 // [Recall that the symbol & retrieves the
60 //   address of the variable 'age'. That is,
61 //   &age is a pointer to the variable 'age'.]
62
63 printf ("Inside main(), after increment(): "
64        "The number is now %d.\n", age);
65
66 // Let's do it again, but this time
67 //   using the increment2 function
68 //   that we created:
69 age = 20;
70 increment2 (age);
71 printf ("Inside main(), after increment2(): "
72        "The number is now %d.\n", age);
73
74 /* Passing by Value vs. Passing by Reference
75
76    When we compile and run this program, we
77    will notice that the result of increment()
78    is different from the result of
79    increment2(). Specifically, only one of them
80    will increment the value in the variable
81    'age'.
82
83    The reason has to do in the way that we
84    pass variables to functions:
85    (1) 'Pass-by-value' means that we write
86        the name of a variable as an argument
87        to the function. The compiler replaces
```



```
88     the name of the variable with the
89     underlying value (e.g., 20), and the
90     function uses this value. Once the
91     function exits, no change will be
92     made to the actual variable that we
93     passed to the function.
94     (2) 'Pass-by-reference' means that we
95     provide the address of a variable as
96     an argument to the function. Then,
97     inside the function, we can access
98     the memory location at that address
99     and make any changes to the variable
100    itself. That means that change WILL
101    affect variables outside the function.
102
103    Follow-up question:
104    Based on the above, which one of
105    increment1() and increment2() will
106    change the number inside the 'age'
107    variable?
108    */
109
110    printf ("-----\n");
111
112    // Let's create an array containing 40 'A'
113    // characters!
114    char * heap_arr = allocate_and_populate (40, 'A');
115
116    // !heap_arr is the same as checking that:
117    // heap_arr == NULL
118    if (!heap_arr)
119    {
120        printf ("Error occured while creating "
121               "the heap array.\n");
122        exit (EXIT_FAILURE);
123    }
124
125    printf ("The C string on the heap contains: %s.\n"
126           "These are %zu letters in total!\n",
127           heap_arr, strlen(heap_arr));
128
129    // Don't forget to free arrays that were
130    // created by other functions!
131    free (heap_arr);
```



```
132
133     printf ("-----\n");
134
135     return EXIT_SUCCESS;
136 }
137
138 // Function returning the sign of a
139 //   floating-point number: -1 if negative,
140 //   0 if zero, or 1 if positive.
141 short sign (double num)
142 {
143     if (num < 0)
144         return -1;
145     if (num > 0)
146         return 1;
147     return 0;
148 }
149
150 // Function incrementing the variable whose
151 //   pointer is given as an argument and
152 //   prints the incremented value, as measured
153 //   inside the function.
154 void increment (int * num)
155 {
156     *num += 1;
157     // Alternative to the above line of code:
158     //   num[0] += 1;
159     // When we put a * to the left of a pointer,
160     //   we access the memory location to which
161     //   the pointer points. This is called
162     //   'de-referencing'.
163     // Summary:
164     // (1) &var - returns the address of a value
165     // (2) *var - return the value at address 'var'
166     // In other words, & and * are opposite
167     //   operations.
168
169     printf ("Inside increment(): The number "
170            "is now %d.\n", *num);
171
172     // Since the return type of the function is
173     //   'void', we don't return any value.
174     //   You may, however, type:
175     return;
```



```
176 }
177
178 // Function incrementing the variable that is
179 //   given as an argument and prints the
180 //   incremented value, as measured inside
181 //   the function.
182 void increment2 (int num)
183 {
184     num += 1;
185     printf ("Inside increment2(): The number "
186           "is now %d.\n", num);
187     return;
188 }
189
190 // Function allocating a heap array of size
191 //   'length' and populating it with the
192 //   character 'content'. A pointer to this new
193 //   array is returned. That is, the created
194 //   array will be accessible even after we
195 //   return from the function. It is the
196 //   programmer's responsibility to free() the
197 //   array later in the program.
198 char * allocate_and_populate (int length, char content)
199 {
200     // Allocate an array on the heap:
201     char * heap_array
202         = malloc ((length + 1) * sizeof(char));
203     if (heap_array == NULL)
204         return NULL;
205     // Populate the entire array with the
206     //   character inside 'content':
207     int i;
208     for (i = 0; i < length; i++)
209         heap_array[i] = content;
210     // Place the null character at the end:
211     heap_array[length] = '\0';
212
213     // Return a pointer (address) to the
214     //   beginning of the array:
215     return heap_array;
216 }
217
```



```
1  /* A C program introducing C structs (structures).
2  *
3  *   Miriam Briskman, 2/8/2023
4  *   CISC 3350, Brooklyn College
5  *   Licensed under CC BY-NC 4.0
6  */
7
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h> // String library
12
13 // C's 'objects':
14 struct Student
15 {
16     char name [100];
17     int age;
18     double GPA;
19     char GPA_letter;
20 }; // <--- Don't forget the semi-colon here at
21     //      the end of the structure definition!
22
23
24 // No methods can be defined within structs!
25 // You can only include variables, arrays,
26 //   and other stuctures. These are
27 //   collectively called 'data fields'.
28
29 int main ()
30 {
31     // You may choose to initialize a structure (set
32     //   the data fields of a new struct instance)
33     //   of) inline using curly braces:
34     struct Student std1 = {"Jane Doe", 20, 3.8, 'A'};
35
36     // Or instantiate one without initializing it
37     //   right away:
38     struct Student std2;
39
40     // At some later point in the program, you can
41     //   start filling out the stucture's data. You
42     //   can access data fields using the dot (.)
```




```
43 // notation:
44 strcpy (std2.name, "Sean Chen");
45 std2.age = 22;
46 std2.GPA = 3.8;
47 std2.GPA_letter = 'A';
48
49 // Data fields of structs are used wherever
50 // you would use regular variables and
51 // arrays. For example, we can print them
52 // out:
53 printf ("The student %s, who is %d years old, "
54         "has %.3f GPA, which is %c.\n",
55         std1.name, std1.age,
56         std1.GPA, std1.GPA_letter);
57
58 // Fact: All the data fields in a struct are
59 // always 'public': data fields can't be
60 // set to 'private' or 'protected' modes.
61
62 // Aside from creating new structures,
63 // programmers in the Linux environment
64 // also frequently use existing structs
65 // that are defined in libraries. We will
66 // encounter more structures in future
67 // chapters and learn how we can use them!
68 //
69 // For example, when we cover Topic 9, we
70 // will learn that the time limit that
71 // a program is allowed to run on a
72 // Linux computer without being
73 // interrupted is represented by a struct
74 // called 'timespec', whose definition is
75 // already provided by the scheduling
76 // library of Linux (<sched.h>). The
77 // definition of this struct is:
78 //
79 // struct timespec
80 // {
81 //     time_t tv_sec; // seconds
82 //     long tv_nsec; // nanoseconds
83 // };
84 //
85 // That is, this struct simply stores
```



```
86 // the duration of that time limit
87 // in seconds and nanoseconds.
88 // Above, 'time_t' is a data type whose
89 // purpose is to store a number
90 // associated with seconds. Whether
91 // 'time_t' is internally defined as
92 // an 'int', 'float', or 'double' is
93 // computer-specific (every computer
94 // might choose to define 'time_t' as
95 // it wishes.)
96
97 return EXIT_SUCCESS;
98 }
99
```



```
1  /* This program reminds us that errors must be
2  *   checked after a user entered some input:
3  *   we shouldn't depend on the user's
4  *   attention to detail and integrity when
5  *   requesting data from the user. Whether
6  *   the user makes a mistake and enters an
7  *   incorrect datum, or if they do so
8  *   intentionally, our program must be
9  *   capable of detecting it and, possibly,
10 *   asking the user to re-enter data.
11 *
12 *   Miriam Briskman, 2/8/2023
13 *   CISC 3350, Brooklyn College
14 *   Licensed under CC BY-NC 4.0
15 */
16
17 #include <stdio.h>
18 #include <stdlib.h>
19
20 int main ()
21 {
22     printf ("Please enter an integer "
23            "larger than 5: >> ");
24
25     int input; // Will store the user's input.
26
27     scanf ("%d", &input); // Read keyboard input.
28
29     // "The additional arguments should point
30     // to already allocated objects of the type
31     // specified by their corresponding format
32     // specifier within the format string."
33     // (Taken from:
34     // http://www.cplusplus.com/reference/cstdio/scanf/)
35     //
36     // The above quote means that, we you decide
37     // to read in input from the user, ensure
38     // that you are reading it into an existing
39     // variable or array, and that the data
40     // type of the variable or array matches the
41     // format that you are trying to read in.
42     // For example, in the above scanf()
```



```
43 // statement, we wish to read in "%d", which
44 // is the format specifier of an integer,
45 // and, indeed, 'input' is a variable of the
46 // type 'integer'.
47
48
49 // Checking whether the correct input has been
50 // entered:
51 if (input > 5)
52     printf ("The integer you entered "
53            "is: %d.\n", input);
54 else
55 {
56     printf ("Error: you entered %d, "
57            "which is less than 5!\n", input);
58     exit(EXIT_FAILURE);
59 }
60
61 return EXIT_SUCCESS;
62 }
63
```



```
1  /* This program shows how to work with command-line
2  *   arguments. So far, we haven't touched this
3  *   aspect of C-programming, but, as we will
4  *   learn later in this course, command-line
5  *   arguments turn out to be super useful and
6  *   convenient to use!
7  *
8  *   Miriam Briskman, 2/8/2023
9  *   CISC 3350, Brooklyn College
10  *   Licensed under CC BY-NC 4.0
11  */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15
16 /* In the header of the main() function:
17  (1) 'argc' has the amount of command-line
18  arguments that were passed to the
19  program.
20  (2) 'argv' is an array of strings carrying
21  the arguments themselves.
22  */
23 int main (int argc, char * argv[])
24 {
25     /* FAQ:
26     Q: Why do we need to have 'int argc',
27        and cannot have 'char * argv[]' alone?
28     A: Because in C, we need to maintain a
29        counter for the number of elements in
30        an array, since arrays/strings in C are
31        not objects, unlike their Java counterparts.
32        In Java, the String object has an accessible
33        size data type, which C doesn't have, so
34        we must keep track of the number of items
35        in this array.
36
37     Q: How do you run a program with command
38        line arguments?
39     A: Suppose that you compile a certain C
40        file called 'prog.c' as:
41           gcc -o prog prog.c
42        You would then run it as:
```



```
43         ./prog
44     In this case, you'd have a single
45     command-line argument provided to
46     the program:
47     argv[0] ---> "./prog"
48
49     If you want to add more command-line
50     arguments, you would run it like:
51         ./prog hello hi 2 -18.5
52     You now have 5 command-line
53     arguments:
54     argv[0] ---> "./prog"
55     argv[1] ---> "hello"
56     argv[2] ---> "hi"
57     argv[3] ---> "2"
58     argv[4] ---> "-18.5"
59
60     All of the command-line arguments are
61     provided as char arrays (strings).
62     For instance, if you want to assign
63     argv[3] to an int variable, you first
64     need to convert the string to an int.
65     One such C function that converts a
66     string to an integer is atoi(), which
67     is defined in the library <stdlib.h>.
68     More on atoi() here:
69     https://cplusplus.com/reference/cstdlib/atoi/
70
71     Fun fact: You've been calling C
72     programs with arguments all along!
73
74     When you type:
75         ls -a
76     in the Linux terminal, you run a
77     program called "ls" and provide
78     the argument "-a" to it, which
79     instructs "ls" to print the contents
80     of the directory, including hidden
81     files (files whose names start with
82     a dot (.))
83     Isn't it cool?
84     */
85
```



```
86 // Let's print how many arguments are provided:
87 printf ("The amount of command line arguments "
88         "is: %d\n", argc);
89 // The 1st command line argument is ALWAYS the
90 //   name of the executable (= C program:)
91 printf ("The name of this program is: %s\n",
92         argv[0]);
93
94 // Therefore, there is ALWAYS going to be at
95 //   least one argument provided to the
96 //   program! We simply ignored it so far.
97 // This also means that 'argc' is going to be
98 //   equal to 1 or greater.
99
100 // Note that, just as in Java, array indices
101 //   begin with 0, so argv[0] is the first
102 //   argument, argv[1] is the 2nd argument,
103 //   etc.
104
105 // Finally, let's print the rest of the command
106 //   -line arguments (if such exist):
107 if (argc == 1)
108     printf ("No command line arguments passed "
109            "besides the program's name.\n");
110 else
111 {
112     int i;
113     for (i = 1; i < argc; i++)
114         printf ("Argument number %d is: "
115                "%s\n", i + 1, argv[i]);
116 }
117
118 // After compiling this source code, we will
119 //   run this program with a few command-line
120 //   arguments.
121
122 // Follow-up question:
123 // What do you think will happen when we provide
124 //   2 or more of our space-separated arguments
125 //   in quotes, like: "wow yummy"?
126 //   For instance:
127 //       ./command_line_arguments "wow yummy"
128 //
```




```
129 // How many arguments will such a quoted
130 // string result in: will they still be
131 // considered as two separate arguments?
132
133 return EXIT_SUCCESS;
134 }
135
```



Topic 4: File I/O

Note: A copy of each one of the sources listed below is included in this packet. To jump to the location of a source copy, click the page link under “Page”.

In case a source isn’t included (e.g., a YouTube video,) click the  (external link) symbol under “Page” to open the external source in a browser window.

| # | Citation & Source Link | Page |
|----|---|------|
| 1 | “stdin(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/stdin.3.html | 459 |
| 2 | “errno(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/errno.3.html | 462 |
| 3 | “open(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/open.2.html | 479 |
| 4 | “read(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/read.2.html | 503 |
| 5 | “write(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/write.2.html | 508 |
| 6 | “fsync(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/fsync.2.html | 514 |
| 7 | “sync(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/sync.2.html | 518 |
| 8 | “close(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/close.2.html | 522 |
| 9 | “lseek(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/lseek.2.html | 527 |
| 10 | “pread(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/pread.2.html | 532 |
| 11 | “truncate(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/truncate.2.html | 536 |
| 12 | “select(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/select.2.html | 541 |
| 13 | “poll(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/poll.2.html | 553 |
| 14 | Briskman, Miriam. “Materials for Topic 4: File I/O.” <i>Topic 4: File I/O — CISC 3350 Materials</i> , 2023. URL: www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_04/ | 564 |
| 15 | Briskman, Miriam. “file_io_syscalls.c .” (C source code) 27 Feb. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_04/file_io_syscalls.c | 566 |

| # | Citation & Source Link | Page |
|----|--|------|
| 16 | Love, Robert. “select_syscall_example.c .” (C source code), <i>Linux System Programming: Talking Directly to the Kernel and C Library</i> , O’Reilly Media, Inc., Sebastopol, CA, 2013, pp. 55–56. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_04/select_syscall_example.c | 577 |
| 17 | Love, Robert. “poll_syscall_example.c .” (C source code), <i>Linux System Programming: Talking Directly to the Kernel and C Library</i> , O’Reilly Media, Inc., Sebastopol, CA, 2013, pp. 60–61. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_04/poll_syscall_example.c | 580 |

stdin(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 stdin(3)

Library Functions Manual

stdin(3)**NAME** [top](#)

stdin, stdout, stderr - standard I/O streams

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
extern FILE *stdin;  
extern FILE *stdout;  
extern FILE *stderr;
```

DESCRIPTION [top](#)

Under normal circumstances every UNIX program has three streams opened for it when it starts up, one for input, one for output, and one for printing diagnostic or error messages. These are typically attached to the user's terminal (see [tty\(4\)](#)) but might instead refer to files or other devices, depending on what the parent process chose to set up. (See also the "Redirection" section of [sh\(1\)](#).)



The input stream is referred to as "standard input"; the output stream is referred to as "standard output"; and the error stream is referred to as "standard error". These terms are abbreviated to form the symbols used to refer to these files, namely *stdin*, *stdout*, and *stderr*.

Each of these symbols is a `stdio(3)` macro of type pointer to *FILE*, and can be used with functions like `fprintf(3)` or `fread(3)`.

Since *FILEs* are a buffering wrapper around UNIX file descriptors, the same underlying files may also be accessed using the raw UNIX file interface, that is, the functions like `read(2)` and `lseek(2)`.

On program startup, the integer file descriptors associated with the streams *stdin*, *stdout*, and *stderr* are 0, 1, and 2, respectively. The preprocessor symbols `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO` are defined with these values in `<unistd.h>`. (Applying `freopen(3)` to one of these streams can change the file descriptor number associated with the stream.)

Note that mixing use of *FILEs* and raw file descriptors can produce unexpected results and should generally be avoided. (For the masochistic among you: POSIX.1, section 8.2.3, describes in detail how this interaction is supposed to work.) A general rule is that file descriptors are handled in the kernel, while `stdio` is just a library. This means for example, that after an `exec(3)`, the child inherits all open file descriptors, but all old streams have become inaccessible.

Since the symbols *stdin*, *stdout*, and *stderr* are specified to be macros, assigning to them is nonportable. The standard streams can be made to refer to different files with help of the library function `freopen(3)`, specially introduced to make it possible to reassign *stdin*, *stdout*, and *stderr*. The standard streams are closed by a call to `exit(3)` and by normal program termination.

STANDARDS [top](#)

C11, POSIX.1-2008.

The standards also stipulate that these three streams shall be open at program startup.



HISTORY [top](#)

C89, POSIX.1-2001.

NOTES [top](#)

The stream `stderr` is unbuffered. The stream `stdout` is line-buffered when it points to a terminal. Partial lines will not appear until `fflush(3)` or `exit(3)` is called, or a newline is printed. This can produce unexpected results, especially with debugging output. The buffering mode of the standard streams (or any other stream) can be changed using the `setbuf(3)` or `setvbuf(3)` call. Note that in case `stdin` is associated with a terminal, there may also be input buffering in the terminal driver, entirely unrelated to `stdio` buffering. (Indeed, normally terminal input is line buffered in the kernel.) This kernel input handling can be modified using calls like `tcsetattr(3)`; see also `stty(1)`, and `termios(3)`.

SEE ALSO [top](#)

`csch(1)`, `sh(1)`, `open(2)`, `fopen(3)`, `stdio(3)`

Linux man-pages (unreleased) (date) [stdin\(3\)](#)

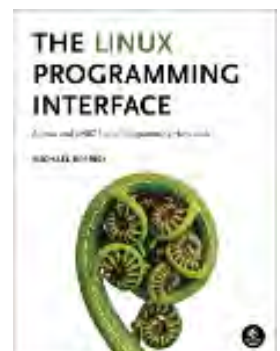
Pages that refer to this page: [intro\(1\)](#), [FILE\(3type\)](#), [stdio\(3\)](#), [pam_exec\(8\)](#)

[Copyright and license for this manual page](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



errno(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [NOTES](#) | [SEE ALSO](#)

errno(3)

Library Functions Manual

errno(3)

NAME [top](#)

`errno` - number of last error

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <errno.h>
```

DESCRIPTION [top](#)

The `<errno.h>` header file defines the integer variable *errno*, which is set by system calls and some library functions in the event of an error to indicate what went wrong.

errno

The value in *errno* is significant only when the return value of the call indicated an error (i.e., -1 from most system calls; -1 or NULL from most library functions); a function that succeeds *is* allowed to change *errno*. The value of *errno* is never set to zero by any system call or library function.

For some system calls and library functions (e.g., [getpriority\(2\)](#)), -1 is a valid return on success. In such cases,



a successful return can be distinguished from an error return by setting `errno` to zero before the call, and then, if the call returns a status that indicates that an error may have occurred, checking to see if `errno` has a nonzero value.

`errno` is defined by the ISO C standard to be a modifiable lvalue of type `int`, and must not be explicitly declared; `errno` may be a macro. `errno` is thread-local; setting it in one thread does not affect its value in any other thread.

Error numbers and names

Valid error numbers are all positive numbers. The `<errno.h>` header file defines symbolic names for each of the possible error numbers that may appear in `errno`.

All the error names specified by POSIX.1 must have distinct values, with the exception of **EAGAIN** and **EWOULDBLOCK**, which may be the same. On Linux, these two have the same value on all architectures.

The error numbers that correspond to each symbolic name vary across UNIX systems, and even across different architectures on Linux. Therefore, numeric values are not included as part of the list of error names below. The `perror(3)` and `strerror(3)` functions can be used to convert these names to corresponding textual error messages.

On any particular Linux system, one can obtain a list of all symbolic error names and the corresponding error numbers using the `errno(1)` command (part of the `moreutils` package):

```
$ errno -1
EPERM 1 Operation not permitted
ENOENT 2 No such file or directory
ESRCH 3 No such process
EINTR 4 Interrupted system call
EIO 5 Input/output error
...
```

The `errno(1)` command can also be used to look up individual error numbers and names, and to search for errors using strings from the error description, as in the following examples:

```
$ errno 2
```

```
ENOENT 2 No such file or directory
$ errno ESRCH
ESRCH 3 No such process
$ errno -s permission
EACCES 13 Permission denied
```

List of error names

In the list of the symbolic error names below, various names are marked as follows:

POSIX.1-2001

The name is defined by POSIX.1-2001, and is defined in later POSIX.1 versions, unless otherwise indicated.

POSIX.1-2008

The name is defined in POSIX.1-2008, but was not present in earlier POSIX.1 standards.

C99 The name is defined by C99.

Below is a list of the symbolic error names that are defined on Linux:

E2BIG Argument list too long (POSIX.1-2001).

EACCES Permission denied (POSIX.1-2001).

EADDRINUSE

Address already in use (POSIX.1-2001).

EADDRNOTAVAIL

Address not available (POSIX.1-2001).

EAFNOSUPPORT

Address family not supported (POSIX.1-2001).

EAGAIN Resource temporarily unavailable (may be the same value as **EWOULDBLOCK**) (POSIX.1-2001).

EALREADY

Connection already in progress (POSIX.1-2001).

EBADE Invalid exchange.



EBADF Bad file descriptor (POSIX.1-2001).

EBADFD File descriptor in bad state.

EBADMSG

Bad message (POSIX.1-2001).

EBADR Invalid request descriptor.

EBADRQC

Invalid request code.

EBADSLT

Invalid slot.

EBUSY Device or resource busy (POSIX.1-2001).

ECANCELED

Operation canceled (POSIX.1-2001).

ECHILD No child processes (POSIX.1-2001).

ECHRNG Channel number out of range.

ECOMM Communication error on send.

ECONNABORTED

Connection aborted (POSIX.1-2001).

ECONNREFUSED

Connection refused (POSIX.1-2001).

ECONNRESET

Connection reset (POSIX.1-2001).

EDEADLK

Resource deadlock avoided (POSIX.1-2001).

EDEADLOCK

On most architectures, a synonym for **EDEADLK**. On some architectures (e.g., Linux MIPS, PowerPC, SPARC), it is a separate error code "File locking deadlock error".

EDESTADDRREQ



Destination address required (POSIX.1-2001).

EDOM Mathematics argument out of domain of function (POSIX.1, C99).

EDQUOT Disk quota exceeded (POSIX.1-2001).

EEXIST File exists (POSIX.1-2001).

EFAULT Bad address (POSIX.1-2001).

EFBIG File too large (POSIX.1-2001).

EHOSTDOWN

Host is down.

EHOSTUNREACH

Host is unreachable (POSIX.1-2001).

EHWPOISON

Memory page has hardware error.

EIDRM Identifier removed (POSIX.1-2001).

EILSEQ Invalid or incomplete multibyte or wide character (POSIX.1, C99).

The text shown here is the glibc error description; in POSIX.1, this error is described as "Illegal byte sequence".

EINPROGRESS

Operation in progress (POSIX.1-2001).

EINTR Interrupted function call (POSIX.1-2001); see [signal\(7\)](#).

EINVAL Invalid argument (POSIX.1-2001).

EIO Input/output error (POSIX.1-2001).

EISCONN

Socket is connected (POSIX.1-2001).

EISDIR Is a directory (POSIX.1-2001).



EISNAM Is a named type file.

EKEYEXPIRED

Key has expired.

EKEYREJECTED

Key was rejected by service.

EKEYREVOKED

Key has been revoked.

EL2HLT Level 2 halted.

EL2NSYNC

Level 2 not synchronized.

EL3HLT Level 3 halted.

EL3RST Level 3 reset.

ELIBACC

Cannot access a needed shared library.

ELIBBAD

Accessing a corrupted shared library.

ELIBMAX

Attempting to link in too many shared libraries.

ELIBSCN

.lib section in a.out corrupted

ELIBEXEC

Cannot exec a shared library directly.

ELNRNG Link number out of range.

ELOOP Too many levels of symbolic links (POSIX.1-2001).

EMEDIUMTYPE

Wrong medium type.

EMFILE Too many open files (POSIX.1-2001). Commonly caused by



exceeding the **RLIMIT_NOFILE** resource limit described in [getrlimit\(2\)](#). Can also be caused by exceeding the limit specified in [/proc/sys/fs/nr_open](#).

EMLINK Too many links (POSIX.1-2001).

EMSGSIZE

Message too long (POSIX.1-2001).

EMULTIHOP

Multihop attempted (POSIX.1-2001).

ENAMETOOLONG

Filename too long (POSIX.1-2001).

ENETDOWN

Network is down (POSIX.1-2001).

ENETRESET

Connection aborted by network (POSIX.1-2001).

ENETUNREACH

Network unreachable (POSIX.1-2001).

ENFILE Too many open files in system (POSIX.1-2001). On Linux, this is probably a result of encountering the [/proc/sys/fs/file-max](#) limit (see [proc\(5\)](#)).

ENOANO No anode.

ENOBUFS

No buffer space available (POSIX.1 (XSI STREAMS option)).

ENODATA

The named attribute does not exist, or the process has no access to this attribute; see [xattr\(7\)](#).

In POSIX.1-2001 (XSI STREAMS option), this error was described as "No message is available on the STREAM head read queue".

ENODEV No such device (POSIX.1-2001).

ENOENT No such file or directory (POSIX.1-2001).

Typically, this error results when a specified pathname does not exist, or one of the components in the directory prefix of a pathname does not exist, or the specified pathname is a dangling symbolic link.

ENOEXEC

Exec format error (POSIX.1-2001).

ENOKEY Required key not available.

ENOLCK No locks available (POSIX.1-2001).

ENOLINK

Link has been severed (POSIX.1-2001).

ENOMEDIUM

No medium found.

ENOMEM Not enough space/cannot allocate memory (POSIX.1-2001).

ENOMSG No message of the desired type (POSIX.1-2001).

ENONET Machine is not on the network.

ENOPKG Package not installed.

ENOPROTOPT

Protocol not available (POSIX.1-2001).

ENOSPC No space left on device (POSIX.1-2001).

ENOSR No STREAM resources (POSIX.1 (XSI STREAMS option)).

ENOSTR Not a STREAM (POSIX.1 (XSI STREAMS option)).

ENOSYS Function not implemented (POSIX.1-2001).

ENOTBLK

Block device required.

ENOTCONN

The socket is not connected (POSIX.1-2001).

ENOTDIR

Not a directory (POSIX.1-2001).

ENOTEMPTY

Directory not empty (POSIX.1-2001).

ENOTRECOVERABLE

State not recoverable (POSIX.1-2008).

ENOTSOCK

Not a socket (POSIX.1-2001).

ENOTSUP

Operation not supported (POSIX.1-2001).

ENOTTY Inappropriate I/O control operation (POSIX.1-2001).

ENOTUNIQ

Name not unique on network.

ENXIO No such device or address (POSIX.1-2001).

EOPNOTSUPP

Operation not supported on socket (POSIX.1-2001).

(**ENOTSUP** and **EOPNOTSUPP** have the same value on Linux, but according to POSIX.1 these error values should be distinct.)

E_OVERFLOW

Value too large to be stored in data type (POSIX.1-2001).

EOWNERDEAD

Owner died (POSIX.1-2008).

EPERM Operation not permitted (POSIX.1-2001).

EPFNOSUPPORT

Protocol family not supported.

EPIPE Broken pipe (POSIX.1-2001).

EPROTO Protocol error (POSIX.1-2001).



EPROTONOSUPPORT

Protocol not supported (POSIX.1-2001).

EPROTOTYPE

Protocol wrong type for socket (POSIX.1-2001).

ERANGE Result too large (POSIX.1, C99).

EREMCHG

Remote address changed.

EREMOTE

Object is remote.

EREMOTEIO

Remote I/O error.

ERESTART

Interrupted system call should be restarted.

ERFKILL

Operation not possible due to RF-kill.

EROFS Read-only filesystem (POSIX.1-2001).

ESHUTDOWN

Cannot send after transport endpoint shutdown.

ESPIPE Invalid seek (POSIX.1-2001).

ESOCKTNOSUPPORT

Socket type not supported.

ESRCH No such process (POSIX.1-2001).

ESTALE Stale file handle (POSIX.1-2001).

This error can occur for NFS and for other filesystems.

ESTRPIPE

Streams pipe error.

ETIME Timer expired (POSIX.1 (XSI STREAMS option)).

(POSIX.1 says "STREAM `ioctl(2)` timeout".)

ETIMEDOUT

Connection timed out (POSIX.1-2001).

ETOOMANYREFS

Too many references: cannot splice.

ETXTBSY

Text file busy (POSIX.1-2001).

EUCLEAN

Structure needs cleaning.

EUNATCH

Protocol driver not attached.

EUSERS Too many users.

EWouldBlock

Operation would block (may be same value as **EAGAIN**) (POSIX.1-2001).

EXDEV Invalid cross-device link (POSIX.1-2001).

EXFULL Exchange full.

NOTES [top](#)

A common mistake is to do

```
if (somecall() == -1) {
    printf("somecall() failed\n");
    if (errno == ...) { ... }
}
```

where `errno` no longer needs to have the value it had upon return from `somecall()` (i.e., it may have been changed by the `printf(3)`). If the value of `errno` should be preserved across a library call, it must be saved:

```
if (somecall() == -1) {
    int errsv = errno;
```



```

    printf("somecall() failed\n");
    if (errsv == ...) { ... }
}

```

Note that the POSIX threads APIs do *not* set *errno* on error. Instead, on failure they return an error number as the function result. These error numbers have the same meanings as the error numbers returned in *errno* by other APIs.

On some ancient systems, `<errno.h>` was not present or did not declare *errno*, so that it was necessary to declare *errno* manually (i.e., *extern int errno*). **Do not do this.** It long ago ceased to be necessary, and it will cause problems with modern versions of the C library.

SEE ALSO [top](#)

[errno\(1\)](#), [err\(3\)](#), [error\(3\)](#), [perror\(3\)](#), [strerror\(3\)](#)

Linux man-pages (unreleased) (date) [errno\(3\)](#)

Pages that refer to this page: [errno.h\(0p\)](#), [netdb.h\(0p\)](#), [signal.h\(0p\)](#), [gawk\(1\)](#), [mv\(1p\)](#), [strace\(1\)](#), [accept\(2\)](#), [access\(2\)](#), [acct\(2\)](#), [add_key\(2\)](#), [adjtimex\(2\)](#), [alloc_hugepages\(2\)](#), [arch_prctl\(2\)](#), [bdflush\(2\)](#), [bind\(2\)](#), [bpf\(2\)](#), [brk\(2\)](#), [cacheflush\(2\)](#), [capget\(2\)](#), [chdir\(2\)](#), [chmod\(2\)](#), [chown\(2\)](#), [chroot\(2\)](#), [clock_getres\(2\)](#), [clone\(2\)](#), [close\(2\)](#), [close_range\(2\)](#), [connect\(2\)](#), [copy_file_range\(2\)](#), [create_module\(2\)](#), [delete_module\(2\)](#), [dup\(2\)](#), [epoll_create\(2\)](#), [epoll_ctl\(2\)](#), [epoll_wait\(2\)](#), [eventfd\(2\)](#), [execve\(2\)](#), [execveat\(2\)](#), [fallocate\(2\)](#), [fanotify_init\(2\)](#), [fanotify_mark\(2\)](#), [fcntl\(2\)](#), [flock\(2\)](#), [fork\(2\)](#), [fsync\(2\)](#), [futex\(2\)](#), [futimesat\(2\)](#), [getcpu\(2\)](#), [getdents\(2\)](#), [getdomainname\(2\)](#), [getgid\(2\)](#), [getgroups\(2\)](#), [gethostname\(2\)](#), [getitimer\(2\)](#), [get_kernel_syms\(2\)](#), [get_mempolicy\(2\)](#), [getpeername\(2\)](#), [getpriority\(2\)](#), [getrandom\(2\)](#), [getresuid\(2\)](#), [getrlimit\(2\)](#), [getrusage\(2\)](#), [getsid\(2\)](#), [getsockname\(2\)](#), [getsockopt\(2\)](#), [gettimeofday\(2\)](#), [getuid\(2\)](#), [getunwind\(2\)](#), [getxattr\(2\)](#), [init_module\(2\)](#), [inotify_add_watch\(2\)](#), [inotify_init\(2\)](#), [inotify_rm_watch\(2\)](#), [intro\(2\)](#), [io_cancel\(2\)](#), [ioctl\(2\)](#), [ioctl_console\(2\)](#), [ioctl_fat\(2\)](#), [ioctl_ficlonerange\(2\)](#), [ioctl_fideduperange\(2\)](#), [ioctl_fslabel\(2\)](#), [ioctl_getfsmmap\(2\)](#), [ioctl_tty\(2\)](#), [ioctl_userfaultfd\(2\)](#), [ioctl_xfs_ag_geometry\(2\)](#), [ioctl_xfs_bulkstat\(2\)](#), [ioctl_xfs_fsbulkstat\(2\)](#), [ioctl_xfs_fscounts\(2\)](#), [ioctl_xfs_fsgeometry\(2\)](#), [ioctl_xfs_fsgetxattr\(2\)](#), [ioctl_xfs_fsinumbers\(2\)](#), [ioctl_xfs_getbmapx\(2\)](#), [ioctl_xfs_getresblks\(2\)](#), [ioctl_xfs_goingdown\(2\)](#), [ioctl_xfs_inumbers\(2\)](#), [ioctl_xfs_scrub_metadata\(2\)](#), [io_destroy\(2\)](#), [io_getevents\(2\)](#), [ioperm\(2\)](#), [ioprio\(2\)](#), [ioprio_set\(2\)](#), [io_setup\(2\)](#), [io_submit\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [io_uring_register\(2\)](#), [io_uring_setup\(2\)](#), [kcmp\(2\)](#), [kexec_load\(2\)](#), [keyctl\(2\)](#), [kill\(2\)](#), [link\(2\)](#), [listen\(2\)](#), [listxattr\(2\)](#), [llseek\(2\)](#), [lookup_dcookie\(2\)](#), [lseek\(2\)](#), [madvise\(2\)](#), [mbind\(2\)](#),

membarrier(2), memfd_create(2), memfd_secret(2), migrate_pages(2), mincore(2), mkdir(2), mknod(2), mlock(2), mmap2(2), mmap(2), modify_ldt(2), mount(2), mount_setattr(2), move_pages(2), mprotect(2), mremap(2), msgctl(2), msgget(2), msgop(2), msync(2), nanosleep(2), nfsservctl(2), nice(2), open(2), openat2(2), open_by_handle_at(2), pause(2), pciconfig_read(2), perf_event_open(2), perfmonctl(2), personality(2), pidfd_getfd(2), pidfd_open(2), pidfd_send_signal(2), pipe(2), pivot_root(2), pkey_alloc(2), poll(2), prctl(2), pread(2), process_madvise(2), process_vm_readv(2), ptrace(2), query_module(2), quotactl(2), read(2), readahead(2), readdir(2), readlink(2), readv(2), reboot(2), recv(2), recvmmsg(2), remap_file_pages(2), removexattr(2), rename(2), request_key(2), restart_syscall(2), rmdir(2), rt_sigqueueinfo(2), s390_guarded_storage(2), s390_pci_mmio_write(2), s390_runtime_instr(2), s390_sthyi(2), sched_get_priority_max(2), sched_rr_get_interval(2), sched_setaffinity(2), sched_setattr(2), sched_setparam(2), sched_setscheduler(2), sched_yield(2), seccomp(2), seccomp_unotify(2), select(2), select_tut(2), semctl(2), semget(2), semop(2), send(2), sendfile(2), sendmmsg(2), seteuid(2), setfsuid(2), setfsuid(2), setgid(2), set_mempolicy(2), setns(2), setpgid(2), setresuid(2), setreuid(2), setsid(2), set_thread_area(2), setuid(2), setxattr(2), shmctl(2), shmget(2), shmop(2), shutdown(2), sigaction(2), sigaltstack(2), signal(2), signalfd(2), sigpending(2), sigprocmask(2), sigreturn(2), sigsuspend(2), sigwaitinfo(2), socket(2), socketpair(2), splice(2), spu_create(2), spu_run(2), stat(2), statfs(2), statx(2), stime(2), swapon(2), symlink(2), sync(2), sync_file_range(2), syscall(2), _syscall(2), syscalls(2), sysctl(2), sysfs(2), sysinfo(2), syslog(2), tee(2), time(2), timer_create(2), timer_delete(2), timerfd_create(2), timer_getoverrun(2), timer_settime(2), times(2), tkill(2), truncate(2), umount(2), uname(2), unimplemented(2), unlink(2), unshare(2), uselib(2), userfaultfd(2), ustat(2), utime(2), utimensat(2), vhangup(2), vm86(2), vmsplice(2), wait(2), write(2), accept(3p), access(3p), acl_add_perm(3), acl_calc_mask(3), acl_check(3), acl_clear_perms(3), acl_cmp(3), acl_copy_entry(3), acl_copy_ext(3), acl_copy_int(3), acl_create_entry(3), acl_delete_def_file(3), acl_delete_entry(3), acl_delete_perm(3), acl_dup(3), acl_entries(3), acl_equiv_mode(3), acl_extended_fd(3), acl_extended_file(3), acl_free(3), acl_from_mode(3), acl_from_text(3), acl_get_entry(3), acl_get_fd(3), acl_get_file(3), acl_get_perm(3), acl_get_permset(3), acl_get_qualifier(3), acl_get_tag_type(3), acl_init(3), acl_set_fd(3), acl_set_file(3), acl_set_permset(3), acl_set_qualifier(3), acl_set_tag_type(3), acl_size(3), acl_to_any_text(3), acl_to_text(3), acl_valid(3), acos(3), acos(3p), acosh(3), acosh(3p), adjtime(3), aio_cancel(3), aio_cancel(3p), aio_error(3), aio_error(3p), aio_fsync(3), aio_fsync(3p), aio_read(3), aio_read(3p), aio_return(3), aio_return(3p), aio_suspend(3), aio_suspend(3p), aio_write(3), aio_write(3p), alphasort(3p), asin(3), asin(3p), asinh(3p), atan2(3p), atan(3p), atanh(3), atanh(3p), atoi(3), attr_get(3), attr_list(3), attr_multi(3), attr_remove(3), attr_set(3), audit_open(3), avc_add_callback(3), avc_compute_create(3), avc_context_to_sid(3), avc_has_perm(3), avc_init(3), avc_netlink_loop(3), avc_open(3), bind(3p), bindresvport(3), btree(3), calloc(3p), canonicalize_file_name(3), cap_clear(3), cap_from_text(3), cap_get_file(3), cap_get_proc(3), cap_iab(3), cap_init(3), cap_launch(3), catclose(3p), catgets(3p), catopen(3), catopen(3p), ceil(3), cfree(3), cfsetispeed(3p), cfsetospeed(3p), chdir(3p), chmod(3p), chown(3p), clearerr(3p), clock_getres(3p), close(3p),



closedir(3), closedir(3p), closelog(3p), confstr(3), confstr(3p), connect(3p), context_new(3), cos(3), cos(3p), cosh(3), cosh(3p), crypt(3p), ctime(3), daemon(3), dbopen(3), dirfd(3), dirfd(3p), dup(3p), duplocale(3), duplocale(3p), encrypt(3), encrypt(3p), endgrent(3p), endpwent(3p), erf(3), erf(3p), erfc(3), erfc(3p), err(3), errno(3p), error(3), euidaccess(3), exec(3), exec(3p), _Exit(3p), exp10(3), exp2(3p), exp(3), exp(3p), expm1(3), expm1(3p), fattach(3p), fchdir(3p), fchmod(3p), fchown(3p), fclose(3), fclose(3p), fcntl(3p), fdatasync(3p), fdetach(3p), fdim(3), fdim(3p), fdopen(3p), fdopendir(3p), feof(3p), ferror(3), ferror(3p), fexecve(3), fflush(3), fflush(3p), fgetc(3p), fgetgrent(3), fgetpos(3p), fgetpwent(3), fgets(3p), fgetwc(3), fgetwc(3p), fgetws(3p), fileno(3), fileno(3p), floor(3), fma(3), fma(3p), fmemopen(3), fmemopen(3p), fmod(3), fmod(3p), fopen(3), fopen(3p), fork(3p), form(3x), form_cursor(3x), form_driver(3x), form_field(3x), form_field_attributes(3x), form_field_buffer(3x), form_field_info(3x), form_field_just(3x), form_field_opts(3x), form_fieldtype(3x), form_field_validation(3x), form_hook(3x), form_opts(3x), form_page(3x), form_post(3x), form_win(3x), fpathconf(3), fpathconf(3p), fprintf(3p), fpurge(3), fputc(3p), fputs(3p), fputwc(3), fputwc(3p), fputws(3p), fread(3p), freeaddrinfo(3p), freopen(3p), fscanf(3p), fseek(3), fseek(3p), fseeko(3), fsetpos(3p), fstat(3p), fstatat(3p), fstatvfs(3p), fsync(3p), ftell(3p), ftok(3), ftok(3p), ftruncate(3p), fts(3), ftw(3p), futimens(3p), futimes(3), fwide(3p), fwprintf(3p), fwrite(3p), fwscanf(3p), getaddrinfo(3), getcontext(3), getcwd(3), getcwd(3p), getdate(3), getdate(3p), getdelim(3p), getdirentries(3), getegid(3p), getentropy(3), geteuid(3p), getfilecon(3), getgid(3p), getgrent(3), getgrgid(3p), getgrnam(3), getgrnam(3p), getgroups(3p), gethostid(3), getifaddrs(3), getitimer(3p), getline(3), getlogin(3), getlogin(3p), getmsg(3p), getnameinfo(3), getnameinfo(3p), getopt(3p), getpass(3), getpeername(3p), getpgid(3p), get_phys_pages(3), getpriority(3p), getpt(3), getpw(3), getpwent(3), getpwnam(3), getpwnam(3p), getpwuid(3p), getrlimit(3p), getrusage(3p), gets(3p), getseuserbyname(3), getsid(3p), getsockname(3p), getsockopt(3p), getspnam(3), getuid(3p), getutent(3), getwchar(3), glob(3), glob(3p), gmtime(3p), gnutls_transport_set_errno(3), grantpt(3), grantpt(3p), handle(3), hash(3), hsearch(3), hypot(3), hypot(3p), iconv(3), iconv(3p), iconv_close(3), iconv_close(3p), iconv_open(3), iconv_open(3p), if_indextoname(3p), if_nameindex(3), if_nameindex(3p), if_nametoindex(3), ilogb(3), ilogb(3p), inet_net_pton(3), inet_ntop(3), inet_ntop(3p), inet_pton(3), initgroups(3), intro(3), ioctl(3p), io_uring_prep_accept(3), io_uring_prep_accept_direct(3), io_uring_prep_close(3), io_uring_prep_close_direct(3), io_uring_prep_connect(3), io_uring_prep_fadvise(3), io_uring_prep_fallocate(3), io_uring_prep_fsync(3), io_uring_prep_futex_wait(3), io_uring_prep_futex_waitv(3), io_uring_prep_futex_wake(3), io_uring_prep_link(3), io_uring_prep_linkat(3), io_uring_prep_madvise(3), io_uring_prep_mkdir(3), io_uring_prep_mkdirat(3), io_uring_prep_multishot_accept(3), io_uring_prep_multishot_accept_direct(3), io_uring_prep_openat2(3), io_uring_prep_openat2_direct(3), io_uring_prep_openat(3), io_uring_prep_openat_direct(3), io_uring_prep_poll_add(3), io_uring_prep_poll_multishot(3), io_uring_prep_read(3), io_uring_prep_read_fixed(3), io_uring_prep_read_multishot(3), io_uring_prep_readv2(3), io_uring_prep_readv(3), io_uring_prep_recv(3), io_uring_prep_recvmsg(3), io_uring_prep_recvmsg_multishot(3), io_uring_prep_recv_multishot(3),



[io_uring_prep_rename\(3\)](#), [io_uring_prep_renameat\(3\)](#), [io_uring_prep_send\(3\)](#),
[io_uring_prep_sendmsg\(3\)](#), [io_uring_prep_sendmsg_zc\(3\)](#), [io_uring_prep_sendto\(3\)](#),
[io_uring_prep_send_zc\(3\)](#), [io_uring_prep_send_zc_fixed\(3\)](#),
[io_uring_prep_shutdown\(3\)](#), [io_uring_prep_socket\(3\)](#), [io_uring_prep_socket_direct\(3\)](#),
[io_uring_prep_socket_direct_alloc\(3\)](#), [io_uring_prep_splice\(3\)](#),
[io_uring_prep_statx\(3\)](#), [io_uring_prep_symlink\(3\)](#), [io_uring_prep_symlinkat\(3\)](#),
[io_uring_prep_sync_file_range\(3\)](#), [io_uring_prep_tee\(3\)](#), [io_uring_prep_unlink\(3\)](#),
[io_uring_prep_unlinkat\(3\)](#), [io_uring_prep_waitid\(3\)](#), [io_uring_prep_write\(3\)](#),
[io_uring_prep_write_fixed\(3\)](#), [io_uring_prep_writev2\(3\)](#), [io_uring_prep_writev\(3\)](#),
[isastream\(3p\)](#), [isatty\(3\)](#), [isatty\(3p\)](#), [isfdtype\(3\)](#), [j0\(3\)](#), [j0\(3p\)](#), [keyctl_capabilities\(3\)](#),
[keyctl_chown\(3\)](#), [keyctl_clear\(3\)](#), [keyctl_describe\(3\)](#), [keyctl_get_keyring_ID\(3\)](#),
[keyctl_get_persistent\(3\)](#), [keyctl_get_security\(3\)](#), [keyctl_instantiate\(3\)](#),
[keyctl_invalidate\(3\)](#), [keyctl_join_session_keyring\(3\)](#), [keyctl_link\(3\)](#), [keyctl_move\(3\)](#),
[keyctl_pkey_encrypt\(3\)](#), [keyctl_pkey_query\(3\)](#), [keyctl_pkey_sign\(3\)](#), [keyctl_read\(3\)](#),
[keyctl_restrict_keyring\(3\)](#), [keyctl_revoke\(3\)](#), [keyctl_search\(3\)](#),
[keyctl_session_to_parent\(3\)](#), [keyctl_setperm\(3\)](#), [keyctl_set_reqkey_keyring\(3\)](#),
[keyctl_set_timeout\(3\)](#), [keyctl_update\(3\)](#), [keyctl_watch_key\(3\)](#), [kill\(3p\)](#), [killpg\(3\)](#),
[lchown\(3p\)](#), [ldap_dup\(3\)](#), [ldap_get_dn\(3\)](#), [ldap_open\(3\)](#), [ldexp\(3\)](#), [ldexp\(3p\)](#),
[lgamma\(3\)](#), [lgamma\(3p\)](#), [libcap\(3\)](#), [libmagic\(3\)](#), [libpsx\(3\)](#), [link\(3p\)](#), [lio_listio\(3\)](#),
[lio_listio\(3p\)](#), [listen\(3p\)](#), [llrint\(3p\)](#), [llround\(3p\)](#), [localtime\(3p\)](#), [lockf\(3\)](#), [lockf\(3p\)](#),
[log10\(3p\)](#), [log1p\(3\)](#), [log1p\(3p\)](#), [log2\(3p\)](#), [log\(3\)](#), [log\(3p\)](#), [logb\(3\)](#), [logb\(3p\)](#), [lrint\(3\)](#),
[lrint\(3p\)](#), [lround\(3\)](#), [lround\(3p\)](#), [lseek\(3p\)](#), [makecontext\(3\)](#), [malloc\(3\)](#), [malloc\(3p\)](#),
[malloc_info\(3\)](#), [mallopt\(3\)](#), [matherr\(3\)](#), [mblen\(3p\)](#), [mbrlen\(3\)](#), [mbrlen\(3p\)](#),
[mbrtowc\(3\)](#), [mbrtowc\(3p\)](#), [mbsnrtowcs\(3\)](#), [mbsrtowcs\(3\)](#), [mbsrtowcs\(3p\)](#),
[mbstowcs\(3p\)](#), [mbtowc\(3p\)](#), [menu\(3x\)](#), [menu_attributes\(3x\)](#), [menu_cursor\(3x\)](#),
[menu_driver\(3x\)](#), [menu_format\(3x\)](#), [menu_hook\(3x\)](#), [menu_items\(3x\)](#),
[menu_mark\(3x\)](#), [menu_new\(3x\)](#), [menu_opts\(3x\)](#), [menu_pattern\(3x\)](#), [menu_post\(3x\)](#),
[menu_win\(3x\)](#), [mitem_current\(3x\)](#), [mitem_new\(3x\)](#), [mitem_opts\(3x\)](#),
[mitem_value\(3x\)](#), [mkdir\(3p\)](#), [mkdtemp\(3\)](#), [mkdtemp\(3p\)](#), [mkfifo\(3\)](#), [mkfifo\(3p\)](#),
[mknod\(3p\)](#), [mkstemp\(3\)](#), [mktemp\(3\)](#), [mktime\(3p\)](#), [mlock\(3p\)](#), [mlockall\(3p\)](#),
[mmap\(3p\)](#), [mmv_stats_init\(3\)](#), [mmv_stats_registry\(3\)](#), [mpool\(3\)](#), [mprotect\(3p\)](#),
[mq_close\(3\)](#), [mq_close\(3p\)](#), [mq_getattr\(3\)](#), [mq_getattr\(3p\)](#), [mq_notify\(3\)](#),
[mq_notify\(3p\)](#), [mq_open\(3\)](#), [mq_open\(3p\)](#), [mq_receive\(3\)](#), [mq_receive\(3p\)](#),
[mq_send\(3\)](#), [mq_send\(3p\)](#), [mq_setattr\(3p\)](#), [mq_unlink\(3\)](#), [mq_unlink\(3p\)](#),
[msgctl\(3p\)](#), [msgget\(3p\)](#), [msgrcv\(3p\)](#), [msgsnd\(3p\)](#), [msync\(3p\)](#), [munmap\(3p\)](#),
[nanosleep\(3p\)](#), [newlocale\(3\)](#), [newlocale\(3p\)](#), [nextafter\(3\)](#), [nextafter\(3p\)](#), [nftw\(3p\)](#),
[nice\(3p\)](#), [numa\(3\)](#), [open\(3p\)](#), [opendir\(3\)](#), [open_memstream\(3\)](#),
[open_memstream\(3p\)](#), [openpty\(3\)](#), [pause\(3p\)](#), [pclose\(3p\)](#), [pcpintro\(3\)](#), [perror\(3\)](#),
[perror\(3p\)](#), [pipe\(3p\)](#), [pmfault\(3\)](#), [pmfstring\(3\)](#), [pmrecord\(3\)](#), [poll\(3p\)](#), [popen\(3\)](#),
[popen\(3p\)](#), [posix_fallocate\(3\)](#), [posix_memalign\(3\)](#), [posix_openpt\(3\)](#),
[posix_openpt\(3p\)](#), [posix_typed_mem_open\(3p\)](#), [pow\(3\)](#), [pow\(3p\)](#), [printf\(3\)](#),
[probe::kprocess.exec_complete\(3stap\)](#), [pselect\(3p\)](#), [psiginfo\(3p\)](#),
[pthread_sigmask\(3p\)](#), [ptsname\(3\)](#), [ptsname\(3p\)](#), [putenv\(3\)](#), [putenv\(3p\)](#), [putmsg\(3p\)](#),
[putpwent\(3\)](#), [puts\(3p\)](#), [putwchar\(3\)](#), [raise\(3p\)](#), [random\(3\)](#), [random_r\(3\)](#), [rcmd\(3\)](#),
[read\(3p\)](#), [readdir\(3\)](#), [readdir\(3p\)](#), [readlink\(3p\)](#), [realloc\(3p\)](#), [realpath\(3\)](#), [realpath\(3p\)](#),
[recno\(3\)](#), [recursive_key_scan\(3\)](#), [recv\(3p\)](#), [recvfrom\(3p\)](#), [recvmsg\(3p\)](#),
[regcomp\(3p\)](#), [remainder\(3\)](#), [remainder\(3p\)](#), [remove\(3\)](#), [remquo\(3\)](#), [remquo\(3p\)](#),



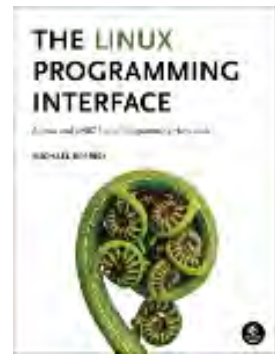
rename(3p), rewind(3p), rint(3), rint(3p), rmdir(3p), round(3), rpmatch(3), rtime(3), scalb(3), scalbln(3), scalbln(3p), scandir(3), scanf(3), sched_getcpu(3), sched_getparam(3p), sched_get_priority_max(3p), sched_getscheduler(3p), sched_rr_get_interval(3p), sched_setparam(3p), sched_setscheduler(3p), sched_yield(3p), sctp_bindx(3), sctp_connectx(3), sctp_peeloff(3), sd_bus_error(3), sd_bus_error_add_map(3), sd-bus-errors(3), sd_bus_message_new_method_error(3), sd_bus_process(3), sd_bus_wait(3), sd_journal_print(3), seccomp_export_bpf(3), seccomp_init(3), seccomp_load(3), seccomp_notify_alloc(3), seccomp_precompute(3), seccomp_rule_add(3), security_class_to_string(3), selabel_digest(3), selabel_lookup(3), selabel_lookup_best_match(3), selabel_open(3), selinux_file_context_verify(3), selinux_raw_context_to_color(3), selinux_restorecon(3), selinux_restorecon_default_handle(3), selinux_restorecon_set_alt_rootpath(3), selinux_restorecon_xattr(3), selinux_set_callback(3), selinux_set_mapping(3), sem_close(3), sem_close(3p), semctl(3p), sem_destroy(3), sem_destroy(3p), semget(3p), sem_getvalue(3), sem_getvalue(3p), sem_init(3), sem_init(3p), semop(3p), sem_open(3), sem_open(3p), sem_post(3), sem_post(3p), sem_timedwait(3p), sem_trywait(3p), sem_unlink(3), sem_unlink(3p), sem_wait(3), send(3p), sendmsg(3p), sendto(3p), setbuf(3), setbuf(3p), setegid(3p), setenv(3), setenv(3p), seteuid(3p), setfilecon(3), setgid(3p), setkey(3p), setpgid(3p), setregid(3p), setreuid(3p), setsid(3p), setsockopt(3p), setuid(3p), setvbuf(3p), shmat(3p), shmctl(3p), shmctl(3p), shmget(3p), shm_open(3), shm_open(3p), shm_unlink(3p), shutdown(3p), sigaction(3p), sigaddset(3p), sigaltstack(3p), sigdelset(3p), sigemptyset(3p), sigfillset(3p), sighold(3p), siginterrupt(3), siginterrupt(3p), sigismember(3p), signal(3p), sigpause(3), sigpending(3p), sigqueue(3), sigqueue(3p), sigset(3), sigsetops(3), sigsuspend(3p), sigtimedwait(3p), sigvec(3), sin(3), sin(3p), sincos(3), sinh(3), sinh(3p), socketatmark(3), socketatmark(3p), socket(3p), socketpair(3p), sqrt(3), sqrt(3p), statvfs(3), strcoll(3p), strdup(3), strdup(3p), strerror(3), strerror(3p), strfmon(3), strfmon(3p), strptime(3), strsignal(3p), strtod(3), strtod(3p), strtointmax(3), strtointmax(3p), strtol(3), strtol(3p), strtoul(3), strtoul(3p), strxfrm(3p), symlink(3p), sysconf(3), sysconf(3p), syslog(3), system(3), system(3p), tan(3), tan(3p), tanh(3p), tcdrain(3p), tcflow(3p), tcflush(3p), tcgetattr(3p), tcgetpgrp(3), tcgetpgrp(3p), tcgetsid(3), tcgetsid(3p), tcsendbreak(3p), tcsetattr(3p), tcsetpgrp(3p), telldir(3), tempnam(3), tempnam(3p), termios(3), tgamma(3), tgamma(3p), timegm(3), timer_create(3p), timer_delete(3p), timer_getoverrun(3p), times(3p), tmpfile(3), tmpfile(3p), towctrans(3p), tracefs_filter_string_append(3), truncate(3p), ttyname(3), ttyname(3p), udev_device_new_from_syspath(3), ulimit(3), ulimit(3p), uname(3p), ungetwc(3), unlink(3p), unlockpt(3), unlockpt(3p), unsetenv(3p), uselocale(3), uselocale(3p), usleep(3), utime(3p), wait(3p), waitid(3p), wctomb(3), wctomb(3p), wcscoll(3p), wcsdup(3), wcsdup(3p), wcsnrtombs(3), wcsrtombs(3), wcsrtombs(3p), wcstod(3p), wcstointmax(3p), wcstol(3p), wcstoul(3p), wcsxfrm(3p), wctrans(3p), wordexp(3p), write(3p), writev(3p), y0(3), y0(3p), random(4), proc(5), selabel_file(5), sudo_plugin(5), systemd.exec(5), cpuset(7), fanotify(7), io_uring(7), ip(7), man-pages(7), math_error(7), pipe(7), pthreads(7), signal-safety(7), socket(7), spufs(7), systemd.journal-fields(7), tcp(7), unix(7)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



open(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

open(2)

System Calls Manual

open(2)**NAME** [top](#)

open, openat, creat - open and possibly create a file

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <fcntl.h>
```

```
int open(const char *pathname, int flags, ...
        /* mode_t mode */ );
```

```
int creat(const char *pathname, mode_t mode);
```

```
int openat(int dirfd, const char *pathname, int flags, ...
        /* mode_t mode */ );
```

```
/* Documented separately, in openat2\(2\): */
int openat2(int dirfd, const char *pathname,
            const struct open_how *how, size_t size);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
openat():
    Since glibc 2.10:
        _POSIX_C_SOURCE >= 200809L
```

```
Before glibc 2.10:  
_ATFILE_SOURCE
```

DESCRIPTION [top](#)

The **open()** system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if **O_CREAT** is specified in *flags*) be created by **open()**.

The return value of **open()** is a file descriptor, a small, nonnegative integer that is an index to an entry in the process's table of open file descriptors. The file descriptor is used in subsequent system calls (**read(2)**, **write(2)**, **lseek(2)**, **fcntl(2)**, etc.) to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

By default, the new file descriptor is set to remain open across an **execve(2)** (i.e., the **FD_CLOEXEC** file descriptor flag described in **fcntl(2)** is initially disabled); the **O_CLOEXEC** flag, described below, can be used to change this default. The file offset is set to the beginning of the file (see **lseek(2)**).

A call to **open()** creates a new *open file description*, an entry in the system-wide table of open files. The open file description records the file offset and the file status flags (see below). A file descriptor is a reference to an open file description; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file. For further details on open file descriptions, see NOTES.

The argument *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

In addition, zero or more file creation flags and file status flags can be bitwise ORed in *flags*. The *file creation flags* are **O_CLOEXEC**, **O_CREAT**, **O_DIRECTORY**, **O_EXCL**, **O_NOCTTY**, **O_NOFOLLOW**, **O_TMPFILE**, and **O_TRUNC**. The *file status flags* are all of the remaining flags listed below. The distinction between these two groups of flags is that the file creation flags affect the semantics of the open operation itself, while the file status flags affect the semantics of subsequent I/O operations. The file status flags can be retrieved and (in some cases) modified; see **fcntl(2)** for details.

The full list of file creation flags and file status flags is as follows:

O_APPEND

The file is opened in append mode. Before each `write(2)`, the file offset is positioned at the end of the file, as if with `lseek(2)`. The modification of the file offset and the write operation are performed as a single atomic step.

O_APPEND may lead to corrupted files on NFS filesystems if more than one process appends data to a file at once. This is because NFS does not support appending to a file, so the client kernel has to simulate it, which can't be done without a race condition.

O_ASYNC

Enable signal-driven I/O: generate a signal (**SIGIO** by default, but this can be changed via `fcntl(2)`) when input or output becomes possible on this file descriptor. This feature is available only for terminals, pseudoterminals, sockets, and (since Linux 2.6) pipes and FIFOs. See `fcntl(2)` for further details. See also BUGS, below.

O_CLOEXEC (since Linux 2.6.23)

Enable the close-on-exec flag for the new file descriptor. Specifying this flag permits a program to avoid additional `fcntl(2)` **F_SETFD** operations to set the **FD_CLOEXEC** flag.

Note that the use of this flag is essential in some multithreaded programs, because using a separate `fcntl(2)` **F_SETFD** operation to set the **FD_CLOEXEC** flag does not suffice to avoid race conditions where one thread opens a file descriptor and attempts to set its close-on-exec flag using `fcntl(2)` at the same time as another thread does a `fork(2)` plus `execve(2)`. Depending on the order of execution, the race may lead to the file descriptor returned by `open()` being unintentionally leaked to the program executed by the child process created by `fork(2)`. (This kind of race is in principle possible for any system call that creates a file descriptor whose close-on-exec flag should be set, and various other Linux system calls provide an equivalent of the **O_CLOEXEC** flag to deal with this problem.)

O_CREAT

If *pathname* does not exist, create it as a regular file.

The owner (user ID) of the new file is set to the effective user ID of the process.

The group ownership (group ID) of the new file is set either to the effective group ID of the process (System V semantics) or to the group ID of the parent directory (BSD semantics). On Linux, the behavior depends on whether the set-group-ID mode bit is set on the parent directory: if that bit is set, then BSD semantics apply; otherwise, System V semantics apply. For some filesystems, the behavior also depends on the *bsdgroups* and *sysvgroups* mount options described in [mount\(8\)](#).

The *mode* argument specifies the file mode bits to be applied when a new file is created. If neither **O_CREAT** nor **O_TMPFILE** is specified in *flags*, then *mode* is ignored (and can thus be specified as 0, or simply omitted). The *mode* argument **must** be supplied if **O_CREAT** or **O_TMPFILE** is specified in *flags*; if it is not supplied, some arbitrary bytes from the stack will be applied as the file mode.

The effective mode is modified by the process's *umask* in the usual way: in the absence of a default ACL, the mode of the created file is $(mode \& \sim umask)$.

Note that *mode* applies only to future accesses of the newly created file; the **open()** call that creates a read-only file may well return a read/write file descriptor.

The following symbolic constants are provided for *mode*:

S_IRWXU 00700 user (file owner) has read, write, and execute permission

S_IRUSR 00400 user has read permission

S_IWUSR 00200 user has write permission

S_IXUSR 00100 user has execute permission

S_IRWXG 00070 group has read, write, and execute permission

S_IRGRP 00040 group has read permission

S_IWGRP 00020 group has write permission



S_IXGRP 00010 group has execute permission

S_IRWXO 00007 others have read, write, and execute permission

S_IROTH 00004 others have read permission

S_IWOTH 00002 others have write permission

S_IXOTH 00001 others have execute permission

According to POSIX, the effect when other bits are set in *mode* is unspecified. On Linux, the following bits are also honored in *mode*:

S_ISUID 0004000 set-user-ID bit

S_ISGID 0002000 set-group-ID bit (see [inode\(7\)](#)).

S_ISVTX 0001000 sticky bit (see [inode\(7\)](#)).

O_DIRECT (since Linux 2.4.10)

Try to minimize cache effects of the I/O to and from this file. In general this will degrade performance, but it is useful in special situations, such as when applications do their own caching. File I/O is done directly to/from user-space buffers. The **O_DIRECT** flag on its own makes an effort to transfer data synchronously, but does not give the guarantees of the **O_SYNC** flag that data and necessary metadata are transferred. To guarantee synchronous I/O, **O_SYNC** must be used in addition to **O_DIRECT**. See NOTES below for further discussion.

A semantically similar (but deprecated) interface for block devices is described in [raw\(8\)](#).

O_DIRECTORY

If *pathname* is not a directory, cause the open to fail. This flag was added in Linux 2.1.126, to avoid denial-of-service problems if [opendir\(3\)](#) is called on a FIFO or tape device.

O_DSYNC

Write operations on the file will complete according to the requirements of synchronized I/O *data* integrity completion.

By the time `write(2)` (and similar) return, the output data has been transferred to the underlying hardware, along with any file metadata that would be required to retrieve that data (i.e., as though each `write(2)` was followed by a call to `fdatasync(2)`). See *NOTES below*.

O_EXCL Ensure that this call creates the file: if this flag is specified in conjunction with **O_CREAT**, and `pathname` already exists, then `open()` fails with the error **EEXIST**.

When these two flags are specified, symbolic links are not followed: if `pathname` is a symbolic link, then `open()` fails regardless of where the symbolic link points.

In general, the behavior of **O_EXCL** is undefined if it is used without **O_CREAT**. There is one exception: on Linux 2.6 and later, **O_EXCL** can be used without **O_CREAT** if `pathname` refers to a block device. If the block device is in use by the system (e.g., mounted), `open()` fails with the error **EBUSY**.

On NFS, **O_EXCL** is supported only when using NFSv3 or later on kernel 2.6 or later. In NFS environments where **O_EXCL** support is not provided, programs that rely on it for performing locking tasks will contain a race condition. Portable programs that want to perform atomic file locking using a lockfile, and need to avoid reliance on NFS support for **O_EXCL**, can create a unique file on the same filesystem (e.g., incorporating hostname and PID), and use `link(2)` to make a link to the lockfile. If `link(2)` returns 0, the lock is successful. Otherwise, use `stat(2)` on the unique file to check if its link count has increased to 2, in which case the lock is also successful.

O_LARGEFILE

(LFS) Allow files whose sizes cannot be represented in an `off_t` (but can be represented in an `off64_t`) to be opened. The **_LARGEFILE64_SOURCE** macro must be defined (before including *any* header files) in order to obtain this definition. Setting the **_FILE_OFFSET_BITS** feature test macro to 64 (rather than using **O_LARGEFILE**) is the preferred method of accessing large files on 32-bit systems (see `feature_test_macros(7)`).

O_NOATIME (since Linux 2.6.8)

Do not update the file last access time (`st_atime` in the inode) when the file is `read(2)`.

This flag can be employed only if one of the following conditions is true:

- The effective UID of the process matches the owner UID of the file.
- The calling process has the **CAP_FOWNER** capability in its user namespace and the owner UID of the file has a mapping in the namespace.

This flag is intended for use by indexing or backup programs, where its use can significantly reduce the amount of disk activity. This flag may not be effective on all filesystems. One example is NFS, where the server maintains the access time.

O_NOCTTY

If *pathname* refers to a terminal device—see [tty\(4\)](#)—it will not become the process's controlling terminal even if the process does not have one.

O_NOFOLLOW

If the trailing component (i.e., basename) of *pathname* is a symbolic link, then the open fails, with the error **ELOOP**. Symbolic links in earlier components of the pathname will still be followed. (Note that the **ELOOP** error that can occur in this case is indistinguishable from the case where an open fails because there are too many symbolic links found while resolving components in the prefix part of the pathname.)

This flag is a FreeBSD extension, which was added in Linux 2.1.126, and has subsequently been standardized in POSIX.1-2008.

See also **O_PATH** below.

O_NONBLOCK or **O_NDELAY**

When possible, the file is opened in nonblocking mode. Neither the **open()** nor any subsequent I/O operations on the file descriptor which is returned will cause the calling process to wait.

Note that the setting of this flag has no effect on the operation of [poll\(2\)](#), [select\(2\)](#), [epoll\(7\)](#), and similar, since those interfaces merely inform the caller about

whether a file descriptor is "ready", meaning that an I/O operation performed on the file descriptor with the **O_NONBLOCK** flag *clear* would not block.

Note that this flag has no effect for regular files and block devices; that is, I/O operations will (briefly) block when device activity is required, regardless of whether **O_NONBLOCK** is set. Since **O_NONBLOCK** semantics might eventually be implemented, applications should not depend upon blocking behavior when specifying this flag for regular files and block devices.

For the handling of FIFOs (named pipes), see also [fifo\(7\)](#). For a discussion of the effect of **O_NONBLOCK** in conjunction with mandatory file locks and with file leases, see [fcntl\(2\)](#).

O_PATH (since Linux 2.6.39)

Obtain a file descriptor that can be used for two purposes: to indicate a location in the filesystem tree and to perform operations that act purely at the file descriptor level. The file itself is not opened, and other file operations (e.g., [read\(2\)](#), [write\(2\)](#), [fchmod\(2\)](#), [fchown\(2\)](#), [fgetxattr\(2\)](#), [ioctl\(2\)](#), [mmap\(2\)](#)) fail with the error **EBADF**.

The following operations *can* be performed on the resulting file descriptor:

- [close\(2\)](#).
- [fchdir\(2\)](#), if the file descriptor refers to a directory (since Linux 3.5).
- [fstat\(2\)](#) (since Linux 3.6).
- [fstatfs\(2\)](#) (since Linux 3.12).
- Duplicating the file descriptor ([dup\(2\)](#), [fcntl\(2\)](#) **F_DUPFD**, etc.).
- Getting and setting file descriptor flags ([fcntl\(2\)](#) **F_GETFD** and **F_SETFD**).
- Retrieving open file status flags using the [fcntl\(2\)](#) **F_GETFL** operation: the returned flags will include the bit **O_PATH**.



- Passing the file descriptor as the *dirfd* argument of **openat()** and the other "*at()" system calls. This includes **linkat(2)** with **AT_EMPTY_PATH** (or via procfs using **AT_SYMLINK_FOLLOW**) even if the file is not a directory.
- Passing the file descriptor to another process via a UNIX domain socket (see **SCM_RIGHTS** in **unix(7)**).

When **O_PATH** is specified in *flags*, flag bits other than **O_CLOEXEC**, **O_DIRECTORY**, and **O_NOFOLLOW** are ignored.

Opening a file or directory with the **O_PATH** flag requires no permissions on the object itself (but does require execute permission on the directories in the path prefix). Depending on the subsequent operation, a check for suitable file permissions may be performed (e.g., **fchdir(2)** requires execute permission on the directory referred to by its file descriptor argument). By contrast, obtaining a reference to a filesystem object by opening it with the **O_RDONLY** flag requires that the caller have read permission on the object, even when the subsequent operation (e.g., **fchdir(2)**, **fstat(2)**) does not require read permission on the object.

If *pathname* is a symbolic link and the **O_NOFOLLOW** flag is also specified, then the call returns a file descriptor referring to the symbolic link. This file descriptor can be used as the *dirfd* argument in calls to **fchownat(2)**, **fstatat(2)**, **linkat(2)**, and **readlinkat(2)** with an empty *pathname* to have the calls operate on the symbolic link.

If *pathname* refers to an automount point that has not yet been triggered, so no other filesystem is mounted on it, then the call returns a file descriptor referring to the automount directory without triggering a mount. **fstatfs(2)** can then be used to determine if it is, in fact, an untriggered automount point (**.f_type == AUTOFS_SUPER_MAGIC**).

One use of **O_PATH** for regular files is to provide the equivalent of POSIX.1's **O_EXEC** functionality. This permits us to open a file for which we have execute permission but not read permission, and then execute that file, with steps something like the following:

```
char buf[PATH_MAX];
fd = open("some_prog", O_PATH);
snprintf(buf, PATH_MAX, "/proc/self/fd/%d", fd);
execl(buf, "some_prog", (char *) NULL);
```

An **O_PATH** file descriptor can also be passed as the argument of [fexecve\(3\)](#).

O_SYNC Write operations on the file will complete according to the requirements of synchronized I/O *file* integrity completion (by contrast with the synchronized I/O *data* integrity completion provided by **O_DSYNC**.)

By the time [write\(2\)](#) (or similar) returns, the output data and associated file metadata have been transferred to the underlying hardware (i.e., as though each [write\(2\)](#) was followed by a call to [fsync\(2\)](#)). *See NOTES below.*

O_TMPFILE (since Linux 3.11)

Create an unnamed temporary regular file. The *pathname* argument specifies a directory; an unnamed inode will be created in that directory's filesystem. Anything written to the resulting file will be lost when the last file descriptor is closed, unless the file is given a name.

O_TMPFILE must be specified with one of **O_RDWR** or **O_WRONLY** and, optionally, **O_EXCL**. If **O_EXCL** is not specified, then [linkat\(2\)](#) can be used to link the temporary file into the filesystem, making it permanent, using code like the following:

```
char path[PATH_MAX];
fd = open("/path/to/dir", O_TMPFILE | O_RDWR,
          S_IRUSR | S_IWUSR);

/* File I/O on 'fd'... */

linkat(fd, "", AT_FDCWD, "/path/for/file", AT_EMPTY_PATH);

/* If the caller doesn't have the CAP_DAC_READ_SEARCH
   capability (needed to use AT_EMPTY_PATH with linkat(2)),
   and there is a proc(5) filesystem mounted, then the
   linkat(2) call above can be replaced with:

snprintf(path, PATH_MAX, "/proc/self/fd/%d", fd);
linkat(AT_FDCWD, path, AT_FDCWD, "/path/for/file",
       AT_SYMLINK_FOLLOW);
```


*/

In this case, the `open()` *mode* argument determines the file permission mode, as with `O_CREAT`.

Specifying `O_EXCL` in conjunction with `O_TMPFILE` prevents a temporary file from being linked into the filesystem in the above manner. (Note that the meaning of `O_EXCL` in this case is different from the meaning of `O_EXCL` otherwise.)

There are two main use cases for `O_TMPFILE`:

- Improved `tmpfile(3)` functionality: race-free creation of temporary files that (1) are automatically deleted when closed; (2) can never be reached via any pathname; (3) are not subject to symlink attacks; and (4) do not require the caller to devise unique names.
- Creating a file that is initially invisible, which is then populated with data and adjusted to have appropriate filesystem attributes (`fchown(2)`, `fchmod(2)`, `fsetxattr(2)`, etc.) before being atomically linked into the filesystem in a fully formed state (using `linkat(2)` as described above).

`O_TMPFILE` requires support by the underlying filesystem; only a subset of Linux filesystems provide that support. In the initial implementation, support was provided in the `ext2`, `ext3`, `ext4`, `UDF`, `Minix`, and `tmpfs` filesystems. Support for other filesystems has subsequently been added as follows: `XFS` (Linux 3.15); `Btrfs` (Linux 3.16); `F2FS` (Linux 3.16); and `ubifs` (Linux 4.9)

`O_TRUNC`

If the file already exists and is a regular file and the access mode allows writing (i.e., is `O_RDWR` or `O_WRONLY`) it will be truncated to length 0. If the file is a FIFO or terminal device file, the `O_TRUNC` flag is ignored. Otherwise, the effect of `O_TRUNC` is unspecified.

`creat()`

A call to `creat()` is equivalent to calling `open()` with *flags* equal to `O_CREAT|O_WRONLY|O_TRUNC`.

`openat()`

The `openat()` system call operates in exactly the same way as

open(), except for the differences described here.

The *dirfd* argument is used in conjunction with the *pathname* argument as follows:

- If the pathname given in *pathname* is absolute, then *dirfd* is ignored.
- If the pathname given in *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like **open()**).
- If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **open()** for a relative pathname). In this case, *dirfd* must be a directory that was opened for reading (**O_RDONLY**) or using the **O_PATH** flag.

If the pathname given in *pathname* is relative, and *dirfd* is not a valid file descriptor, an error (**EBADF**) results. (Specifying an invalid file descriptor number in *dirfd* can be used as a means to ensure that *pathname* is absolute.)

openat2(2)

The **openat2(2)** system call is an extension of **openat()**, and provides a superset of the features of **openat()**. It is documented separately, in **openat2(2)**.

RETURN VALUE [top](#)

On success, **open()**, **openat()**, and **creat()** return the new file descriptor (a nonnegative integer). On error, -1 is returned and *errno* is set to indicate the error.

ERRORS [top](#)

open(), **openat()**, and **creat()** can fail with the following errors:

EACCES The requested access to the file is not allowed, or search permission is denied for one of the directories in the path prefix of *pathname*, or the file did not exist yet and write access to the parent directory is not allowed. (See

also [path_resolution\(7\)](#).)

- EACCES** Where **O_CREAT** is specified, the *protected_fifos* or *protected_regular* sysctl is enabled, the file already exists and is a FIFO or regular file, the owner of the file is neither the current user nor the owner of the containing directory, and the containing directory is both world- or group-writable and sticky. For details, see the descriptions of [/proc/sys/fs/protected_fifos](#) and [/proc/sys/fs/protected_regular](#) in [proc\(5\)](#).
- EBADF** ([openat\(\)](#)) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.
- EBUSY** **O_EXCL** was specified in *flags* and *pathname* refers to a block device that is in use by the system (e.g., it is mounted).
- EDQUOT** Where **O_CREAT** is specified, the file does not exist, and the user's quota of disk blocks or inodes on the filesystem has been exhausted.
- EEXIST** *pathname* already exists and **O_CREAT** and **O_EXCL** were used.
- EFAULT** *pathname* points outside your accessible address space.
- EFBIG** See **EOVERFLOW**.
- EINTR** While blocked waiting to complete an open of a slow device (e.g., a FIFO; see [fifo\(7\)](#)), the call was interrupted by a signal handler; see [signal\(7\)](#).
- EINVAL** The filesystem does not support the **O_DIRECT** flag. See **NOTES** for more information.
- EINVAL** Invalid value in *flags*.
- EINVAL** **O_TMPFILE** was specified in *flags*, but neither **O_WRONLY** nor **O_RDWR** was specified.
- EINVAL** **O_CREAT** was specified in *flags* and the final component ("basename") of the new file's *pathname* is invalid (e.g., it contains characters not permitted by the underlying filesystem).
- EINVAL** The final component ("basename") of *pathname* is invalid (e.g., it contains characters not permitted by the



underlying filesystem).

EISDIR *pathname* refers to a directory and the access requested involved writing (that is, **O_WRONLY** or **O_RDWR** is set).

EISDIR *pathname* refers to an existing directory, **O_TMPFILE** and one of **O_WRONLY** or **O_RDWR** were specified in *flags*, but this kernel version does not provide the **O_TMPFILE** functionality.

ELOOP Too many symbolic links were encountered in resolving *pathname*.

ELOOP *pathname* was a symbolic link, and *flags* specified **O_NOFOLLOW** but not **O_PATH**.

EMFILE The per-process limit on the number of open file descriptors has been reached (see the description of **RLIMIT_NOFILE** in [getrlimit\(2\)](#)).

ENAMETOOLONG

pathname was too long.

ENFILE The system-wide limit on the total number of open files has been reached.

ENODEV *pathname* refers to a device special file and no corresponding device exists. (This is a Linux kernel bug; in this situation **ENXIO** must be returned.)

ENOENT **O_CREAT** is not set and the named file does not exist.

ENOENT A directory component in *pathname* does not exist or is a dangling symbolic link.

ENOENT *pathname* refers to a nonexistent directory, **O_TMPFILE** and one of **O_WRONLY** or **O_RDWR** were specified in *flags*, but this kernel version does not provide the **O_TMPFILE** functionality.

ENOMEM The named file is a FIFO, but memory for the FIFO buffer can't be allocated because the per-user hard limit on memory allocation for pipes has been reached and the caller is not privileged; see [pipe\(7\)](#).

ENOMEM Insufficient kernel memory was available.



ENOSPC *pathname* was to be created but the device containing *pathname* has no room for the new file.

ENOTDIR

A component used as a directory in *pathname* is not, in fact, a directory, or **O_DIRECTORY** was specified and *pathname* was not a directory.

ENOTDIR

(**openat()**) *pathname* is a relative pathname and *dirfd* is a file descriptor referring to a file other than a directory.

ENXIO **O_NONBLOCK** | **O_WRONLY** is set, the named file is a FIFO, and no process has the FIFO open for reading.

ENXIO The file is a device special file and no corresponding device exists.

ENXIO The file is a UNIX domain socket.

EOPNOTSUPP

The filesystem containing *pathname* does not support **O_TMPFILE**.

E_OVERFLOW

pathname refers to a regular file that is too large to be opened. The usual scenario here is that an application compiled on a 32-bit platform without **-D_FILE_OFFSET_BITS=64** tried to open a file whose size exceeds $(1 \ll 31) - 1$ bytes; see also **O_LARGEFILE** above. This is the error specified by POSIX.1; before Linux 2.6.24, Linux gave the error **EFBIG** for this case.

EPERM The **O_NOATIME** flag was specified, but the effective user ID of the caller did not match the owner of the file and the caller was not privileged.

EPERM The operation was prevented by a file seal; see [fcntl\(2\)](#).

EROFS *pathname* refers to a file on a read-only filesystem and write access was requested.

ETXTBSY

pathname refers to an executable image which is currently being executed and write access was requested.



ETXTBSY

pathname refers to a file that is currently in use as a swap file, and the **O_TRUNC** flag was specified.

ETXTBSY

pathname refers to a file that is currently being read by the kernel (e.g., for module/firmware loading), and write access was requested.

EWOULDBLOCK

The **O_NONBLOCK** flag was specified, and an incompatible lease was held on the file (see [fcntl\(2\)](#)).

VERSIONS

[top](#)

The (undefined) effect of **O_RDONLY | O_TRUNC** varies among implementations. On many systems the file is actually truncated.

Synchronized I/O

The POSIX.1-2008 "synchronized I/O" option specifies different variants of synchronized I/O, and specifies the **open()** flags **O_SYNC**, **O_DSYNC**, and **O_RSYNC** for controlling the behavior. Regardless of whether an implementation supports this option, it must at least support the use of **O_SYNC** for regular files.

Linux implements **O_SYNC** and **O_DSYNC**, but not **O_RSYNC**. Somewhat incorrectly, glibc defines **O_RSYNC** to have the same value as **O_SYNC**. (**O_RSYNC** is defined in the Linux header file [<asm/fcntl.h>](#) on HP PA-RISC, but it is not used.)

O_SYNC provides synchronized I/O *file* integrity completion, meaning write operations will flush data and all associated metadata to the underlying hardware. **O_DSYNC** provides synchronized I/O *data* integrity completion, meaning write operations will flush data to the underlying hardware, but will only flush metadata updates that are required to allow a subsequent read operation to complete successfully. Data integrity completion can reduce the number of disk operations that are required for applications that don't need the guarantees of file integrity completion.

To understand the difference between the two types of completion, consider two pieces of file metadata: the file last modification timestamp (*st_mtime*) and the file length. All write operations will update the last file modification timestamp, but only writes that add data to the end of the file will change the file length.



The last modification timestamp is not needed to ensure that a read completes successfully, but the file length is. Thus, **O_DSYNC** would only guarantee to flush updates to the file length metadata (whereas **O_SYNC** would also always flush the last modification timestamp metadata).

Before Linux 2.6.33, Linux implemented only the **O_SYNC** flag for **open()**. However, when that flag was specified, most filesystems actually provided the equivalent of synchronized I/O *data* integrity completion (i.e., **O_SYNC** was actually implemented as the equivalent of **O_DSYNC**).

Since Linux 2.6.33, proper **O_SYNC** support is provided. However, to ensure backward binary compatibility, **O_DSYNC** was defined with the same value as the historical **O_SYNC**, and **O_SYNC** was defined as a new (two-bit) flag value that includes the **O_DSYNC** flag value. This ensures that applications compiled against new headers get at least **O_DSYNC** semantics before Linux 2.6.33.

C library/kernel differences

Since glibc 2.26, the glibc wrapper function for **open()** employs the **openat()** system call, rather than the kernel's **open()** system call. For certain architectures, this is also true before glibc 2.26.

STANDARDS [top](#)

open()
creat()
openat()
POSIX.1-2008.

[openat2\(2\)](#) Linux.

The **O_DIRECT**, **O_NOATIME**, **O_PATH**, and **O_TMPFILE** flags are Linux-specific. One must define **_GNU_SOURCE** to obtain their definitions.

The **O_CLOEXEC**, **O_DIRECTORY**, and **O_NOFOLLOW** flags are not specified in POSIX.1-2001, but are specified in POSIX.1-2008. Since glibc 2.12, one can obtain their definitions by defining either **_POSIX_C_SOURCE** with a value greater than or equal to 200809L or **_XOPEN_SOURCE** with a value greater than or equal to 700. In glibc 2.11 and earlier, one obtains the definitions by defining **_GNU_SOURCE**.

HISTORY [top](#)

open()
creat()
SVr4, 4.3BSD, POSIX.1-2001.

openat()
POSIX.1-2008. Linux 2.6.16, glibc 2.4.

NOTES [top](#)

Under Linux, the **O_NONBLOCK** flag is sometimes used in cases where one wants to open but does not necessarily have the intention to read or write. For example, this may be used to open a device in order to get a file descriptor for use with [ioctl\(2\)](#).

Note that **open()** can open device special files, but **creat()** cannot create them; use [mknod\(2\)](#) instead.

If the file is newly created, its *st_atime*, *st_ctime*, *st_mtime* fields (respectively, time of last access, time of last status change, and time of last modification; see [stat\(2\)](#)) are set to the current time, and so are the *st_ctime* and *st_mtime* fields of the parent directory. Otherwise, if the file is modified because of the **O_TRUNC** flag, its *st_ctime* and *st_mtime* fields are set to the current time.

The files in the */proc/pid/fd* directory show the open file descriptors of the process with the PID *pid*. The files in the */proc/pid/fdinfo* directory show even more information about these file descriptors. See [proc\(5\)](#) for further details of both of these directories.

The Linux header file `<asm/fcntl.h>` doesn't define **O_ASYNC**; the (BSD-derived) **FASYNC** synonym is defined instead.

Open file descriptions

The term open file description is the one used by POSIX to refer to the entries in the system-wide table of open files. In other contexts, this object is variously also called an "open file object", a "file handle", an "open file table entry", or—in kernel-developer parlance—a *struct file*.

When a file descriptor is duplicated (using [dup\(2\)](#) or similar), the duplicate refers to the same open file description as the original file descriptor, and the two file descriptors

consequently share the file offset and file status flags. Such sharing can also occur between processes: a child process created via `fork(2)` inherits duplicates of its parent's file descriptors, and those duplicates refer to the same open file descriptions.

Each `open()` of a file creates a new open file description; thus, there may be multiple open file descriptions corresponding to a file inode.

On Linux, one can use the `kcmp(2)` `KCMP_FILE` operation to test whether two file descriptors (in the same process or in two different processes) refer to the same open file description.

NFS

There are many infelicities in the protocol underlying NFS, affecting amongst others `O_SYNC` and `O_NDELAY`.

On NFS filesystems with UID mapping enabled, `open()` may return a file descriptor but, for example, `read(2)` requests are denied with `EACCES`. This is because the client performs `open()` by checking the permissions, but UID mapping is performed by the server upon read and write requests.

FIFOs

Opening the read or write end of a FIFO blocks until the other end is also opened (by another process or thread). See `fifo(7)` for further details.

File access mode

Unlike the other values that can be specified in `flags`, the `access mode` values `O_RDONLY`, `O_WRONLY`, and `O_RDWR` do not specify individual bits. Rather, they define the low order two bits of `flags`, and are defined respectively as 0, 1, and 2. In other words, the combination `O_RDONLY | O_WRONLY` is a logical error, and certainly does not have the same meaning as `O_RDWR`.

Linux reserves the special, nonstandard access mode 3 (binary 11) in `flags` to mean: check for read and write permission on the file and return a file descriptor that can't be used for reading or writing. This nonstandard access mode is used by some Linux drivers to return a file descriptor that is to be used only for device-specific `ioctl(2)` operations.

Rationale for `openat()` and other directory file descriptor APIs

`openat()` and the other system calls and library functions that take a directory file descriptor argument (i.e., `execveat(2)`, `faccessat(2)`, `fanotify_mark(2)`, `fchmodat(2)`, `fchownat(2)`,



`fspick(2)`, `fstatat(2)`, `futimesat(2)`, `linkat(2)`, `mkdirat(2)`, `mknodat(2)`, `mount_setattr(2)`, `move_mount(2)`, `name_to_handle_at(2)`, `open_tree(2)`, `openat2(2)`, `readlinkat(2)`, `renameat(2)`, `renameat2(2)`, `statx(2)`, `symlinkat(2)`, `unlinkat(2)`, `utimensat(2)`, `mkfifoat(3)`, and `scandirat(3)`) address two problems with the older interfaces that preceded them. Here, the explanation is in terms of the `openat()` call, but the rationale is analogous for the other interfaces.

First, `openat()` allows an application to avoid race conditions that could occur when using `open()` to open files in directories other than the current working directory. These race conditions result from the fact that some component of the directory prefix given to `open()` could be changed in parallel with the call to `open()`. Suppose, for example, that we wish to create the file `dir1/dir2/xxx.dep` if the file `dir1/dir2/xxx` exists. The problem is that between the existence check and the file-creation step, `dir1` or `dir2` (which might be symbolic links) could be modified to point to a different location. Such races can be avoided by opening a file descriptor for the target directory, and then specifying that file descriptor as the `dirfd` argument of (say) `fstatat(2)` and `openat()`. The use of the `dirfd` file descriptor also has other benefits:

- the file descriptor is a stable reference to the directory, even if the directory is renamed; and
- the open file descriptor prevents the underlying filesystem from being dismounted, just as when a process has a current working directory on a filesystem.

Second, `openat()` allows the implementation of a per-thread "current working directory", via file descriptor(s) maintained by the application. (This functionality can also be obtained by tricks based on the use of `/proc/self/fd/dirfd`, but less efficiently.)

The `dirfd` argument for these APIs can be obtained by using `open()` or `openat()` to open a directory (with either the `O_RDONLY` or the `O_PATH` flag). Alternatively, such a file descriptor can be obtained by applying `dirfd(3)` to a directory stream created using `opendir(3)`.

When these APIs are given a `dirfd` argument of `AT_FDCWD` or the specified pathname is absolute, then they handle their pathname argument in the same way as the corresponding conventional APIs. However, in this case, several of the APIs have a `flags` argument



that provides access to functionality that is not available with the corresponding conventional APIs.

O_DIRECT

The **O_DIRECT** flag may impose alignment restrictions on the length and address of user-space buffers and the file offset of I/Os. In Linux alignment restrictions vary by filesystem and kernel version and might be absent entirely. The handling of misaligned **O_DIRECT** I/Os also varies; they can either fail with **EINVAL** or fall back to buffered I/O.

Since Linux 6.1, **O_DIRECT** support and alignment restrictions for a file can be queried using [statx\(2\)](#), using the **STATX_DIOALIGN** flag. Support for **STATX_DIOALIGN** varies by filesystem; see [statx\(2\)](#).

Some filesystems provide their own interfaces for querying **O_DIRECT** alignment restrictions, for example the **XFS_IOC_DIOINFO** operation in [xfctl\(3\)](#). **STATX_DIOALIGN** should be used instead when it is available.

If none of the above is available, then direct I/O support and alignment restrictions can only be assumed from known characteristics of the filesystem, the individual file, the underlying storage device(s), and the kernel version. In Linux 2.4, most filesystems based on block devices require that the file offset and the length and memory address of all I/O segments be multiples of the filesystem block size (typically 4096 bytes). In Linux 2.6.0, this was relaxed to the logical block size of the block device (typically 512 bytes). A block device's logical block size can be determined using the [ioctl\(2\)](#) **BLKSSZGET** operation or from the shell using the command:

```
blockdev --getss
```

O_DIRECT I/Os should never be run concurrently with the [fork\(2\)](#) system call, if the memory buffer is a private mapping (i.e., any mapping created with the [mmap\(2\)](#) **MAP_PRIVATE** flag; this includes memory allocated on the heap and statically allocated buffers). Any such I/Os, whether submitted via an asynchronous I/O interface or from another thread in the process, should be completed before [fork\(2\)](#) is called. Failure to do so can result in data corruption and undefined behavior in parent and child processes. This restriction does not apply when the memory buffer for the **O_DIRECT** I/Os was created using [shmat\(2\)](#) or [mmap\(2\)](#) with the **MAP_SHARED** flag. Nor does this restriction apply when the memory buffer has been advised as **MADV_DONTFORK**



with `madvise(2)`, ensuring that it will not be available to the child after `fork(2)`.

The **O_DIRECT** flag was introduced in SGI IRIX, where it has alignment restrictions similar to those of Linux 2.4. IRIX has also a `fcntl(2)` call to query appropriate alignments, and sizes. FreeBSD 4.x introduced a flag of the same name, but without alignment restrictions.

O_DIRECT support was added in Linux 2.4.10. Older Linux kernels simply ignore this flag. Some filesystems may not implement the flag, in which case `open()` fails with the error **EINVAL** if it is used.

Applications should avoid mixing **O_DIRECT** and normal I/O to the same file, and especially to overlapping byte regions in the same file. Even when the filesystem correctly handles the coherency issues in this situation, overall I/O throughput is likely to be slower than using either mode alone. Likewise, applications should avoid mixing `mmap(2)` of files with direct I/O to the same files.

The behavior of **O_DIRECT** with NFS will differ from local filesystems. Older kernels, or kernels configured in certain ways, may not support this combination. The NFS protocol does not support passing the flag to the server, so **O_DIRECT** I/O will bypass the page cache only on the client; the server may still cache the I/O. The client asks the server to make the I/O synchronous to preserve the synchronous semantics of **O_DIRECT**. Some servers will perform poorly under these circumstances, especially if the I/O size is small. Some servers may also be configured to lie to clients about the I/O having reached stable storage; this will avoid the performance penalty at some risk to data integrity in the event of server power failure. The Linux NFS client places no alignment restrictions on **O_DIRECT** I/O.

In summary, **O_DIRECT** is a potentially powerful tool that should be used with caution. It is recommended that applications treat use of **O_DIRECT** as a performance option which is disabled by default.

BUGS [top](#)

Currently, it is not possible to enable signal-driven I/O by specifying **O_ASYNC** when calling `open()`; use `fcntl(2)` to enable this flag.



One must check for two different error codes, **EISDIR** and **ENOENT**, when trying to determine whether the kernel supports **O_TMPFILE** functionality.

When both **O_CREAT** and **O_DIRECTORY** are specified in *flags* and the file specified by *pathname* does not exist, **open()** will create a regular file (i.e., **O_DIRECTORY** is ignored).

SEE ALSO [top](#)

[chmod\(2\)](#), [chown\(2\)](#), [close\(2\)](#), [dup\(2\)](#), [fcntl\(2\)](#), [link\(2\)](#), [lseek\(2\)](#), [mknod\(2\)](#), [mmap\(2\)](#), [mount\(2\)](#), [open_by_handle_at\(2\)](#), [openat2\(2\)](#), [read\(2\)](#), [socket\(2\)](#), [stat\(2\)](#), [umask\(2\)](#), [unlink\(2\)](#), [write\(2\)](#), [fopen\(3\)](#), [acl\(5\)](#), [fifo\(7\)](#), [inode\(7\)](#), [path_resolution\(7\)](#), [symlink\(7\)](#)

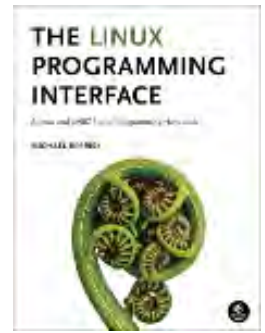
Linux man-pages (unreleased) (date) [open\(2\)](#)

Pages that refer to this page: [pv\(1\)](#), [strace\(1\)](#), [systemd-nspawn\(1\)](#), [accept\(2\)](#), [access\(2\)](#), [chmod\(2\)](#), [chown\(2\)](#), [chroot\(2\)](#), [clone\(2\)](#), [close\(2\)](#), [copy_file_range\(2\)](#), [dup\(2\)](#), [epoll_create\(2\)](#), [epoll_ctl\(2\)](#), [eventfd\(2\)](#), [execveat\(2\)](#), [fanotify_init\(2\)](#), [fanotify_mark\(2\)](#), [fcntl\(2\)](#), [flock\(2\)](#), [fork\(2\)](#), [fsync\(2\)](#), [futimesat\(2\)](#), [getrlimit\(2\)](#), [getxattr\(2\)](#), [inotify_init\(2\)](#), [ioctl\(2\)](#), [ioctl_fat\(2\)](#), [ioctl_tty\(2\)](#), [ioprio_set\(2\)](#), [io_submit\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [kcmp\(2\)](#), [link\(2\)](#), [listxattr\(2\)](#), [llseek\(2\)](#), [lseek\(2\)](#), [memfd_create\(2\)](#), [memfd_secret\(2\)](#), [mkdir\(2\)](#), [mknod\(2\)](#), [mount\(2\)](#), [mount_setattr\(2\)](#), [msgget\(2\)](#), [openat2\(2\)](#), [open_by_handle_at\(2\)](#), [perf_event_open\(2\)](#), [personality\(2\)](#), [pidfd_getfd\(2\)](#), [pipe\(2\)](#), [prctl\(2\)](#), [read\(2\)](#), [readlink\(2\)](#), [readv\(2\)](#), [recv\(2\)](#), [removexattr\(2\)](#), [rename\(2\)](#), [seccomp\(2\)](#), [seccomp_unotify\(2\)](#), [semget\(2\)](#), [send\(2\)](#), [sendfile\(2\)](#), [setxattr\(2\)](#), [shmget\(2\)](#), [signalfd\(2\)](#), [socket\(2\)](#), [spu_create\(2\)](#), [stat\(2\)](#), [statfs\(2\)](#), [statx\(2\)](#), [symlink\(2\)](#), [syscalls\(2\)](#), [timerfd_create\(2\)](#), [truncate\(2\)](#), [umask\(2\)](#), [unlink\(2\)](#), [uselib\(2\)](#), [userfaultfd\(2\)](#), [utimensat\(2\)](#), [write\(2\)](#), [catopen\(3\)](#), [dbopen\(3\)](#), [dirfd\(3\)](#), [euidaccess\(3\)](#), [ferror\(3\)](#), [fileno\(3\)](#), [fopen\(3\)](#), [fpathconf\(3\)](#), [fts\(3\)](#), [getcwd\(3\)](#), [getdirentries\(3\)](#), [getdtablesize\(3\)](#), [getfilecon\(3\)](#), [getpt\(3\)](#), [getutent\(3\)](#), [grantpt\(3\)](#), [handle\(3\)](#), [io_uring_prep_openat\(3\)](#), [io_uring_prep_openat_direct\(3\)](#), [mkfifo\(3\)](#), [mkstemp\(3\)](#), [mode_t\(3type\)](#), [mq_open\(3\)](#), [opendir\(3\)](#), [popen\(3\)](#), [posix_openpt\(3\)](#), [posix_spawn\(3\)](#), [pthread_setname_np\(3\)](#), [remove\(3\)](#), [scandir\(3\)](#), [selinux_status_open\(3\)](#), [sem_open\(3\)](#), [setfilecon\(3\)](#), [shm_open\(3\)](#), [statvfs\(3\)](#), [stdin\(3\)](#), [stdio\(3\)](#), [tempnam\(3\)](#), [tmpnam\(3\)](#), [tracefs_trace_pipe_stream\(3\)](#), [lp\(4\)](#), [random\(4\)](#), [st\(4\)](#), [proc\(5\)](#), [attributes\(7\)](#), [capabilities\(7\)](#), [cpuset\(7\)](#), [credentials\(7\)](#), [daemon\(7\)](#), [epoll\(7\)](#), [feature_test_macros\(7\)](#), [fifo\(7\)](#), [inode\(7\)](#), [inotify\(7\)](#), [landlock\(7\)](#), [mq_overview\(7\)](#), [path_resolution\(7\)](#), [pipe\(7\)](#), [pty\(7\)](#), [shm_overview\(7\)](#), [signal\(7\)](#), [signal-safety\(7\)](#), [symlink\(7\)](#), [unix\(7\)](#), [mount\(8\)](#), [mount.fuse3\(8\)](#), [xfs_io\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



read(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

 read(2)

System Calls Manual

read(2)

NAME [top](#)

read - read from a file descriptor

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
ssize_t read(int fd, void buf[.count], size_t count);
```

DESCRIPTION [top](#)

`read()` attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and `read()` returns zero.

If *count* is zero, `read()` may detect the errors described below. In the absence of any errors, or if `read()` does not check for

errors, a **read()** with a *count* of 0 returns zero and has no other effects.

According to POSIX.1, if *count* is greater than **SSIZE_MAX**, the result is implementation-defined; see NOTES for the upper limit on Linux.

RETURN VALUE [top](#)

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because **read()** was interrupted by a signal. See also NOTES.

On error, -1 is returned, and *errno* is set to indicate the error. In this case, it is left unspecified whether the file position (if any) changes.

ERRORS [top](#)

EAGAIN The file descriptor *fd* refers to a file other than a socket and has been marked nonblocking (**O_NONBLOCK**), and the read would block. See [open\(2\)](#) for further details on the **O_NONBLOCK** flag.

EAGAIN or **EWOULDBLOCK**

The file descriptor *fd* refers to a socket and has been marked nonblocking (**O_NONBLOCK**), and the read would block. POSIX.1-2001 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

EBADF *fd* is not a valid file descriptor or is not open for reading.

EFAULT *buf* is outside your accessible address space.

- EINTR** The call was interrupted by a signal before any data was read; see [signal\(7\)](#).
- EINVAL** *fd* is attached to an object which is unsuitable for reading; or the file was opened with the **O_DIRECT** flag, and either the address specified in *buf*, the value specified in *count*, or the file offset is not suitably aligned.
- EINVAL** *fd* was created via a call to [timerfd_create\(2\)](#) and the wrong size buffer was given to **read()**; see [timerfd_create\(2\)](#) for further information.
- EIO** I/O error. This will happen for example when the process is in a background process group, tries to read from its controlling terminal, and either it is ignoring or blocking **SIGTTIN** or its process group is orphaned. It may also occur when there is a low-level I/O error while reading from a disk or tape. A further possible cause of **EIO** on networked filesystems is when an advisory lock had been taken out on the file descriptor and this lock has been lost. See the *Lost Locks* section of [fcntl\(2\)](#) for further details.
- EISDIR** *fd* refers to a directory.

Other errors may occur, depending on the object connected to *fd*.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

SVr4, 4.3BSD, POSIX.1-2001.

NOTES [top](#)

On Linux, **read()** (and similar system calls) will transfer at most `0x7ffff000` (2,147,479,552) bytes, returning the number of bytes actually transferred. (This is true on both 32-bit and 64-bit

systems.)

On NFS filesystems, reading small amounts of data will update the timestamp only the first time, subsequent calls may not do so. This is caused by client side attribute caching, because most if not all NFS clients leave `st_atime` (last file access time) updates to the server, and client side reads satisfied from the client's cache will not cause `st_atime` updates on the server as there are no server-side reads. UNIX semantics can be obtained by disabling client-side attribute caching, but in most situations this will substantially increase server load and decrease performance.

BUGS [top](#)

According to POSIX.1-2008/SUSv4 Section XSI 2.9.7 ("Thread Interactions with Regular File Operations"):

All of the following functions shall be atomic with respect to each other in the effects specified in POSIX.1-2008 when they operate on regular files or symbolic links: ...

Among the APIs subsequently listed are `read()` and `readv(2)`. And among the effects that should be atomic across threads (and processes) are updates of the file offset. However, before Linux 3.14, this was not the case: if two processes that share an open file description (see `open(2)`) perform a `read()` (or `readv(2)`) at the same time, then the I/O operations were not atomic with respect updating the file offset, with the result that the reads in the two processes might (incorrectly) overlap in the blocks of data that they obtained. This problem was fixed in Linux 3.14.

SEE ALSO [top](#)

`close(2)`, `fcntl(2)`, `ioctl(2)`, `lseek(2)`, `open(2)`, `pread(2)`, `readdir(2)`, `readlink(2)`, `readv(2)`, `select(2)`, `write(2)`, `fread(3)`

Linux man-pages (unreleased) (date) [read\(2\)](#)

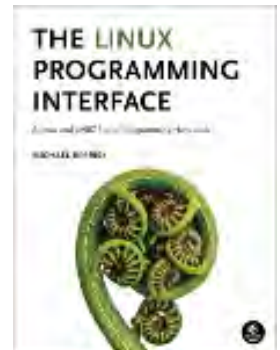
Pages that refer to this page: [grep\(1\)](#), [hardlink\(1\)](#), [ps\(1\)](#), [pv\(1\)](#), [strace\(1\)](#), [telnet-probe\(1\)](#), [close\(2\)](#), [epoll_ctl\(2\)](#), [eventfd\(2\)](#), [fanotify_init\(2\)](#), [fcntl\(2\)](#), [getrandom\(2\)](#), [inotify_add_watch\(2\)](#), [ioctl_tty\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [open\(2\)](#),

[perf_event_open\(2\)](#), [perfmonctl\(2\)](#), [pidfd_open\(2\)](#), [pipe\(2\)](#), [pread\(2\)](#), [ptrace\(2\)](#), [readahead\(2\)](#), [readv\(2\)](#), [recv\(2\)](#), [seccomp\(2\)](#), [seccomp_unotify\(2\)](#), [select\(2\)](#), [select_tut\(2\)](#), [sendfile\(2\)](#), [setpgid\(2\)](#), [signalfd\(2\)](#), [socket\(2\)](#), [socketpair\(2\)](#), [syscalls\(2\)](#), [timerfd_create\(2\)](#), [userfaultfd\(2\)](#), [write\(2\)](#), [aio_error\(3\)](#), [aio_read\(3\)](#), [aio_return\(3\)](#), [curs_getch\(3x\)](#), [dbopen\(3\)](#), [fgetc\(3\)](#), [fopen\(3\)](#), [fread\(3\)](#), [getline\(3\)](#), [gets\(3\)](#), [io_uring_prep_read\(3\)](#), [io_uring_prep_readv2\(3\)](#), [io_uring_prep_readv\(3\)](#), [libexpect\(3\)](#), [mkfifo\(3\)](#), [mpool\(3\)](#), [readdir\(3\)](#), [rtime\(3\)](#), [size_t\(3type\)](#), [stdin\(3\)](#), [stdio\(3\)](#), [termios\(3\)](#), [xdr\(3\)](#), [xfstcl\(3\)](#), [dsp56k\(4\)](#), [fuse\(4\)](#), [lirc\(4\)](#), [null\(4\)](#), [random\(4\)](#), [rtc\(4\)](#), [st\(4\)](#), [proc\(5\)](#), [systemd.exec\(5\)](#), [aio\(7\)](#), [cpuset\(7\)](#), [epoll\(7\)](#), [fanotify\(7\)](#), [inode\(7\)](#), [inotify\(7\)](#), [io_uring\(7\)](#), [landlock\(7\)](#), [pipe\(7\)](#), [signal\(7\)](#), [signal-safety\(7\)](#), [socket\(7\)](#), [spufs\(7\)](#), [tcp\(7\)](#), [vsock\(7\)](#), [x25\(7\)](#), [mount.fuse3\(8\)](#), [netsniff-ng\(8\)](#), [xfs_io\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



write(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

 write(2)

System Calls Manual

write(2)**NAME** [top](#)

write - write to a file descriptor

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
ssize_t write(int fd, const void buf[.count], size_t count);
```

DESCRIPTION [top](#)

`write()` writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

The number of bytes written may be less than *count* if, for example, there is insufficient space on the underlying physical medium, or the **RLIMIT_FSIZE** resource limit is encountered (see [setrlimit\(2\)](#)), or the call was interrupted by a signal handler after having written less than *count* bytes. (See also [pipe\(7\)](#).)

For a seekable file (i.e., one to which [lseek\(2\)](#) may be applied,

for example, a regular file) writing takes place at the file offset, and the file offset is incremented by the number of bytes actually written. If the file was `open(2)`ed with `O_APPEND`, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.

POSIX requires that a `read(2)` that can be proved to occur after a `write()` has returned will return the new data. Note that not all filesystems are POSIX conforming.

According to POSIX.1, if `count` is greater than `SSIZE_MAX`, the result is implementation-defined; see NOTES for the upper limit on Linux.

RETURN VALUE [top](#)

On success, the number of bytes written is returned. On error, `-1` is returned, and `errno` is set to indicate the error.

Note that a successful `write()` may transfer fewer than `count` bytes. Such partial writes can occur for various reasons; for example, because there was insufficient space on the disk device to write all of the requested bytes, or because a blocked `write()` to a socket, pipe, or similar was interrupted by a signal handler after it had transferred some, but before it had transferred all of the requested bytes. In the event of a partial write, the caller can make another `write()` call to transfer the remaining bytes. The subsequent call will either transfer further bytes or may result in an error (e.g., if the disk is now full).

If `count` is zero and `fd` refers to a regular file, then `write()` may return a failure status if one of the errors below is detected. If no errors are detected, or error detection is not performed, `0` is returned without causing any other effect. If `count` is zero and `fd` refers to a file other than a regular file, the results are not specified.

ERRORS [top](#)

EAGAIN The file descriptor `fd` refers to a file other than a socket and has been marked nonblocking (`O_NONBLOCK`), and

the write would block. See [open\(2\)](#) for further details on the **O_NONBLOCK** flag.

EAGAIN or **EWOULDBLOCK**

The file descriptor *fd* refers to a socket and has been marked nonblocking (**O_NONBLOCK**), and the write would block. POSIX.1-2001 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

EBADF *fd* is not a valid file descriptor or is not open for writing.

EDESTADDRREQ

fd refers to a datagram socket for which a peer address has not been set using [connect\(2\)](#).

EDQUOT The user's quota of disk blocks on the filesystem containing the file referred to by *fd* has been exhausted.

EFAULT *buf* is outside your accessible address space.

EFBIG An attempt was made to write a file that exceeds the implementation-defined maximum file size or the process's file size limit, or to write at a position past the maximum allowed offset.

EINTR The call was interrupted by a signal before any data was written; see [signal\(7\)](#).

EINVAL *fd* is attached to an object which is unsuitable for writing; or the file was opened with the **O_DIRECT** flag, and either the address specified in *buf*, the value specified in *count*, or the file offset is not suitably aligned.

EIO A low-level I/O error occurred while modifying the inode. This error may relate to the write-back of data written by an earlier **write()**, which may have been issued to a different file descriptor on the same file. Since Linux 4.13, errors from write-back come with a promise that they *may* be reported by subsequent **write()** requests, and *will* be reported by a subsequent [fsync\(2\)](#) (whether or not they



were also reported by `write()`). An alternate cause of **EIO** on networked filesystems is when an advisory lock had been taken out on the file descriptor and this lock has been lost. See the *Lost Locks* section of `fcntl(2)` for further details.

ENOSPC The device containing the file referred to by `fd` has no room for the data.

EPERM The operation was prevented by a file seal; see `fcntl(2)`.

EPIPE `fd` is connected to a pipe or socket whose reading end is closed. When this happens the writing process will also receive a **SIGPIPE** signal. (Thus, the write return value is seen only if the program catches, blocks or ignores this signal.)

Other errors may occur, depending on the object connected to `fd`.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

SVr4, 4.3BSD, POSIX.1-2001.

Under SVr4 a write may be interrupted and return **EINTR** at any point, not just before any data is written.

NOTES [top](#)

A successful return from `write()` does not make any guarantee that data has been committed to disk. On some filesystems, including NFS, it does not even guarantee that space has successfully been reserved for the data. In this case, some errors might be delayed until a future `write()`, `fsync(2)`, or even `close(2)`. The only way to be sure is to call `fsync(2)` after you are done writing all your data.

If a `write()` is interrupted by a signal handler before any bytes

are written, then the call fails with the error **EINTR**; if it is interrupted after at least one byte has been written, the call succeeds, and returns the number of bytes written.

On Linux, **write()** (and similar system calls) will transfer at most `0x7ffff000` (2,147,479,552) bytes, returning the number of bytes actually transferred. (This is true on both 32-bit and 64-bit systems.)

An error return value while performing **write()** using direct I/O does not mean the entire write has failed. Partial data may be written and the data at the file offset on which the **write()** was attempted should be considered inconsistent.

BUGS [top](#)

According to POSIX.1-2008/SUSv4 Section XSI 2.9.7 ("Thread Interactions with Regular File Operations"):

All of the following functions shall be atomic with respect to each other in the effects specified in POSIX.1-2008 when they operate on regular files or symbolic links: ...

Among the APIs subsequently listed are **write()** and [writev\(2\)](#). And among the effects that should be atomic across threads (and processes) are updates of the file offset. However, before Linux 3.14, this was not the case: if two processes that share an open file description (see [open\(2\)](#)) perform a **write()** (or [writev\(2\)](#)) at the same time, then the I/O operations were not atomic with respect to updating the file offset, with the result that the blocks of data output by the two processes might (incorrectly) overlap. This problem was fixed in Linux 3.14.

SEE ALSO [top](#)

[close\(2\)](#), [fcntl\(2\)](#), [fsync\(2\)](#), [ioctl\(2\)](#), [lseek\(2\)](#), [open\(2\)](#), [pwrite\(2\)](#), [read\(2\)](#), [select\(2\)](#), [writev\(2\)](#), [fwrite\(3\)](#)

Linux man-pages (unreleased) (date) [write\(2\)](#)

Pages that refer to this page: [ps\(1\)](#), [pv\(1\)](#), [strace\(1\)](#), [telnet-probe\(1\)](#), [close\(2\)](#), [epoll_ctl\(2\)](#), [eventfd\(2\)](#), [fcntl\(2\)](#), [fsync\(2\)](#), [getpeername\(2\)](#), [getrlimit\(2\)](#),

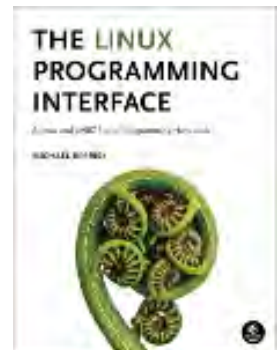


[io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [lseek\(2\)](#), [memfd_create\(2\)](#), [mmap\(2\)](#), [open\(2\)](#), [pipe\(2\)](#), [pread\(2\)](#), [read\(2\)](#), [readv\(2\)](#), [seccomp\(2\)](#), [select\(2\)](#), [select_tut\(2\)](#), [send\(2\)](#), [sendfile\(2\)](#), [socket\(2\)](#), [socketpair\(2\)](#), [sync\(2\)](#), [syscalls\(2\)](#), [aio_error\(3\)](#), [aio_return\(3\)](#), [aio_write\(3\)](#), [curs_print\(3x\)](#), [dbopen\(3\)](#), [fclose\(3\)](#), [fflush\(3\)](#), [fgetc\(3\)](#), [fopen\(3\)](#), [fread\(3\)](#), [gets\(3\)](#), [io_uring_prep_write\(3\)](#), [io_uring_prep_writev2\(3\)](#), [io_uring_prep_writev\(3\)](#), [libexpect\(3\)](#), [mkfifo\(3\)](#), [mpool\(3\)](#), [puts\(3\)](#), [size_t\(3type\)](#), [stdio\(3\)](#), [xdr\(3\)](#), [xfctl\(3\)](#), [dsp56k\(4\)](#), [fuse\(4\)](#), [lirc\(4\)](#), [st\(4\)](#), [proc\(5\)](#), [systemd.exec\(5\)](#), [aio\(7\)](#), [cgroups\(7\)](#), [cpuset\(7\)](#), [epoll\(7\)](#), [fanotify\(7\)](#), [inode\(7\)](#), [inotify\(7\)](#), [landlock\(7\)](#), [pipe\(7\)](#), [sched\(7\)](#), [signal\(7\)](#), [signal-safety\(7\)](#), [socket\(7\)](#), [spufs\(7\)](#), [tcp\(7\)](#), [time_namespaces\(7\)](#), [udp\(7\)](#), [user_namespaces\(7\)](#), [vsock\(7\)](#), [x25\(7\)](#), [fsfreeze\(8\)](#), [netsniff-ng\(8\)](#), [wipefs\(8\)](#), [xfs_io\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fsync(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 fsync(2)

System Calls Manual

fsync(2)

NAME [top](#)

`fsync`, `fdatasync` - synchronize a file's in-core state with storage device

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int fsync(int fd);
```

```
int fdatasync(int fd);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

fsync():

glibc 2.16 and later:

 No feature test macros need be defined

glibc up to and including 2.15:

```
    _BSD_SOURCE || _XOPEN_SOURCE
```

```
    || /* Since glibc 2.8: */ _POSIX_C_SOURCE >= 200112L
```

fdatasync():

```
_POSIX_C_SOURCE >= 199309L || _XOPEN_SOURCE >= 500
```

DESCRIPTION [top](#)

fsync() transfers ("flushes") all modified in-core data of (i.e., modified buffer cache pages for) the file referred to by the file descriptor *fd* to the disk device (or other permanent storage device) so that all changed information can be retrieved even if the system crashes or is rebooted. This includes writing through or flushing a disk cache if present. The call blocks until the device reports that the transfer has completed.

As well as flushing the file data, **fsync()** also flushes the metadata information associated with the file (see [inode\(7\)](#)).

Calling **fsync()** does not necessarily ensure that the entry in the directory containing the file has also reached disk. For that an explicit **fsync()** on a file descriptor for the directory is also needed.

fdatasync() is similar to **fsync()**, but does not flush modified metadata unless that metadata is needed in order to allow a subsequent data retrieval to be correctly handled. For example, changes to *st_atime* or *st_mtime* (respectively, time of last access and time of last modification; see [inode\(7\)](#)) do not require flushing because they are not necessary for a subsequent data read to be handled correctly. On the other hand, a change to the file size (*st_size*, as made by say [ftruncate\(2\)](#)), would require a metadata flush.

The aim of **fdatasync()** is to reduce disk activity for applications that do not require all metadata to be synchronized with the disk.

RETURN VALUE [top](#)

On success, these system calls return zero. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EBADF *fd* is not a valid open file descriptor.

EINTR The function was interrupted by a signal; see [signal\(7\)](#).

EIO An error occurred during synchronization. This error may relate to data written to some other file descriptor on the same file. Since Linux 4.13, errors from write-back will be reported to all file descriptors that might have written the data which triggered the error. Some filesystems (e.g., NFS) keep close track of which data came through which file descriptor, and give more precise reporting. Other filesystems (e.g., most local filesystems) will report errors to all file descriptors that were open on the file when the error was recorded.

ENOSPC Disk space was exhausted while synchronizing.

EROFS, EINVAL

fd is bound to a special file (e.g., a pipe, FIFO, or socket) which does not support synchronization.

ENOSPC, EDQUOT

fd is bound to a file on NFS or another filesystem which does not allocate space at the time of a [write\(2\)](#) system call, and some previous write failed due to insufficient storage space.

VERSIONS [top](#)

On POSIX systems on which [fdatasync\(\)](#) is available, **_POSIX_SYNCHRONIZED_IO** is defined in [<unistd.h>](#) to a value greater than 0. (See also [sysconf\(3\)](#).)

On some UNIX systems (but not Linux), *fd* must be a *writable* file descriptor.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, 4.3BSD.

In Linux 2.2 and earlier, **fdatasync()** is equivalent to **fsync()**, and so has no performance advantage.

The **fsync()** implementations in older kernels and lesser used filesystems do not know how to flush disk caches. In these cases disk caches need to be disabled using **hdparm(8)** or **sdparm(8)** to guarantee safe operation.

SEE ALSO [top](#)

[sync\(1\)](#), [bdflush\(2\)](#), [open\(2\)](#), [posix_fadvise\(2\)](#), [pwritev\(2\)](#), [sync\(2\)](#), [sync_file_range\(2\)](#), [fflush\(3\)](#), [fileno\(3\)](#), [hdparm\(8\)](#), [mount\(8\)](#)

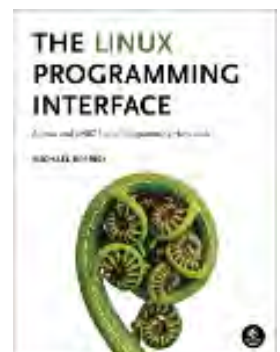
Linux man-pages (unreleased) (date) [fsync\(2\)](#)

Pages that refer to this page: [pv\(1\)](#), [sync\(1\)](#), [bdflush\(2\)](#), [close\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [mount\(2\)](#), [open\(2\)](#), [posix_fadvise\(2\)](#), [statx\(2\)](#), [sync\(2\)](#), [sync_file_range\(2\)](#), [syscalls\(2\)](#), [write\(2\)](#), [aio_error\(3\)](#), [aio_fsync\(3\)](#), [aio_return\(3\)](#), [dbopen\(3\)](#), [fclose\(3\)](#), [fflush\(3\)](#), [io_uring_prep_fsync\(3\)](#), [cups-files.conf\(5\)](#), [systemd.exec\(5\)](#), [aio\(7\)](#), [signal-safety\(7\)](#), [mount\(8\)](#), [sfdisk\(8\)](#), [xfs_io\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sync(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [BUGS](#) | [SEE ALSO](#)

 sync(2)

System Calls Manual

sync(2)**NAME** [top](#)

sync, syncfs - commit filesystem caches to disk

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
void sync(void);
```

```
int syncfs(int fd);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
sync():
    _XOPEN_SOURCE >= 500
    || /* Since glibc 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _BSD_SOURCE
```

```
syncfs():
    _GNU_SOURCE
```

DESCRIPTION [top](#)

sync() causes all pending modifications to filesystem metadata and cached file data to be written to the underlying filesystems.

syncfs() is like **sync()**, but synchronizes just the filesystem containing file referred to by the open file descriptor *fd*.

RETURN VALUE [top](#)

syncfs() returns 0 on success; on error, it returns -1 and sets *errno* to indicate the error.

ERRORS [top](#)

sync() is always successful.

syncfs() can fail for at least the following reasons:

EBADF *fd* is not a valid file descriptor.

EIO An error occurred during synchronization. This error may relate to data written to any file on the filesystem, or on metadata related to the filesystem itself.

ENOSPC Disk space was exhausted while synchronizing.

ENOSPC, EDQUOT

Data was written to a file on NFS or another filesystem which does not allocate space at the time of a **write(2)** system call, and some previous write failed due to insufficient storage space.

VERSIONS [top](#)

According to the standard specification (e.g., POSIX.1-2001), **sync()** schedules the writes, but may return before the actual writing is done. However Linux waits for I/O completions, and thus **sync()** or **syncfs()** provide the same guarantees as **fsync()** called on every file in the system or filesystem respectively.

STANDARDS [top](#)

sync() POSIX.1-2008.

syncfs()
Linux.

HISTORY [top](#)

sync() POSIX.1-2001, SVr4, 4.3BSD.

syncfs()
Linux 2.6.39, glibc 2.14.

Since glibc 2.2.2, the Linux prototype for **sync()** is as listed above, following the various standards. In glibc 2.2.1 and earlier, it was "int sync(void)", and **sync()** always returned 0.

In mainline kernel versions prior to Linux 5.8, **syncfs()** will fail only when passed a bad file descriptor (**EBADF**). Since Linux 5.8, **syncfs()** will also report an error if one or more inodes failed to be written back since the last **syncfs()** call.

BUGS [top](#)

Before Linux 1.3.20, Linux did not wait for I/O to complete before returning.

SEE ALSO [top](#)

[sync\(1\)](#), [fdatasync\(2\)](#), [fsync\(2\)](#)

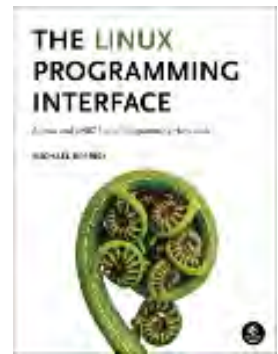
Linux man-pages (unreleased) [\(date\)](#) [sync\(2\)](#)

Pages that refer to this page: [sync\(1\)](#), [systemd-nspawn\(1\)](#), [bdflush\(2\)](#), [fsync\(2\)](#), [mount\(2\)](#), [reboot\(2\)](#), [sync_file_range\(2\)](#), [syscalls\(2\)](#), [fclose\(3\)](#), [fflush\(3\)](#), [nfs\(5\)](#), [ctrlaltdel\(8\)](#), [fdisk\(8\)](#), [fsck.minix\(8\)](#), [mke2fs\(8\)](#), [mount\(8\)](#), [xfs_io\(8\)](#), [xfs_quota\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



close(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 close(2)

System Calls Manual

close(2)**NAME** [top](#)

close - close a file descriptor

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int close(int fd);
```

DESCRIPTION [top](#)

close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see [fcntl\(2\)](#)) held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If *fd* is the last file descriptor referring to the underlying open file description (see [open\(2\)](#)), the resources associated with the open file description are freed; if the file descriptor

was the last reference to a file which has been removed using `unlink(2)`, the file is deleted.

RETURN VALUE [top](#)

`close()` returns zero on success. On error, -1 is returned, and `errno` is set to indicate the error.

ERRORS [top](#)

EBADF `fd` isn't a valid open file descriptor.

EINTR The `close()` call was interrupted by a signal; see [signal\(7\)](#).

EIO An I/O error occurred.

ENOSPC, EDQUOT

On NFS, these errors are not normally reported against the first write which exceeds the available storage space, but instead against a subsequent `write(2)`, `fsync(2)`, or `close()`.

See NOTES for a discussion of why `close()` should not be retried after an error.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.3BSD.

NOTES [top](#)

A successful close does not guarantee that the data has been successfully saved to disk, as the kernel uses the buffer cache to defer writes. Typically, filesystems do not flush buffers when a file is closed. If you need to be sure that the data is

physically stored on the underlying disk, use `fsync(2)`. (It will depend on the disk hardware at this point.)

The `close-on-exec` file descriptor flag can be used to ensure that a file descriptor is automatically closed upon a successful `execve(2)`; see `fcntl(2)` for details.

Multithreaded processes and `close()`

It is probably unwise to close file descriptors while they may be in use by system calls in other threads in the same process. Since a file descriptor may be reused, there are some obscure race conditions that may cause unintended side effects.

Furthermore, consider the following scenario where two threads are performing operations on the same file descriptor:

- (1) One thread is blocked in an I/O system call on the file descriptor. For example, it is trying to `write(2)` to a pipe that is already full, or trying to `read(2)` from a stream socket which currently has no available data.
- (2) Another thread closes the file descriptor.

The behavior in this situation varies across systems. On some systems, when the file descriptor is closed, the blocking system call returns immediately with an error.

On Linux (and possibly some other systems), the behavior is different: the blocking I/O system call holds a reference to the underlying open file description, and this reference keeps the description open until the I/O system call completes. (See `open(2)` for a discussion of open file descriptions.) Thus, the blocking system call in the first thread may successfully complete after the `close()` in the second thread.

Dealing with error returns from `close()`

A careful programmer will check the return value of `close()`, since it is quite possible that errors on a previous `write(2)` operation are reported only on the final `close()` that releases the open file description. Failing to check the return value when closing a file may lead to *silent* loss of data. This can especially be observed with NFS and with disk quota.

Note, however, that a failure return should be used only for

diagnostic purposes (i.e., a warning to the application that there may still be I/O pending or there may have been failed I/O) or remedial purposes (e.g., writing the file once more or creating a backup).

Retrying the `close()` after a failure return is the wrong thing to do, since this may cause a reused file descriptor from another thread to be closed. This can occur because the Linux kernel *always* releases the file descriptor early in the close operation, freeing it for reuse; the steps that may return an error, such as flushing data to the filesystem or device, occur only later in the close operation.

Many other implementations similarly always close the file descriptor (except in the case of `EBADF`, meaning that the file descriptor was invalid) even if they subsequently report an error on return from `close()`. POSIX.1 is currently silent on this point, but there are plans to mandate this behavior in the next major release of the standard.

A careful programmer who wants to know about I/O errors may precede `close()` with a call to `fsync(2)`.

The `EINTR` error is a somewhat special case. Regarding the `EINTR` error, POSIX.1-2008 says:

If `close()` is interrupted by a signal that is to be caught, it shall return -1 with `errno` set to `EINTR` and the state of `fildes` is unspecified.

This permits the behavior that occurs on Linux and many other implementations, where, as with other errors that may be reported by `close()`, the file descriptor is guaranteed to be closed. However, it also permits another possibility: that the implementation returns an `EINTR` error and keeps the file descriptor open. (According to its documentation, HP-UX's `close()` does this.) The caller must then once more use `close()` to close the file descriptor, to avoid file descriptor leaks. This divergence in implementation behaviors provides a difficult hurdle for portable applications, since on many implementations, `close()` must not be called again after an `EINTR` error, and on at least one, `close()` must be called again. There are plans to address this conundrum for the next major release of the POSIX.1 standard.



SEE ALSO [top](#)

[close_range\(2\)](#), [fcntl\(2\)](#), [fsync\(2\)](#), [open\(2\)](#), [shutdown\(2\)](#),
[unlink\(2\)](#), [fclose\(3\)](#)

Linux man-pages (unreleased) **(date)**

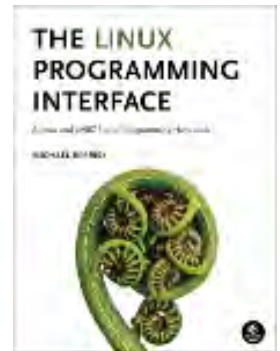
close(2)

Pages that refer to this page: [bpf\(2\)](#), [close_range\(2\)](#), [dup\(2\)](#), [epoll_create\(2\)](#),
[eventfd\(2\)](#), [flock\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [open\(2\)](#), [perfmonctl\(2\)](#),
[read\(2\)](#), [shutdown\(2\)](#), [signalfd\(2\)](#), [socket\(2\)](#), [spu_create\(2\)](#), [spu_run\(2\)](#), [syscalls\(2\)](#),
[timerfd_create\(2\)](#), [write\(2\)](#), [closedir\(3\)](#), [dbopen\(3\)](#), [fclose\(3\)](#), [fcloseall\(3\)](#), [fts\(3\)](#),
[getdtablesize\(3\)](#), [io_uring_prep_close\(3\)](#), [io_uring_prep_close_direct\(3\)](#),
[io_uring_prep_fixed_fd_install\(3\)](#), [mkfifo\(3\)](#), [__pmconnectlogger\(3\)](#), [posix_spawn\(3\)](#),
[shm_open\(3\)](#), [stdio\(3\)](#), [nfs\(5\)](#), [systemd.socket\(5\)](#), [cpuset\(7\)](#), [epoll\(7\)](#), [fanotify\(7\)](#),
[inotify\(7\)](#), [pipe\(7\)](#), [shm_overview\(7\)](#), [signal-safety\(7\)](#), [socket\(7\)](#), [spufs\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of
[The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



lseek(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 lseek(2)

System Calls Manual

lseek(2)**NAME** [top](#)

lseek - reposition read/write file offset

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
off_t lseek(int fd, off_t offset, int whence);
```

DESCRIPTION [top](#)

`lseek()` repositions the file offset of the open file description associated with the file descriptor *fd* to the argument *offset* according to the directive *whence* as follows:

SEEK_SET

The file offset is set to *offset* bytes.

SEEK_CUR

The file offset is set to its current location plus *offset* bytes.

SEEK_END

The file offset is set to the size of the file plus *offset* bytes.

lseek() allows the file offset to be set beyond the end of the file (but this does not change the size of the file). If data is later written at this point, subsequent reads of the data in the gap (a "hole") return null bytes ('\0') until data is actually written into the gap.

Seeking file data and holes

Since Linux 3.1, Linux supports the following additional values for *whence*:

SEEK_DATA

Adjust the file offset to the next location in the file greater than or equal to *offset* containing data. If *offset* points to data, then the file offset is set to *offset*.

SEEK_HOLE

Adjust the file offset to the next hole in the file greater than or equal to *offset*. If *offset* points into the middle of a hole, then the file offset is set to *offset*. If there is no hole past *offset*, then the file offset is adjusted to the end of the file (i.e., there is an implicit hole at the end of any file).

In both of the above cases, **lseek()** fails if *offset* points past the end of the file.

These operations allow applications to map holes in a sparsely allocated file. This can be useful for applications such as file backup tools, which can save space when creating backups and preserve holes, if they have a mechanism for discovering holes.

For the purposes of these operations, a hole is a sequence of zeros that (normally) has not been allocated in the underlying file storage. However, a filesystem is not obliged to report holes, so these operations are not a guaranteed mechanism for mapping the storage space actually allocated to a file. (Furthermore, a sequence of zeros that actually has been written to the underlying storage may not be reported as a hole.) In the



simplest implementation, a filesystem can support the operations by making **SEEK_HOLE** always return the offset of the end of the file, and making **SEEK_DATA** always return *offset* (i.e., even if the location referred to by *offset* is a hole, it can be considered to consist of data that is a sequence of zeros).

The **_GNU_SOURCE** feature test macro must be defined in order to obtain the definitions of **SEEK_DATA** and **SEEK_HOLE** from *<unistd.h>*.

The **SEEK_HOLE** and **SEEK_DATA** operations are supported for the following filesystems:

- Btrfs (since Linux 3.1)
- OCFS (since Linux 3.2)
- XFS (since Linux 3.5)
- ext4 (since Linux 3.8)
- [tmpfs\(5\)](#) (since Linux 3.8)
- NFS (since Linux 3.18)
- FUSE (since Linux 4.5)
- GFS2 (since Linux 4.15)

RETURN VALUE [top](#)

Upon successful completion, **lseek()** returns the resulting offset location as measured in bytes from the beginning of the file. On error, the value (*off_t*) *-1* is returned and *errno* is set to indicate the error.

ERRORS [top](#)

EBADF *fd* is not an open file descriptor.

EINVAL *whence* is not valid. Or: the resulting file offset would be negative, or beyond the end of a seekable device.

ENXIO *whence* is **SEEK_DATA** or **SEEK_HOLE**, and *offset* is beyond the end of the file, or *whence* is **SEEK_DATA** and *offset* is within a hole at the end of the file.

E_OVERFLOW

The resulting file offset cannot be represented in an *off_t*.

ESPIPE *fd* is associated with a pipe, socket, or FIFO.

VERSIONS [top](#)

On Linux, using **lseek()** on a terminal device fails with the error **ESPIPE**.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.3BSD.

SEEK_DATA and **SEEK_HOLE** are nonstandard extensions also present in Solaris, FreeBSD, and DragonFly BSD; they are proposed for inclusion in the next POSIX revision (Issue 8).

NOTES [top](#)

See [open\(2\)](#) for a discussion of the relationship between file descriptors, open file descriptions, and files.

If the **O_APPEND** file status flag is set on the open file description, then a [write\(2\)](#) *always* moves the file offset to the end of the file, regardless of the use of **lseek()**.

Some devices are incapable of seeking and POSIX does not specify which devices must support **lseek()**.

SEE ALSO [top](#)

[dup\(2\)](#), [fallocate\(2\)](#), [fork\(2\)](#), [open\(2\)](#), [fseek\(3\)](#), [lseek64\(3\)](#),
[posix_fallocate\(3\)](#)

Linux man-pages (unreleased) **(date)**

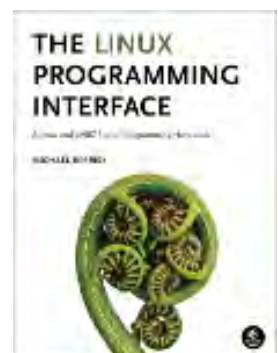
Lseek(2)

Pages that refer to this page: [copy_file_range\(2\)](#), [dup\(2\)](#), [llseek\(2\)](#), [open\(2\)](#),
[pread\(2\)](#), [read\(2\)](#), [readahead\(2\)](#), [readv\(2\)](#), [syscalls\(2\)](#), [write\(2\)](#), [fseek\(3\)](#),
[getdirentries\(3\)](#), [lseek64\(3\)](#), [off_t\(3type\)](#), [posix_fallocate\(3\)](#), [seekdir\(3\)](#), [stdin\(3\)](#),
[cpuid\(4\)](#), [proc\(5\)](#), [pipe\(7\)](#), [signal-safety\(7\)](#), [spufs\(7\)](#), [user_namespaces\(7\)](#), [xfs_io\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of
The Linux Programming Interface.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



pread(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

 pread(2)

System Calls Manual

pread(2)

NAME [top](#)

pread, pwrite - read from or write to a file descriptor at a given offset

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
ssize_t pread(int fd, void buf[.count], size_t count,  
              off_t offset);  
ssize_t pwrite(int fd, const void buf[.count], size_t count,  
              off_t offset);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
pread(), pwrite():  
    _XOPEN_SOURCE >= 500  
    || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
```

DESCRIPTION [top](#)

pread() reads up to *count* bytes from file descriptor *fd* at offset *offset* (from the start of the file) into the buffer starting at *buf*. The file offset is not changed.

pwrite() writes up to *count* bytes from the buffer starting at *buf* to the file descriptor *fd* at offset *offset*. The file offset is not changed.

The file referenced by *fd* must be capable of seeking.

RETURN VALUE [top](#)

On success, **pread()** returns the number of bytes read (a return of zero indicates end of file) and **pwrite()** returns the number of bytes written.

Note that it is not an error for a successful call to transfer fewer bytes than requested (see [read\(2\)](#) and [write\(2\)](#)).

On error, -1 is returned and *errno* is set to indicate the error.

ERRORS [top](#)

pread() can fail and set *errno* to any error specified for [read\(2\)](#) or [lseek\(2\)](#). **pwrite()** can fail and set *errno* to any error specified for [write\(2\)](#) or [lseek\(2\)](#).

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

Added in Linux 2.1.60; the entries in the i386 system call table were added in Linux 2.1.69. C library support (including emulation using [lseek\(2\)](#) on older kernels without the system calls) was added in glibc 2.1.

C library/kernel differences

On Linux, the underlying system calls were renamed in Linux 2.6: **pread()** became **pread64()**, and **pwrite()** became **pwrite64()**. The system call numbers remained the same. The glibc **pread()** and **pwrite()** wrapper functions transparently deal with the change.

On some 32-bit architectures, the calling signature for these system calls differ, for the reasons described in [syscall\(2\)](#).

NOTES [top](#)

The **pread()** and **pwrite()** system calls are especially useful in multithreaded applications. They allow multiple threads to perform I/O on the same file descriptor without being affected by changes to the file offset by other threads.

BUGS [top](#)

POSIX requires that opening a file with the **O_APPEND** flag should have no effect on the location at which **pwrite()** writes data. However, on Linux, if a file is opened with **O_APPEND**, **pwrite()** appends data to the end of the file, regardless of the value of *offset*.

SEE ALSO [top](#)

[lseek\(2\)](#), [read\(2\)](#), [readv\(2\)](#), [write\(2\)](#)

Linux man-pages (unreleased) [\(date\)](#) [pread\(2\)](#)

Pages that refer to this page: [ps\(1\)](#), [fcntl\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [read\(2\)](#), [readv\(2\)](#), [syscall\(2\)](#), [syscalls\(2\)](#), [write\(2\)](#), [off_t\(3type\)](#), [cpuid\(4\)](#), [proc\(5\)](#), [io_uring\(7\)](#), [socket\(7\)](#), [spufs\(7\)](#), [user_namespaces\(7\)](#), [xfs_io\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



truncate(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

 truncate(2)

System Calls Manual

truncate(2)

NAME [top](#)

truncate, ftruncate - truncate a file to a specified length

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int truncate(const char *path, off_t length);  
int ftruncate(int fd, off_t length);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

truncate():

```
  _XOPEN_SOURCE >= 500  
  || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L  
  || /* glibc <= 2.19: */ _BSD_SOURCE
```

ftruncate():

```
  _XOPEN_SOURCE >= 500  
  || /* Since glibc 2.3.5: */ _POSIX_C_SOURCE >= 200112L  
  || /* glibc <= 2.19: */ _BSD_SOURCE
```


DESCRIPTION [top](#)

The **truncate()** and **ftruncate()** functions cause the regular file named by *path* or referenced by *fd* to be truncated to a size of precisely *length* bytes.

If the file previously was larger than this size, the extra data is lost. If the file previously was shorter, it is extended, and the extended part reads as null bytes ('\0').

The file offset is not changed.

If the size changed, then the `st_ctime` and `st_mtime` fields (respectively, time of last status change and time of last modification; see [inode\(7\)](#)) for the file are updated, and the set-user-ID and set-group-ID mode bits may be cleared.

With **ftruncate()**, the file must be open for writing; with **truncate()**, the file must be writable.

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

For **truncate()**:

EACCES Search permission is denied for a component of the path prefix, or the named file is not writable by the user. (See also [path_resolution\(7\)](#).)

EFAULT The argument *path* points outside the process's allocated address space.

EFBIG The argument *length* is larger than the maximum file size. (XSI)

EINTR While blocked waiting to complete, the call was interrupted by a signal handler; see [fcntl\(2\)](#) and [signal\(7\)](#).



EINVAL The argument *length* is negative or larger than the maximum file size.

EIO An I/O error occurred updating the inode.

EISDIR The named file is a directory.

ELOOP Too many symbolic links were encountered in translating the pathname.

ENAMETOOLONG

A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.

ENOENT The named file does not exist.

ENOTDIR

A component of the path prefix is not a directory.

EPERM The underlying filesystem does not support extending a file beyond its current size.

EPERM The operation was prevented by a file seal; see [fcntl\(2\)](#).

EROFS The named file resides on a read-only filesystem.

ETXTBSY

The file is an executable file that is being executed.

For **ftruncate()** the same errors apply, but instead of things that can be wrong with *path*, we now have things that can be wrong with the file descriptor, *fd*:

EBADF *fd* is not a valid file descriptor.

EBADF or **EINVAL**

fd is not open for writing.

EINVAL *fd* does not reference a regular file or a POSIX shared memory object.

EINVAL or **EBADF**

The file descriptor *fd* is not open for writing. POSIX

permits, and portable applications should handle, either error for this case. (Linux produces **EINVAL**.)

VERSIONS [top](#)

The details in DESCRIPTION are for XSI-compliant systems. For non-XSI-compliant systems, the POSIX standard allows two behaviors for **ftruncate()** when *length* exceeds the file length (note that **truncate()** is not specified at all in such an environment): either returning an error, or extending the file. Like most UNIX implementations, Linux follows the XSI requirement when dealing with native filesystems. However, some nonnative filesystems do not permit **truncate()** and **ftruncate()** to be used to extend a file beyond its current length: a notable example on Linux is VFAT.

On some 32-bit architectures, the calling signature for these system calls differ, for the reasons described in [syscall\(2\)](#).

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, 4.4BSD, SVr4 (first appeared in 4.2BSD).

The original Linux **truncate()** and **ftruncate()** system calls were not designed to handle large file offsets. Consequently, Linux 2.4 added **truncate64()** and **ftruncate64()** system calls that handle large files. However, these details can be ignored by applications using glibc, whose wrapper functions transparently employ the more recent system calls where they are available.

NOTES [top](#)

ftruncate() can also be used to set the size of a POSIX shared memory object; see [shm_open\(3\)](#).

BUGS [top](#)

A header file bug in glibc 2.12 meant that the minimum value of `_POSIX_C_SOURCE` required to expose the declaration of `ftruncate()` was 200809L instead of 200112L. This has been fixed in later glibc versions.

SEE ALSO [top](#)

[truncate\(1\)](#), [open\(2\)](#), [stat\(2\)](#), [path_resolution\(7\)](#)

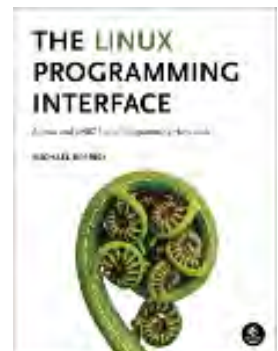
Linux man-pages (unreleased) (date) [truncate\(2\)](#)

Pages that refer to this page: [truncate\(1\)](#), [fallocate\(2\)](#), [fcntl\(2\)](#), [fsync\(2\)](#), [getrlimit\(2\)](#), [memfd_create\(2\)](#), [memfd_secret\(2\)](#), [mmap\(2\)](#), [syscall\(2\)](#), [syscalls\(2\)](#), [off_t\(3type\)](#), [shm_open\(3\)](#), [inode\(7\)](#), [inotify\(7\)](#), [landlock\(7\)](#), [shm_overview\(7\)](#), [signal-safety\(7\)](#), [xfs_io\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



select(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [EXAMPLES](#) | [SEE ALSO](#)

 [select\(2\)](#)

System Calls Manual

[select\(2\)](#)

NAME [top](#)

select, pselect, FD_CLR, FD_ISSET, FD_SET, FD_ZERO, fd_set - synchronous I/O multiplexing

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/select.h>
```

```
typedef /* ... */ fd_set;
```

```
int select(int nfds, fd_set *_Nullable restrict readfds,  
          fd_set *_Nullable restrict writefds,  
          fd_set *_Nullable restrict exceptfds,  
          struct timeval *_Nullable restrict timeout);
```

```
void FD_CLR(int fd, fd_set *set);  
int  FD_ISSET(int fd, fd_set *set);  
void FD_SET(int fd, fd_set *set);  
void FD_ZERO(fd_set *set);
```

```
int pselect(int nfds, fd_set *_Nullable restrict readfds,  
          fd_set *_Nullable restrict writefds,  
          fd_set *_Nullable restrict exceptfds,  
          const struct timespec *_Nullable restrict timeout,
```

```
const sigset_t *_Nullable restrict sigmask);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
pselect():  
    _POSIX_C_SOURCE >= 200112L
```

DESCRIPTION [top](#)

WARNING: `select()` can monitor only file descriptors numbers that are less than `FD_SETSIZE` (1024)—an unreasonably low limit for many modern applications—and this limitation will not change. All modern applications should instead use `poll(2)` or `epoll(7)`, which do not suffer this limitation.

`select()` allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform a corresponding I/O operation (e.g., `read(2)`, or a sufficiently small `write(2)`) without blocking.

`fd_set`

A structure type that can represent a set of file descriptors. According to POSIX, the maximum number of file descriptors in an `fd_set` structure is the value of the macro `FD_SETSIZE`.

File descriptor sets

The principal arguments of `select()` are three "sets" of file descriptors (declared with the type `fd_set`), which allow the caller to wait for three classes of events on the specified set of file descriptors. Each of the `fd_set` arguments may be specified as NULL if no file descriptors are to be watched for the corresponding class of events.

Note well: Upon return, each of the file descriptor sets is modified in place to indicate which file descriptors are currently "ready". Thus, if using `select()` within a loop, the sets *must be reinitialized* before each call.

The contents of a file descriptor set can be manipulated using the following macros:

`FD_ZERO()`

This macro clears (removes all file descriptors from) *set*. It should be employed as the first step in initializing a file descriptor set.

FD_SET()

This macro adds the file descriptor *fd* to *set*. Adding a file descriptor that is already present in the set is a no-op, and does not produce an error.

FD_CLR()

This macro removes the file descriptor *fd* from *set*. Removing a file descriptor that is not present in the set is a no-op, and does not produce an error.

FD_ISSET()

select() modifies the contents of the sets according to the rules described below. After calling **select()**, the **FD_ISSET()** macro can be used to test if a file descriptor is still present in a set. **FD_ISSET()** returns nonzero if the file descriptor *fd* is present in *set*, and zero if it is not.

Arguments

The arguments of **select()** are as follows:

readfds

The file descriptors in this set are watched to see if they are ready for reading. A file descriptor is ready for reading if a read operation will not block; in particular, a file descriptor is also ready on end-of-file.

After **select()** has returned, *readfds* will be cleared of all file descriptors except for those that are ready for reading.

writefds

The file descriptors in this set are watched to see if they are ready for writing. A file descriptor is ready for writing if a write operation will not block. However, even if a file descriptor indicates as writable, a large write may still block.

After **select()** has returned, *writefds* will be cleared of all file descriptors except for those that are ready for



writing.

exceptfds

The file descriptors in this set are watched for "exceptional conditions". For examples of some exceptional conditions, see the discussion of **POLLPRI** in [poll\(2\)](#).

After **select()** has returned, *exceptfds* will be cleared of all file descriptors except for those for which an exceptional condition has occurred.

nfds This argument should be set to the highest-numbered file descriptor in any of the three sets, plus 1. The indicated file descriptors in each set are checked, up to this limit (but see BUGS).

timeout

The *timeout* argument is a *timeval* structure (shown below) that specifies the interval that **select()** should block waiting for a file descriptor to become ready. The call will block until either:

- a file descriptor becomes ready;
- the call is interrupted by a signal handler; or
- the timeout expires.

Note that the *timeout* interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount.

If both fields of the *timeval* structure are zero, then **select()** returns immediately. (This is useful for polling.)

If *timeout* is specified as NULL, **select()** blocks indefinitely waiting for a file descriptor to become ready.

pselect()

The **pselect()** system call allows an application to safely wait until either a file descriptor becomes ready or until a signal is



caught.

The operation of **select()** and **pselect()** is identical, other than these three differences:

- **select()** uses a timeout that is a *struct timeval* (with seconds and microseconds), while **pselect()** uses a *struct timespec* (with seconds and nanoseconds).
- **select()** may update the *timeout* argument to indicate how much time was left. **pselect()** does not change this argument.
- **select()** has no *sigmask* argument, and behaves as **pselect()** called with NULL *sigmask*.

sigmask is a pointer to a signal mask (see [sigprocmask\(2\)](#)); if it is not NULL, then **pselect()** first replaces the current signal mask by the one pointed to by *sigmask*, then does the "select" function, and then restores the original signal mask. (If *sigmask* is NULL, the signal mask is not modified during the **pselect()** call.)

Other than the difference in the precision of the *timeout* argument, the following **pselect()** call:

```
ready = pselect(nfds, &readfds, &writefds, &exceptfds,  
               timeout, &sigmask);
```

is equivalent to *atomically* executing the following calls:

```
sigset_t origmask;  
  
pthread_sigmask(SIG_SETMASK, &sigmask, &origmask);  
ready = select(nfds, &readfds, &writefds, &exceptfds, timeout);  
pthread_sigmask(SIG_SETMASK, &origmask, NULL);
```

The reason that **pselect()** is needed is that if one wants to wait for either a signal or for a file descriptor to become ready, then an atomic test is needed to prevent race conditions. (Suppose the signal handler sets a global flag and returns. Then a test of this global flag followed by a call of **select()** could hang indefinitely if the signal arrived just after the test but just before the call. By contrast, **pselect()** allows one to first block signals, handle the signals that have come in, then call **pselect()** with the desired *sigmask*, avoiding the race.)



The `timeout`

The `timeout` argument for `select()` is a structure of the following type:

```
struct timeval {
    time_t      tv_sec;          /* seconds */
    suseconds_t tv_usec;       /* microseconds */
};
```

The corresponding argument for `pselect()` is a `timespec(3)` structure.

On Linux, `select()` modifies `timeout` to reflect the amount of time not slept; most other implementations do not do this. (POSIX.1 permits either behavior.) This causes problems both when Linux code which reads `timeout` is ported to other operating systems, and when code is ported to Linux that reuses a `struct timeval` for multiple `select()`s in a loop without reinitializing it. Consider `timeout` to be undefined after `select()` returns.

RETURN VALUE [top](#)

On success, `select()` and `pselect()` return the number of file descriptors contained in the three returned descriptor sets (that is, the total number of bits that are set in `readfds`, `writefds`, `exceptfds`). The return value may be zero if the timeout expired before any file descriptors became ready.

On error, `-1` is returned, and `errno` is set to indicate the error; the file descriptor sets are unmodified, and `timeout` becomes undefined.

ERRORS [top](#)

EBADF An invalid file descriptor was given in one of the sets. (Perhaps a file descriptor that was already closed, or one on which an error has occurred.) However, see **BUGS**.

EINTR A signal was caught; see [signal\(7\)](#).

EINVAL `nfds` is negative or exceeds the `RLIMIT_NOFILE` resource limit (see [getrlimit\(2\)](#)).

EINVAL The value contained within *timeout* is invalid.

ENOMEM Unable to allocate memory for internal tables.

VERSIONS [top](#)

On some other UNIX systems, **select()** can fail with the error **EAGAIN** if the system fails to allocate kernel-internal resources, rather than **ENOMEM** as Linux does. POSIX specifies this error for **poll(2)**, but not for **select()**. Portable programs may wish to check for **EAGAIN** and loop, just as with **EINTR**.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

select()

POSIX.1-2001, 4.4BSD (first appeared in 4.2BSD).

Generally portable to/from non-BSD systems supporting clones of the BSD socket layer (including System V variants). However, note that the System V variant typically sets the timeout variable before returning, but the BSD variant does not.

pselect()

Linux 2.6.16. POSIX.1g, POSIX.1-2001.

Prior to this, it was emulated in glibc (but see BUGS).

fd_set POSIX.1-2001.

NOTES [top](#)

The following header also provides the *fd_set* type: `<sys/time.h>`.

An *fd_set* is a fixed size buffer. Executing **FD_CLR()** or **FD_SET()** with a value of *fd* that is negative or is equal to or larger than **FD_SETSIZE** will result in undefined behavior. Moreover, POSIX requires *fd* to be a valid file descriptor.

The operation of **select()** and **pselect()** is not affected by the **O_NONBLOCK** flag.

The self-pipe trick

On systems that lack **pselect()**, reliable (and more portable) signal trapping can be achieved using the self-pipe trick. In this technique, a signal handler writes a byte to a pipe whose other end is monitored by **select()** in the main program. (To avoid possibly blocking when writing to a pipe that may be full or reading from a pipe that may be empty, nonblocking I/O is used when reading from and writing to the pipe.)

Emulating **usleep(3)**

Before the advent of **usleep(3)**, some code employed a call to **select()** with all three sets empty, *nfds* zero, and a non-NULL *timeout* as a fairly portable way to sleep with subsecond precision.

Correspondence between **select()** and **poll()** notifications

Within the Linux kernel source, we find the following definitions which show the correspondence between the readable, writable, and exceptional condition notifications of **select()** and the event notifications provided by **poll(2)** and **epoll(7)**:

```
#define POLLIN_SET  (EPOLLRDNORM | EPOLLRDBAND | EPOLLIN |  
                   EPOLLHUP | EPOLLERR)  
/* Ready for reading */  
#define POLLOUT_SET (EPOLLWRBAND | EPOLLWRNORM | EPOLLOUT |  
                   EPOLLERR)  
/* Ready for writing */  
#define POLLEX_SET (EPOLLPRI)  
/* Exceptional condition */
```

Multithreaded applications

If a file descriptor being monitored by **select()** is closed in another thread, the result is unspecified. On some UNIX systems, **select()** unblocks and returns, with an indication that the file descriptor is ready (a subsequent I/O operation will likely fail with an error, unless another process reopens the file descriptor between the time **select()** returned and the I/O operation is performed). On Linux (and some other systems), closing the file descriptor in another thread has no effect on **select()**. In summary, any application that relies on a particular behavior in this scenario must be considered buggy.



C library/kernel differences

The Linux kernel allows file descriptor sets of arbitrary size, determining the length of the sets to be checked from the value of *nfds*. However, in the glibc implementation, the *fd_set* type is fixed in size. See also BUGS.

The **pselect()** interface described in this page is implemented by glibc. The underlying Linux system call is named **pselect6()**. This system call has somewhat different behavior from the glibc wrapper function.

The Linux **pselect6()** system call modifies its *timeout* argument. However, the glibc wrapper function hides this behavior by using a local variable for the timeout argument that is passed to the system call. Thus, the glibc **pselect()** function does not modify its *timeout* argument; this is the behavior required by POSIX.1-2001.

The final argument of the **pselect6()** system call is not a *sigset_t ** pointer, but is instead a structure of the form:

```
struct {
    const kernel_sigset_t *ss;    /* Pointer to signal set */
    size_t ss_len;               /* Size (in bytes) of object
                                pointed to by 'ss' */
};
```

This allows the system call to obtain both a pointer to the signal set and its size, while allowing for the fact that most architectures support a maximum of 6 arguments to a system call. See [sigprocmask\(2\)](#) for a discussion of the difference between the kernel and libc notion of the signal set.

Historical glibc details

glibc 2.0 provided an incorrect version of **pselect()** that did not take a *sigmask* argument.

From glibc 2.1 to glibc 2.2.1, one must define **_GNU_SOURCE** in order to obtain the declaration of **pselect()** from `<sys/select.h>`.

BUGS [top](#)

POSIX allows an implementation to define an upper limit, advertised via the constant **FD_SETSIZE**, on the range of file

descriptors that can be specified in a file descriptor set. The Linux kernel imposes no fixed limit, but the glibc implementation makes *fd_set* a fixed-size type, with **FD_SETSIZE** defined as 1024, and the **FD_*()** macros operating according to that limit. To monitor file descriptors greater than 1023, use [poll\(2\)](#) or [epoll\(7\)](#) instead.

The implementation of the *fd_set* arguments as value-result arguments is a design error that is avoided in [poll\(2\)](#) and [epoll\(7\)](#).

According to POSIX, **select()** should check all specified file descriptors in the three file descriptor sets, up to the limit *nfds-1*. However, the current implementation ignores any file descriptor in these sets that is greater than the maximum file descriptor number that the process currently has open. According to POSIX, any such file descriptor that is specified in one of the sets should result in the error **EBADF**.

Starting with glibc 2.1, glibc provided an emulation of **pselect()** that was implemented using [sigprocmask\(2\)](#) and **select()**. This implementation remained vulnerable to the very race condition that **pselect()** was designed to prevent. Modern versions of glibc use the (race-free) **pselect()** system call on kernels where it is provided.

On Linux, **select()** may report a socket file descriptor as "ready for reading", while nevertheless a subsequent read blocks. This could for example happen when data has arrived but upon examination has the wrong checksum and is discarded. There may be other circumstances in which a file descriptor is spuriously reported as ready. Thus it may be safer to use **O_NONBLOCK** on sockets that should not block.

On Linux, **select()** also modifies *timeout* if the call is interrupted by a signal handler (i.e., the **EINTR** error return). This is not permitted by POSIX.1. The Linux **pselect()** system call has the same behavior, but the glibc wrapper hides this behavior by internally copying the *timeout* to a local variable and passing that variable to the system call.

EXAMPLES [top](#)

```
#include <stdio.h>
#include <stdlib.h>
```



```
#include <sys/select.h>

int
main(void)
{
    int          retval;
    fd_set       rfds;
    struct timeval tv;

    /* Watch stdin (fd 0) to see when it has input. */

    FD_ZERO(&rfds);
    FD_SET(0, &rfds);

    /* Wait up to five seconds. */

    tv.tv_sec = 5;
    tv.tv_usec = 0;

    retval = select(1, &rfds, NULL, NULL, &tv);
    /* Don't rely on the value of tv now! */

    if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Data is available now.\n");
        /* FD_ISSET(0, &rfds) will be true. */
    else
        printf("No data within five seconds.\n");

    exit(EXIT_SUCCESS);
}
```

SEE ALSO [top](#)

[accept\(2\)](#), [connect\(2\)](#), [poll\(2\)](#), [read\(2\)](#), [recv\(2\)](#),
[restart_syscall\(2\)](#), [send\(2\)](#), [sigprocmask\(2\)](#), [write\(2\)](#),
[timespec\(3\)](#), [epoll\(7\)](#), [time\(7\)](#)

For a tutorial with discussion and examples, see [select_tut\(2\)](#).

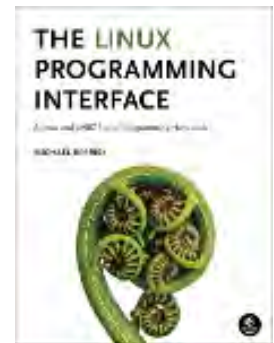
Linux man-pages (unreleased) (date) [select\(2\)](#)

Pages that refer to this page: [strace\(1\)](#), [accept\(2\)](#), [alarm\(2\)](#), [connect\(2\)](#), [epoll_wait\(2\)](#), [eventfd\(2\)](#), [fcntl\(2\)](#), [futex\(2\)](#), [ioctl_tty\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [migrate_pages\(2\)](#), [open\(2\)](#), [pause\(2\)](#), [perf_event_open\(2\)](#), [perfmonctl\(2\)](#), [personality\(2\)](#), [pidfd_open\(2\)](#), [poll\(2\)](#), [prctl\(2\)](#), [read\(2\)](#), [recv\(2\)](#), [restart_syscall\(2\)](#), [seccomp_unotify\(2\)](#), [select_tut\(2\)](#), [send\(2\)](#), [signalfd\(2\)](#), [socket\(2\)](#), [syscalls\(2\)](#), [timerfd_create\(2\)](#), [userfaultfd\(2\)](#), [write\(2\)](#), [avc_netlink_loop\(3\)](#), [ldap_get_option\(3\)](#), [ldap_result\(3\)](#), [pcap\(3pcap\)](#), [pcap_get_required_select_timeout\(3pcap\)](#), [pcap_get_selectable_fd\(3pcap\)](#), [pmrecord\(3\)](#), [pmtime\(3\)](#), [rpc\(3\)](#), [sctp_connectx\(3\)](#), [timeval\(3type\)](#), [ualarm\(3\)](#), [usleep\(3\)](#), [random\(4\)](#), [rtc\(4\)](#), [proc\(5\)](#), [slapd-asynctrans\(5\)](#), [slapd-ldap\(5\)](#), [slapd-meta\(5\)](#), [systemd.exec\(5\)](#), [epoll\(7\)](#), [fanotify\(7\)](#), [inotify\(7\)](#), [mq_overview\(7\)](#), [pipe\(7\)](#), [pty\(7\)](#), [signal\(7\)](#), [signal-safety\(7\)](#), [socket\(7\)](#), [system_data_types\(7\)](#), [tcp\(7\)](#), [time\(7\)](#), [udp\(7\)](#), [setarch\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



poll(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [EXAMPLES](#) | [SEE ALSO](#)

 poll(2)

System Calls Manual

poll(2)**NAME** [top](#)

poll, ppoll - wait for some event on a file descriptor

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <poll.h>

int poll(struct pollfd *fds, nfd_t nfd, int timeout);

#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <poll.h>

int ppoll(struct pollfd *fds, nfd_t nfd,
          const struct timespec *_Nullable tmo_p,
          const sigset_t *_Nullable sigmask);
```

DESCRIPTION [top](#)

poll() performs a similar task to **select(2)**: it waits for one of a set of file descriptors to become ready to perform I/O. The Linux-specific **epoll(7)** API performs a similar task, but offers features beyond those found in **poll()**.

The set of file descriptors to be monitored is specified in the *fds* argument, which is an array of structures of the following

form:

```
struct pollfd {
    int    fd;          /* file descriptor */
    short events;      /* requested events */
    short revents;     /* returned events */
};
```

The caller should specify the number of items in the *fds* array in *nfds*.

The field *fd* contains a file descriptor for an open file. If this field is negative, then the corresponding *events* field is ignored and the *revents* field returns zero. (This provides an easy way of ignoring a file descriptor for a single `poll()` call: simply set the *fd* field to its bitwise complement.)

The field *events* is an input parameter, a bit mask specifying the events the application is interested in for the file descriptor *fd*. This field may be specified as zero, in which case the only events that can be returned in *revents* are **POLLHUP**, **POLLERR**, and **POLLNVAL** (see below).

The field *revents* is an output parameter, filled by the kernel with the events that actually occurred. The bits returned in *revents* can include any of those specified in *events*, or one of the values **POLLERR**, **POLLHUP**, or **POLLNVAL**. (These three bits are meaningless in the *events* field, and will be set in the *revents* field whenever the corresponding condition is true.)

If none of the events requested (and no error) has occurred for any of the file descriptors, then `poll()` blocks until one of the events occurs.

The *timeout* argument specifies the number of milliseconds that `poll()` should block waiting for a file descriptor to become ready. The call will block until either:

- a file descriptor becomes ready;
- the call is interrupted by a signal handler; or
- the timeout expires.

Being "ready" means that the requested operation will not block; thus, `poll()`ing regular files, block devices, and other files with no reasonable polling semantic *always* returns instantly as



ready to read and write.

Note that the *timeout* interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount. Specifying a negative value in *timeout* means an infinite timeout. Specifying a *timeout* of zero causes `poll()` to return immediately, even if no file descriptors are ready.

The bits that may be set/returned in *events* and *revents* are defined in `<poll.h>`:

POLLIN There is data to read.

POLLPRI

There is some exceptional condition on the file descriptor. Possibilities include:

- There is out-of-band data on a TCP socket (see [tcp\(7\)](#)).
- A pseudoterminal master in packet mode has seen a state change on the slave (see [ioctl_tty\(2\)](#)).
- A *cgroup.events* file has been modified (see [cgroups\(7\)](#)).

POLLOUT

Writing is now possible, though a write larger than the available space in a socket or pipe will still block (unless **O_NONBLOCK** is set).

POLLRDHUP (since Linux 2.6.17)

Stream socket peer closed connection, or shut down writing half of connection. The **_GNU_SOURCE** feature test macro must be defined (before including *any* header files) in order to obtain this definition.

POLLERR

Error condition (only returned in *revents*; ignored in *events*). This bit is also set for a file descriptor referring to the write end of a pipe when the read end has been closed.

POLLHUP

Hang up (only returned in *revents*; ignored in *events*). Note that when reading from a channel such as a pipe or a stream socket, this event merely indicates that the peer

closed its end of the channel. Subsequent reads from the channel will return `0` (end of file) only after all outstanding data in the channel has been consumed.

POLLNVAL

Invalid request: *fd* not open (only returned in *revents*; ignored in *events*).

When compiling with `_XOPEN_SOURCE` defined, one also has the following, which convey no further information beyond the bits listed above:

POLLRDNORM

Equivalent to `POLLIN`.

POLLRDBAND

Priority band data can be read (generally unused on Linux).

POLLWRNORM

Equivalent to `POLLOUT`.

POLLWRBAND

Priority data may be written.

Linux also knows about, but does not use `POLLMSG`.

ppoll()

The relationship between `poll()` and `ppoll()` is analogous to the relationship between `select(2)` and `pselect(2)`: like `pselect(2)`, `ppoll()` allows an application to safely wait until either a file descriptor becomes ready or until a signal is caught.

Other than the difference in the precision of the *timeout* argument, the following `ppoll()` call:

```
ready = ppoll(&fds, nfds, tmo_p, &sigmask);
```

is nearly equivalent to *atomically* executing the following calls:

```
sigset_t origmask;
int timeout;

timeout = (tmo_p == NULL) ? -1 :
          (tmo_p->tv_sec * 1000 + tmo_p->tv_nsec / 1000000);
pthread_sigmask(SIG_SETMASK, &sigmask, &origmask);
ready = poll(&fds, nfds, timeout);
```



```
pthread_sigmask(SIG_SETMASK, &origmask, NULL);
```

The above code segment is described as *nearly* equivalent because whereas a negative *timeout* value for `poll()` is interpreted as an infinite timeout, a negative value expressed in **tmo_p* results in an error from `ppoll()`.

See the description of `pselect(2)` for an explanation of why `ppoll()` is necessary.

If the *sigmask* argument is specified as NULL, then no signal mask manipulation is performed (and thus `ppoll()` differs from `poll()` only in the precision of the *timeout* argument).

The *tmo_p* argument specifies an upper limit on the amount of time that `ppoll()` will block. This argument is a pointer to a `timespec(3)` structure.

If *tmo_p* is specified as NULL, then `ppoll()` can block indefinitely.

RETURN VALUE [top](#)

On success, `poll()` returns a nonnegative value which is the number of elements in the *pollfds* whose *revents* fields have been set to a nonzero value (indicating an event or an error). A return value of zero indicates that the system call timed out before any file descriptors became ready.

On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EFAULT *fds* points outside the process's accessible address space. The array given as argument was not contained in the calling program's address space.

EINTR A signal occurred before any requested event; see [signal\(7\)](#).

EINVAL The *nfds* value exceeds the `RLIMIT_NOFILE` value.

EINVAL (`ppoll()`) The timeout value expressed in **tmo_p* is invalid (negative).



ENOMEM Unable to allocate memory for kernel data structures.

VERSIONS [top](#)

On some other UNIX systems, **poll()** can fail with the error **EAGAIN** if the system fails to allocate kernel-internal resources, rather than **ENOMEM** as Linux does. POSIX permits this behavior. Portable programs may wish to check for **EAGAIN** and loop, just as with **EINTR**.

Some implementations define the nonstandard constant **INFTIM** with the value -1 for use as a *timeout* for **poll()**. This constant is not provided in glibc.

C library/kernel differences

The Linux **ppoll()** system call modifies its *tmo_p* argument. However, the glibc wrapper function hides this behavior by using a local variable for the timeout argument that is passed to the system call. Thus, the glibc **ppoll()** function does not modify its *tmo_p* argument.

The raw **ppoll()** system call has a fifth argument, *size_t sigsetsize*, which specifies the size in bytes of the *sigmask* argument. The glibc **ppoll()** wrapper function specifies this argument as a fixed value (equal to *sizeof(kernel_sigset_t)*). See [sigprocmask\(2\)](#) for a discussion on the differences between the kernel and the libc notion of the sigset.

STANDARDS [top](#)

poll() POSIX.1-2008.

ppoll()
Linux.

HISTORY [top](#)

poll() POSIX.1-2001. Linux 2.1.23.

On older kernels that lack this system call, the glibc **poll()** wrapper function provides emulation using [select\(2\)](#).

ppoll()

Linux 2.6.16, glibc 2.4.

NOTES [top](#)

The operation of **poll()** and **ppoll()** is not affected by the **O_NONBLOCK** flag.

For a discussion of what may happen if a file descriptor being monitored by **poll()** is closed in another thread, see [select\(2\)](#).

BUGS [top](#)

See the discussion of spurious readiness notifications under the **BUGS** section of [select\(2\)](#).

EXAMPLES [top](#)

The program below opens each of the files named in its command-line arguments and monitors the resulting file descriptors for readiness to read (**POLLIN**). The program loops, repeatedly using **poll()** to monitor the file descriptors, printing the number of ready file descriptors on return. For each ready file descriptor, the program:

- displays the returned *revents* field in a human-readable form;
- if the file descriptor is readable, reads some data from it, and displays that data on standard output; and
- if the file descriptor was not readable, but some other event occurred (presumably **POLLHUP**), closes the file descriptor.

Suppose we run the program in one terminal, asking it to open a FIFO:

```
$ mkfifo myfifo
$ ./poll_input myfifo
```

In a second terminal window, we then open the FIFO for writing, write some data to it, and close the FIFO:

```
$ echo aaaabbbbcccc > myfifo
```

In the terminal where we are running the program, we would then

see:

```
Opened "myfifo" on fd 3
About to poll()
Ready: 1
  fd=3; events: POLLIN POLLHUP
  read 10 bytes: aaaaabbbbb
About to poll()
Ready: 1
  fd=3; events: POLLIN POLLHUP
  read 6 bytes: ccccc

About to poll()
Ready: 1
  fd=3; events: POLLHUP
  closing fd 3
All file descriptors closed; bye
```

In the above output, we see that **poll()** returned three times:

- On the first return, the bits returned in the *revents* field were **POLLIN**, indicating that the file descriptor is readable, and **POLLHUP**, indicating that the other end of the FIFO has been closed. The program then consumed some of the available input.
- The second return from **poll()** also indicated **POLLIN** and **POLLHUP**; the program then consumed the last of the available input.
- On the final return, **poll()** indicated only **POLLHUP** on the FIFO, at which point the file descriptor was closed and the program terminated.

Program source

```
/* poll_input.c

Licensed under GNU General Public License v2 or later.
*/
#include <fcntl.h>
#include <poll.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
```




```
    } while (0)

int
main(int argc, char *argv[])
{
    int          ready;
    char         buf[10];
    nfd_t        num_open_fds, nfd;
    ssize_t      s;
    struct pollfd *pfd;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s file...\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    num_open_fds = nfd = argc - 1;
    pfd = calloc(nfd, sizeof(struct pollfd));
    if (pfd == NULL)
        errExit("malloc");

    /* Open each file on command line, and add it to 'pfd' array. */

    for (nfd_t j = 0; j < nfd; j++) {
        pfd[j].fd = open(argv[j + 1], O_RDONLY);
        if (pfd[j].fd == -1)
            errExit("open");

        printf("Opened \"%s\" on fd %d\n", argv[j + 1], pfd[j].fd);

        pfd[j].events = POLLIN;
    }

    /* Keep calling poll() as long as at least one file descriptor is
       open. */

    while (num_open_fds > 0) {
        printf("About to poll()\n");
        ready = poll(pfd, nfd, -1);
        if (ready == -1)
            errExit("poll");

        printf("Ready: %d\n", ready);

        /* Deal with array returned by poll(). */

        for (nfd_t j = 0; j < nfd; j++) {
```



```

if (pfds[j].revents != 0) {
    printf(" fd=%d; events: %s%s%s\n", pfd[j].fd,
           (pfds[j].revents & POLLIN) ? "POLLIN " : "",
           (pfds[j].revents & POLLHUP) ? "POLLHUP " : "",
           (pfds[j].revents & POLLERR) ? "POLLERR " : "");

    if (pfds[j].revents & POLLIN) {
        s = read(pfd[j].fd, buf, sizeof(buf));
        if (s == -1)
            errExit("read");
        printf(" read %zd bytes: %.*s\n",
              s, (int) s, buf);
    } else {
        /* POLLERR | POLLHUP */
        printf(" closing fd %d\n", pfd[j].fd);
        if (close(pfd[j].fd) == -1)
            errExit("close");
        num_open_fds--;
    }
}
}
}

printf("All file descriptors closed; bye\n");
exit(EXIT_SUCCESS);
}

```

SEE ALSO [top](#)

[restart_syscall\(2\)](#), [select\(2\)](#), [select_tut\(2\)](#), [timespec\(3\)](#),
[epoll\(7\)](#), [time\(7\)](#)

Linux man-pages (unreleased)

(date)

[poll\(2\)](#)

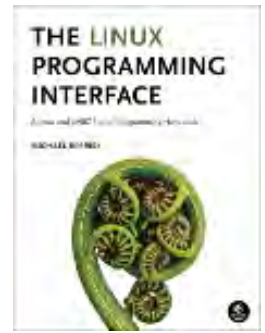
Pages that refer to this page: [less\(1\)](#), [accept\(2\)](#), [connect\(2\)](#), [epoll_ctl\(2\)](#), [epoll_wait\(2\)](#), [eventfd\(2\)](#), [fcntl\(2\)](#), [futexp\(2\)](#), [ioctl_tty\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [open\(2\)](#), [perf_event_open\(2\)](#), [perfmonctl\(2\)](#), [personality\(2\)](#), [pidfd_open\(2\)](#), [prctl\(2\)](#), [ptrace\(2\)](#), [recv\(2\)](#), [restart_syscall\(2\)](#), [seccomp_unotify\(2\)](#), [select\(2\)](#), [select_tut\(2\)](#), [sigaction\(2\)](#), [signalfd\(2\)](#), [syscalls\(2\)](#), [timerfd_create\(2\)](#), [userfaultfd\(2\)](#), [io_uring_prep_poll_add\(3\)](#), [io_uring_prep_poll_multishot\(3\)](#), [ldap_get_option\(3\)](#), [pcap\(3pcap\)](#), [pcap_get_required_select_timeout\(3pcap\)](#), [pcap_get_selectable_fd\(3pcap\)](#), [rtime\(3\)](#), [sctp_connectx\(3\)](#), [sd_bus_get_fd\(3\)](#), [sd_bus_wait\(3\)](#), [sd_journal_get_fd\(3\)](#), [sd_login_monitor_new\(3\)](#), [sd_notify\(3\)](#), [random\(4\)](#), [proc\(5\)](#), [slapd-asynctrans\(5\)](#), [slapd-ldap\(5\)](#), [slapd-meta\(5\)](#), [systemd.exec\(5\)](#), [cgroups\(7\)](#), [epoll\(7\)](#), [fanotify\(7\)](#), [inotify\(7\)](#), [mq_overview\(7\)](#), [pipe\(7\)](#), [signal\(7\)](#), [signal-safety\(7\)](#), [socket\(7\)](#), [spufs\(7\)](#), [system_data_types\(7\)](#), [tcp\(7\)](#), [udp\(7\)](#), [setarch\(8\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



Materials for Topic 4: File I/O

Full C Programs

- [file_io_syscalls.c](#) - a C program that demonstrates uses of the `creat()`, `open()`, `read()`, `write()`, `pread()`, `pwrite()`, `ftruncate()`, `fsync()`, `sync()`, `lseek()`, and `close()` file I/O system calls.
- [select_syscall_example.c](#) - a C program that demonstrates a use of the `select()` system call, which continuously checks for the availability of several I/O files.
- [poll_syscall_example.c](#) - a C program that demonstrates a use of the `poll()` system call, a more efficient function than `select()` in some aspects. It continuously checks for the availability of several I/O files.

Runnable Linux Commands

- The command:

```
gcc -Wall -Wextra -O2 -g -o program program.c
```

compiles the C source code located inside the file `program.c`. See more details [here](#).

- The command:

```
./short_prompt
```

executes code inside a file named `short_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).

- The command:

```
. ./long_prompt
```

executes code inside a file named `long_prompt` and sources it (applies all the changes to the current session.)

See more details [here](#).



This website by [Miriam Briskman](#) is licensed under [CC BY-NC 4.0](#).

```
1  /* A C program that demonstrates uses of the creat(),
2  *   open(), read(), write(), pread(), pwrite(),
3  *   ftruncate(), fsync(), sync(), lseek(), and
4  *   close() file I/O system calls.
5  *
6  *   Miriam Briskman, 2/27/2023
7  *   CISC 3350, Brooklyn College
8  *   Licensed under CC BY-NC 4.0
9  */
10
11 #define _XOPEN_SOURCE 500
12
13 // Library defining types such as size_t (usually
14 //   as an unsigned integer:)
15 #include <sys/types.h>
16 // Library defining modes such as S_IRUSR
17 //   (=0000400, r for owner:)
18 #include <sys/stat.h>
19 // Library defining the open() syscall and flags
20 //   such as O_RDONLY:
21 #include <fcntl.h>
22 // Library defining standard symbolic constants and
23 //   types. Needed for read(), write(), fsync(),
24 //   sync(), lseek(), ftruncate(), and close():
25 #include <unistd.h>
26 // Needed for perror():
27 #include <stdio.h>
28 // Needed for 'EXIT_SUCCESS' and 'EXIT_FAILURE':
29 #include <stdlib.h>
30 // Needed for strlen():
31 #include <string.h>
32
33 // An unsigned integer constant representing the
34 //   size of an array that we create.
35 const size_t BUF_SIZE = 100;
36
37 int main ()
38 {
39     // An array consisting of the alphabet:
40     char alphabet [] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
41
42     //////////////////////////////////////
```



```
43 // Creating and writing into a file //
44 ///////////////////////////////////////////////////////////////////
45
46 // Create a new file and open it for writing.
47 //   The file's permissions will be: read &
48 //   write for the owner, and read-only for the
49 //   owner's group and other users.
50 int file_descriptor = creat ("content.txt",
51                             S_IWUSR | S_IRUSR |
52                             S_IRGRP | S_IROTH);
53 // The above call to creat() is equivalent to
54 // the following call to open():
55 //
56 // open ("content.txt", O_WRONLY | O_CREAT | O_TRUNC,
57 //       S_IWUSR | S_IRUSR | S_IRGRP | S_IROTH);
58
59 if (file_descriptor == -1) // Checking for errors!
60 {
61     perror ("creat");
62     exit (EXIT_FAILURE);
63 }
64
65 // We now use the write() system call to write
66 // the alphabet into the file:
67 ssize_t chars_written = write (file_descriptor,
68                                alphabet,
69                                strlen (alphabet));
70 if (chars_written == -1)
71 {
72     perror ("write");
73 exit (EXIT_FAILURE);
74 }
75
76 // Since we are not guaranteed that this writing
77 // will actually be saved to the disk right
78 // away, and since we do want to save the change
79 // to the disk without waiting, we can force it
80 // to happen with fsync():
81 int output = fsync (file_descriptor);
82 if (output == -1)
83 {
84     perror ("fsync");
85     exit (EXIT_FAILURE);
```



```
86     }
87
88     // We are ready now to close the file.
89     // Instead of doing:
90     //     int result = close (file_descriptor);
91     //     if (result == -1)
92     //     {
93     //         perror ("close");
94     //         exit (EXIT_FAILURE);
95     //     }
96     // We use a shorter approach:
97     if (close (file_descriptor) == -1)
98     {
99         perror ("close");
100    exit (EXIT_FAILURE);
101    }
102
103
104    //////////////////////////////////////
105    // Reading from a file and printing to the terminal //
106    //////////////////////////////////////
107
108    // We now re-open the same one file that we created
109    //     earlier, but now for reading only:
110    file_descriptor = open ("content.txt", O_RDONLY);
111    if (file_descriptor == -1) // Checking for errors!
112    {
113        perror ("open");
114        exit (EXIT_FAILURE);
115    }
116
117    // Create a char array that into which we will store
118    //     the content that we read in:
119    char buf [BUF_SIZE];
120
121    // Attempt reading up to BUF_SIZE - 1 characters. We
122    //     need the last spot to store the null character
123    //     '\0' in it:
124    ssize_t number_of_chars_read_in = read (file_descriptor,
125                                           buf,
126                                           BUF_SIZE - 1);
127
128    // Checking for errors:
129    if (number_of_chars_read_in == -1)
```




```
129     {
130         perror ("read");
131         exit (EXIT_FAILURE);
132     }
133
134     // If we got to this point, we know that
135     //     'number_of_chars_read_in' tells how many characters
136     //     up to 99 were read from the file into the array
137     //     'buf' that we created above. Let's put the null
138     //     char right after the data we read in, and print the
139     //     contents of 'buf'!
140     buf[number_of_chars_read_in] = '\0';
141
142     char message1 [] = "1. We read the following "
143                       "content from the file:\n",
144     message2 [] = "2. We read the following "
145                  "content from the file:\n",
146     message3 [] = "3. We read the following "
147                  "content from the file:\n",
148     message4 [] = "4. We read the following "
149                  "content from the file:\n";
150
151     // The following write() system calls write content
152     //     to the screen. The screen is considered in
153     //     Linux as an open file with the descriptor of
154     //     STDOUT_FILENO.
155     if (write (STDOUT_FILENO,
156              message1,
157              strlen (message1))
158         == -1)
159     {
160         perror ("write");
161         exit (EXIT_FAILURE);
162     }
163     if (write (STDOUT_FILENO,
164              buf,
165              number_of_chars_read_in)
166         == -1)
167     {
168         perror ("write");
169         exit (EXIT_FAILURE);
170     }
171     if (write (STDOUT_FILENO, "\n", 1) == -1)
```



```
172     {
173         perror ("write");
174         exit (EXIT_FAILURE);
175     }
176
177     // Close the file now:
178     if (close (file_descriptor) == -1)
179     {
180         perror ("close");
181         exit (EXIT_FAILURE);
182     }
183
184     //////////////////////////////////////
185     // Reading from and writing into a file //
186     //////////////////////////////////////
187
188     // We now re-open the same one file that we
189     //     created earlier, but now for both
190     //     reading and writing:
191     file_descriptor = open ("content.txt", O_RDWR);
192     // Checking for errors:
193     if (file_descriptor == -1)
194     {
195         perror ("open");
196         exit (EXIT_FAILURE);
197     }
198
199     // Let's truncate the size of the file to only
200     //     10 characters:
201     if (ftruncate(file_descriptor, 10) == -1)
202     {
203         perror ("ftruncate");
204         exit (EXIT_FAILURE);
205     }
206
207     // After this truncation, let's read the content
208     //     of the file!
209     number_of_chars_read_in = read (file_descriptor,
210                                     buf,
211                                     BUF_SIZE - 1);
212
213     // Checking for errors:
214     if (number_of_chars_read_in == -1)
215     {
```



```
215     perror ("read");
216     exit (EXIT_FAILURE);
217 }
218
219 buf[number_of_chars_read_in] = '\0';
220
221 // Let's print what we read to the screen:
222 if (write (STDOUT_FILENO,
223         message2,
224         strlen (message2))
225     == -1)
226 {
227     perror ("write");
228     exit (EXIT_FAILURE);
229 }
230 if (write (STDOUT_FILENO,
231         buf,
232         number_of_chars_read_in)
233     == -1)
234 {
235     perror ("write");
236     exit (EXIT_FAILURE);
237 }
238 if (write (STDOUT_FILENO, "\n", 1) == -1)
239 {
240     perror ("write");
241     exit (EXIT_FAILURE);
242 }
243
244 // While we were reading the file, the reading
245 // position moved to the end of the file.
246 // To change the reading and writing
247 // position, you will use the lseek() system
248 // call. We now want to return the position
249 // to the top of the file:
250 off_t position = lseek (file_descriptor,
251                        (off_t) 0,
252                        SEEK_SET);
253 // Above, (off_t) is used to convert the integer
254 // 0 to a literal of the off_t data type.
255 // SEEK_SET tells that we want to jump to the
256 // exact position of 0 inside the file.
257 if (position == (off_t) -1)
```



```
258     {
259         perror ("lseek");
260         exit (EXIT_FAILURE);
261     }
262
263     // We now write the word "Hello" at the
264     //     beginning of the file. We will later see
265     //     how it will affect its content:
266     chars_written = write (file_descriptor,
267                           "Hello",
268                           strlen ("Hello"));
269     if (chars_written == -1)
270     {
271         perror ("write");
272         exit (EXIT_FAILURE);
273     }
274
275     // Since we just made a change to the content,
276     //     let's commit this change to the disk right
277     //     away by using sync():
278     sync();
279     // sync() doesn't return a value and never fails,
280     //     so we don't check for errors.
281     // sync() will commit the changes for ALL the
282     //     modified files that the program currently
283     //     has opened.
284
285     // We change the position of the file to the
286     //     beginning again because we want to read
287     //     from it again:
288     position = lseek (file_descriptor,
289                     (off_t) 0,
290                     SEEK_SET);
291     if (position == (off_t) -1)
292     {
293         perror ("lseek");
294         exit (EXIT_FAILURE);
295     }
296
297     // Let's read the file's content and print it:
298     number_of_chars_read_in = read (file_descriptor,
299                                   buf,
300                                   BUF_SIZE - 1);
```



```
301 // Checking for errors:
302 if (number_of_chars_read_in == -1)
303 {
304     perror ("read");
305     exit (EXIT_FAILURE);
306 }
307
308 buf[number_of_chars_read_in] = '\0';
309
310 // Let's print what we read to the screen:
311 if (write (STDOUT_FILENO,
312           message3,
313           strlen (message3))
314     == -1)
315 {
316     perror ("write");
317     exit (EXIT_FAILURE);
318 }
319 if (write (STDOUT_FILENO,
320           buf,
321           number_of_chars_read_in)
322     == -1)
323 {
324     perror ("write");
325     exit (EXIT_FAILURE);
326 }
327 if (write (STDOUT_FILENO, "\n", 1) == -1)
328 {
329     perror ("write");
330     exit (EXIT_FAILURE);
331 }
332
333 // As the last exercise for this program, we
334 // will attempt to write more content at the
335 // end of the file - but at a spot that is
336 // further beyond the end. We'll see what
337 // such a reading does to the file when we
338 // later run this program.
339
340 // We write the word: "Wow" at position 20 into
341 // the file. Instead of moving the position
342 // of the file as we did for the previous
343 // calls to read() and write(), we now use
```



```
344 // the syscalls pread() and pwrite(), which
345 // do the same thing as read() and write()
346 // but don't change the position inside the
347 // file. In the call to pwrite() below, the
348 // argument 20 (the last argument) indicates
349 // at what position we wish to start writing
350 // that content inside the file.
351 chars_written = pwrite (file_descriptor,
352                         "Wow",
353                         strlen ("Wow"),
354                         20);
355 if (chars_written == -1)
356 {
357     perror ("pwrite");
358     exit (EXIT_FAILURE);
359 }
360
361 // Next, we read the file from the beginning. As
362 // we did it for pwrite(), we pass 0 as the
363 // 4th argument to pread() to tell that we
364 // want to start reading from the beginning
365 // of the file, without changing the position
366 // inside the file:
367 number_of_chars_read_in = pread (file_descriptor,
368                                  buf,
369                                  BUF_SIZE - 1,
370                                  0);
371 // Checking for errors:
372 if (number_of_chars_read_in == -1)
373 {
374     perror ("pread");
375     exit (EXIT_FAILURE);
376 }
377
378 buf[number_of_chars_read_in] = '\0';
379
380 // Let's print what we read to the screen:
381 if (write (STDOUT_FILENO,
382           message4,
383           strlen (message4))
384     == -1)
385 {
386     perror ("write");
```



```
387     exit (EXIT_FAILURE);
388 }
389 if (write (STDOUT_FILENO,
390         buf,
391         number_of_chars_read_in)
392     == -1)
393 {
394     perror ("write");
395     exit (EXIT_FAILURE);
396 }
397 if (write (STDOUT_FILENO, "\n", 1) == -1)
398 {
399     perror ("write");
400     exit (EXIT_FAILURE);
401 }
402
403 if (write (STDOUT_FILENO,
404         "In the last reading, we "
405         "actually read a total of ",
406         strlen ("In the last reading, we "
407             "actually read a total of "))
408     == -1)
409 {
410     perror ("write");
411     exit (EXIT_FAILURE);
412 }
413
414 // Copy the integer value in
415 //     'number_of_chars_read_in' to the 'buf'
416 //     char array:
417 sprintf(buf, "%zu", number_of_chars_read_in);
418
419 if (write (STDOUT_FILENO, buf, strlen(buf)) == -1)
420 {
421     perror ("write");
422     exit (EXIT_FAILURE);
423 }
424 if (write (STDOUT_FILENO,
425         " bytes.\n",
426         strlen(" bytes.\n"))
427     == -1)
428 {
429     perror ("write");
```



```
430     exit (EXIT_FAILURE);
431 }
432
433 // Finally, close the file:
434 if (close (file_descriptor) == -1)
435 {
436     perror ("close");
437     exit (EXIT_FAILURE);
438 }
439
440 if (write (STDOUT_FILENO,
441           "The space between "
442           "locations 10 and 20 is "
443           "actually padded with "
444           "null characters.\n",
445           80)
446     == -1)
447 {
448     perror ("write");
449     exit (EXIT_FAILURE);
450 }
451
452 if (write (STDOUT_FILENO,
453           "That is, on disk, the file looks "
454           "as follows:\nHelloFGHIJ\\0\\0\\0"
455           "\\0\\0\\0\\0\\0\\0\\0\\0\\0Wow\n",
456           80)
457     == -1)
458 {
459     perror ("write");
460     exit (EXIT_FAILURE);
461 }
462
463 return EXIT_SUCCESS;
464 }
465
```




```
1 // The following program exemplifies a call to the
2 // select() system call. We wait (block) only
3 // on one file: the keyboard (fd = 0 = STDIN_FILENO)
4 // for 5 seconds. If the user types some content and
5 // presses ENTER before the 5 seconds end, the
6 // content will be output to the terminal. Otherwise,
7 // the 5 seconds will elapse, so the system call will
8 // return without gathering input from the user.
9 //
10 // This program is taken from Linux System Programming:
11 // Talking Directly to the Kernel and C Library,
12 // 2nd Edition, by Love. ISBN: 978-1-44933953-1,
13 // pages 55-56.
14
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <sys/time.h>
18 #include <sys/types.h>
19 #include <unistd.h>
20
21 #define TIMEOUT 5 /* select timeout in seconds */
22 #define BUF_LEN 1024 /* read buffer in bytes */
23
24 int main (void)
25 {
26     struct timeval tv;
27     fd_set readfds;
28     int ret;
29
30     /* Wait on stdin for input. */
31     FD_ZERO(&readfds);
32     FD_SET(STDIN_FILENO, &readfds);
33
34     /* Wait up to five seconds. */
35     tv.tv_sec = TIMEOUT;
36     tv.tv_usec = 0;
37
38     /* All right, now block! */
39     ret = select (STDIN_FILENO + 1,
40                 &readfds,
41                 NULL,
42                 NULL,
```



```
43         &tv);
44
45     if (ret == -1)
46     {
47         perror ("select");
48         exit (EXIT_FAILURE);
49     }
50     else if (!ret)
51     {
52         printf ("%d seconds elapsed.\n", TIMEOUT);
53         return EXIT_SUCCESS;
54     }
55
56     /*
57     * Is our file descriptor ready to read?
58     * (It must be, as it was the only fd that
59     * we provided and the call returned
60     * nonzero, but we will humor ourselves.)
61     */
62     if (FD_ISSET(STDIN_FILENO, &readfds)) {
63         char buf[BUF_LEN+1];
64         int len;
65
66         /* guaranteed to not block */
67         len = read (STDIN_FILENO, buf, BUF_LEN);
68         if (len == -1)
69         {
70             perror ("read");
71             exit (EXIT_FAILURE);
72         }
73
74         if (len) // Same as if (len != 0)
75         {
76             buf[len] = '\0';
77             printf ("read: %s\n", buf);
78         }
79
80         return EXIT_SUCCESS;
81     }
82
83     fprintf (stderr, "This should not happen!\n");
84     return EXIT_FAILURE;
```



85 | }
86 |

```
1 // The following program exemplifies a call to the
2 // poll() system call. We wait (block) on 2 files:
3 // the keyboard (fd = 0 = STDIN_FILENO) and the
4 // screen (fd = 1 = STDOUT_FILENO) for 5 seconds.
5 // We expect the screen to be responsive right
6 // away, so we'll see the message: "stdout is
7 // writable". Since poll() will return right
8 // away, the program will soon thereafter end.
9 // As such, to check also for the keyboard,
10 // create some file (say, a .txt file) and
11 // call the poll_syscall_example program this
12 // way: ./poll_syscall_example < text.txt
13 // where text.txt is the file you created. Then,
14 // the following additional message will be
15 // output: "stdin is readable" besides "stdout
16 // is writable".
17 //
18 // This program is taken from Linux System Programming:
19 // Talking Directly to the Kernel and C Library,
20 // 2nd Edition, by Love. ISBN: 978-1-44933953-1,
21 // pages 60-61.
22
23 #include <stdio.h>
24 #include <stdlib.h>
25 #include <unistd.h>
26 #include <poll.h>
27
28 #define TIMEOUT 5 /* poll timeout, in seconds */
29
30 int main (void)
31 {
32     struct pollfd fds[2];
33     int ret;
34
35     /* watch stdin for input */
36     fds[0].fd = STDIN_FILENO;
37     fds[0].events = POLLIN;
38
39     /* watch stdout for ability to write (almost always true) */
40     fds[1].fd = STDOUT_FILENO;
41     fds[1].events = POLLOUT;
42
```



```
43  /* All set, block! */
44  ret = poll (fds, 2, TIMEOUT * 1000);
45
46  if (ret == -1)
47  {
48      perror ("poll");
49      exit (EXIT_FAILURE);
50  }
51
52  if (!ret) // Same as saying: if (ret == 0)
53  {
54      printf ("%d seconds elapsed.\n", TIMEOUT);
55      return EXIT_SUCCESS;
56  }
57
58  if (fds[0].revents & POLLIN)
59      printf ("stdin is readable\n");
60
61  if (fds[1].revents & POLLOUT)
62      printf ("stdout is writable\n");
63
64  return EXIT_SUCCESS;
65 }
66
```



Topic 5: Buffered File I/O

Note: A copy of each one of the sources listed below is included in this packet. To jump to the location of a source copy, click the page link under “Page”.

In case a source isn’t included (e.g., a YouTube video,) click the [↗](#) (external link) symbol under “Page” to open the external source in a browser window.

| # | Citation & Source Link | Page |
|----|--|------|
| 1 | “time(1) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man1/time.1.html | 584 |
| 2 | Rubin, Paul, et al. “dd(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/dd.1.html | 590 |
| 3 | Meskes, Michael. “stat(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/stat.1.html | 595 |
| 4 | “stdio(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/stdio.3.html | 601 |
| 5 | “fopen(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/fopen.3.html | 606 |
| 6 | “fdopen(3p) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/fdopen.3p.html | 613 |
| 7 | “fclose(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/fclose.3.html | 618 |
| 8 | “fcloseall(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/fcloseall.3.html | 621 |
| 9 | “fgetc(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/fgetc.3.html | 623 |
| 10 | “ungetc(3p) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/ungetc.3p.html | 626 |
| 11 | “fgets(3p) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/fgets.3p.html | 630 |
| 12 | “fread(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/fread.3.html | 634 |
| 13 | “ferror(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/ferror.3.html | 638 |
| 14 | “fputc(3p) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/fputc.3p.html | 641 |
| 15 | “fputs(3p) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/fputs.3p.html | 645 |
| 16 | “fseek(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/fseek.3.html | 649 |

| # | Citation & Source Link | Page |
|----|---|------|
| 17 | “ftell(3p) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/ftell.3p.html | 653 |
| 18 | “fsetpos(3p) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/fsetpos.3p.html | 657 |
| 19 | “rewind(3p) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/rewind.3p.html | 662 |
| 20 | “fflush(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/fflush.3.html | 665 |
| 21 | “fileno(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/fileno.3.html | 668 |
| 22 | “setvbuf(3p) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/setvbuf.3p.html | 671 |
| 23 | “flockfile(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/flockfile.3.html | 675 |
| 24 | Briskman, Miriam. “Materials for Topic 5: Buffered File I/O.” <i>Topic 5: Buffered File I/O — CISC 3350 Materials</i> , 2023. URL: www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_05/ | 679 |
| 25 | Briskman, Miriam. “files.c .” (C source code) 2 Feb. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_03/files.c | 435 |
| 26 | Briskman, Miriam. “buffered_io.c .” (C source code) 8 Mar. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_05/buffered_io.c | 682 |

time(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [EXIT STATUS](#) | [ENVIRONMENT](#) | [GNU VERSION](#) | [BUGS](#) | [SEE ALSO](#)

 time(1)

General Commands Manual

time(1)

NAME [top](#)

`time` - time a simple command or give resource usage

SYNOPSIS [top](#)

`time` [*option* ...] *command* [*argument* ...]

DESCRIPTION [top](#)

The `time` command runs the specified program *command* with the given arguments. When *command* finishes, `time` writes a message to standard error giving timing statistics about this program run. These statistics consist of (i) the elapsed real time between invocation and termination, (ii) the user CPU time (the sum of the *tms_utime* and *tms_cutime* values in a *struct tms* as returned by [times\(2\)](#)), and (iii) the system CPU time (the sum of the *tms_stime* and *tms_cstime* values in a *struct tms* as returned by [times\(2\)](#)).

Note: some shells (e.g., [bash\(1\)](#)) have a built-in `time` command that provides similar information on the usage of time and possibly other resources. To access the real command, you may need to specify its pathname (something like */usr/bin/time*).

OPTIONS [top](#)

-p When in the POSIX locale, use the precise traditional format

```
"real %f\nuser %f\nsys %f\n"
```

(with numbers in seconds) where the number of decimals in the output for %f is unspecified but is sufficient to express the clock tick accuracy, and at least one.

EXIT STATUS [top](#)

If *command* was invoked, the exit status is that of *command*. Otherwise, it is 127 if *command* could not be found, 126 if it could be found but could not be invoked, and some other nonzero value (1-125) if something else went wrong.

ENVIRONMENT [top](#)

The variables **LANG**, **LC_ALL**, **LC_CTYPE**, **LC_MESSAGES**, **LC_NUMERIC**, and **NLSPATH** are used for the text and formatting of the output. **PATH** is used to search for *command*.

GNU VERSION [top](#)

Below a description of the GNU 1.7 version of **time**. Disregarding the name of the utility, GNU makes it output lots of useful information, not only about time used, but also on other resources like memory, I/O and IPC calls (where available). The output is formatted using a format string that can be specified using the *-f* option or the **TIME** environment variable.

The default format string is:

```
%User %Ssystem %Eelapsed %PCPU (%Xtext+%Ddata %Mmax)k  
%Iinputs+%Ooutputs (%Fmajor+%Rminor)pagefaults %Wswaps
```

When the *-p* option is given, the (portable) output format is used:

```
real %e  
user %U
```

sys %S

The format string

The format is interpreted in the usual printf-like way. Ordinary characters are directly copied, tab, newline, and backslash are escaped using `\t`, `\n`, and `\\`, a percent sign is represented by `%%`, and otherwise `%` indicates a conversion. The program **time** will always add a trailing newline itself. The conversions follow. All of those used by **tcsh(1)** are supported.

Time

- %E** Elapsed real time (in [hours:]minutes:seconds).
- %e** (Not in **tcsh(1)**.) Elapsed real time (in seconds).
- %S** Total number of CPU-seconds that the process spent in kernel mode.
- %U** Total number of CPU-seconds that the process spent in user mode.
- %P** Percentage of the CPU that this job got, computed as $(\%U + \%S) / \%E$.

Memory

- %M** Maximum resident set size of the process during its lifetime, in Kbytes.
- %t** (Not in **tcsh(1)**.) Average resident set size of the process, in Kbytes.
- %K** Average total (data+stack+text) memory use of the process, in Kbytes.
- %D** Average size of the process's unshared data area, in Kbytes.
- %p** (Not in **tcsh(1)**.) Average size of the process's unshared stack space, in Kbytes.
- %X** Average size of the process's shared text space, in Kbytes.



- %Z** (Not in **tcsh**(1).) System's page size, in bytes. This is a per-system constant, but varies between systems.
- %F** Number of major page faults that occurred while the process was running. These are faults where the page has to be read in from disk.
- %R** Number of minor, or recoverable, page faults. These are faults for pages that are not valid but which have not yet been claimed by other virtual pages. Thus the data in the page is still valid but the system tables must be updated.
- %W** Number of times the process was swapped out of main memory.
- %c** Number of times the process was context-switched involuntarily (because the time slice expired).
- %w** Number of waits: times that the program was context-switched voluntarily, for instance while waiting for an I/O operation to complete.
- I/O**
- %I** Number of filesystem inputs by the process.
- %O** Number of filesystem outputs by the process.
- %r** Number of socket messages received by the process.
- %s** Number of socket messages sent by the process.
- %k** Number of signals delivered to the process.
- %C** (Not in **tcsh**(1).) Name and command-line arguments of the command being timed.
- %x** (Not in **tcsh**(1).) Exit status of the command.

GNU options

-f *format*, **--format=***format*

Specify output format, possibly overriding the format specified in the environment variable TIME.



-p, --portability

Use the portable output format.

-o *file*, --output=*file*

Do not send the results to *stderr*, but overwrite the specified file.

-a, --append

(Used together with -o.) Do not overwrite but append.

-v, --verbose

Give very verbose output about all the program knows about.

-q, --quiet

Don't report abnormal program termination (where *command* is terminated by a signal) or nonzero exit status.

GNU standard options

--help Print a usage message on standard output and exit successfully.

-V, --version

Print version information on standard output, then exit successfully.

-- Terminate option list.

BUGS [top](#)

Not all resources are measured by all versions of UNIX, so some of the values might be reported as zero. The present selection was mostly inspired by the data provided by 4.2 or 4.3BSD.

GNU time version 1.7 is not yet localized. Thus, it does not implement the POSIX requirements.

The environment variable **TIME** was badly chosen. It is not unusual for systems like **autoconf(1)** or **make(1)** to use environment variables with the name of a utility to override the utility to be used. Uses like MORE or TIME for options to programs (instead of program pathnames) tend to lead to

difficulties.

It seems unfortunate that `-o` overwrites instead of appends. (That is, the `-a` option should be the default.)

Mail suggestions and bug reports for GNU **time** to bug-time@gnu.org. Please include the version of **time**, which you can get by running

```
time --version
```

and the operating system and C compiler you used.

SEE ALSO [top](#)

[bash\(1\)](#), [tcsh\(1\)](#), [times\(2\)](#), [wait3\(2\)](#)

Linux man-pages (unreleased) (date)

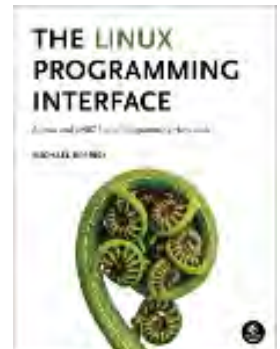
[time\(1\)](#)

Pages that refer to this page: [strace\(1\)](#), [times\(2\)](#), [time\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



dd(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [AUTHOR](#) | [REPORTING BUGS](#) | [COPYRIGHT](#) | [SEE ALSO](#) | [COLOPHON](#)

 DD(1)

User Commands

DD(1)**NAME** [top](#)

dd - convert and copy a file

SYNOPSIS [top](#)

```
dd [OPERAND]...  
dd OPTION
```

DESCRIPTION [top](#)

Copy a file, converting and formatting according to the operands.

bs=BYTES

read and write up to BYTES bytes at a time (default: 512);
overrides ibs and obs

cbs=BYTES

convert BYTES bytes at a time

conv=CONVS

convert the file as per the comma separated symbol list

count=N

copy only N input blocks

ibs=BYTES

read up to BYTES bytes at a time (default: 512)

if=FILE

read from FILE instead of stdin

iflag=FLAGS

read as per the comma separated symbol list

obs=BYTES

write BYTES bytes at a time (default: 512)

of=FILE

write to FILE instead of stdout

oflag=FLAGS

write as per the comma separated symbol list

seek=N (or oseek=N) skip N obs-sized output blocks

skip=N (or iseek=N) skip N ibs-sized input blocks

status=LEVEL

The LEVEL of information to print to stderr; 'none' suppresses everything but error messages, 'noxfer' suppresses the final transfer statistics, 'progress' shows periodic transfer statistics

N and BYTES may be followed by the following multiplicative suffixes: c=1, w=2, b=512, kB=1000, K=1024, MB=1000*1000, M=1024*1024, xM=M, GB=1000*1000*1000, G=1024*1024*1024, and so on for T, P, E, Z, Y, R, Q. Binary prefixes can be used, too: KiB=K, MiB=M, and so on. If N ends in 'B', it counts bytes not blocks.

Each CONV symbol may be:

ascii from EBCDIC to ASCII

ebcdic from ASCII to EBCDIC

ibm from ASCII to alternate EBCDIC

block pad newline-terminated records with spaces to cbs-size



unblock replace trailing spaces in cbs-size records with newline

lcase change upper case to lower case

ucase change lower case to upper case

sparse try to seek rather than write all-NUL output blocks

swab swap every pair of input bytes

sync pad every input block with NULs to ibs-size; when used with block or unblock, pad with spaces rather than NULs

excl fail if the output file already exists

nocreat
do not create the output file

notrunc
do not truncate the output file

noerror
continue after read errors

fdatsync
physically write output file data before finishing

fsync likewise, but also write metadata

Each FLAG symbol may be:

append append mode (makes sense only for output; conv=notrunc suggested)

direct use direct I/O for data

directory
fail unless a directory

dsync use synchronized I/O for data

sync likewise, but also for metadata



fullblock
accumulate full blocks of input (iflag only)

nonblock
use non-blocking I/O

noatime
do not update access time

nocache
Request to drop cache. See also oflag=sync

noctty do not assign controlling terminal from file

nofollow
do not follow symlinks

Sending a USR1 signal to a running 'dd' process makes it print I/O statistics to standard error and then resume copying.

Options are:

--help display this help and exit

--version
output version information and exit

AUTHOR [top](#)

Written by Paul Rubin, David MacKenzie, and Stuart Kemp.

REPORTING BUGS [top](#)

GNU coreutils online help:
<<https://www.gnu.org/software/coreutils/>>
Report any translation bugs to
<<https://translationproject.org/team/>>

COPYRIGHT [top](#)

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>. This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO [top](#)

Full documentation <<https://www.gnu.org/software/coreutils/dd>> or available locally via: info '(coreutils) dd invocation'

COLOPHON [top](#)

This page is part of the *coreutils* (basic file, shell and text manipulation utilities) project. Information about the project can be found at <<http://www.gnu.org/software/coreutils/>>. If you have a bug report for this manual page, see <<http://www.gnu.org/software/coreutils/>>. This page was obtained from the tarball *coreutils-9.4.tar.xz* fetched from <<http://ftp.gnu.org/gnu/coreutils/>> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

GNU coreutils 9.4

August 2023

DD(1)

Pages that refer to this page: [pipesz\(1\)](#), [truncate\(1\)](#), [xfs\(5\)](#), [fdisk\(8\)](#), [sfdisk\(8\)](#), [swapon\(8\)](#), [xfs_copy\(8\)](#), [xfs_repair\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



stat(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [AUTHOR](#) | [REPORTING BUGS](#) | [COPYRIGHT](#) | [SEE ALSO](#) | [COLOPHON](#)

 STAT(1)

User Commands

STAT(1)**NAME** [top](#)

stat - display file or file system status

SYNOPSIS [top](#)

stat [*OPTION*]... *FILE*...

DESCRIPTION [top](#)

Display file or file system status.

Mandatory arguments to long options are mandatory for short options too.

-L, --dereference
follow links

-f, --file-system
display file system status instead of file status

--cached=MODE
specify how to use cached attributes; useful on remote file systems. See MODE below

-c --format=FORMAT
use the specified FORMAT instead of the default; output a



newline after each use of `FORMAT`

--printf=FORMAT

like **--format**, but interpret backslash escapes, and do not output a mandatory trailing newline; if you want a newline, include `\n` in `FORMAT`

-t, --terse

print the information in terse form

--help display this help and exit

--version

output version information and exit

The `MODE` argument of **--cached** can be: `always`, `never`, or `default`. `'always'` will use cached attributes if available, while `'never'` will try to synchronize with the latest attributes, and `'default'` will leave it up to the underlying file system.

The valid format sequences for files (without **--file-system**):

| | |
|------------------|---|
| <code>%a</code> | permission bits in octal (note '#' and '0' printf flags) |
| <code>%A</code> | permission bits and file type in human readable form |
| <code>%b</code> | number of blocks allocated (see <code>%B</code>) |
| <code>%B</code> | the size in bytes of each block reported by <code>%b</code> |
| <code>%C</code> | SELinux security context string |
| <code>%d</code> | device number in decimal (<code>st_dev</code>) |
| <code>%D</code> | device number in hex (<code>st_dev</code>) |
| <code>%Hd</code> | major device number in decimal |
| <code>%Ld</code> | minor device number in decimal |
| <code>%f</code> | raw mode in hex |
| <code>%F</code> | file type |

| | |
|-----|--|
| %g | group ID of owner |
| %G | group name of owner |
| %h | number of hard links |
| %i | inode number |
| %m | mount point |
| %n | file name |
| %N | quoted file name with dereference if symbolic link |
| %o | optimal I/O transfer size hint |
| %s | total size, in bytes |
| %r | device type in decimal (st_rdev) |
| %R | device type in hex (st_rdev) |
| %Hr | major device type in decimal, for character/block device special files |
| %Lr | minor device type in decimal, for character/block device special files |
| %t | major device type in hex, for character/block device special files |
| %T | minor device type in hex, for character/block device special files |
| %u | user ID of owner |
| %U | user name of owner |
| %w | time of file birth, human-readable; - if unknown |
| %W | time of file birth, seconds since Epoch; 0 if unknown |
| %x | time of last access, human-readable |



%X time of last access, seconds since Epoch
%y time of last data modification, human-readable
%Y time of last data modification, seconds since Epoch
%z time of last status change, human-readable
%Z time of last status change, seconds since Epoch

Valid format sequences for file systems:

%a free blocks available to non-superuser
%b total data blocks in file system
%c total file nodes in file system
%d free file nodes in file system
%f free blocks in file system
%i file system ID in hex
%l maximum length of filenames
%n file name
%s block size (for faster transfers)
%S fundamental block size (for block counts)
%t file system type in hex
%T file system type in human readable form

--terse is equivalent to the following FORMAT:

%n %s %b %f %u %g %D %i %h %t %T %X %Y %Z %W %o %C

--terse --file-system is equivalent to the following FORMAT:

%n %i %l %t %s %S %b %f %a %c %d

NOTE: your shell may have its own version of stat, which usually



supersedes the version described here. Please refer to your shell's documentation for details about the options it supports.

AUTHOR [top](#)

Written by Michael Meskes.

REPORTING BUGS [top](#)

GNU coreutils online help:
<<https://www.gnu.org/software/coreutils/>>
Report any translation bugs to
<<https://translationproject.org/team/>>

COPYRIGHT [top](#)

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>. This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO [top](#)

[stat\(2\)](#), [statfs\(2\)](#), [statx\(2\)](#)

Full documentation <<https://www.gnu.org/software/coreutils/stat>> or available locally via: info '(coreutils) stat invocation'

COLOPHON [top](#)

This page is part of the *coreutils* (basic file, shell and text manipulation utilities) project. Information about the project can be found at <<http://www.gnu.org/software/coreutils/>>. If you have a bug report for this manual page, see <<http://www.gnu.org/software/coreutils/>>. This page was obtained from the tarball coreutils-9.4.tar.xz fetched from <<http://ftp.gnu.org/gnu/coreutils/>> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for



the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

GNU coreutils 9.4

August 2023

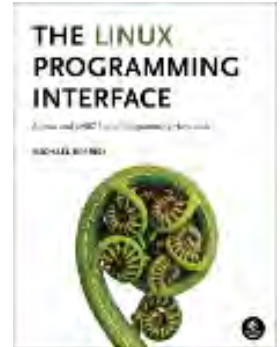
STAT(1)

Pages that refer to this page: [namei\(1\)](#), [stat\(2\)](#), [statx\(2\)](#), [inode\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



stdio(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 stdio(3)

Library Functions Manual

stdio(3)**NAME** [top](#)

stdio - standard input/output library functions

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
FILE *stdin;  
FILE *stdout;  
FILE *stderr;
```

DESCRIPTION [top](#)

The standard I/O library provides a simple and efficient buffered stream I/O interface. Input and output is mapped into logical data streams and the physical I/O characteristics are concealed. The functions and macros are listed below; more information is available from the individual man pages.

A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve creating a new file. Creating an existing file causes its former contents

to be discarded. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a *file position indicator* associated with the stream is positioned at the start of the file (byte zero), unless the file is opened with append mode. If append mode is used, it is unspecified whether the position indicator will be placed at the start or the end of the file. The position indicator is maintained by subsequent reads, writes, and positioning requests. All input occurs as if the characters were read by successive calls to the `fgetc(3)` function; all output takes place as if all characters were written by successive calls to the `fputc(3)` function.

A file is disassociated from a stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are transferred to the host environment) before the stream is disassociated from the file. The value of a pointer to a *FILE* object is indeterminate after a file is closed (garbage).

A file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at the start). If the main function returns to its original caller, or the `exit(3)` function is called, all open files are closed (hence all output streams are flushed) before program termination. Other methods of program termination, such as `abort(3)` do not bother about closing files properly.

At program startup, three text streams are predefined and need not be opened explicitly: *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). These streams are abbreviated *stdin*, *stdout*, and *stderr*. When opened, the standard error stream is not fully buffered; the standard input and output streams are fully buffered if and only if the streams do not refer to an interactive device.

Output streams that refer to terminal devices are always line buffered by default; pending output to such streams is written automatically whenever an input stream that refers to a terminal device is read. In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to `fflush(3)` the standard output before going off and computing so that the output will appear.

The *stdio* library is a part of the library **libc** and routines are



automatically loaded as needed by `cc(1)`. The SYNOPSIS sections of the following manual pages indicate which include files are to be used, what the compiler declaration for the function looks like and which external variables are of interest.

The following are defined as macros; these names may not be reused without first removing their current definitions with `#undef`: `BUFSIZ`, `EOF`, `FILENAME_MAX`, `FOPEN_MAX`, `L_cuserid`, `L_ctermid`, `L_tmpnam`, `NULL`, `SEEK_END`, `SEEK_SET`, `SEEK_CUR`, `TMP_MAX`, `clearerr`, `feof`, `ferror`, `fileno`, `getc`, `getchar`, `putc`, `putchar`, `stderr`, `stdin`, `stdout`. Function versions of the macro functions `feof`, `ferror`, `clearerr`, `fileno`, `getc`, `getchar`, `putc`, and `putchar` exist and will be used if the macros definitions are explicitly removed.

List of functions

| Function | Description |
|-----------------------------|--|
| clearerr(3) | check and reset stream status |
| fclose(3) | close a stream |
| fdopen(3) | stream open functions |
| feof(3) | check and reset stream status |
| ferror(3) | check and reset stream status |
| fflush(3) | flush a stream |
| fgetc(3) | get next character or word from input stream |
| fgetpos(3) | reposition a stream |
| fgets(3) | get a line from a stream |
| fileno(3) | return the integer descriptor of the argument stream |
| fopen(3) | stream open functions |
| fprintf(3) | formatted output conversion |
| fpurge(3) | flush a stream |
| fputc(3) | output a character or word to a stream |
| fputs(3) | output a line to a stream |
| fread(3) | binary stream input/output |
| freopen(3) | stream open functions |
| fscanf(3) | input format conversion |
| fseek(3) | reposition a stream |
| fsetpos(3) | reposition a stream |
| ftell(3) | reposition a stream |
| fwrite(3) | binary stream input/output |
| getc(3) | get next character or word from input stream |
| getchar(3) | get next character or word from input stream |
| gets(3) | get a line from a stream |
| getw(3) | get next character or word from input stream |

| | |
|--------------------------------|--|
| mktemp(3) | make temporary filename (unique) |
| perror(3) | system error messages |
| printf(3) | formatted output conversion |
| putc(3) | output a character or word to a stream |
| putchar(3) | output a character or word to a stream |
| puts(3) | output a line to a stream |
| putw(3) | output a character or word to a stream |
| remove(3) | remove directory entry |
| rewind(3) | reposition a stream |
| scanf(3) | input format conversion |
| setbuf(3) | stream buffering operations |
| setbuffer(3) | stream buffering operations |
| setlinebuf(3) | stream buffering operations |
| setvbuf(3) | stream buffering operations |
| sprintf(3) | formatted output conversion |
| sscanf(3) | input format conversion |
| strerror(3) | system error messages |
| sys_errlist(3) | system error messages |
| sys_nerr(3) | system error messages |
| tempnam(3) | temporary file routines |
| tmpfile(3) | temporary file routines |
| tmpnam(3) | temporary file routines |
| ungetc(3) | un-get character from input stream |
| vfprintf(3) | formatted output conversion |
| vscanf(3) | input format conversion |
| vprintf(3) | formatted output conversion |
| vscanf(3) | input format conversion |
| vsprintf(3) | formatted output conversion |
| vsscanf(3) | input format conversion |

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

C89, POSIX.1-2001.

SEE ALSO [top](#)

[close\(2\)](#), [open\(2\)](#), [read\(2\)](#), [write\(2\)](#), [stdout\(3\)](#),
[unlocked_stdio\(3\)](#)

Linux man-pages (unreleased) (date)

stdio(3)

Pages that refer to this page: [pmsnap\(1\)](#), [_exit\(2\)](#), [fcntl\(2\)](#), [vfork\(2\)](#), [curs_addch\(3x\)](#), [exit\(3\)](#), [ferror\(3\)](#), [FILE\(3type\)](#), [fileno\(3\)](#), [popen\(3\)](#), [printf\(3\)](#), [stdin\(3\)](#), [unlocked_stdio\(3\)](#), [scr_dump\(5\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fopen(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

 fopen(3)

Library Functions Manual

fopen(3)**NAME** [top](#)

fopen, fdopen, freopen - stream open functions

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
FILE *fopen(const char *restrict pathname, const char *restrict mode);  
FILE *fdopen(int fd, const char *mode);  
FILE *freopen(const char *restrict pathname, const char *restrict mode,  
              FILE *restrict stream);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
fdopen():  
    _POSIX_C_SOURCE
```

DESCRIPTION [top](#)

The **fopen()** function opens the file whose name is the string pointed to by *pathname* and associates a stream with it.

The argument *mode* points to a string beginning with one of the following sequences (possibly followed by additional characters,

as described below):

- r** Open text file for reading. The stream is positioned at the beginning of the file.
- r+** Open for reading and writing. The stream is positioned at the beginning of the file.
- w** Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
- w+** Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
- a** Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
- a+** Open for reading and appending (writing at end of file). The file is created if it does not exist. Output is always appended to the end of the file. POSIX is silent on what the initial read position is when using this mode. For glibc, the initial file position for reading is at the beginning of the file, but for Android/BSD/MacOS, the initial file position for reading is at the end of the file.

The *mode* string can also include the letter 'b' either as a last character or as a character between the characters in any of the two-character strings described above. This is strictly for compatibility with ISO C and has no effect; the 'b' is ignored on all POSIX conforming systems, including Linux. (Other systems may treat text files and binary files differently, and adding the 'b' may be a good idea if you do I/O to a binary file and expect that your program may be ported to non-UNIX environments.)

See NOTES below for details of glibc extensions for *mode*.

Any created file will have the mode **S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH** (0666), as modified by the process's `umask` value (see `umask(2)`).

Reads and writes may be intermixed on read/write streams in any order. Note that ANSI C requires that a file positioning function intervene between output and input, unless an input operation encounters end-of-file. (If this condition is not met,



then a read is allowed to return the result of writes other than the most recent.) Therefore it is good practice (and indeed sometimes necessary under Linux) to put an `fseek(3)` or `fsetpos(3)` operation between write and read operations on such a stream. This operation may be an apparent no-op (as in `fseek(..., 0L, SEEK_CUR)` called for its synchronizing side effect).

Opening a file in append mode (`a` as the first character of `mode`) causes all subsequent write operations to this stream to occur at end-of-file, as if preceded by the call:

```
fseek(stream, 0, SEEK_END);
```

The file descriptor associated with the stream is opened as if by a call to `open(2)` with the following flags:

| <code>fopen()</code> mode | <code>open()</code> flags |
|---------------------------|--|
| <code>r</code> | <code>O_RDONLY</code> |
| <code>w</code> | <code>O_WRONLY</code> <code>O_CREAT</code> <code>O_TRUNC</code> |
| <code>a</code> | <code>O_WRONLY</code> <code>O_CREAT</code> <code>O_APPEND</code> |
| <code>r+</code> | <code>O_RDWR</code> |
| <code>w+</code> | <code>O_RDWR</code> <code>O_CREAT</code> <code>O_TRUNC</code> |
| <code>a+</code> | <code>O_RDWR</code> <code>O_CREAT</code> <code>O_APPEND</code> |

`fdopen()`

The `fdopen()` function associates a stream with the existing file descriptor, `fd`. The `mode` of the stream (one of the values "r", "r+", "w", "w+", "a", "a+") must be compatible with the mode of the file descriptor. The file position indicator of the new stream is set to that belonging to `fd`, and the error and end-of-file indicators are cleared. Modes "w" or "w+" do not cause truncation of the file. The file descriptor is not dup'ed, and will be closed when the stream created by `fdopen()` is closed. The result of applying `fdopen()` to a shared memory object is undefined.

`freopen()`

The `freopen()` function opens the file whose name is the string pointed to by `pathname` and associates the stream pointed to by `stream` with it. The original stream (if it exists) is closed. The `mode` argument is used just as in the `fopen()` function.

If the *pathname* argument is a null pointer, **freopen()** changes the mode of the stream to that specified in *mode*; that is, **freopen()** reopens the pathname that is associated with the stream. The specification for this behavior was added in the C99 standard, which says:

In this case, the file descriptor associated with the stream need not be closed if the call to **freopen()** succeeds. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

The primary use of the **freopen()** function is to change the file associated with a standard text stream (*stderr*, *stdin*, or *stdout*).

RETURN VALUE [top](#)

Upon successful completion **fopen()**, **fdopen()**, and **freopen()** return a *FILE* pointer. Otherwise, NULL is returned and *errno* is set to indicate the error.

ERRORS [top](#)

EINVAL The *mode* provided to **fopen()**, **fdopen()**, or **freopen()** was invalid.

The **fopen()**, **fdopen()**, and **freopen()** functions may also fail and set *errno* for any of the errors specified for the routine [malloc\(3\)](#).

The **fopen()** function may also fail and set *errno* for any of the errors specified for the routine [open\(2\)](#).

The **fdopen()** function may also fail and set *errno* for any of the errors specified for the routine [fcntl\(2\)](#).

The **freopen()** function may also fail and set *errno* for any of the errors specified for the routines [open\(2\)](#), [fclose\(3\)](#), and [fflush\(3\)](#).

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|---|---------------|---------|
| fopen() , fdopen() , freopen() | Thread safety | MT-Safe |

STANDARDS [top](#)

fopen()
freopen()
 C11, POSIX.1-2008.

fdopen()
 POSIX.1-2008.

HISTORY [top](#)

fopen()
freopen()
 POSIX.1-2001, C89.

fdopen()
 POSIX.1-2001.

NOTES [top](#)

glibc notes

The GNU C library allows the following extensions for the string specified in *mode*:

- c** (since glibc 2.3.3)
 Do not make the open operation, or subsequent read and write operations, thread cancelation points. This flag is ignored for **fdopen()**.
- e** (since glibc 2.7)
 Open the file with the **O_CLOEXEC** flag. See [open\(2\)](#) for more information. This flag is ignored for **fdopen()**.
- m** (since glibc 2.3)
 Attempt to access the file using [mmap\(2\)](#), rather than I/O system calls ([read\(2\)](#), [write\(2\)](#)). Currently, use of [mmap\(2\)](#) is attempted only for a file opened for reading.
- x** Open the file exclusively (like the **O_EXCL** flag of

`open(2)`). If the file already exists, `fopen()` fails, and sets `errno` to `EEXIST`. This flag is ignored for `fdopen()`.

In addition to the above characters, `fopen()` and `freopen()` support the following syntax in `mode`:

`,ccs=string`

The given `string` is taken as the name of a coded character set and the stream is marked as wide-oriented. Thereafter, internal conversion functions convert I/O to and from the character set `string`. If the `,ccs=string` syntax is not specified, then the wide-orientation of the stream is determined by the first file operation. If that operation is a wide-character operation, the stream is marked wide-oriented, and functions to convert to the coded character set are loaded.

BUGS [top](#)

When parsing for individual flag characters in `mode` (i.e., the characters preceding the "ccs" specification), the glibc implementation of `fopen()` and `freopen()` limits the number of characters examined in `mode` to 7 (or, before glibc 2.14, to 6, which was not enough to include possible specifications such as "rb+cmxe"). The current implementation of `fdopen()` parses at most 5 characters in `mode`.

SEE ALSO [top](#)

[open\(2\)](#), [fclose\(3\)](#), [fileno\(3\)](#), [fmemopen\(3\)](#), [fopencookie\(3\)](#), [open_memstream\(3\)](#)

Linux man-pages (unreleased) (date) [fopen\(3\)](#)

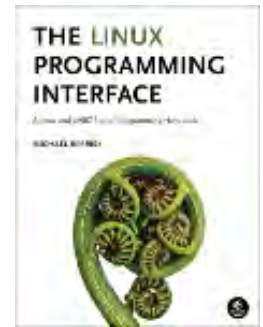
Pages that refer to this page: [open\(2\)](#), [fclose\(3\)](#), [fcloseall\(3\)](#), [ferror\(3\)](#), [fflush\(3\)](#), [fgetc\(3\)](#), [fgetgrent\(3\)](#), [fgetpwent\(3\)](#), [fgetwc\(3\)](#), [fgetws\(3\)](#), [FILE\(3type\)](#), [fileno\(3\)](#), [fmemopen\(3\)](#), [fopencookie\(3\)](#), [fputwc\(3\)](#), [fputws\(3\)](#), [getline\(3\)](#), [getmntent\(3\)](#), [gets\(3\)](#), [libexpect\(3\)](#), [open_memstream\(3\)](#), [popenlog\(3\)](#), [popen\(3\)](#), [procio\(3\)](#), [pthread_getattr_np\(3\)](#), [puts\(3\)](#), [setbuf\(3\)](#), [stdin\(3\)](#), [stdio\(3\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fdopen(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 FDOPEN(3P)

POSIX Programmer's Manual

FDOPEN(3P)**PROLOG** [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

`fdopen` – associate a stream with a file descriptor

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
FILE *fdopen(int fil-des, const char *mode);
```

DESCRIPTION [top](#)

The `fdopen()` function shall associate a stream with a file descriptor.

The `mode` argument is a character string having one of the following values:

`r` or `rb` Open a file for reading.

- w* or *wb* Open a file for writing.
- a* or *ab* Open a file for writing at end-of-file.
- r+* or *rb+* or *r+b*
 Open a file for update (reading and writing).
- w+* or *wb+* or *w+b*
 Open a file for update (reading and writing).
- a+* or *ab+* or *a+b*
 Open a file for update (reading and writing) at
 end-of-file.

The meaning of these flags is exactly as specified in *fopen()*, except that modes beginning with *w* shall not cause truncation of the file.

Additional values for the *mode* argument may be supported by an implementation.

The application shall ensure that the mode of the stream as expressed by the *mode* argument is allowed by the file access mode of the open file description to which *fil-des* refers. The file position indicator associated with the new stream is set to the position indicated by the file offset associated with the file descriptor.

The error and end-of-file indicators for the stream shall be cleared. The *fdopen()* function may cause the last data access timestamp of the underlying file to be marked for update.

If *fil-des* refers to a shared memory object, the result of the *fdopen()* function is unspecified.

If *fil-des* refers to a typed memory object, the result of the *fdopen()* function is unspecified.

The *fdopen()* function shall preserve the offset maximum previously set for the open file description corresponding to *fil-des*.

RETURN VALUE [top](#)

Upon successful completion, *fdopen()* shall return a pointer to a stream; otherwise, a null pointer shall be returned and *errno* set to indicate the error.

ERRORS [top](#)

The *fdopen()* function shall fail if:

EMFILE {STREAM_MAX} streams are currently open in the calling process.

The *fdopen()* function may fail if:

EBADF The *fildevs* argument is not a valid file descriptor.

EINVAL The *mode* argument is not a valid mode.

EMFILE {FOPEN_MAX} streams are currently open in the calling process.

ENOMEM Insufficient space to allocate a buffer.

The following sections are informative.

EXAMPLES [top](#)

None.

APPLICATION USAGE [top](#)

File descriptors are obtained from calls like *open()*, *dup()*, *creat()*, or *pipe()*, which open files but do not return streams.

RATIONALE [top](#)

The file descriptor may have been obtained from *open()*, *creat()*, *pipe()*, *dup()*, *fcntl()*, or *socket()*; inherited through *fork()*, *posix_spawn()*, or *exec*; or perhaps obtained by other means.

The meanings of the *mode* arguments of *fdopen()* and *fopen()* differ. With *fdopen()*, open for write (*w* or *w+*) does not truncate, and append (*a* or *a+*) cannot create for writing. The *mode* argument formats that include a *b* are allowed for consistency with the ISO C standard function *fopen()*. The *b* has no effect on the resulting stream. Although not explicitly required by this volume of POSIX.1-2017, a good implementation of append (*a*) mode would cause the `O_APPEND` flag to be set.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

Section 2.5.1, Interaction of File Descriptors and Standard I/O Streams, [fclose\(3p\)](#), [fmemopen\(3p\)](#), [fopen\(3p\)](#), [open\(3p\)](#), [open_memstream\(3p\)](#), [posix_spawn\(3p\)](#), [socket\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [stdio.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .



Pages that refer to this page: [stdio.h\(0p\)](#), [fileno\(3p\)](#), [fmemopen\(3p\)](#), [fopen\(3p\)](#), [freopen\(3p\)](#), [open_memstream\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *[The Linux Programming Interface](#)*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fclose(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 fclose(3)

Library Functions Manual

fclose(3)

NAME [top](#)

fclose - close a stream

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
int fclose(FILE *stream);
```

DESCRIPTION [top](#)

The `fclose()` function flushes the stream pointed to by *stream* (writing any buffered output data using `fflush(3)`) and closes the underlying file descriptor.

RETURN VALUE [top](#)

Upon successful completion, 0 is returned. Otherwise, **EOF** is returned and `errno` is set to indicate the error. In either case, any further access (including another call to `fclose()`) to the stream results in undefined behavior.



ERRORS [top](#)

EBADF The file descriptor underlying *stream* is not valid.

The **fclose()** function may also fail and set *errno* for any of the errors specified for the routines `close(2)`, `write(2)`, or `fflush(3)`.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------------------|---------------|---------|
| <code>fclose()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

C89, POSIX.1-2001.

NOTES [top](#)

Note that **fclose()** flushes only the user-space buffers provided by the C library. To ensure that the data is physically stored on disk the kernel buffers must be flushed too, for example, with `sync(2)` or `fsync(2)`.

SEE ALSO [top](#)

`close(2)`, `fcloseall(3)`, `fflush(3)`, `fileno(3)`, `fopen(3)`, `setbuf(3)`

Linux man-pages (unreleased) (date)

fclose(3)



Pages that refer to this page: [close\(2\)](#), [abort\(3\)](#), [fcloseall\(3\)](#), [fflush\(3\)](#), [FILE\(3type\)](#), [fileno\(3\)](#), [fmemopen\(3\)](#), [fopen\(3\)](#), [fopencookie\(3\)](#), [getmntent\(3\)](#), [open_memstream\(3\)](#), [popen\(3\)](#), [setbuf\(3\)](#), [stdio\(3\)](#), [xdr\(3\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fcloseall(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [SEE ALSO](#)

 fcloseall(3)

Library Functions Manual

fcloseall(3)

NAME [top](#)

fcloseall - close all open streams

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#define _GNU_SOURCE          /* See feature_test_macros(7) */
#include <stdio.h>

int fcloseall(void);
```

DESCRIPTION [top](#)

The `fcloseall()` function closes all of the calling process's open streams. Buffered output for each stream is written before it is closed (as for `fflush(3)`); buffered input is discarded.

The standard streams, *stdin*, *stdout*, and *stderr* are also closed.

RETURN VALUE [top](#)

This function returns 0 if all files were successfully closed; on error, **EOF** is returned.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|--------------------|---------------|------------------------|
| fcloseall() | Thread safety | MT-Unsafe race:streams |

The **fcloseall()** function does not lock the streams, so it is not thread-safe.

STANDARDS [top](#)

GNU.

SEE ALSO [top](#)

[close\(2\)](#), [fclose\(3\)](#), [fflush\(3\)](#), [fopen\(3\)](#), [setbuf\(3\)](#)

Linux man-pages (unreleased) (date)

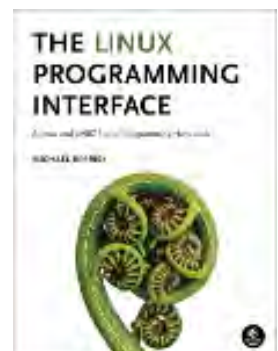
[fcloseall\(3\)](#)

Pages that refer to this page: [fclose\(3\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fgetc(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 fgetc(3)

Library Functions Manual

fgetc(3)

NAME [top](#)

fgetc, fgets, getc, getchar, ungetc - input of characters and strings

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
int fgetc(FILE *stream);
```

```
int getc(FILE *stream);
```

```
int getchar(void);
```

```
char *fgets(char s[restrict .size], int size, FILE *restrict stream);
```

```
int ungetc(int c, FILE *stream);
```

DESCRIPTION [top](#)

fgetc() reads the next character from *stream* and returns it as an *unsigned char* cast to an *int*, or **EOF** on end of file or error.

getc() is equivalent to **fgetc()** except that it may be implemented as a macro which evaluates *stream* more than once.

getchar() is equivalent to **getc(stdin)**.

fgetc() reads in at most one less than *size* characters from *stream* and stores them into the buffer pointed to by *s*. Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

ungetc() pushes *c* back to *stream*, cast to *unsigned char*, where it is available for subsequent read operations. Pushed-back characters will be returned in reverse order; only one pushback is guaranteed.

Calls to the functions described here can be mixed with each other and with calls to other input functions from the *stdio* library for the same input stream.

For nonlocking counterparts, see [unlocked_stdio\(3\)](#).

RETURN VALUE [top](#)

fgetc(), **getc()**, and **getchar()** return the character read as an *unsigned char* cast to an *int* or **EOF** on end of file or error.

fgetc() returns *s* on success, and NULL on error or when end of file occurs while no characters have been read.

ungetc() returns *c* on success, or **EOF** on error.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|--|---------------|---------|
| fgetc() , fgetc() , getc() , getchar() , ungetc() | Thread safety | MT-Safe |

STANDARDS [top](#)

C11, POSIX.1-2008.



HISTORY [top](#)

POSIX.1-2001, C89.

NOTES [top](#)

It is not advisable to mix calls to input functions from the `stdio` library with low-level calls to `read(2)` for the file descriptor associated with the input stream; the results will be undefined and very probably not what you want.

SEE ALSO [top](#)

`read(2)`, `write(2)`, `ferror(3)`, `fgetwc(3)`, `fgetws(3)`, `fopen(3)`, `fread(3)`, `fseek(3)`, `getline(3)`, `gets(3)`, `getwchar(3)`, `puts(3)`, `scanf(3)`, `ungetwc(3)`, `unlocked_stdio(3)`, `feature_test_macros(7)`

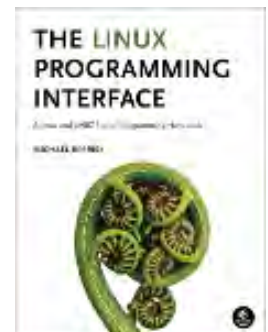
Linux man-pages (unreleased) [\(date\)](#) [fgetc\(3\)](#)

Pages that refer to this page: `EOF(3const)`, `ferror(3)`, `fgetwc(3)`, `fgetws(3)`, `flockfile(3)`, `fpurge(3)`, `fseek(3)`, `getline(3)`, `gets(3)`, `getw(3)`, `getwchar(3)`, `puts(3)`, `rpmatch(3)`, `scanf(3)`, `sscanf(3)`, `stdio(3)`, `ungetwc(3)`

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



ungetc(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 UNGETC(3P)

POSIX Programmer's Manual

UNGETC(3P)

PROLOG [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

`ungetc` – push byte back into input stream

SYNOPSIS [top](#)

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

DESCRIPTION [top](#)

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The `ungetc()` function shall push the byte specified by `c` (converted to an **unsigned char**) back onto the input stream

pointed to by *stream*. The pushed-back bytes shall be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file-positioning function (*fseek()*, *fseeko()*, *fsetpos()*, or *rewind()*) or *fflush()* shall discard any pushed-back bytes for the stream. The external storage corresponding to the stream shall be unchanged.

One byte of push-back shall be provided. If *ungetc()* is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail.

If the value of *c* equals that of the macro EOF, the operation shall fail and the input stream shall be left unchanged.

A successful call to *ungetc()* shall clear the end-of-file indicator for the stream. The value of the file-position indicator for the stream after all pushed-back bytes have been read, or discarded by calling *fseek()*, *fseeko()*, *fsetpos()*, or *rewind()* (but not *fflush()*), shall be the same as it was before the bytes were pushed back. The file-position indicator is decremented by each successful call to *ungetc()*; if its value was 0 before a call, its value is unspecified after the call.

RETURN VALUE [top](#)

Upon successful completion, *ungetc()* shall return the byte pushed back after conversion. Otherwise, it shall return EOF.

ERRORS [top](#)

No errors are defined.

The following sections are informative.

EXAMPLES [top](#)

None.

APPLICATION USAGE [top](#)

None.

RATIONALE [top](#)

None.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

Section 2.5, Standard I/O Streams, [fseek\(3p\)](#), [getc\(3p\)](#), [fsetpos\(3p\)](#), [read\(3p\)](#), [rewind\(3p\)](#), [setbuf\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [stdio.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

IEEE/The Open Group

2017

UNGETC(3P)



Pages that refer to this page: [stdio.h\(0p\)](#), [fgetc\(3p\)](#), [fgetpos\(3p\)](#), [fgets\(3p\)](#), [fseek\(3p\)](#), [fsetpos\(3p\)](#), [stdin\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *[The Linux Programming Interface](#)*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fgets(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 FGETS(3P)

POSIX Programmer's Manual

FGETS(3P)

PROLOG [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

`fgets` – get a string from a stream

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
char *fgets(char *restrict s, int n, FILE *restrict stream);
```

DESCRIPTION [top](#)

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The `fgets()` function shall read bytes from *stream* into the array pointed to by *s* until *n*-1 bytes are read, or a <newline> is read and transferred to *s*, or an end-of-file condition is encountered. A null byte shall be written immediately after the last byte read into the array. If the end-of-file condition is encountered before any bytes are read, the contents of the array pointed to by *s* shall not be changed.

The `fgets()` function may mark the last data access timestamp of the file associated with `stream` for update. The last data access timestamp shall be marked for update by the first successful execution of `fgetc()`, `fgets()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `getdelim()`, `getline()`, `gets()`, or `scanf()` using `stream` that returns data not supplied by a prior call to `ungetc()`.

RETURN VALUE [top](#)

Upon successful completion, `fgets()` shall return `s`. If the stream is at end-of-file, the end-of-file indicator for the stream shall be set and `fgets()` shall return a null pointer. If a read error occurs, the error indicator for the stream shall be set, `fgets()` shall return a null pointer, and shall set `errno` to indicate the error.

ERRORS [top](#)

Refer to [fgetc\(3p\)](#).

The following sections are informative.

EXAMPLES [top](#)

Reading Input

The following example uses `fgets()` to read lines of input. It assumes that the file it is reading is a text file and that lines in this text file are no longer than 16384 (or `{LINE_MAX}` if it is less than 16384 on the implementation where it is running) bytes long. (Note that the standard utilities have no line length limit if `sysconf(_SC_LINE_MAX)` returns -1 without setting `errno`. This example assumes that `sysconf(_SC_LINE_MAX)` will not fail.)

```
#include <limits.h>
#include <stdio.h>
#include <unistd.h>
#define MYLIMIT 16384

char *line;
int line_max;
if (LINE_MAX >= MYLIMIT) {
    // Use maximum line size of MYLIMIT. If LINE_MAX is
    // bigger than our limit, sysconf() cannot report a
    // smaller limit.
    line_max = MYLIMIT;
}
```

```
    } else {
        long limit = sysconf(_SC_LINE_MAX);
        line_max = (limit < 0 || limit > MYLIMIT) ? MYLIMIT : (int)limit;
    }

    // line_max + 1 leaves room for the null byte added by fgets().
    line = malloc(line_max + 1);
    if (line == NULL) {
        // out of space
        ...
        return error;
    }

    while (fgets(line, line_max + 1, fp) != NULL) {
        // Verify that a full line has been read ...
        // If not, report an error or prepare to treat the
        // next time through the loop as a read of a
        // continuation of the current line.
        ...
        // Process line ...
        ...
    }
    free(line);
    ...

```

APPLICATION USAGE [top](#)

None.

RATIONALE [top](#)

None.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

Section 2.5, Standard I/O Streams, [fgetc\(3p\)](#), [fopen\(3p\)](#), [fread\(3p\)](#), [fscanf\(3p\)](#), [getc\(3p\)](#), [getchar\(3p\)](#), [getdelim\(3p\)](#), [gets\(3p\)](#), [ungetc\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [stdio.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

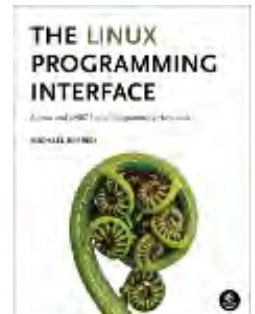
IEEE/The Open Group**2017****FGETS(3P)**

Pages that refer to this page: [stdio.h\(0p\)](#), [fgetc\(3p\)](#), [getdelim\(3p\)](#), [gets\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fread(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [EXAMPLES](#) | [SEE ALSO](#)

 fread(3)

Library Functions Manual

fread(3)

NAME [top](#)

fread, fwrite - binary stream input/output

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
size_t fread(void ptr[restrict .size * .nmemb],
             size_t size, size_t nmemb,
             FILE *restrict stream);
size_t fwrite(const void ptr[restrict .size * .nmemb],
             size_t size, size_t nmemb,
             FILE *restrict stream);
```

DESCRIPTION [top](#)

The function `fread()` reads `nmemb` items of data, each `size` bytes long, from the stream pointed to by `stream`, storing them at the location given by `ptr`.

The function `fwrite()` writes `nmemb` items of data, each `size` bytes long, to the stream pointed to by `stream`, obtaining them from the

location given by *ptr*.

For nonlocking counterparts, see [unlocked_stdio\(3\)](#).

RETURN VALUE [top](#)

On success, **fread()** and **fwrite()** return the number of items read or written. This number equals the number of bytes transferred only when *size* is 1. If an error occurs, or the end of the file is reached, the return value is a short item count (or zero).

The file position indicator for the stream is advanced by the number of bytes successfully read or written.

fread() does not distinguish between end-of-file and error, and callers must use [feof\(3\)](#) and [ferror\(3\)](#) to determine which occurred.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|----------------------------------|---------------|---------|
| fread() , fwrite() | Thread safety | MT-Safe |

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, C89.

EXAMPLES [top](#)

The program below demonstrates the use of **fread()** by parsing `/bin/sh` ELF executable in binary mode and printing its magic and

```
class:
```

```
$ ./a.out
ELF magic: 0x7f454c46
Class: 0x02
```

Program source

```
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))

int
main(void)
{
    FILE          *fp;
    size_t        ret;
    unsigned char  buffer[4];

    fp = fopen("/bin/sh", "rb");
    if (!fp) {
        perror("fopen");
        return EXIT_FAILURE;
    }

    ret = fread(buffer, sizeof(*buffer), ARRAY_SIZE(buffer), fp);
    if (ret != ARRAY_SIZE(buffer)) {
        fprintf(stderr, "fread() failed: %zu\n", ret);
        exit(EXIT_FAILURE);
    }

    printf("ELF magic: %#04x%02x%02x%02x\n", buffer[0], buffer[1],
          buffer[2], buffer[3]);

    ret = fread(buffer, 1, 1, fp);
    if (ret != 1) {
        fprintf(stderr, "fread() failed: %zu\n", ret);
        exit(EXIT_FAILURE);
    }

    printf("Class: %#04x\n", buffer[0]);

    fclose(fp);
}
```

```
    exit(EXIT_SUCCESS);  
}
```

SEE ALSO [top](#)

[read\(2\)](#), [write\(2\)](#), [feof\(3\)](#), [ferror\(3\)](#), [unlocked_stdio\(3\)](#)

Linux man-pages (unreleased) (date)

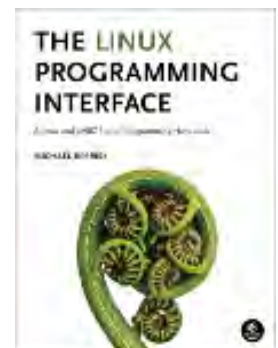
fread(3)

Pages that refer to this page: [pmlogger\(1\)](#), [read\(2\)](#), [write\(2\)](#), [fgetc\(3\)](#), [FILE\(3type\)](#), [getline\(3\)](#), [gets\(3\)](#), [getw\(3\)](#), [puts\(3\)](#), [setbuf\(3\)](#), [size_t\(3type\)](#), [stdin\(3\)](#), [stdio\(3\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



ferror(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [CAVEATS](#) | [SEE ALSO](#)

 ferror(3)

Library Functions Manual

ferror(3)

NAME [top](#)

clearerr, feof, ferror - check and reset stream status

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
void clearerr(FILE *stream);  
int feof(FILE *stream);  
int ferror(FILE *stream);
```

DESCRIPTION [top](#)

The function `clearerr()` clears the end-of-file and error indicators for the stream pointed to by *stream*.

The function `feof()` tests the end-of-file indicator for the stream pointed to by *stream*, returning nonzero if it is set. The end-of-file indicator can be cleared only by the function `clearerr()`.

The function **ferror()** tests the error indicator for the stream pointed to by *stream*, returning nonzero if it is set. The error indicator can be reset only by the **clearerr()** function.

For nonlocking counterparts, see [unlocked_stdio\(3\)](#).

RETURN VALUE [top](#)

The **feof()** function returns nonzero if the end-of-file indicator is set for *stream*; otherwise, it returns zero.

The **ferror()** function returns nonzero if the error indicator is set for *stream*; otherwise, it returns zero.

ERRORS [top](#)

These functions should not fail and do not set *errno*.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|---|---------------|---------|
| clearerr() , feof() , ferror() | Thread safety | MT-Safe |

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

C89, POSIX.1-2001.

NOTES [top](#)

POSIX.1-2008 specifies that these functions shall not change the value of `errno` if `stream` is valid.

CAVEATS [top](#)

Normally, programs should read the return value of an input function, such as `fgetc(3)`, before using functions of the `feof(3)` family. Only when the function returned the sentinel value `EOF` it makes sense to distinguish between the end of a file or an error with `feof(3)` or `ferror(3)`.

SEE ALSO [top](#)

`open(2)`, `fdopen(3)`, `fileno(3)`, `stdio(3)`, `unlocked_stdio(3)`

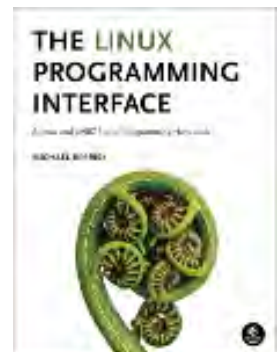
Linux man-pages (unreleased) [\(date\)](#) [ferror\(3\)](#)

Pages that refer to this page: `EOF(3const)`, `ferror(3)`, `fgetc(3)`, `fread(3)`, `fseek(3)`, `gets(3)`, `getw(3)`, `puts(3)`, `scanf(3)`, `stdio(3)`

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fputc(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 fputc(3P)

POSIX Programmer's Manual

fputc(3P)

PROLOG [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

fputc – put a byte on a stream

SYNOPSIS [top](#)

```
#include <stdio.h>

int fputc(int c, FILE *stream);
```

DESCRIPTION [top](#)

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The `fputc()` function shall write the byte specified by `c` (converted to an **unsigned char**) to the output stream pointed to

by *stream*, at the position indicated by the associated file-position indicator for the stream (if defined), and shall advance the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the byte shall be appended to the output stream.

The last data modification and last file status change timestamps of the file shall be marked for update between the successful execution of *fputc()* and the next successful completion of a call to *fflush()* or *fclose()* on the same stream or a call to *exit()* or *abort()*.

RETURN VALUE [top](#)

Upon successful completion, *fputc()* shall return the value it has written. Otherwise, it shall return EOF, the error indicator for the stream shall be set, and *errno* shall be set to indicate the error.

ERRORS [top](#)

The *fputc()* function shall fail if either the *stream* is unbuffered or the *stream*'s buffer needs to be flushed, and:

- EAGAIN** The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the thread would be delayed in the write operation.
- EBADF** The file descriptor underlying *stream* is not a valid file descriptor open for writing.
- EFBIG** An attempt was made to write to a file that exceeds the maximum file size.
- EFBIG** An attempt was made to write to a file that exceeds the file size limit of the process.
- EFBIG** The file is a regular file and an attempt was made to write at or beyond the offset maximum.
- EINTR** The write operation was terminated due to the receipt of a signal, and no data was transferred.



EIO A physical I/O error has occurred, or the process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the calling thread is not blocking SIGTTOU, the process is not ignoring SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.

ENOSPC There was no free space remaining on the device containing the file.

EPIPE An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal shall also be sent to the thread.

The `fputc()` function may fail if:

ENOMEM Insufficient storage space is available.

ENXIO A request was made of a nonexistent device, or the request was outside the capabilities of the device.

The following sections are informative.

EXAMPLES [top](#)

None.

APPLICATION USAGE [top](#)

None.

RATIONALE [top](#)

None.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

[Section 2.5, Standard I/O Streams](#), [ferror\(3p\)](#), [fopen\(3p\)](#), [getrlimit\(3p\)](#), [putc\(3p\)](#), [puts\(3p\)](#), [setbuf\(3p\)](#), [ulimit\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [stdio.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

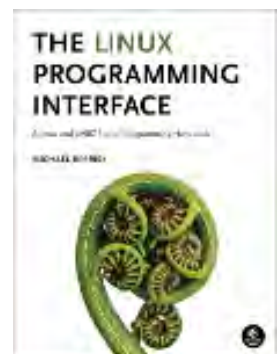
IEEE/The Open Group**2017*****FPUTC(3P)***

Pages that refer to this page: [stdio.h\(0p\)](#), [fprintf\(3p\)](#), [fputs\(3p\)](#), [fwrite\(3p\)](#), [perror\(3p\)](#), [psiginfo\(3p\)](#), [putc\(3p\)](#), [putchar\(3p\)](#), [puts\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fputs(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 FPUTS(3P)

POSIX Programmer's Manual

FPUTS(3P)**PROLOG** [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

fputs – put a string on a stream

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
int fputs(const char *restrict s, FILE *restrict stream);
```

DESCRIPTION [top](#)

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The *fputs()* function shall write the null-terminated string pointed to by *s* to the stream pointed to by *stream*. The

terminating null byte shall not be written.

The last data modification and last file status change timestamps of the file shall be marked for update between the successful execution of `fputs()` and the next successful completion of a call to `fflush()` or `fclose()` on the same stream or a call to `exit()` or `abort()`.

RETURN VALUE [top](#)

Upon successful completion, `fputs()` shall return a non-negative number. Otherwise, it shall return EOF, set an error indicator for the stream, and set `errno` to indicate the error.

ERRORS [top](#)

Refer to `fputc(3p)`.

The following sections are informative.

EXAMPLES [top](#)

Printing to Standard Output

The following example gets the current time, converts it to a string using `localtime()` and `asctime()`, and prints it to standard output using `fputs()`. It then prints the number of minutes to an event for which it is waiting.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
fputs(asctime(localtime(&now)), stdout);
printf("There are still %d minutes to the event.\n",
      minutes_to_event);
...
```

APPLICATION USAGE [top](#)

The `puts()` function appends a <newline> while `fputs()` does not.

This volume of POSIX.1-2017 requires that successful completion simply return a non-negative integer. There are at least three known different implementation conventions for this requirement:

- * Return a constant value.
- * Return the last character written.
- * Return the number of bytes written. Note that this implementation convention cannot be adhered to for strings longer than `{INT_MAX}` bytes as the value would not be representable in the return type of the function. For backwards-compatibility, implementations can return the number of bytes for strings of up to `{INT_MAX}` bytes, and return `{INT_MAX}` for all longer strings.

RATIONALE [top](#)

The `fputs()` function is one whose source code was specified in the referenced *The C Programming Language*. In the original edition, the function had no defined return value, yet many practical implementations would, as a side-effect, return the value of the last character written as that was the value remaining in the accumulator used as a return value. In the second edition of the book, either the fixed value `0` or `EOF` would be returned depending upon the return value of `ferror()`; however, for compatibility with extant implementations, several implementations would, upon success, return a positive value representing the last byte written.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

[Section 2.5, Standard I/O Streams](#), [fopen\(3p\)](#), [putc\(3p\)](#), [puts\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [stdio.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

IEEE/The Open Group

2017

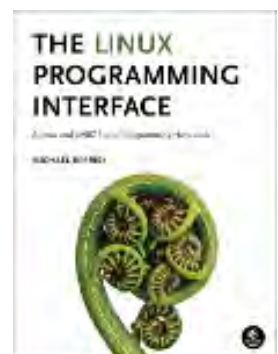
FPUTS(3P)

Pages that refer to this page: [stdio.h\(0p\)](#), [puts\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fseek(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 fseek(3)

Library Functions Manual

fseek(3)

NAME [top](#)

fgetpos, fseek, fsetpos, ftell, rewind - reposition a stream

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
int fseek(FILE *stream, long offset, int whence);  
long ftell(FILE *stream);
```

```
void rewind(FILE *stream);
```

```
int fgetpos(FILE *restrict stream, fpos_t *restrict pos);  
int fsetpos(FILE *stream, const fpos_t *pos);
```

DESCRIPTION [top](#)

The `fseek()` function sets the file position indicator for the stream pointed to by `stream`. The new position, measured in bytes, is obtained by adding `offset` bytes to the position specified by `whence`. If `whence` is set to `SEEK_SET`, `SEEK_CUR`, or

SEEK_END, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the **fseek()** function clears the end-of-file indicator for the stream and undoes any effects of the **ungetc(3)** function on the same stream.

The **ftell()** function obtains the current value of the file position indicator for the stream pointed to by *stream*.

The **rewind()** function sets the file position indicator for the stream pointed to by *stream* to the beginning of the file. It is equivalent to:

```
(void) fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared (see **clearerr(3)**).

The **fgetpos()** and **fsetpos()** functions are alternate interfaces equivalent to **ftell()** and **fseek()** (with *whence* set to **SEEK_SET**), setting and storing the current value of the file offset into or from the object referenced by *pos*. On some non-UNIX systems, an *fpos_t* object may be a complex object and these routines may be the only way to portably reposition a text stream.

If the stream refers to a regular file and the resulting stream offset is beyond the size of the file, subsequent writes will extend the file with a hole, up to the offset, before committing any data. See **lseek(2)** for details on file seeking semantics.

RETURN VALUE [top](#)

The **rewind()** function returns no value. Upon successful completion, **fgetpos()**, **fseek()**, **fsetpos()** return 0, and **ftell()** returns the current offset. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS [top](#)

EINVAL The *whence* argument to **fseek()** was not **SEEK_SET**, **SEEK_END**, or **SEEK_CUR**. Or: the resulting file offset would be negative.

ESPIPE The file descriptor underlying *stream* is not seekable (e.g., it refers to a pipe, FIFO, or socket).

The functions **fgetpos()**, **fseek()**, **fsetpos()**, and **ftell()** may also fail and set *errno* for any of the errors specified for the routines **fflush(3)**, **fstat(2)**, **lseek(2)**, and **malloc(3)**.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|---|---------------|---------|
| fseek() , ftell() , rewind() , fgetpos() , fsetpos() | Thread safety | MT-Safe |

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, C89.

SEE ALSO [top](#)

[lseek\(2\)](#), [fseeko\(3\)](#)

Linux man-pages (unreleased) [\(date\)](#) [fseek\(3\)](#)

Pages that refer to this page: [lseek\(2\)](#), [fgetc\(3\)](#), [fmemopen\(3\)](#), [fopen\(3\)](#), [fopencookie\(3\)](#), [fseeko\(3\)](#), [gets\(3\)](#), [open_memstream\(3\)](#), [puts\(3\)](#), [stdio\(3\)](#), [feature_test_macros\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



ftell(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 FTELL(3P)

POSIX Programmer's Manual

FTELL(3P)**PROLOG** [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

ftell, ftello – return a file offset in a stream

SYNOPSIS [top](#)

```
#include <stdio.h>

long ftell(FILE *stream);
off_t ftello(FILE *stream);
```

DESCRIPTION [top](#)

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The *ftell()* function shall obtain the current value of the file-

position indicator for the stream pointed to by *stream*.

The *ftell()* function shall not change the setting of *errno* if successful.

The *ftello()* function shall be equivalent to *ftell()*, except that the return value is of type **off_t** and the *ftello()* function may change the setting of *errno* if successful.

RETURN VALUE [top](#)

Upon successful completion, *ftell()* and *ftello()* shall return the current value of the file-position indicator for the stream measured in bytes from the beginning of the file.

Otherwise, *ftell()* and *ftello()* shall return -1, and set *errno* to indicate the error.

ERRORS [top](#)

The *ftell()* and *ftello()* functions shall fail if:

EBADF The file descriptor underlying *stream* is not an open file descriptor.

EOverflow

For *ftell()*, the current file offset cannot be represented correctly in an object of type **long**.

EOverflow

For *ftello()*, the current file offset cannot be represented correctly in an object of type **off_t**.

ESPIPE The file descriptor underlying *stream* is associated with a pipe, FIFO, or socket.

The following sections are informative.

EXAMPLES [top](#)

None.



APPLICATION USAGE [top](#)

None.

RATIONALE [top](#)

None.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

Section 2.5, Standard I/O Streams, [fgetpos\(3p\)](#), [fopen\(3p\)](#), [fseek\(3p\)](#), [lseek\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [stdio.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

IEEE/The Open Group

2017

FTELL(3P)



Pages that refer to this page: [stdio.h\(0p\)](#), [fgetpos\(3p\)](#), [fseek\(3p\)](#), [fsetpos\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fsetpos(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 FSETPOS(3P)

POSIX Programmer's Manual

FSETPOS(3P)

PROLOG [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

fsetpos – set current file position

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

DESCRIPTION [top](#)

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The `fsetpos()` function shall set the file position and state indicators for the stream pointed to by `stream` according to the

value of the object pointed to by *pos*, which the application shall ensure is a value obtained from an earlier call to *fgetpos()* on the same stream. If a read or write error occurs, the error indicator for the stream shall be set and *fsetpos()* fails.

A successful call to the *fsetpos()* function shall clear the end-of-file indicator for the stream and undo any effects of *ungetc()* on the same stream. After an *fsetpos()* call, the next operation on an update stream may be either input or output.

The behavior of *fsetpos()* on devices which are incapable of seeking is implementation-defined. The value of the file offset associated with such a device is undefined.

The *fsetpos()* function shall not change the setting of *errno* if successful.

RETURN VALUE [top](#)

The *fsetpos()* function shall return 0 if it succeeds; otherwise, it shall return a non-zero value and set *errno* to indicate the error.

ERRORS [top](#)

The *fsetpos()* function shall fail if, either the *stream* is unbuffered or the *stream*'s buffer needed to be flushed, and the call to *fsetpos()* causes an underlying *lseek()* or *write()* to be invoked, and:

EAGAIN The `O_NONBLOCK` flag is set for the file descriptor and the thread would be delayed in the write operation.

EBADF The file descriptor underlying the stream file is not open for writing or the stream's buffer needed to be flushed and the file is not open.

EFBIG An attempt was made to write a file that exceeds the maximum file size.

EFBIG An attempt was made to write a file that exceeds the file



size limit of the process.

- EFBIG** The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
- EINTR** The write operation was terminated due to the receipt of a signal, and no data was transferred.
- EIO** A physical I/O error has occurred, or the process is a member of a background process group attempting to perform a `write()` to its controlling terminal, TOSTOP is set, the calling thread is not blocking SIGTTOU, the process is not ignoring SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.
- ENOSPC** There was no free space remaining on the device containing the file.
- EPIPE** An attempt was made to write to a pipe or FIFO that is not open for reading by any process; a SIGPIPE signal shall also be sent to the thread.
- ESPIPE** The file descriptor underlying `stream` is associated with a pipe, FIFO, or socket.

The `fsetpos()` function may fail if:

- ENXIO** A request was made of a nonexistent device, or the request was outside the capabilities of the device.

The following sections are informative.

EXAMPLES [top](#)

None.

APPLICATION USAGE [top](#)

None.

RATIONALE [top](#)

None.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

Section 2.5, Standard I/O Streams, [fopen\(3p\)](#), [ftell\(3p\)](#), [lseek\(3p\)](#), [rewind\(3p\)](#), [ungetc\(3p\)](#), [write\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [stdio.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

IEEE/The Open Group

2017

FSETPOS(3P)

Pages that refer to this page: [stdio.h\(0p\)](#), [fseek\(3p\)](#), [ungetc\(3p\)](#), [ungetwc\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



rewind(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 REWIND(3P)

POSIX Programmer's Manual

REWIND(3P)**PROLOG** [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

`rewind` – reset the file position indicator in a stream

SYNOPSIS [top](#)

```
#include <stdio.h>

void rewind(FILE *stream);
```

DESCRIPTION [top](#)

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The call:

```
rewind(stream)
```

shall be equivalent to:

```
(void) fseek(stream, 0L, SEEK_SET)
```

except that *rewind()* shall also clear the error indicator.

Since *rewind()* does not return a value, an application wishing to detect errors should clear *errno*, then call *rewind()*, and if *errno* is non-zero, assume an error has occurred.

RETURN VALUE [top](#)

The *rewind()* function shall not return a value.

ERRORS [top](#)

Refer to [fseek\(3p\)](#) with the exception of **[EINVAL]** which does not apply.

The following sections are informative.

EXAMPLES [top](#)

None.

APPLICATION USAGE [top](#)

None.

RATIONALE [top](#)

None.

FUTURE DIRECTIONS [top](#)

None.



SEE ALSO [top](#)

[Section 2.5, Standard I/O Streams](#), [fseek\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [stdio.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

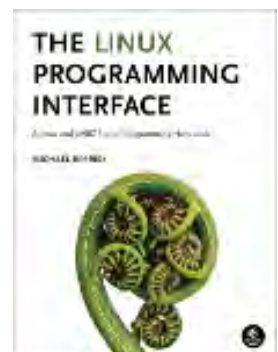
IEEE/The Open Group**2017****REWIND(3P)**

Pages that refer to this page: [stdio.h\(0p\)](#), [fgetpos\(3p\)](#), [fseek\(3p\)](#), [fsetpos\(3p\)](#), [ungetc\(3p\)](#), [ungetwc\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fflush(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 fflush(3)

Library Functions Manual

fflush(3)

NAME [top](#)

fflush - flush a stream

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
int fflush(FILE *_Nullable stream);
```

DESCRIPTION [top](#)

For output streams, **fflush()** forces a write of all user-space buffered data for the given output or update *stream* via the stream's underlying write function.

For input streams associated with seekable files (e.g., disk files, but not pipes or terminals), **fflush()** discards any buffered data that has been fetched from the underlying file, but has not been consumed by the application.

The open status of the stream is unaffected.

If the *stream* argument is NULL, **fflush()** flushes *all* open output streams.

For a nonlocking counterpart, see [unlocked_stdio\(3\)](#).

RETURN VALUE [top](#)

Upon successful completion 0 is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error.

ERRORS [top](#)

EBADF *stream* is not an open stream, or is not open for writing.

The function **fflush()** may also fail and set *errno* for any of the errors specified for [write\(2\)](#).

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------------|---------------|---------|
| fflush() | Thread safety | MT-Safe |

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

C89, POSIX.1-2001, POSIX.1-2008.

POSIX.1-2001 did not specify the behavior for flushing of input streams, but the behavior is specified in POSIX.1-2008.

NOTES [top](#)

Note that **fflush()** flushes only the user-space buffers provided by the C library. To ensure that the data is physically stored on disk the kernel buffers must be flushed too, for example, with `sync(2)` or `fsync(2)`.

SEE ALSO [top](#)

`fsync(2)`, `sync(2)`, `write(2)`, `fclose(3)`, `fileno(3)`, `fopen(3)`, `fpurge(3)`, `setbuf(3)`, `unlocked_stdio(3)`

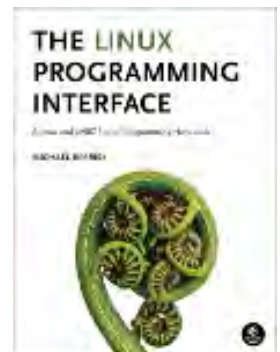
Linux man-pages (unreleased) (date) ***fflush(3)***

Pages that refer to this page: `fsync(2)`, `fclose(3)`, `fcloseall(3)`, `fmemopen(3)`, `fopen(3)`, `fpurge(3)`, `fseek(3)`, `open_memstream(3)`, `popen(3)`, `setbuf(3)`, `stdin(3)`, `stdio(3)`, `xdr(3)`

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fileno(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 fileno(3)

Library Functions Manual

fileno(3)

NAME [top](#)

`fileno` - obtain file descriptor of a stdio stream

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
int fileno(FILE *stream);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
fileno():  
    _POSIX_C_SOURCE
```

DESCRIPTION [top](#)

The function `fileno()` examines the argument *stream* and returns the integer file descriptor used to implement this stream. The file descriptor is still owned by *stream* and will be closed when `fclose(3)` is called. Duplicate the file descriptor with `dup(2)`

before passing it to code that might close it.

For the nonlocking counterpart, see [unlocked_stdio\(3\)](#).

RETURN VALUE [top](#)

On success, `fileno()` returns the file descriptor associated with `stream`. On failure, `-1` is returned and `errno` is set to indicate the error.

ERRORS [top](#)

`EBADF` `stream` is not associated with a file.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------------------|---------------|---------|
| <code>fileno()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

SEE ALSO [top](#)

[open\(2\)](#), [fdopen\(3\)](#), [stdio\(3\)](#), [unlocked_stdio\(3\)](#)

Linux man-pages (unreleased) (date)

[fileno\(3\)](#)



Pages that refer to this page: [fsync\(2\)](#), [fclose\(3\)](#), [ferror\(3\)](#), [fflush\(3\)](#), [fmemopen\(3\)](#), [fopen\(3\)](#), [open_memstream\(3\)](#), [pcap_file\(3pcap\)](#), [stdio\(3\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



setvbuf(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 SETVBUF(3P)

POSIX Programmer's Manual

SETVBUF(3P)**PROLOG** [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

setvbuf – assign buffering to a stream

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
int setvbuf(FILE *restrict stream, char *restrict buf, int type,  
           size_t size);
```

DESCRIPTION [top](#)

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The `setvbuf()` function may be used after the stream pointed to by

stream is associated with an open file but before any other operation (other than an unsuccessful call to *setvbuf()*) is performed on the stream. The argument *type* determines how *stream* shall be buffered, as follows:

- * `{_IOFBF}` shall cause input/output to be fully buffered.
- * `{_IOLBF}` shall cause input/output to be line buffered.
- * `{_IONBF}` shall cause input/output to be unbuffered.

If *buf* is not a null pointer, the array it points to may be used instead of a buffer allocated by *setvbuf()* and the argument *size* specifies the size of the array; otherwise, *size* may determine the size of a buffer allocated by the *setvbuf()* function. The contents of the array at any time are unspecified.

For information about streams, see *Section 2.5, Standard I/O Streams*.

RETURN VALUE [top](#)

Upon successful completion, *setvbuf()* shall return 0. Otherwise, it shall return a non-zero value if an invalid value is given for *type* or if the request cannot be honored, and may set *errno* to indicate the error.

ERRORS [top](#)

The *setvbuf()* function may fail if:

EBADF The file descriptor underlying *stream* is not valid.

The following sections are informative.

EXAMPLES [top](#)

None.

APPLICATION USAGE [top](#)

A common source of error is allocating buffer space as an ```automatic''` variable in a code block, and then failing to close the stream in the same block.

With `setvbuf()`, allocating a buffer of `size` bytes does not necessarily imply that all of `size` bytes are used for the buffer area.

Applications should note that many implementations only provide line buffering on input from terminal devices.

RATIONALE [top](#)

None.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

Section 2.5, Standard I/O Streams, [fopen\(3p\)](#), [setbuf\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [stdio.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see



https://www.kernel.org/doc/man-pages/reporting_bugs.html .

IEEE/The Open Group

2017

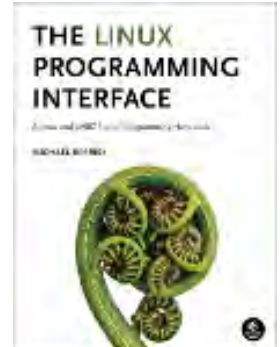
SETVBUF(3P)

Pages that refer to this page: [stdio.h\(0p\)](#), [cat\(1p\)](#), [setbuf\(3p\)](#), [stdin\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



flockfile(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 flockfile(3)

Library Functions Manual

flockfile(3)

NAME [top](#)

flockfile, ftrylockfile, funlockfile - lock FILE for stdio

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
void flockfile(FILE *filehandle);
int ftrylockfile(FILE *filehandle);
void funlockfile(FILE *filehandle);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

All functions shown above:

```
/* Since glibc 2.24: */ _POSIX_C_SOURCE >= 199309L
|| /* glibc <= 2.23: */ _POSIX_C_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

DESCRIPTION [top](#)

The `stdio` functions are thread-safe. This is achieved by assigning to each *FILE* object a lockcount and (if the lockcount is nonzero) an owning thread. For each library call, these functions wait until the *FILE* object is no longer locked by a different thread, then lock it, do the requested I/O, and unlock the object again.

(Note: this locking has nothing to do with the file locking done by functions like `flock(2)` and `lockf(3)`.)

All this is invisible to the C-programmer, but there may be two reasons to wish for more detailed control. On the one hand, maybe a series of I/O actions by one thread belongs together, and should not be interrupted by the I/O of some other thread. On the other hand, maybe the locking overhead should be avoided for greater efficiency.

To this end, a thread can explicitly lock the *FILE* object, then do its series of I/O actions, then unlock. This prevents other threads from coming in between. If the reason for doing this was to achieve greater efficiency, one does the I/O with the nonlocking versions of the `stdio` functions: with `getc_unlocked(3)` and `putc_unlocked(3)` instead of `getc(3)` and `putc(3)`.

The `flockfile()` function waits for **filehandle* to be no longer locked by a different thread, then makes the current thread owner of **filehandle*, and increments the lockcount.

The `funlockfile()` function decrements the lock count.

The `ftrylockfile()` function is a nonblocking version of `flockfile()`. It does nothing in case some other thread owns **filehandle*, and it obtains ownership and increments the lockcount otherwise.

RETURN VALUE [top](#)

The `ftrylockfile()` function returns zero for success (the lock was obtained), and nonzero for failure.

ERRORS [top](#)

None.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|--|---------------|---------|
| <code>flockfile()</code> , <code>ftrylockfile()</code> , <code>funlockfile()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

These functions are available when `_POSIX_THREAD_SAFE_FUNCTIONS` is defined.

SEE ALSO [top](#)

[unlocked_stdio\(3\)](#)

Linux man-pages (unreleased) (date) [flockfile\(3\)](#)

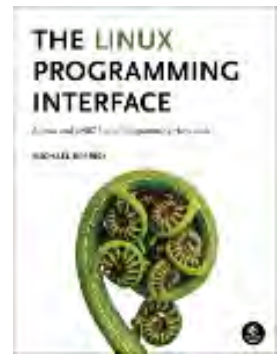
Pages that refer to this page: [FILE\(3type\)](#), [stdio_ext\(3\)](#), [unlocked_stdio\(3\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *[The Linux Programming Interface](#)*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



Materials for Topic 5: Buffered File I/O

Full C Programs

- [files.c](#) - a C program using file pointers and calls to the following functions: `fopen()`, `fprintf()`, `getc()`, `fgets()`, and `fclose()`.
- [buffered_io.c](#) - a C program using: `fopen()`, `fwrite()`, `fread()`, and `fclose()` to read a binary data from and write a binary data into a structure.

Runnable Linux Commands

Quick Links:

- [gcc](#)
- [./short_prompt](#)
- [./long_prompt](#)
- [time](#)
- [dd bs=1 count=2097152](#)
- [dd bs=1024 count=2048](#)
- [stat](#)
- [xxd](#)

-
- The command:

```
gcc -Wall -Wextra -O2 -g -o program program.c
```

compiles the C source code located inside the file `program.c`. See more details [here](#).

- The command:

```
./short_prompt
```

executes code inside a file named `short_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).

- The command:

```
./long_prompt
```

executes code inside a file named `long_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).

- The command:

```
time SomeCommand
```

runs the command given by `SomeCommand` and, after the command finishes running, displays how much time the command executed, provided in terms of the (i) the elapsed real time between invocation and termination, (ii) the user CPU time, and (iii) the system CPU time.

- The command:

```
dd bs=1 count=2097152 if=/dev/zero of=pirate1
```

copies 2097152 blocks of size 1 byte each from the file at `/dev/zero` (which is a disk-like file providing an infinite stream of zeros, for testing and debugging purposes) to a file named `pirate1`.

- The command:

```
dd bs=1024 count=2048 if=/dev/zero of=pirate2
```

copies 2048 blocks of size 1024 bytes each from the file at `/dev/zero` (which is a disk-like file providing an infinite stream of zeros, for testing and debugging purposes) to a file named `pirate2`. Since disks process data in terms of blocks (which are usually multiples of 512,) we expect this call to run much faster than the call

```
dd bs=1 count=2097152 if=/dev/zero of=pirate1
```

.

- The command:

```
stat fileOrDiskName
```

shows some statistics about the file or disk given by `fileOrDiskName`. For example,

```
stat /dev/sda
```

 shows some statistics about the disk `sda`. The context in which this command is mentioned in the lecture notes is that it lets you view the block size that the disk uses, such as 4K = 4096.

- The command:

```
xxd myFile
```

prints the contents of the file `myFile` in hexadecimal. A hexadecimal view allows you to understand what characters exist in the file, even if they are invisible in decimal (normal) view, such as null characters and other special ASCII characters.



This website by [Miriam Briskman](#) is licensed under [CC BY-NC 4.0](#).



```
1  /* The following program exemplifies calls to
2  *   fopen() fwrite(), fread(), and fclose(). We
3  *   write the binary data of a single structure
4  *   into a file. This same file is then opened
5  *   for reading, and we read this data back in
6  *   again as binary data into another structure
7  *   instance, s. We then print the data fields
8  *   of s to the terminal.
9  *
10 *   Miriam Briskman, 3/8/2023
11 *   CISC 3350, Brooklyn College
12 *   Licensed under CC BY-NC 4.0
13 */
14
15 #include <stdio.h>
16 #include <stdlib.h> // For EXIT_FAILURE, EXIT_SUCCESS.
17
18 struct Dog
19 {
20     char breed[100];
21     unsigned int age; // in years
22     char nature[100];
23 };
24
25 int main (void)
26 {
27     FILE *in_file, *out_file;
28
29     struct Dog lucky = {"Labrador retriever",
30                        5,
31                        "Playful, Loving"};
32     struct Dog stella;
33
34     // Above, 'lucky' and 'stella' are both
35     // instances of the 'Dog' structure.
36     // Specifically, 'lucky' and 'stella'
37     // are twins! Only 'lucky' was
38     // initiated so far, and we will
39     // initiate 'stella' now via file i/o!
40
41     // Create a new file 'data' for writing:
42     out_file = fopen ("data", "w");
```



```
43  if (!out_file) // Same as (out_file == NULL)
44  {
45      perror ("fopen");
46      exit (EXIT_FAILURE);
47  }
48
49  // Write lucky's data in binary manner into
50  //   the file:
51  if (!fwrite (&lucky,
52              sizeof (struct Dog),
53              1,
54              out_file))
55  {
56      perror ("fwrite");
57      exit (EXIT_FAILURE);
58  }
59
60  /* Explanation:
61   Above, fwrite() accesses the address in
62   memory where the information about
63   lucky is stored. Because all the data
64   fields of a structure are stored
65   consecutively in memory, the function
66   needs to know how many bytes a Dog
67   structure uses in memory and copy
68   that data over to the file. The 'sizeof'
69   operator finds this information.
70   We need to copy only one instance of
71   the Dog struct, so the 3rd argument to
72   the function is 1.
73  */
74
75  // Close the file 'out_file':
76  if (fclose (out_file))
77  {
78      perror ("fclose");
79      exit (EXIT_FAILURE);
80  }
81
82  // Open the 'data' file for reading:
83  in_file = fopen ("data", "r");
84  if (!in_file)
85  {
```



```
86     perror ("fopen");
87     exit (EXIT_FAILURE);
88 }
89
90 // Read in the binary data:
91 if (!fread (&stella,
92           sizeof (struct Dog),
93           1,
94           in_file))
95 {
96     perror ("fread");
97     exit (EXIT_FAILURE);
98 }
99
100 // Close the file 'in_file':
101 if (fclose (in_file))
102 {
103     perror ("fclose");
104     exit (EXIT_FAILURE);
105 }
106
107 // Finally, print information about stella
108 //     in the terminal:
109 printf ("Information about Stella:\n"
110        "Breed: %s\n"
111        "Age: %i\n"
112        "Nature: %s\n",
113        stella.breed, stella.age, stella.nature);
114
115 return EXIT_SUCCESS;
116 }
117
```



Topic 6: Vectored I/O & Memory-Mapping

Note: A copy of each one of the sources listed below is included in this packet. To jump to the location of a source copy, click the page link under “Page”.

In case a source isn’t included (e.g., a YouTube video,) click the ↗ (external link) symbol under “Page” to open the external source in a browser window.

| # | Citation & Source Link | Page |
|----|---|------|
| 1 | “readv(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/readv.2.html | 687 |
| 2 | “mmap(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/mmap.2.html | 694 |
| 3 | “sysconf(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/sysconf.3.html | 710 |
| 4 | “getpagesize(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/getpagesize.2.html | 718 |
| 5 | “fstat(3p) - Linux manual page”, <i>man7.org</i> , 2017. URL: https://man7.org/linux/man-pages/man3/fstat.3p.html | 721 |
| 6 | “putchar(3p) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/putchar.3p.html | 726 |
| 7 | “mremap(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/mremap.2.html | 729 |
| 8 | “mprotect(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/mprotect.2.html | 735 |
| 9 | “msync(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/msync.2.html | 742 |
| 10 | “madvise(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/madvise.2.html | 745 |
| 11 | Briskman, Miriam. “Materials for Topic 6: Vectored I/O & Memory-Mapping.” <i>Topic 6: Vectored I/O & Memory-Mapping — CISC 3350 Materials</i> , 2023. URL: www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_06/ | 759 |
| 12 | Briskman, Miriam. “writev_example.c .” (C source code) 8 Mar. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_06/writev_example.c | 761 |
| 13 | Briskman, Miriam. “readv_example.cc .” (C source code) 8 Mar. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_06/readv_example.c | 763 |

| # | Citation & Source Link | Page |
|----|--|------|
| 14 | Love, Robert. "mmap_example.c ." (C source code), <i>Linux System Programming: Talking Directly to the Kernel and C Library</i> , O'Reilly Media, Inc., Sebastopol, CA, 2013, pp. 109-110. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_06/mmap_example.c | 766 |

readv(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [EXAMPLES](#) | [SEE ALSO](#)

 readv(2)

System Calls Manual

readv(2)

NAME [top](#)

readv, writev, preadv, pwritev, preadv2, pwritev2 - read or write data into multiple buffers

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/uio.h>
```

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);  
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

```
ssize_t preadv(int fd, const struct iovec *iov, int iovcnt,  
               off_t offset);  
ssize_t pwritev(int fd, const struct iovec *iov, int iovcnt,  
                off_t offset);
```

```
ssize_t preadv2(int fd, const struct iovec *iov, int iovcnt,  
                off_t offset, int flags);  
ssize_t pwritev2(int fd, const struct iovec *iov, int iovcnt,  
                 off_t offset, int flags);
```

Feature Test Macro Requirements for glibc (see

[feature_test_macros\(7\)](#)):

preadv(), **pwritev()**:
Since glibc 2.19:
 _DEFAULT_SOURCE
glibc 2.19 and earlier:
 _BSD_SOURCE

DESCRIPTION [top](#)

The **readv()** system call reads *iovcnt* buffers from the file associated with the file descriptor *fd* into the buffers described by *iov* ("scatter input").

The **writev()** system call writes *iovcnt* buffers of data described by *iov* to the file associated with the file descriptor *fd* ("gather output").

The pointer *iov* points to an array of *iovec* structures, described in [iovec\(3type\)](#).

The **readv()** system call works just like [read\(2\)](#) except that multiple buffers are filled.

The **writev()** system call works just like [write\(2\)](#) except that multiple buffers are written out.

Buffers are processed in array order. This means that **readv()** completely fills *iov[0]* before proceeding to *iov[1]*, and so on. (If there is insufficient data, then not all buffers pointed to by *iov* may be filled.) Similarly, **writev()** writes out the entire contents of *iov[0]* before proceeding to *iov[1]*, and so on.

The data transfers performed by **readv()** and **writev()** are atomic: the data written by **writev()** is written as a single block that is not intermingled with output from writes in other processes; analogously, **readv()** is guaranteed to read a contiguous block of data from the file, regardless of read operations performed in other threads or processes that have file descriptors referring to the same open file description (see [open\(2\)](#)).

preadv() and **pwritev()**

The **preadv()** system call combines the functionality of **readv()**

and `pread(2)`. It performs the same task as `readv()`, but adds a fourth argument, *offset*, which specifies the file offset at which the input operation is to be performed.

The `pwritev()` system call combines the functionality of `writev()` and `pwrite(2)`. It performs the same task as `writev()`, but adds a fourth argument, *offset*, which specifies the file offset at which the output operation is to be performed.

The file offset is not changed by these system calls. The file referred to by *fd* must be capable of seeking.

`preadv2()` and `pwritev2()`

These system calls are similar to `preadv()` and `pwritev()` calls, but add a fifth argument, *flags*, which modifies the behavior on a per-call basis.

Unlike `preadv()` and `pwritev()`, if the *offset* argument is -1, then the current file offset is used and updated.

The *flags* argument contains a bitwise OR of zero or more of the following flags:

RWF_DSYNC (since Linux 4.7)

Provide a per-write equivalent of the **O_DSYNC** `open(2)` flag. This flag is meaningful only for `pwritev2()`, and its effect applies only to the data range written by the system call.

RWF_HIPRI (since Linux 4.6)

High priority read/write. Allows block-based filesystems to use polling of the device, which provides lower latency, but may use additional resources. (Currently, this feature is usable only on a file descriptor opened using the **O_DIRECT** flag.)

RWF_SYNC (since Linux 4.7)

Provide a per-write equivalent of the **O_SYNC** `open(2)` flag. This flag is meaningful only for `pwritev2()`, and its effect applies only to the data range written by the system call.

RWF_NOWAIT (since Linux 4.14)

Do not wait for data which is not immediately available.



If this flag is specified, the **preadv2()** system call will return instantly if it would have to read data from the backing storage or wait for a lock. If some data was successfully read, it will return the number of bytes read. If no bytes were read, it will return -1 and set *errno* to **EAGAIN** (but see **BUGS**). Currently, this flag is meaningful only for **preadv2()**.

RWF_APPEND (since Linux 4.16)

Provide a per-write equivalent of the **O_APPEND** **open(2)** flag. This flag is meaningful only for **pwritev2()**, and its effect applies only to the data range written by the system call. The *offset* argument does not affect the write operation; the data is always appended to the end of the file. However, if the *offset* argument is -1, the current file offset is updated.

RETURN VALUE [top](#)

On success, **readv()**, **preadv()**, and **preadv2()** return the number of bytes read; **writev()**, **pwritev()**, and **pwritev2()** return the number of bytes written.

Note that it is not an error for a successful call to transfer fewer bytes than requested (see **read(2)** and **write(2)**).

On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

The errors are as given for **read(2)** and **write(2)**. Furthermore, **preadv()**, **preadv2()**, **pwritev()**, and **pwritev2()** can also fail for the same reasons as **lseek(2)**. Additionally, the following errors are defined:

EINVAL The sum of the *iov_len* values overflows an *ssize_t* value.

EINVAL The vector count, *iovcnt*, is less than zero or greater than the permitted maximum.

EOPNOTSUPP

An unknown flag is specified in *flags*.

VERSIONS [top](#)

C library/kernel differences

The raw `preadv()` and `pwritev()` system calls have call signatures that differ slightly from that of the corresponding GNU C library wrapper functions shown in the SYNOPSIS. The final argument, *offset*, is unpacked by the wrapper functions into two arguments in the system calls:

`unsigned long pos_l, unsigned long pos`

These arguments contain, respectively, the low order and high order 32 bits of *offset*.

STANDARDS [top](#)

`readv()`
`writev()`
POSIX.1-2008.

`preadv()`
`pwritev()`
BSD.

`preadv2()`
`pwritev2()`
Linux.

HISTORY [top](#)

`readv()`
`writev()`
POSIX.1-2001, 4.4BSD (first appeared in 4.2BSD).

`preadv()`, `pwritev()`: Linux 2.6.30, glibc 2.10.

`preadv2()`, `pwritev2()`: Linux 4.6, glibc 2.26.

Historical C library/kernel differences

To deal with the fact that `IOV_MAX` was so low on early versions of Linux, the glibc wrapper functions for `readv()` and `writev()` did some extra work if they detected that the underlying kernel

system call failed because this limit was exceeded. In the case of `readv()`, the wrapper function allocated a temporary buffer large enough for all of the items specified by `iov`, passed that buffer in a call to `read(2)`, copied data from the buffer to the locations specified by the `iov_base` fields of the elements of `iov`, and then freed the buffer. The wrapper function for `writev()` performed the analogous task using a temporary buffer and a call to `write(2)`.

The need for this extra effort in the glibc wrapper functions went away with Linux 2.2 and later. However, glibc continued to provide this behavior until glibc 2.10. Starting with glibc 2.9, the wrapper functions provide this behavior only if the library detects that the system is running a Linux kernel older than Linux 2.6.18 (an arbitrarily selected kernel version). And since glibc 2.20 (which requires a minimum of Linux 2.6.32), the glibc wrapper functions always just directly invoke the system calls.

NOTES [top](#)

POSIX.1 allows an implementation to place a limit on the number of items that can be passed in `iov`. An implementation can advertise its limit by defining `IOV_MAX` in `<limits.h>` or at run time via the return value from `sysconf(_SC_IOV_MAX)`. On modern Linux systems, the limit is 1024. Back in Linux 2.0 days, this limit was 16.

BUGS [top](#)

Linux 5.9 and Linux 5.10 have a bug where `preadv2()` with the `RWF_NOWAIT` flag may return 0 even when not at end of file.

EXAMPLES [top](#)

The following code sample demonstrates the use of `writev()`:

```
char          *str0 = "hello ";
char          *str1 = "world\n";
ssize_t       nwritten;
struct iovec  iov[2];
```

```
iov[0].iov_base = str0;
iov[0].iov_len = strlen(str0);
iov[1].iov_base = str1;
iov[1].iov_len = strlen(str1);

nwritten = writev(STDOUT_FILENO, iov, 2);
```

SEE ALSO [top](#)

[pread\(2\)](#), [read\(2\)](#), [write\(2\)](#)

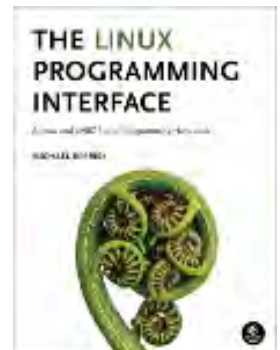
Linux man-pages (unreleased) (date) [readv\(2\)](#)

Pages that refer to this page: [strace\(1\)](#), [fcntl\(2\)](#), [fsync\(2\)](#), [io_submit\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [io_uring_setup\(2\)](#), [pread\(2\)](#), [process_vm_readv\(2\)](#), [read\(2\)](#), [recv\(2\)](#), [send\(2\)](#), [syscall\(2\)](#), [syscalls\(2\)](#), [write\(2\)](#), [iovec\(3type\)](#), [size_t\(3type\)](#), [io_uring\(7\)](#), [signal\(7\)](#), [socket\(7\)](#), [spufs\(7\)](#), [xfs_io\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



mmap(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [EXAMPLES](#) | [SEE ALSO](#)

 mmap(2)

System Calls Manual

mmap(2)

NAME [top](#)

mmap, munmap - map or unmap files or devices into memory

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/mman.h>
```

```
void *mmap(void addr[.length], size_t length, int prot, int flags,  
           int fd, off_t offset);  
int munmap(void addr[.length], size_t length);
```

See NOTES for information on feature test macro requirements.

DESCRIPTION [top](#)

`mmap()` creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in *addr*. The *length* argument specifies the length of the mapping (which must be greater than 0).

If *addr* is NULL, then the kernel chooses the (page-aligned)



address at which to create the mapping; this is the most portable method of creating a new mapping. If *addr* is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the kernel will pick a nearby page boundary (but always above or equal to the value specified by */proc/sys/vm/mmap_min_addr*) and attempt to create the mapping there. If another mapping already exists there, the kernel picks a new address that may or may not depend on the hint. The address of the new mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping; see **MAP_ANONYMOUS** below), are initialized using *length* bytes starting at offset *offset* in the file (or other object) referred to by the file descriptor *fd*. *offset* must be a multiple of the page size as returned by *sysconf(_SC_PAGE_SIZE)*.

After the **mmap()** call has returned, the file descriptor, *fd*, can be closed immediately without invalidating the mapping.

The *prot* argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). It is either **PROT_NONE** or the bitwise OR of one or more of the following flags:

PROT_EXEC

Pages may be executed.

PROT_READ

Pages may be read.

PROT_WRITE

Pages may be written.

PROT_NONE

Pages may not be accessed.

The flags argument

The *flags* argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file. This behavior is determined by including exactly one of the following values in *flags*:

MAP_SHARED

Share this mapping. Updates to the mapping are visible to



other processes mapping the same region, and (in the case of file-backed mappings) are carried through to the underlying file. (To precisely control when updates are carried through to the underlying file requires the use of `msync(2)`.)

MAP_SHARED_VALIDATE (since Linux 4.15)

This flag provides the same behavior as **MAP_SHARED** except that **MAP_SHARED** mappings ignore unknown flags in *flags*. By contrast, when creating a mapping using **MAP_SHARED_VALIDATE**, the kernel verifies all passed flags are known and fails the mapping with the error **EOPNOTSUPP** for unknown flags. This mapping type is also required to be able to use some mapping flags (e.g., **MAP_SYNC**).

MAP_PRIVATE

Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the `mmap()` call are visible in the mapped region.

Both **MAP_SHARED** and **MAP_PRIVATE** are described in POSIX.1-2001 and POSIX.1-2008. **MAP_SHARED_VALIDATE** is a Linux extension.

In addition, zero or more of the following values can be ORed in *flags*:

MAP_32BIT (since Linux 2.4.20, 2.6)

Put the mapping into the first 2 Gigabytes of the process address space. This flag is supported only on x86-64, for 64-bit programs. It was added to allow thread stacks to be allocated somewhere in the first 2 GB of memory, so as to improve context-switch performance on some early 64-bit processors. Modern x86-64 processors no longer have this performance problem, so use of this flag is not required on those systems. The **MAP_32BIT** flag is ignored when **MAP_FIXED** is set.

MAP_ANON

Synonym for **MAP_ANONYMOUS**; provided for compatibility with other implementations.

MAP_ANONYMOUS

The mapping is not backed by any file; its contents are



initialized to zero. The *fd* argument is ignored; however, some implementations require *fd* to be -1 if **MAP_ANONYMOUS** (or **MAP_ANON**) is specified, and portable applications should ensure this. The *offset* argument should be zero. Support for **MAP_ANONYMOUS** in conjunction with **MAP_SHARED** was added in Linux 2.4.

MAP_DENYWRITE

This flag is ignored. (Long ago—Linux 2.0 and earlier—it signaled that attempts to write to the underlying file should fail with **ETXTBSY**. But this was a source of denial-of-service attacks.)

MAP_EXECUTABLE

This flag is ignored.

MAP_FILE

Compatibility flag. Ignored.

MAP_FIXED

Don't interpret *addr* as a hint: place the mapping at exactly that address. *addr* must be suitably aligned: for most architectures a multiple of the page size is sufficient; however, some architectures may impose additional restrictions. If the memory region specified by *addr* and *length* overlaps pages of any existing mapping(s), then the overlapped part of the existing mapping(s) will be discarded. If the specified address cannot be used, **mmap()** will fail.

Software that aspires to be portable should use the **MAP_FIXED** flag with care, keeping in mind that the exact layout of a process's memory mappings is allowed to change significantly between Linux versions, C library versions, and operating system releases. *Carefully read the discussion of this flag in NOTES!*

MAP_FIXED_NO_REPLACE (since Linux 4.17)

This flag provides behavior that is similar to **MAP_FIXED** with respect to the *addr* enforcement, but differs in that **MAP_FIXED_NO_REPLACE** never clobbers a preexisting mapped range. If the requested range would collide with an existing mapping, then this call fails with the error **EEXIST**. This flag can therefore be used as a way to atomically (with respect to other threads) attempt to map



an address range: one thread will succeed; all others will report failure.

Note that older kernels which do not recognize the **MAP_FIXED_NOREPLACE** flag will typically (upon detecting a collision with a preexisting mapping) fall back to a “non-**MAP_FIXED**” type of behavior: they will return an address that is different from the requested address. Therefore, backward-compatible software should check the returned address against the requested address.

MAP_GROWSDOWN

This flag is used for stacks. It indicates to the kernel virtual memory system that the mapping should extend downward in memory. The return address is one page lower than the memory area that is actually created in the process's virtual address space. Touching an address in the "guard" page below the mapping will cause the mapping to grow by a page. This growth can be repeated until the mapping grows to within a page of the high end of the next lower mapping, at which point touching the "guard" page will result in a **SIGSEGV** signal.

MAP_HUGETLB (since Linux 2.6.32)

Allocate the mapping using "huge" pages. See the Linux kernel source file

[Documentation/admin-guide/mm/hugetlbpage.rst](#) for further information, as well as NOTES, below.

MAP_HUGE_2MB, MAP_HUGE_1GB (since Linux 3.8)

Used in conjunction with **MAP_HUGETLB** to select alternative hugetlb page sizes (respectively, 2 MB and 1 GB) on systems that support multiple hugetlb page sizes.

More generally, the desired huge page size can be configured by encoding the base-2 logarithm of the desired page size in the six bits at the offset **MAP_HUGE_SHIFT**. (A value of zero in this bit field provides the default huge page size; the default huge page size can be discovered via the *Hugepagesize* field exposed by [/proc/meminfo](#).) Thus, the above two constants are defined as:

```
#define MAP_HUGE_2MB      (21 << MAP_HUGE_SHIFT)
#define MAP_HUGE_1GB     (30 << MAP_HUGE_SHIFT)
```



The range of huge page sizes that are supported by the system can be discovered by listing the subdirectories in [/sys/kernel/mm/hugepages](#).

MAP_LOCKED (since Linux 2.5.37)

Mark the mapped region to be locked in the same way as [mlock\(2\)](#). This implementation will try to populate (prefault) the whole range but the `mmap()` call doesn't fail with **ENOMEM** if this fails. Therefore major faults might happen later on. So the semantic is not as strong as [mlock\(2\)](#). One should use `mmap()` plus [mlock\(2\)](#) when major faults are not acceptable after the initialization of the mapping. The **MAP_LOCKED** flag is ignored in older kernels.

MAP_NONBLOCK (since Linux 2.5.46)

This flag is meaningful only in conjunction with **MAP_POPULATE**. Don't perform read-ahead: create page tables entries only for pages that are already present in RAM. Since Linux 2.6.23, this flag causes **MAP_POPULATE** to do nothing. One day, the combination of **MAP_POPULATE** and **MAP_NONBLOCK** may be reimplemented.

MAP_NORESERVE

Do not reserve swap space for this mapping. When swap space is reserved, one has the guarantee that it is possible to modify the mapping. When swap space is not reserved one might get **SIGSEGV** upon a write if no physical memory is available. See also the discussion of the file [/proc/sys/vm/overcommit_memory](#) in [proc\(5\)](#). Before Linux 2.6, this flag had effect only for private writable mappings.

MAP_POPULATE (since Linux 2.5.46)

Populate (prefault) page tables for a mapping. For a file mapping, this causes read-ahead on the file. This will help to reduce blocking on page faults later. The `mmap()` call doesn't fail if the mapping cannot be populated (for example, due to limitations on the number of mapped huge pages when using **MAP_HUGETLB**). Support for **MAP_POPULATE** in conjunction with private mappings was added in Linux 2.6.23.

MAP_STACK (since Linux 2.6.27)



Allocate the mapping at an address suitable for a process or thread stack.

This flag is currently a no-op on Linux. However, by employing this flag, applications can ensure that they transparently obtain support if the flag is implemented in the future. Thus, it is used in the glibc threading implementation to allow for the fact that some architectures may (later) require special treatment for stack allocations. A further reason to employ this flag is portability: **MAP_STACK** exists (and has an effect) on some other systems (e.g., some of the BSDs).

MAP_SYNC (since Linux 4.15)

This flag is available only with the **MAP_SHARED_VALIDATE** mapping type; mappings of type **MAP_SHARED** will silently ignore this flag. This flag is supported only for files supporting DAX (direct mapping of persistent memory). For other files, creating a mapping with this flag results in an **EOPNOTSUPP** error.

Shared file mappings with this flag provide the guarantee that while some memory is mapped writable in the address space of the process, it will be visible in the same file at the same offset even after the system crashes or is rebooted. In conjunction with the use of appropriate CPU instructions, this provides users of such mappings with a more efficient way of making data modifications persistent.

MAP_UNINITIALIZED (since Linux 2.6.33)

Don't clear anonymous pages. This flag is intended to improve performance on embedded devices. This flag is honored only if the kernel was configured with the **CONFIG_MMAP_ALLOW_UNINITIALIZED** option. Because of the security implications, that option is normally enabled only on embedded devices (i.e., devices where one has complete control of the contents of user memory).

Of the above flags, only **MAP_FIXED** is specified in POSIX.1-2001 and POSIX.1-2008. However, most systems also support **MAP_ANONYMOUS** (or its synonym **MAP_ANON**).

munmap()

The **munmap()** system call deletes the mappings for the specified



address range, and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.

The address *addr* must be a multiple of the page size (but *Length* need not be). All pages containing a part of the indicated range are unmapped, and subsequent references to these pages will generate **SIGSEGV**. It is not an error if the indicated range does not contain any mapped pages.

RETURN VALUE [top](#)

On success, **mmap()** returns a pointer to the mapped area. On error, the value **MAP_FAILED** (that is, *(void *) -1*) is returned, and *errno* is set to indicate the error.

On success, **munmap()** returns 0. On failure, it returns -1, and *errno* is set to indicate the error (probably to **EINVAL**).

ERRORS [top](#)

EACCES A file descriptor refers to a non-regular file. Or a file mapping was requested, but *fd* is not open for reading. Or **MAP_SHARED** was requested and **PROT_WRITE** is set, but *fd* is not open in read/write (**O_RDWR**) mode. Or **PROT_WRITE** is set, but the file is append-only.

EAGAIN The file has been locked, or too much memory has been locked (see [setrlimit\(2\)](#)).

EBADF *fd* is not a valid file descriptor (and **MAP_ANONYMOUS** was not set).

EEXIST **MAP_FIXED_NOREPLACE** was specified in *flags*, and the range covered by *addr* and *Length* clashes with an existing mapping.

EINVAL We don't like *addr*, *Length*, or *offset* (e.g., they are too large, or not aligned on a page boundary).

EINVAL (since Linux 2.6.12) *Length* was 0.



EINVAL *flags* contained none of **MAP_PRIVATE**, **MAP_SHARED**, or **MAP_SHARED_VALIDATE**.

ENFILE The system-wide limit on the total number of open files has been reached.

ENODEV The underlying filesystem of the specified file does not support memory mapping.

ENOMEM No memory is available.

ENOMEM The process's maximum number of mappings would have been exceeded. This error can also occur for **munmap()**, when unmapping a region in the middle of an existing mapping, since this results in two smaller mappings on either side of the region being unmapped.

ENOMEM (since Linux 4.7) The process's **RLIMIT_DATA** limit, described in [getrlimit\(2\)](#), would have been exceeded.

ENOMEM We don't like *addr*, because it exceeds the virtual address space of the CPU.

EOVERFLOW

On 32-bit architecture together with the large file extension (i.e., using 64-bit *off_t*): the number of pages used for *length* plus number of pages used for *offset* would overflow *unsigned Long* (32 bits).

EPERM The *prot* argument asks for **PROT_EXEC** but the mapped area belongs to a file on a filesystem that was mounted no-exec.

EPERM The operation was prevented by a file seal; see [fcntl\(2\)](#).

EPERM The **MAP_HUGETLB** flag was specified, but the caller was not privileged (did not have the **CAP_IPC_LOCK** capability) and is not a member of the *sysctl_hugetlb_shm_group* group; see the description of */proc/sys/vm/sysctl_hugetlb_shm_group* in

ETXTBSY

MAP_DENYWRITE was set but the object specified by *fd* is open for writing.

Use of a mapped region can result in these signals:

SIGSEGV

Attempted write into a region mapped as read-only.

SIGBUS Attempted access to a page of the buffer that lies beyond the end of the mapped file. For an explanation of the treatment of the bytes in the page that corresponds to the end of a mapped file that is not a multiple of the page size, see NOTES.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|---|---------------|---------|
| <code>mmap()</code> , <code>munmap()</code> | Thread safety | MT-Safe |

VERSIONS [top](#)

On some hardware architectures (e.g., i386), **PROT_WRITE** implies **PROT_READ**. It is architecture dependent whether **PROT_READ** implies **PROT_EXEC** or not. Portable programs should always set **PROT_EXEC** if they intend to execute code in the new mapping.

The portable way to create a mapping is to specify *addr* as 0 (NULL), and omit **MAP_FIXED** from *flags*. In this case, the system chooses the address for the mapping; the address is chosen so as not to conflict with any existing mapping, and will not be 0. If the **MAP_FIXED** flag is specified, and *addr* is 0 (NULL), then the mapped address will be 0 (NULL).

Certain *flags* constants are defined only if suitable feature test macros are defined (possibly by default): **_DEFAULT_SOURCE** with glibc 2.19 or later; or **_BSD_SOURCE** or **_SVID_SOURCE** in glibc 2.19 and earlier. (Employing **_GNU_SOURCE** also suffices, and requiring that macro specifically would have been more logical, since these flags are all Linux-specific.) The relevant flags are: **MAP_32BIT**, **MAP_ANONYMOUS** (and the synonym **MAP_ANON**),



MAP_DENYWRITE, **MAP_EXECUTABLE**, **MAP_FILE**, **MAP_GROWSDOWN**, **MAP_HUGETLB**, **MAP_LOCKED**, **MAP_NONBLOCK**, **MAP_NORESERVE**, **MAP_POPULATE**, and **MAP_STACK**.

C library/kernel differences

This page describes the interface provided by the glibc **mmap()** wrapper function. Originally, this function invoked a system call of the same name. Since Linux 2.4, that system call has been superseded by **mmap2(2)**, and nowadays the glibc **mmap()** wrapper function invokes **mmap2(2)** with a suitably adjusted value for *offset*.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.4BSD.

On POSIX systems on which **mmap()**, **msync(2)**, and **munmap()** are available, **_POSIX_MAPPED_FILES** is defined in *<unistd.h>* to a value greater than 0. (See also **sysconf(3)**.)

NOTES [top](#)

Memory mapped by **mmap()** is preserved across **fork(2)**, with the same attributes.

A file is mapped in multiples of the page size. For a file that is not a multiple of the page size, the remaining bytes in the partial page at the end of the mapping are zeroed when mapped, and modifications to that region are not written out to the file. The effect of changing the size of the underlying file of a mapping on the pages that correspond to added or removed regions of the file is unspecified.

An application can determine which pages of a mapping are currently resident in the buffer/page cache using **mincore(2)**.

Using **MAP_FIXED** safely

The only safe use for **MAP_FIXED** is where the address range

specified by *addr* and *length* was previously reserved using another mapping; otherwise, the use of **MAP_FIXED** is hazardous because it forcibly removes preexisting mappings, making it easy for a multithreaded process to corrupt its own address space.

For example, suppose that thread A looks through */proc/pid/maps* in order to locate an unused address range that it can map using **MAP_FIXED**, while thread B simultaneously acquires part or all of that same address range. When thread A subsequently employs **mmap(MAP_FIXED)**, it will effectively clobber the mapping that thread B created. In this scenario, thread B need not create a mapping directly; simply making a library call that, internally, uses **dlopen(3)** to load some other shared library, will suffice. The **dlopen(3)** call will map the library into the process's address space. Furthermore, almost any library call may be implemented in a way that adds memory mappings to the address space, either with this technique, or by simply allocating memory. Examples include **brk(2)**, **malloc(3)**, **pthread_create(3)**, and the PAM libraries (<http://www.linux-pam.org>).

Since Linux 4.17, a multithreaded program can use the **MAP_FIXED_NO_REPLACE** flag to avoid the hazard described above when attempting to create a mapping at a fixed address that has not been reserved by a preexisting mapping.

Timestamps changes for file-backed mappings

For file-backed mappings, the *st_atime* field for the mapped file may be updated at any time between the **mmap()** and the corresponding unmapping; the first reference to a mapped page will update the field if it has not been already.

The *st_ctime* and *st_mtime* field for a file mapped with **PROT_WRITE** and **MAP_SHARED** will be updated after a write to the mapped region, and before a subsequent **msync(2)** with the **MS_SYNC** or **MS_ASYNC** flag, if one occurs.

Huge page (Huge TLB) mappings

For mappings that employ huge pages, the requirements for the arguments of **mmap()** and **munmap()** differ somewhat from the requirements for mappings that use the native system page size.

For **mmap()**, *offset* must be a multiple of the underlying huge page size. The system automatically aligns *length* to be a multiple of the underlying huge page size.



For `mmap()`, `addr`, and `length` must both be a multiple of the underlying huge page size.

BUGS [top](#)

On Linux, there are no guarantees like those suggested above under `MAP_NORESERVE`. By default, any process can be killed at any moment when the system runs out of memory.

Before Linux 2.6.7, the `MAP_POPULATE` flag has effect only if `prot` is specified as `PROT_NONE`.

SUSv3 specifies that `mmap()` should fail if `length` is 0. However, before Linux 2.6.12, `mmap()` succeeded in this case: no mapping was created and the call returned `addr`. Since Linux 2.6.12, `mmap()` fails with the error `EINVAL` for this case.

POSIX specifies that the system shall always zero fill any partial page at the end of the object and that system will never write any modification of the object beyond its end. On Linux, when you write data to such partial page after the end of the object, the data stays in the page cache even after the file is closed and unmapped and even though the data is never written to the file itself, subsequent mappings may see the modified content. In some cases, this could be fixed by calling `msync(2)` before the unmap takes place; however, this doesn't work on `tmpfs(5)` (for example, when using the POSIX shared memory interface documented in `shm_overview(7)`).

EXAMPLES [top](#)

The following program prints part of the file specified in its first command-line argument to standard output. The range of bytes to be printed is specified via offset and length values in the second and third command-line arguments. The program creates a memory mapping of the required pages of the file and then uses `write(2)` to output the desired bytes.

Program source

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
```



```
#include <sys/stat.h>
#include <unistd.h>

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

int
main(int argc, char *argv[])
{
    int          fd;
    char         *addr;
    off_t        offset, pa_offset;
    size_t       length;
    ssize_t      s;
    struct stat  sb;

    if (argc < 3 || argc > 4) {
        fprintf(stderr, "%s file offset [length]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    fd = open(argv[1], O_RDONLY);
    if (fd == -1)
        handle_error("open");

    if (fstat(fd, &sb) == -1)                /* To obtain file size */
        handle_error("fstat");

    offset = atoi(argv[2]);
    pa_offset = offset & ~(sysconf(_SC_PAGE_SIZE) - 1);
    /* offset for mmap() must be page aligned */

    if (offset >= sb.st_size) {
        fprintf(stderr, "offset is past end of file\n");
        exit(EXIT_FAILURE);
    }

    if (argc == 4) {
        length = atoi(argv[3]);
        if (offset + length > sb.st_size)
            length = sb.st_size - offset;
        /* Can't display bytes past end of file */
    } else { /* No length arg ==> display to end of file */
        length = sb.st_size - offset;
    }
}
```



```
    }

    addr = mmap(NULL, length + offset - pa_offset, PROT_READ,
                MAP_PRIVATE, fd, pa_offset);
    if (addr == MAP_FAILED)
        handle_error("mmap");

    s = write(STDOUT_FILENO, addr + offset - pa_offset, length);
    if (s != length) {
        if (s == -1)
            handle_error("write");

        fprintf(stderr, "partial write");
        exit(EXIT_FAILURE);
    }

    munmap(addr, length + offset - pa_offset);
    close(fd);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO [top](#)

[ftruncate\(2\)](#), [getpagesize\(2\)](#), [memfd_create\(2\)](#), [mincore\(2\)](#), [mlock\(2\)](#), [mmap2\(2\)](#), [mprotect\(2\)](#), [mremap\(2\)](#), [msync\(2\)](#), [remap_file_pages\(2\)](#), [setrlimit\(2\)](#), [shmat\(2\)](#), [userfaultfd\(2\)](#), [shm_open\(3\)](#), [shm_overview\(7\)](#)

The descriptions of the following files in [proc\(5\)](#):
[/proc/pid/maps](#), [/proc/pid/map_files](#), and [/proc/pid/smaps](#).

B.O. Gallmeister, POSIX.4, O'Reilly, pp. 128–129 and 389–391.

Linux man-pages (unreleased) (date) [mmap\(2\)](#)

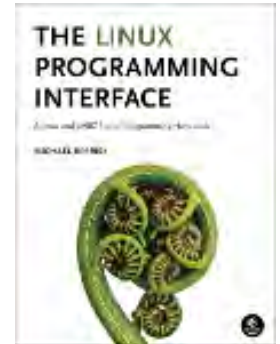
Pages that refer to this page: [memusage\(1\)](#), [alloc_hugepages\(2\)](#), [arch_prctl\(2\)](#), [clone\(2\)](#), [execve\(2\)](#), [fcntl\(2\)](#), [fork\(2\)](#), [futux\(2\)](#), [get_mempolicy\(2\)](#), [getpagesize\(2\)](#), [getrlimit\(2\)](#), [ioctl_userfaultfd\(2\)](#), [io_uring_register\(2\)](#), [io_uring_setup\(2\)](#), [madvise\(2\)](#), [mbind\(2\)](#), [memfd_create\(2\)](#), [memfd_secret\(2\)](#), [mincore\(2\)](#), [mlock\(2\)](#), [mmap2\(2\)](#), [mprotect\(2\)](#), [mremap\(2\)](#), [msync\(2\)](#), [open\(2\)](#), [perf_event_open\(2\)](#), [personality\(2\)](#), [posix_fadvise\(2\)](#), [prctl\(2\)](#), [readahead\(2\)](#), [remap_file_pages\(2\)](#), [seccomp\(2\)](#), [sendfile\(2\)](#), [set_mempolicy\(2\)](#), [shmget\(2\)](#), [shmop\(2\)](#), [statx\(2\)](#), [syscalls\(2\)](#), [uselib\(2\)](#), [userfaultfd\(2\)](#), [vfork\(2\)](#), [avc_init\(3\)](#), [avc_open\(3\)](#), [cap_launch\(3\)](#), [fopen\(3\)](#),

[io_uring_queue_exit\(3\)](#), [io_uring_queue_init\(3\)](#), [io_uring_queue_init_mem\(3\)](#), [io_uring_queue_init_params\(3\)](#), [mallinfo\(3\)](#), [malloc\(3\)](#), [malloc_stats\(3\)](#), [mallopt\(3\)](#), [numa\(3\)](#), [off_t\(3type\)](#), [pthread_attr_setguardsize\(3\)](#), [pthread_attr_setstack\(3\)](#), [selinux_status_open\(3\)](#), [sem_init\(3\)](#), [shm_open\(3\)](#), [core\(5\)](#), [proc\(5\)](#), [systemd.exec\(5\)](#), [tmpfs\(5\)](#), [capabilities\(7\)](#), [fanotify\(7\)](#), [file-hierarchy\(7\)](#), [futex\(7\)](#), [inode\(7\)](#), [inotify\(7\)](#), [io_uring\(7\)](#), [pkeys\(7\)](#), [shm_overview\(7\)](#), [spufs\(7\)](#), [ld.so\(8\)](#), [netsniff-ng\(8\)](#), [setarch\(8\)](#), [trafgen\(8\)](#), [xfs_io\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sysconf(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [BUGS](#) | [SEE ALSO](#)

 sysconf(3)

Library Functions Manual

sysconf(3)

NAME [top](#)

sysconf - get configuration information at run time

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
long sysconf(int name);
```

DESCRIPTION [top](#)

POSIX allows an application to test at compile or run time whether certain options are supported, or what the value is of certain configurable constants or limits.

At compile time this is done by including *<unistd.h>* and/or *<limits.h>* and testing the value of certain macros.

At run time, one can ask for numerical values using the present function **sysconf()**. One can ask for numerical values that may depend on the filesystem in which a file resides using

`fpathconf(3)` and `pathconf(3)`. One can ask for string values using `confstr(3)`.

The values obtained from these functions are system configuration constants. They do not change during the lifetime of a process.

For options, typically, there is a constant `_POSIX_FOO` that may be defined in `<unistd.h>`. If it is undefined, one should ask at run time. If it is defined to `-1`, then the option is not supported. If it is defined to `0`, then relevant functions and headers exist, but one has to ask at run time what degree of support is available. If it is defined to a value other than `-1` or `0`, then the option is supported. Usually the value (such as `200112L`) indicates the year and month of the POSIX revision describing the option. `glibc` uses the value `1` to indicate support as long as the POSIX revision has not been published yet. The `sysconf()` argument will be `_SC_FOO`. For a list of options, see `posixoptions(7)`.

For variables or limits, typically, there is a constant `_FOO`, maybe defined in `<limits.h>`, or `_POSIX_FOO`, maybe defined in `<unistd.h>`. The constant will not be defined if the limit is unspecified. If the constant is defined, it gives a guaranteed value, and a greater value might actually be supported. If an application wants to take advantage of values which may change between systems, a call to `sysconf()` can be made. The `sysconf()` argument will be `_SC_FOO`.

POSIX.1 variables

We give the name of the variable, the name of the `sysconf()` argument used to inquire about its value, and a short description.

First, the POSIX.1 compatible values.

ARG_MAX - `_SC_ARG_MAX`

The maximum length of the arguments to the `exec(3)` family of functions. Must not be less than `_POSIX_ARG_MAX` (4096).

CHILD_MAX - `_SC_CHILD_MAX`

The maximum number of simultaneous processes per user ID. Must not be less than `_POSIX_CHILD_MAX` (25).



HOST_NAME_MAX - _SC_HOST_NAME_MAX

Maximum length of a hostname, not including the terminating null byte, as returned by [gethostname\(2\)](#). Must not be less than **_POSIX_HOST_NAME_MAX** (255).

LOGIN_NAME_MAX - _SC_LOGIN_NAME_MAX

Maximum length of a login name, including the terminating null byte. Must not be less than **_POSIX_LOGIN_NAME_MAX** (9).

NGROUPS_MAX - _SC_NGROUPS_MAX

Maximum number of supplementary group IDs.

clock ticks - _SC_CLK_TCK

The number of clock ticks per second. The corresponding variable is obsolete. It was of course called **CLK_TCK**. (Note: the macro **CLOCKS_PER_SEC** does not give information: it must equal 1000000.)

OPEN_MAX - _SC_OPEN_MAX

The maximum number of files that a process can have open at any time. Must not be less than **_POSIX_OPEN_MAX** (20).

PAGESIZE - _SC_PAGESIZE

Size of a page in bytes. Must not be less than 1.

PAGE_SIZE - _SC_PAGE_SIZE

A synonym for **PAGESIZE/_SC_PAGESIZE**. (Both **PAGESIZE** and **PAGE_SIZE** are specified in POSIX.)

RE_DUP_MAX - _SC_RE_DUP_MAX

The number of repeated occurrences of a BRE permitted by [regex\(3\)](#) and [regcomp\(3\)](#). Must not be less than **_POSIX2_RE_DUP_MAX** (255).

STREAM_MAX - _SC_STREAM_MAX

The maximum number of streams that a process can have open at any time. If defined, it has the same value as the standard C macro **FOPEN_MAX**. Must not be less than **_POSIX_STREAM_MAX** (8).

SYMLOOP_MAX - _SC_SYMLOOP_MAX

The maximum number of symbolic links seen in a pathname before resolution returns **ELOOP**. Must not be less than



__POSIX_SYMLINK_MAX (8).

TTY_NAME_MAX - **__SC_TTY_NAME_MAX**

The maximum length of terminal device name, including the terminating null byte. Must not be less than **__POSIX_TTY_NAME_MAX** (9).

TZNAME_MAX - **__SC_TZNAME_MAX**

The maximum number of bytes in a timezone name. Must not be less than **__POSIX_TZNAME_MAX** (6).

__POSIX_VERSION - **__SC_VERSION**

indicates the year and month the POSIX.1 standard was approved in the format **YYMM**; the value **199009L** indicates the Sept. 1990 revision.

POSIX.2 variables

Next, the POSIX.2 values, giving limits for utilities.

BC_BASE_MAX - **__SC_BC_BASE_MAX**

indicates the maximum *obase* value accepted by the **bc(1)** utility.

BC_DIM_MAX - **__SC_BC_DIM_MAX**

indicates the maximum value of elements permitted in an array by **bc(1)**.

BC_SCALE_MAX - **__SC_BC_SCALE_MAX**

indicates the maximum *scale* value allowed by **bc(1)**.

BC_STRING_MAX - **__SC_BC_STRING_MAX**

indicates the maximum length of a string accepted by **bc(1)**.

COLL_WEIGHTS_MAX - **__SC_COLL_WEIGHTS_MAX**

indicates the maximum numbers of weights that can be assigned to an entry of the **LC_COLLATE order** keyword in the locale definition file.

EXPR_NEST_MAX - **__SC_EXPR_NEST_MAX**

is the maximum number of expressions which can be nested within parentheses by **expr(1)**.

LINE_MAX - **__SC_LINE_MAX**



The maximum length of a utility's input line, either from standard input or from a file. This includes space for a trailing newline.

RE_DUP_MAX - _SC_RE_DUP_MAX

The maximum number of repeated occurrences of a regular expression when the interval notation `\{m,n\}` is used.

POSIX2_VERSION - _SC_2_VERSION

indicates the version of the POSIX.2 standard in the format of YYYYMMML.

POSIX2_C_DEV - _SC_2_C_DEV

indicates whether the POSIX.2 C language development facilities are supported.

POSIX2_FORT_DEV - _SC_2_FORT_DEV

indicates whether the POSIX.2 FORTRAN development utilities are supported.

POSIX2_FORT_RUN - _SC_2_FORT_RUN

indicates whether the POSIX.2 FORTRAN run-time utilities are supported.

_POSIX2_LOCALEDEF - _SC_2_LOCALEDEF

indicates whether the POSIX.2 creation of locales via [localedef\(1\)](#) is supported.

POSIX2_SW_DEV - _SC_2_SW_DEV

indicates whether the POSIX.2 software development utilities option is supported.

These values also exist, but may not be standard.

- _SC_PHYS_PAGES

The number of pages of physical memory. Note that it is possible for the product of this value and the value of **_SC_PAGESIZE** to overflow.

- _SC_AVPHYS_PAGES

The number of currently available pages of physical memory.

- _SC_NPROCESSORS_CONF

The number of processors configured. See also [get_nprocs_conf\(3\)](#).

- **_SC_NPROCESSORS_ONLN**

The number of processors currently online (available). See also [get_nprocs_conf\(3\)](#).

RETURN VALUE [top](#)

The return value of `sysconf()` is one of the following:

- On error, -1 is returned and `errno` is set to indicate the error (for example, `EINVAL`, indicating that `name` is invalid).
- If `name` corresponds to a maximum or minimum limit, and that limit is indeterminate, -1 is returned and `errno` is not changed. (To distinguish an indeterminate limit from an error, set `errno` to zero before the call, and then check whether `errno` is nonzero when -1 is returned.)
- If `name` corresponds to an option, a positive value is returned if the option is supported, and -1 is returned if the option is not supported.
- Otherwise, the current value of the option or limit is returned. This value will not be more restrictive than the corresponding value that was described to the application in `<unistd.h>` or `<limits.h>` when the application was compiled.

ERRORS [top](#)

`EINVAL` `name` is invalid.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------|-----------|-------|
| | | |



| | | |
|------------------|---------------|-------------|
| sysconf() | Thread safety | MT-Safe env |
|------------------|---------------|-------------|

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

BUGS [top](#)

It is difficult to use **ARG_MAX** because it is not specified how much of the argument space for [exec\(3\)](#) is consumed by the user's environment variables.

Some returned values may be huge; they are not suitable for allocating memory.

SEE ALSO [top](#)

[bc\(1\)](#), [expr\(1\)](#), [getconf\(1\)](#), [locale\(1\)](#), [confstr\(3\)](#), [fpathconf\(3\)](#), [pathconf\(3\)](#), [posixoptions\(7\)](#)

Linux man-pages (unreleased) (date)

[sysconf\(3\)](#)

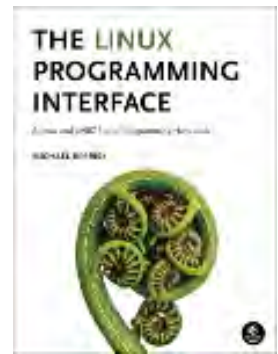
Pages that refer to this page: [systemctl\(1\)](#), [clock_getres\(2\)](#), [execve\(2\)](#), [fsync\(2\)](#), [getgroups\(2\)](#), [getpagesize\(2\)](#), [mlock\(2\)](#), [mmap\(2\)](#), [mprotect\(2\)](#), [msync\(2\)](#), [sched_setaffinity\(2\)](#), [times\(2\)](#), [atexit\(3\)](#), [confstr\(3\)](#), [crypt\(3\)](#), [fpathconf\(3\)](#), [get_nprocs\(3\)](#), [get_phys_pages\(3\)](#), [realpath\(3\)](#), [ulimit\(3\)](#), [posixoptions\(7\)](#), [signal-safety\(7\)](#), [standards\(7\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



getpagesize(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

getpagesize(2)

System Calls Manual

getpagesize(2)

NAME [top](#)

getpagesize - get memory page size

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int getpagesize(void);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

getpagesize():

Since glibc 2.20:

```
    _DEFAULT_SOURCE || ! (_POSIX_C_SOURCE >= 200112L)  
glibc 2.12 to glibc 2.19:
```

```
    _BSD_SOURCE || ! (_POSIX_C_SOURCE >= 200112L)
```

Before glibc 2.12:

```
    _BSD_SOURCE || _XOPEN_SOURCE >= 500
```

DESCRIPTION [top](#)

The function **getpagesize()** returns the number of bytes in a memory page, where "page" is a fixed-length block, the unit for memory allocation and file mapping performed by **mmap(2)**.

STANDARDS [top](#)

None.

HISTORY [top](#)

This call first appeared in 4.2BSD. SVr4, 4.4BSD, SUSv2. In SUSv2 the **getpagesize()** call is labeled LEGACY, and in POSIX.1-2001 it has been dropped; HP-UX does not have this call.

NOTES [top](#)

Portable applications should employ *sysconf(_SC_PAGESIZE)* instead of **getpagesize()**:

```
#include <unistd.h>
long sz = sysconf(_SC_PAGESIZE);
```

(Most systems allow the synonym **_SC_PAGE_SIZE** for **_SC_PAGESIZE**.)

Whether **getpagesize()** is present as a Linux system call depends on the architecture. If it is, it returns the kernel symbol **PAGE_SIZE**, whose value depends on the architecture and machine model. Generally, one uses binaries that are dependent on the architecture but not on the machine model, in order to have a single binary distribution per architecture. This means that a user program should not find **PAGE_SIZE** at compile time from a header file, but use an actual system call, at least for those architectures (like sun4) where this dependency exists. Here glibc 2.0 fails because its **getpagesize()** returns a statically derived value, and does not use a system call. Things are OK in glibc 2.1.

SEE ALSO [top](#)

[mmap\(2\)](#), [sysconf\(3\)](#)

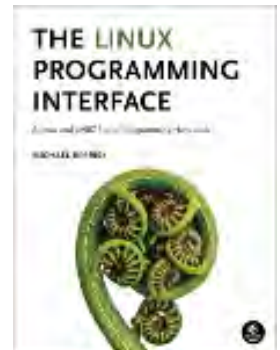
Linux man-pages (unreleased)**(date)*****getpagesize(2)***

Pages that refer to this page: [fcore\(1\)](#), [strace\(1\)](#), [mmap2\(2\)](#), [mmap\(2\)](#), [mremap\(2\)](#), [remap_file_pages\(2\)](#), [syscalls\(2\)](#), [numa\(3\)](#), [posix_memalign\(3\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fstat(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 FSTAT(3P)

POSIX Programmer's Manual

FSTAT(3P)**PROLOG** [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

fstat – get file status

SYNOPSIS [top](#)

```
#include <sys/stat.h>

int fstat(int fildes, struct stat *buf);
```

DESCRIPTION [top](#)

The *fstat()* function shall obtain information about an open file associated with the file descriptor *fildes*, and shall write it to the area pointed to by *buf*.

If *fildes* references a shared memory object, the implementation shall update in the **stat** structure pointed to by the *buf* argument the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the

S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be valid. The implementation may update other fields and flags.

If *fildes* references a typed memory object, the implementation shall update in the **stat** structure pointed to by the *buf* argument the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be valid. The implementation may update other fields and flags.

The *buf* argument is a pointer to a **stat** structure, as defined in `<sys/stat.h>`, into which information is placed concerning the file.

For all other file types defined in this volume of POSIX.1-2017, the structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atim*, *st_ctim*, and *st_mtim* shall have meaningful values and the value of the *st_nlink* member shall be set to the number of links to the file.

An implementation that provides additional or alternative file access control mechanisms may, under implementation-defined conditions, cause *fstat()* to fail.

The *fstat()* function shall update any time-related fields (as described in the Base Definitions volume of POSIX.1-2017, *Section 4.9, File Times Update*), before writing into the **stat** structure.

RETURN VALUE [top](#)

Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.

ERRORS [top](#)

The *fstat()* function shall fail if:

EBADF The *fildes* argument is not a valid file descriptor.

EIO An I/O error occurred while reading from the file system.

EOVERFLOW

The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

The *fstat()* function may fail if:

EOVERFLOW

One of the values is too large to store into the structure pointed to by the *buf* argument.

The following sections are informative.

EXAMPLES [top](#)

Obtaining File Status Information

The following example shows how to obtain file status information for a file named `/home/cnd/mod1`. The structure variable *buffer* is defined for the `stat` structure. The `/home/cnd/mod1` file is opened with read/write privileges and is passed to the open file descriptor *fildes*.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

struct stat buffer;
int      status;
...
fildes = open("/home/cnd/mod1", O_RDWR);
status = fstat(fildes, &buffer);
```

APPLICATION USAGE [top](#)

None.

RATIONALE [top](#)

None.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

[fstatat\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, *Section 4.9, File Times Update*, [sys_stat.h\(0p\)](#), [sys_types.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

IEEE/The Open Group

2017

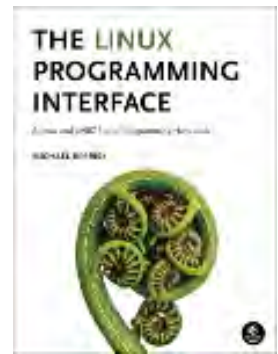
FSTAT(3P)

Pages that refer to this page: [sys_stat.h\(0p\)](#), [fstatat\(3p\)](#), [posix_typed_mem_get_info\(3p\)](#), [posix_typed_mem_open\(3p\)](#), [utime\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *[The Linux Programming Interface](#)*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



putchar(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 PUTCHAR(3P)

POSIX Programmer's Manual

PUTCHAR(3P)**PROLOG** [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

putchar – put a byte on a stdout stream

SYNOPSIS [top](#)

```
#include <stdio.h>

int putchar(int c);
```

DESCRIPTION [top](#)

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The function call `putchar(c)` shall be equivalent to `putc(c, stdout)`.

RETURN VALUE [top](#)

Refer to `fputc(3p)`.

ERRORS [top](#)

Refer to `fputc(3p)`.

The following sections are informative.

EXAMPLES [top](#)

None.

APPLICATION USAGE [top](#)

None.

RATIONALE [top](#)

None.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

Section 2.5, Standard I/O Streams, [putc\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [stdio.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The



Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

IEEE/The Open Group

2017

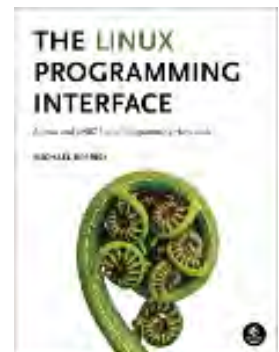
PUTCHAR(3P)

Pages that refer to this page: [stdio.h\(0p\)](#), [getc_unlocked\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



mremap(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

 [mremap\(2\)](#)

System Calls Manual

[mremap\(2\)](#)

NAME [top](#)

mremap - remap a virtual memory address

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#define _GNU_SOURCE          /* See feature_test_macros(7) */
#include <sys/mman.h>

void *mremap(void old_address[.old_size], size_t old_size,
             size_t new_size, int flags, ... /* void *new_address */);
```

DESCRIPTION [top](#)

`mremap()` expands (or shrinks) an existing memory mapping, potentially moving it at the same time (controlled by the *flags* argument and the available virtual address space).

old_address is the old address of the virtual memory block that you want to expand (or shrink). Note that *old_address* has to be page aligned. *old_size* is the old size of the virtual memory block. *new_size* is the requested size of the virtual memory block after the resize. An optional fifth argument, *new_address*, may be provided; see the description of **MREMAP_FIXED** below.

If the value of *old_size* is zero, and *old_address* refers to a



shareable mapping (see `mmap(2)` `MAP_SHARED`), then `mremap()` will create a new mapping of the same pages. `new_size` will be the size of the new mapping and the location of the new mapping may be specified with `new_address`; see the description of `MREMAP_FIXED` below. If a new mapping is requested via this method, then the `MREMAP_MAYMOVE` flag must also be specified.

The `flags` bit-mask argument may be 0, or include the following flags:

`MREMAP_MAYMOVE`

By default, if there is not sufficient space to expand a mapping at its current location, then `mremap()` fails. If this flag is specified, then the kernel is permitted to relocate the mapping to a new virtual address, if necessary. If the mapping is relocated, then absolute pointers into the old mapping location become invalid (offsets relative to the starting address of the mapping should be employed).

`MREMAP_FIXED` (since Linux 2.3.31)

This flag serves a similar purpose to the `MAP_FIXED` flag of `mmap(2)`. If this flag is specified, then `mremap()` accepts a fifth argument, `void *new_address`, which specifies a page-aligned address to which the mapping must be moved. Any previous mapping at the address range specified by `new_address` and `new_size` is unmapped.

If `MREMAP_FIXED` is specified, then `MREMAP_MAYMOVE` must also be specified.

`MREMAP_DONTUNMAP` (since Linux 5.7)

This flag, which must be used in conjunction with `MREMAP_MAYMOVE`, remaps a mapping to a new address but does not unmap the mapping at `old_address`.

The `MREMAP_DONTUNMAP` flag can be used only with private anonymous mappings (see the description of `MAP_PRIVATE` and `MAP_ANONYMOUS` in `mmap(2)`).

After completion, any access to the range specified by `old_address` and `old_size` will result in a page fault. The page fault will be handled by a `userfaultfd(2)` handler if the address is in a range previously registered with `userfaultfd(2)`. Otherwise, the kernel allocates a zero-filled page to handle the fault.



The **MREMAP_DONTUNMAP** flag may be used to atomically move a mapping while leaving the source mapped. See NOTES for some possible applications of **MREMAP_DONTUNMAP**.

If the memory segment specified by *old_address* and *old_size* is locked (using `mlock(2)` or similar), then this lock is maintained when the segment is resized and/or relocated. As a consequence, the amount of memory locked by the process may change.

RETURN VALUE [top](#)

On success `mremap()` returns a pointer to the new virtual memory area. On error, the value **MAP_FAILED** (that is, `(void *) -1`) is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EAGAIN The caller tried to expand a memory segment that is locked, but this was not possible without exceeding the **RLIMIT_MEMLOCK** resource limit.

EFAULT Some address in the range *old_address* to *old_address+old_size* is an invalid virtual memory address for this process. You can also get **EFAULT** even if there exist mappings that cover the whole address space requested, but those mappings are of different types.

EINVAL An invalid argument was given. Possible causes are:

- *old_address* was not page aligned;
- a value other than **MREMAP_MAYMOVE** or **MREMAP_FIXED** or **MREMAP_DONTUNMAP** was specified in *flags*;
- *new_size* was zero;
- *new_size* or *new_address* was invalid;
- the new address range specified by *new_address* and *new_size* overlapped the old address range specified by *old_address* and *old_size*;
- **MREMAP_FIXED** or **MREMAP_DONTUNMAP** was specified without also specifying **MREMAP_MAYMOVE**;

- **MREMAP_DONTUNMAP** was specified, but one or more pages in the range specified by *old_address* and *old_size* were not private anonymous;
- **MREMAP_DONTUNMAP** was specified and *old_size* was not equal to *new_size*;
- *old_size* was zero and *old_address* does not refer to a shareable mapping (but see BUGS);
- *old_size* was zero and the **MREMAP_MAYMOVE** flag was not specified.

ENOMEM Not enough memory was available to complete the operation. Possible causes are:

- The memory area cannot be expanded at the current virtual address, and the **MREMAP_MAYMOVE** flag is not set in *flags*. Or, there is not enough (virtual) memory available.
- **MREMAP_DONTUNMAP** was used causing a new mapping to be created that would exceed the (virtual) memory available. Or, it would exceed the maximum number of allowed mappings.

STANDARDS [top](#)

Linux.

HISTORY [top](#)

Prior to glibc 2.4, glibc did not expose the definition of **MREMAP_FIXED**, and the prototype for `mremap()` did not allow for the *new_address* argument.

NOTES [top](#)

`mremap()` changes the mapping between virtual addresses and memory pages. This can be used to implement a very efficient `realloc(3)`.

In Linux, memory is divided into pages. A process has (one or) several linear virtual memory segments. Each virtual memory segment has one or more mappings to real memory pages (in the

page table). Each virtual memory segment has its own protection (access rights), which may cause a segmentation violation (**SIGSEGV**) if the memory is accessed incorrectly (e.g., writing to a read-only segment). Accessing virtual memory outside of the segments will also cause a segmentation violation.

If **mremap()** is used to move or expand an area locked with **mlock(2)** or equivalent, the **mremap()** call will make a best effort to populate the new area but will not fail with **ENOMEM** if the area cannot be populated.

MREMAP_DONTUNMAP use cases

Possible applications for **MREMAP_DONTUNMAP** include:

- Non-cooperative **userfaultfd(2)**: an application can yank out a virtual address range using **MREMAP_DONTUNMAP** and then employ a **userfaultfd(2)** handler to handle the page faults that subsequently occur as other threads in the process touch pages in the yanked range.
- Garbage collection: **MREMAP_DONTUNMAP** can be used in conjunction with **userfaultfd(2)** to implement garbage collection algorithms (e.g., in a Java virtual machine). Such an implementation can be cheaper (and simpler) than conventional garbage collection techniques that involve marking pages with protection **PROT_NONE** in conjunction with the use of a **SIGSEGV** handler to catch accesses to those pages.

BUGS [top](#)

Before Linux 4.14, if *old_size* was zero and the mapping referred to by *old_address* was a private mapping (**mmap(2)** **MAP_PRIVATE**), **mremap()** created a new private mapping unrelated to the original mapping. This behavior was unintended and probably unexpected in user-space applications (since the intention of **mremap()** is to create a new mapping based on the original mapping). Since Linux 4.14, **mremap()** fails with the error **EINVAL** in this scenario.

SEE ALSO [top](#)

[brk\(2\)](#), [getpagesize\(2\)](#), [getrlimit\(2\)](#), [mlock\(2\)](#), [mmap\(2\)](#), [sbrk\(2\)](#), [malloc\(3\)](#), [realloc\(3\)](#)

Your favorite text book on operating systems for more information on paged memory (e.g., *Modern Operating Systems* by Andrew S.



Tanenbaum, *Inside Linux* by Randolph Bentson, *The Design of the UNIX Operating System* by Maurice J. Bach)

Linux man-pages (unreleased) (date)

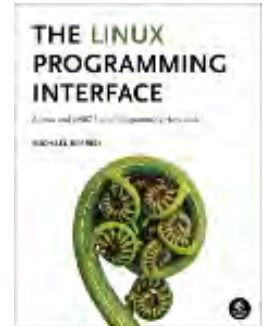
mremap(2)

Pages that refer to this page: [memusage\(1\)](#), [getrlimit\(2\)](#), [ioctl_userfaultfd\(2\)](#), [mmap2\(2\)](#), [mmap\(2\)](#), [prctl\(2\)](#), [remap_file_pages\(2\)](#), [syscalls\(2\)](#), [userfaultfd\(2\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



mprotect(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [EXAMPLES](#) | [SEE ALSO](#)

 mprotect(2)

System Calls Manual

mprotect(2)**NAME** [top](#)

mprotect, pkey_mprotect - set protection on a region of memory

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/mman.h>

int mprotect(void addr[.Len], size_t Len, int prot);

#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <sys/mman.h>

int pkey_mprotect(void addr[.Len], size_t Len, int prot, int pkey);
```

DESCRIPTION [top](#)

mprotect() changes the access protections for the calling process's memory pages containing any part of the address range in the interval [*addr*, *addr+Len-1*]. *addr* must be aligned to a page boundary.

If the calling process tries to access memory in a manner that violates the protections, then the kernel generates a **SIGSEGV**

signal for the process.

prot is a combination of the following access flags: **PROT_NONE** or a bitwise OR of the other values in the following list:

PROT_NONE

The memory cannot be accessed at all.

PROT_READ

The memory can be read.

PROT_WRITE

The memory can be modified.

PROT_EXEC

The memory can be executed.

PROT_SEM (since Linux 2.5.7)

The memory can be used for atomic operations. This flag was introduced as part of the `futex(2)` implementation (in order to guarantee the ability to perform atomic operations required by commands such as **FUTEX_WAIT**), but is not currently used in on any architecture.

PROT_SAO (since Linux 2.6.26)

The memory should have strong access ordering. This feature is specific to the PowerPC architecture (version 2.06 of the architecture specification adds the SAO CPU feature, and it is available on POWER 7 or PowerPC A2, for example).

Additionally (since Linux 2.6.0), *prot* can have one of the following flags set:

PROT_GROWSUP

Apply the protection mode up to the end of a mapping that grows upwards. (Such mappings are created for the stack area on architectures—for example, HP-PARISC—that have an upwardly growing stack.)

PROT_GROWSDOWN

Apply the protection mode down to the beginning of a mapping that grows downward (which should be a stack segment or a segment mapped with the **MAP_GROWSDOWN** flag set).

Like `mprotect()`, `pkey_mprotect()` changes the protection on the pages specified by `addr` and `Len`. The `pkey` argument specifies the protection key (see `pkeys(7)`) to assign to the memory. The protection key must be allocated with `pkey_alloc(2)` before it is passed to `pkey_mprotect()`. For an example of the use of this system call, see `pkeys(7)`.

RETURN VALUE [top](#)

On success, `mprotect()` and `pkey_mprotect()` return zero. On error, these system calls return -1, and `errno` is set to indicate the error.

ERRORS [top](#)

- EACCES** The memory cannot be given the specified access. This can happen, for example, if you `mmap(2)` a file to which you have read-only access, then ask `mprotect()` to mark it `PROT_WRITE`.
- EINVAL** `addr` is not a valid pointer, or not a multiple of the system page size.
- EINVAL** (`pkey_mprotect()`) `pkey` has not been allocated with `pkey_alloc(2)`
- EINVAL** Both `PROT_GROWSUP` and `PROT_GROWSDOWN` were specified in `prot`.
- EINVAL** Invalid flags specified in `prot`.
- EINVAL** (PowerPC architecture) `PROT_SAO` was specified in `prot`, but SAO hardware feature is not available.
- ENOMEM** Internal kernel structures could not be allocated.
- ENOMEM** Addresses in the range [`addr`, `addr+Len-1`] are invalid for the address space of the process, or specify one or more pages that are not mapped. (Before Linux 2.4.19, the error `EFAULT` was incorrectly produced for these cases.)
- ENOMEM** Changing the protection of a memory region would result in the total number of mappings with distinct attributes



(e.g., read versus read/write protection) exceeding the allowed maximum. (For example, making the protection of a range **PROT_READ** in the middle of a region currently protected as **PROT_READ|PROT_WRITE** would result in three mappings: two read/write mappings at each end and a read-only mapping in the middle.)

VERSIONS [top](#)

POSIX says that the behavior of **mprotect()** is unspecified if it is applied to a region of memory that was not obtained via [mmap\(2\)](#).

On Linux, it is always permissible to call **mprotect()** on any address in a process's address space (except for the kernel vsyscall area). In particular, it can be used to change existing code mappings to be writable.

Whether **PROT_EXEC** has any effect different from **PROT_READ** depends on processor architecture, kernel version, and process state. If **READ_IMPLIES_EXEC** is set in the process's personality flags (see [personality\(2\)](#)), specifying **PROT_READ** will implicitly add **PROT_EXEC**.

On some hardware architectures (e.g., i386), **PROT_WRITE** implies **PROT_READ**.

POSIX.1 says that an implementation may permit access other than that specified in *prot*, but at a minimum can allow write access only if **PROT_WRITE** has been set, and must not allow any access if **PROT_NONE** has been set.

Applications should be careful when mixing use of **mprotect()** and **pkey_mprotect()**. On x86, when **mprotect()** is used with *prot* set to **PROT_EXEC** a pkey may be allocated and set on the memory implicitly by the kernel, but only when the pkey was 0 previously.

On systems that do not support protection keys in hardware, **pkey_mprotect()** may still be used, but *pkey* must be set to -1. When called this way, the operation of **pkey_mprotect()** is equivalent to **mprotect()**.

STANDARDS [top](#)

mprotect()
POSIX.1-2008.

pkey_mprotect()
Linux.

HISTORY [top](#)

mprotect()
POSIX.1-2001, SVr4.

pkey_mprotect()
Linux 4.9, glibc 2.27.

NOTES [top](#)

EXAMPLES [top](#)

The program below demonstrates the use of **mprotect()**. The program allocates four pages of memory, makes the third of these pages read-only, and then executes a loop that walks upward through the allocated region modifying bytes.

An example of what we might see when running the program is the following:

```
$ ./a.out
Start of region:          0x804c000
Got SIGSEGV at address: 0x804e000
```

Program source

```
#include <malloc.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <unistd.h>

#define handle_error(msg) \
```



```
do { perror(msg); exit(EXIT_FAILURE); } while (0)

static char *buffer;

static void
handler(int sig, siginfo_t *si, void *unused)
{
    /* Note: calling printf() from a signal handler is not safe
     (and should not be done in production programs), since
     printf() is not async-signal-safe; see signal-safety(7).
     Nevertheless, we use printf() here as a simple way of
     showing that the handler was called. */

    printf("Got SIGSEGV at address: %p\n", si->si_addr);
    exit(EXIT_FAILURE);
}

int
main(void)
{
    int                pagesize;
    struct sigaction   sa;

    sa.sa_flags = SA_SIGINFO;
    sigemptyset(&sa.sa_mask);
    sa.sa_sigaction = handler;
    if (sigaction(SIGSEGV, &sa, NULL) == -1)
        handle_error("sigaction");

    pagesize = sysconf(_SC_PAGE_SIZE);
    if (pagesize == -1)
        handle_error("sysconf");

    /* Allocate a buffer aligned on a page boundary;
     initial protection is PROT_READ | PROT_WRITE. */

    buffer = memalign(pagesize, 4 * pagesize);
    if (buffer == NULL)
        handle_error("memalign");

    printf("Start of region:          %p\n", buffer);

    if (mprotect(buffer + pagesize * 2, pagesize,
                 PROT_READ) == -1)
        handle_error("mprotect");
}
```



```
for (char *p = buffer ; ; )
    *(p++) = 'a';

printf("Loop completed\n");    /* Should never happen */
exit(EXIT_SUCCESS);
}
```

SEE ALSO [top](#)

[mmap\(2\)](#), [sysconf\(3\)](#), [pkeys\(7\)](#)

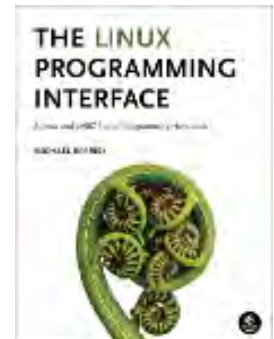
Linux man-pages (unreleased) **(date)** ***mprotect(2)***

Pages that refer to this page: [clone\(2\)](#), [madvise\(2\)](#), [mmap\(2\)](#), [pkey_alloc\(2\)](#), [prctl\(2\)](#), [remap_file_pages\(2\)](#), [seccomp\(2\)](#), [sigaction\(2\)](#), [subpage_prot\(2\)](#), [syscalls\(2\)](#), [pthread_attr_setguardsize\(3\)](#), [pthread_attr_setstack\(3\)](#), [systemd.exec\(5\)](#), [pkeys\(7\)](#), [shm_overview\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



msync(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 msync(2)

System Calls Manual

msync(2)

NAME [top](#)

msync - synchronize a file with a memory map

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/mman.h>
```

```
int msync(void addr[.length], size_t length, int flags);
```

DESCRIPTION [top](#)

`msync()` flushes changes made to the in-core copy of a file that was mapped into memory using `mmap(2)` back to the filesystem. Without use of this call, there is no guarantee that changes are written back before `munmap(2)` is called. To be more precise, the part of the file that corresponds to the memory area starting at *addr* and having length *length* is updated.

The *flags* argument should specify exactly one of `MS_ASYNC` and `MS_SYNC`, and may additionally include the `MS_INVALIDATE` bit. These bits have the following meanings:

MS_ASYNC

Specifies that an update be scheduled, but the call returns immediately.

MS_SYNC

Requests an update and waits for it to complete.

MS_INVALIDATE

Asks to invalidate other mappings of the same file (so that they can be updated with the fresh values just written).

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EBUSY **MS_INVALIDATE** was specified in *flags*, and a memory lock exists for the specified address range.

EINVAL *addr* is not a multiple of `PAGESIZE`; or any bit other than **MS_ASYNC** | **MS_INVALIDATE** | **MS_SYNC** is set in *flags*; or both **MS_SYNC** and **MS_ASYNC** are set in *flags*.

ENOMEM The indicated memory (or part of it) was not mapped.

VERSIONS [top](#)

According to POSIX, either **MS_SYNC** or **MS_ASYNC** must be specified in *flags*, and indeed failure to include one of these flags will cause `msync()` to fail on some systems. However, Linux permits a call to `msync()` that specifies neither of these flags, with semantics that are (currently) equivalent to specifying **MS_ASYNC**. (Since Linux 2.6.19, **MS_ASYNC** is in fact a no-op, since the kernel properly tracks dirty pages and flushes them to storage as necessary.) Notwithstanding the Linux behavior, portable, future-proof applications should ensure that they specify either **MS_SYNC** or **MS_ASYNC** in *flags*.



STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

This call was introduced in Linux 1.3.21, and then used **EFAULT** instead of **ENOMEM**. In Linux 2.4.19, this was changed to the POSIX value **ENOMEM**.

On POSIX systems on which **msync()** is available, both **_POSIX_MAPPED_FILES** and **_POSIX_SYNCHRONIZED_IO** are defined in `<unistd.h>` to a value greater than 0. (See also [sysconf\(3\)](#).)

SEE ALSO [top](#)

[mmap\(2\)](#)

B.O. Gallmeister, POSIX.4, O'Reilly, pp. 128–129 and 389–391.

Linux man-pages (unreleased) (date) [msync\(2\)](#)

Pages that refer to this page: [madvise\(2\)](#), [mmap2\(2\)](#), [mmap\(2\)](#), [remap_file_pages\(2\)](#), [sync_file_range\(2\)](#), [syscalls\(2\)](#), [nfs\(5\)](#), [systemd.exec\(5\)](#), [fanotify\(7\)](#), [inotify\(7\)](#), [xfs_io\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



madvise(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 madvise(2)

System Calls Manual

madvise(2)

NAME [top](#)

madvise - give advice about use of memory

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/mman.h>
```

```
int madvise(void addr[.length], size_t length, int advice);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
madvise():  
    Since glibc 2.19:  
        _DEFAULT_SOURCE  
    Up to and including glibc 2.19:  
        _BSD_SOURCE
```

DESCRIPTION [top](#)

The **madvise()** system call is used to give advice or directions to the kernel about the address range beginning at address *addr* and with size *length*. **madvise()** only operates on whole pages, therefore *addr* must be page-aligned. The value of *length* is rounded up to a multiple of page size. In most cases, the goal of such advice is to improve system or application performance.

Initially, the system call supported a set of "conventional" *advice* values, which are also available on several other implementations. (Note, though, that **madvise()** is not specified in POSIX.) Subsequently, a number of Linux-specific *advice* values have been added.

Conventional advice values

The *advice* values listed below allow an application to tell the kernel how it expects to use some mapped or shared memory areas, so that the kernel can choose appropriate read-ahead and caching techniques. These *advice* values do not influence the semantics of the application (except in the case of **MADV_DONTNEED**), but may influence its performance. All of the *advice* values listed here have analogs in the POSIX-specified **posix_madvise(3)** function, and the values have the same meanings, with the exception of **MADV_DONTNEED**.

The advice is indicated in the *advice* argument, which is one of the following:

MADV_NORMAL

No special treatment. This is the default.

MADV_RANDOM

Expect page references in random order. (Hence, read ahead may be less useful than normally.)

MADV_SEQUENTIAL

Expect page references in sequential order. (Hence, pages in the given range can be aggressively read ahead, and may be freed soon after they are accessed.)

MADV_WILLNEED

Expect access in the near future. (Hence, it might be a good idea to read some pages ahead.)

MADV_DONTNEED



Do not expect access in the near future. (For the time being, the application is finished with the given range, so the kernel can free resources associated with it.)

After a successful **MADV_DONTNEED** operation, the semantics of memory access in the specified region are changed: subsequent accesses of pages in the range will succeed, but will result in either repopulating the memory contents from the up-to-date contents of the underlying mapped file (for shared file mappings, shared anonymous mappings, and shmem-based techniques such as System V shared memory segments) or zero-fill-on-demand pages for anonymous private mappings.

Note that, when applied to shared mappings, **MADV_DONTNEED** might not lead to immediate freeing of the pages in the range. The kernel is free to delay freeing the pages until an appropriate moment. The resident set size (RSS) of the calling process will be immediately reduced however.

MADV_DONTNEED cannot be applied to locked pages, or **VM_PFNMAP** pages. (Pages marked with the kernel-internal **VM_PFNMAP** flag are special memory areas that are not managed by the virtual memory subsystem. Such pages are typically created by device drivers that map the pages into user space.)

Support for Huge TLB pages was added in Linux v5.18. Addresses within a mapping backed by Huge TLB pages must be aligned to the underlying Huge TLB page size, and the range length is rounded up to a multiple of the underlying Huge TLB page size.

Linux-specific advice values

The following Linux-specific *advice* values have no counterparts in the POSIX-specified `posix_madvise(3)`, and may or may not have counterparts in the `madvise()` interface available on other implementations. Note that some of these operations change the semantics of memory accesses.

MADV_REMOVE (since Linux 2.6.16)

Free up a given range of pages and its associated backing store. This is equivalent to punching a hole in the



corresponding range of the backing store (see [fallocate\(2\)](#)). Subsequent accesses in the specified address range will see data with a value of zero.

The specified address range must be mapped shared and writable. This flag cannot be applied to locked pages, or **VM_PFNMAP** pages.

In the initial implementation, only [tmpfs\(5\)](#) supported **MADV_REMOVE**; but since Linux 3.5, any filesystem which supports the [fallocate\(2\)](#) **FALLOC_FL_PUNCH_HOLE** mode also supports **MADV_REMOVE**. Filesystems which do not support **MADV_REMOVE** fail with the error **EOPNOTSUPP**.

Support for the Huge TLB filesystem was added in Linux v4.3.

MADV_DONTFORK (since Linux 2.6.16)

Do not make the pages in this range available to the child after a [fork\(2\)](#). This is useful to prevent copy-on-write semantics from changing the physical location of a page if the parent writes to it after a [fork\(2\)](#). (Such page relocations cause problems for hardware that DMA's into the page.)

MADV_DOFORK (since Linux 2.6.16)

Undo the effect of **MADV_DONTFORK**, restoring the default behavior, whereby a mapping is inherited across [fork\(2\)](#).

MADV_HWPOISON (since Linux 2.6.32)

Poison the pages in the range specified by *addr* and *length* and handle subsequent references to those pages like a hardware memory corruption. This operation is available only for privileged (**CAP_SYS_ADMIN**) processes. This operation may result in the calling process receiving a **SIGBUS** and the page being unmapped.

This feature is intended for testing of memory error-handling code; it is available only if the kernel was configured with **CONFIG_MEMORY_FAILURE**.

MADV_MERGEABLE (since Linux 2.6.32)

Enable Kernel Samepage Merging (KSM) for the pages in the range specified by *addr* and *length*. The kernel regularly



scans those areas of user memory that have been marked as mergeable, looking for pages with identical content. These are replaced by a single write-protected page (which is automatically copied if a process later wants to update the content of the page). KSM merges only private anonymous pages (see [mmap\(2\)](#)).

The KSM feature is intended for applications that generate many instances of the same data (e.g., virtualization systems such as KVM). It can consume a lot of processing power; use with care. See the Linux kernel source file [Documentation/admin-guide/mm/ksm.rst](#) for more details.

The **MADV_MERGEABLE** and **MADV_UNMERGEABLE** operations are available only if the kernel was configured with **CONFIG_KSM**.

MADV_UNMERGEABLE (since Linux 2.6.32)

Undo the effect of an earlier **MADV_MERGEABLE** operation on the specified address range; KSM unmerges whatever pages it had merged in the address range specified by *addr* and *length*.

MADV_SOFT_OFFLINE (since Linux 2.6.33)

Soft offline the pages in the range specified by *addr* and *length*. The memory of each page in the specified range is preserved (i.e., when next accessed, the same content will be visible, but in a new physical page frame), and the original page is offlined (i.e., no longer used, and taken out of normal memory management). The effect of the **MADV_SOFT_OFFLINE** operation is invisible to (i.e., does not change the semantics of) the calling process.

This feature is intended for testing of memory error-handling code; it is available only if the kernel was configured with **CONFIG_MEMORY_FAILURE**.

MADV_HUGEPAGE (since Linux 2.6.38)

Enable Transparent Huge Pages (THP) for pages in the range specified by *addr* and *length*. The kernel will regularly scan the areas marked as huge page candidates to replace them with huge pages. The kernel will also allocate huge pages directly when the region is naturally aligned to the huge page size (see [posix_memalign\(2\)](#)).



This feature is primarily aimed at applications that use large mappings of data and access large regions of that memory at a time (e.g., virtualization systems such as QEMU). It can very easily waste memory (e.g., a 2 MB mapping that only ever accesses 1 byte will result in 2 MB of wired memory instead of one 4 KB page). See the Linux kernel source file [Documentation/admin-guide/mm/transhuge.rst](#) for more details.

Most common kernels configurations provide **MADV_HUGEPAGE**-style behavior by default, and thus **MADV_HUGEPAGE** is normally not necessary. It is mostly intended for embedded systems, where **MADV_HUGEPAGE**-style behavior may not be enabled by default in the kernel. On such systems, this flag can be used in order to selectively enable THP. Whenever **MADV_HUGEPAGE** is used, it should always be in regions of memory with an access pattern that the developer knows in advance won't risk to increase the memory footprint of the application when transparent hugepages are enabled.

Since Linux 5.4, automatic scan of eligible areas and replacement by huge pages works with private anonymous pages (see [mmap\(2\)](#)), shmem pages, and file-backed pages. For all memory types, memory may only be replaced by huge pages on hugepage-aligned boundaries. For file-mapped memory –including tmpfs (see [tmpfs\(2\)](#))– the mapping must also be naturally hugepage-aligned within the file. Additionally, for file-backed, non-tmpfs memory, the file must not be open for write and the mapping must be executable.

The VMA must not be marked **VM_NOHUGEPAGE**, **VM_HUGETLB**, **VM_IO**, **VM_DONTEXPAND**, **VM_MIXEDMAP**, or **VM_PFNMAP**, nor can it be stack memory or backed by a DAX-enabled device (unless the DAX device is hot-plugged as System RAM). The process must also not have **PR_SET_THP_DISABLE** set (see [prctl\(2\)](#)).

The **MADV_HUGEPAGE**, **MADV_NOHUGEPAGE**, and **MADV_COLLAPSE** operations are available only if the kernel was configured with **CONFIG_TRANSPARENT_HUGEPAGE** and file/shmem memory is



only supported if the kernel was configured with **CONFIG_READ_ONLY_THP_FOR_FS**.

MADV_NOHUGEPAGE (since Linux 2.6.38)

Ensures that memory in the address range specified by *addr* and *length* will not be backed by transparent hugepages.

MADV_COLLAPSE (since Linux 6.1)

Perform a best-effort synchronous collapse of the native pages mapped by the memory range into Transparent Huge Pages (THPs). **MADV_COLLAPSE** operates on the current state of memory of the calling process and makes no persistent changes or guarantees on how pages will be mapped, constructed, or faulted in the future.

MADV_COLLAPSE supports private anonymous pages (see [mmap\(2\)](#)), shmem pages, and file-backed pages. See **MADV_HUGEPAGE** for general information on memory requirements for THP. If the range provided spans multiple VMAs, the semantics of the collapse over each VMA is independent from the others. If collapse of a given huge page-aligned/sized region fails, the operation may continue to attempt collapsing the remainder of the specified memory. **MADV_COLLAPSE** will automatically clamp the provided range to be hugepage-aligned.

All non-resident pages covered by the range will first be swapped/faulted-in, before being copied onto a freshly allocated hugepage. If the native pages compose the same PTE-mapped hugepage, and are suitably aligned, allocation of a new hugepage may be elided and collapse may happen in-place. Unmapped pages will have their data directly initialized to 0 in the new hugepage. However, for every eligible hugepage-aligned/sized region to be collapsed, at least one page must currently be backed by physical memory.

MADV_COLLAPSE is independent of any sysfs (see [sysfs\(5\)](#)) setting under */sys/kernel/mm/transparent_hugepage*, both in terms of determining THP eligibility, and allocation semantics. See Linux kernel source file [Documentation/admin-guide/mm/transhuge.rst](#) for more information. **MADV_COLLAPSE** also ignores **huge=** tmpfs mount when operating on tmpfs files. Allocation for the new



hugepage may enter direct reclaim and/or compaction, regardless of VMA flags (though **VM_NOHUGEPAGE** is still respected).

When the system has multiple NUMA nodes, the hugepage will be allocated from the node providing the most native pages.

If all hugepage-sized/aligned regions covered by the provided range were either successfully collapsed, or were already PMD-mapped THPs, this operation will be deemed successful. Note that this doesn't guarantee anything about other possible mappings of the memory. In the event multiple hugepage-aligned/sized areas fail to collapse, only the most-recently-failed code will be set in *errno*.

MADV_DONTDUMP (since Linux 3.4)

Exclude from a core dump those pages in the range specified by *addr* and *length*. This is useful in applications that have large areas of memory that are known not to be useful in a core dump. The effect of **MADV_DONTDUMP** takes precedence over the bit mask that is set via the */proc/pid/coredump_filter* file (see [core\(5\)](#)).

MADV_DODUMP (since Linux 3.4)

Undo the effect of an earlier **MADV_DONTDUMP**.

MADV_FREE (since Linux 4.5)

The application no longer requires the pages in the range specified by *addr* and *len*. The kernel can thus free these pages, but the freeing could be delayed until memory pressure occurs. For each of the pages that has been marked to be freed but has not yet been freed, the free operation will be canceled if the caller writes into the page. After a successful **MADV_FREE** operation, any stale data (i.e., dirty, unwritten pages) will be lost when the kernel frees the pages. However, subsequent writes to pages in the range will succeed and then kernel cannot free those dirtied pages, so that the caller can always see just written data. If there is no subsequent write, the kernel can free the pages at any time. Once pages in the range have been freed, the caller will see zero-fill-on-demand pages upon subsequent page references.



The **MADV_FREE** operation can be applied only to private anonymous pages (see [mmap\(2\)](#)). Before Linux 4.12, when freeing pages on a swapless system, the pages in the given range are freed instantly, regardless of memory pressure.

MADV_WIPEONFORK (since Linux 4.14)

Present the child process with zero-filled memory in this range after a [fork\(2\)](#). This is useful in forking servers in order to ensure that sensitive per-process data (for example, PRNG seeds, cryptographic secrets, and so on) is not handed to child processes.

The **MADV_WIPEONFORK** operation can be applied only to private anonymous pages (see [mmap\(2\)](#)).

Within the child created by [fork\(2\)](#), the **MADV_WIPEONFORK** setting remains in place on the specified address range. This setting is cleared during [execve\(2\)](#).

MADV_KEEPPONFORK (since Linux 4.14)

Undo the effect of an earlier **MADV_WIPEONFORK**.

MADV_COLD (since Linux 5.4)

Deactivate a given range of pages. This will make the pages a more probable reclaim target should there be a memory pressure. This is a nondestructive operation. The advice might be ignored for some pages in the range when it is not applicable.

MADV_PAGEOUT (since Linux 5.4)

Reclaim a given range of pages. This is done to free up memory occupied by these pages. If a page is anonymous, it will be swapped out. If a page is file-backed and dirty, it will be written back to the backing storage. The advice might be ignored for some pages in the range when it is not applicable.

MADV_POPULATE_READ (since Linux 5.14)

"Populate (prefault) page tables readable, faulting in all pages in the range just as if manually reading from each page; however, avoid the actual memory access that would have been performed after handling the fault.

In contrast to **MAP_POPULATE**, **MADV_POPULATE_READ** does not



hide errors, can be applied to (parts of) existing mappings and will always populate (prefault) page tables readable. One example use case is prefaulting a file mapping, reading all file content from disk; however, pages won't be dirtied and consequently won't have to be written back to disk when evicting the pages from memory.

Depending on the underlying mapping, map the shared zeropage, preallocate memory or read the underlying file; files with holes might or might not preallocate blocks. If populating fails, a **SIGBUS** signal is not generated; instead, an error is returned.

If **MADV_POPULATE_READ** succeeds, all page tables have been populated (prefaulted) readable once. If **MADV_POPULATE_READ** fails, some page tables might have been populated.

MADV_POPULATE_READ cannot be applied to mappings without read permissions and special mappings, for example, mappings marked with kernel-internal flags such as **VM_PFNMAP** or **VM_IO**, or secret memory regions created using [memfd_secret\(2\)](#).

Note that with **MADV_POPULATE_READ**, the process can be killed at any moment when the system runs out of memory.

MADV_POPULATE_WRITE (since Linux 5.14)

Populate (prefault) page tables writable, faulting in all pages in the range just as if manually writing to each each page; however, avoid the actual memory access that would have been performed after handling the fault.

In contrast to **MAP_POPULATE**, **MADV_POPULATE_WRITE** does not hide errors, can be applied to (parts of) existing mappings and will always populate (prefault) page tables writable. One example use case is preallocating memory, breaking any CoW (Copy on Write).

Depending on the underlying mapping, preallocate memory or read the underlying file; files with holes will preallocate blocks. If populating fails, a **SIGBUS** signal is not generated; instead, an error is returned.



If **MADV_POPULATE_WRITE** succeeds, all page tables have been populated (prefaulted) writable once. If **MADV_POPULATE_WRITE** fails, some page tables might have been populated.

MADV_POPULATE_WRITE cannot be applied to mappings without write permissions and special mappings, for example, mappings marked with kernel-internal flags such as **VM_PFNMAP** or **VM_IO**, or secret memory regions created using [memfd_secret\(2\)](#).

Note that with **MADV_POPULATE_WRITE**, the process can be killed at any moment when the system runs out of memory.

RETURN VALUE [top](#)

On success, **madvise()** returns zero. On error, it returns -1 and *errno* is set to indicate the error.

ERRORS [top](#)

EACCES *advice* is **MADV_REMOVE**, but the specified address range is not a shared writable mapping.

EAGAIN A kernel resource was temporarily unavailable.

EBADF The map exists, but the area maps something that isn't a file.

EBUSY (for **MADV_COLLAPSE**) Could not charge hugepage to cgroup: cgroup limit exceeded.

EFAULT *advice* is **MADV_POPULATE_READ** or **MADV_POPULATE_WRITE**, and populating (prefaulting) page tables failed because a **SIGBUS** would have been generated on actual memory access and the reason is not a HW poisoned page (HW poisoned pages can, for example, be created using the **MADV_HWPOISON** flag described elsewhere in this page).

EINVAL *addr* is not page-aligned or *length* is negative.

EINVAL *advice* is not a valid.



- EINVAL** *advice* is **MADV_COLD** or **MADV_PAGEOUT** and the specified address range includes locked, Huge TLB pages, or **VM_PFNMAP** pages.
- EINVAL** *advice* is **MADV_DONTNEED** or **MADV_REMOVE** and the specified address range includes locked, Huge TLB pages, or **VM_PFNMAP** pages.
- EINVAL** *advice* is **MADV_MERGEABLE** or **MADV_UNMERGEABLE**, but the kernel was not configured with **CONFIG_KSM**.
- EINVAL** *advice* is **MADV_FREE** or **MADV_WIPEONFORK** but the specified address range includes file, Huge TLB, **MAP_SHARED**, or **VM_PFNMAP** ranges.
- EINVAL** *advice* is **MADV_POPULATE_READ** or **MADV_POPULATE_WRITE**, but the specified address range includes ranges with insufficient permissions or special mappings, for example, mappings marked with kernel-internal flags such a **VM_IO** or **VM_PFNMAP**, or secret memory regions created using [memfd_secret\(2\)](#).
- EIO** (for **MADV_WILLNEED**) Paging in this area would exceed the process's maximum resident set size.
- ENOMEM** (for **MADV_WILLNEED**) Not enough memory: paging in failed.
- ENOMEM** (for **MADV_COLLAPSE**) Not enough memory: could not allocate hugepage.
- ENOMEM** Addresses in the specified range are not currently mapped, or are outside the address space of the process.
- ENOMEM** *advice* is **MADV_POPULATE_READ** or **MADV_POPULATE_WRITE**, and populating (prefaulting) page tables failed because there was not enough memory.
- EPERM** *advice* is **MADV_HWPOISON**, but the caller does not have the **CAP_SYS_ADMIN** capability.
- EHWPOISON**
advice is **MADV_POPULATE_READ** or **MADV_POPULATE_WRITE**, and populating (prefaulting) page tables failed because a HW



poisoned page (HW poisoned pages can, for example, be created using the **MADV_HWPOISON** flag described elsewhere in this page) was encountered.

VERSIONS [top](#)

Versions of this system call, implementing a wide variety of *advice* values, exist on many other implementations. Other implementations typically implement at least the flags listed above under *Conventional advice flags*, albeit with some variation in semantics.

POSIX.1-2001 describes [posix_madvise\(3\)](#) with constants **POSIX_MADV_NORMAL**, **POSIX_MADV_RANDOM**, **POSIX_MADV_SEQUENTIAL**, **POSIX_MADV_WILLNEED**, and **POSIX_MADV_DONTNEED**, and so on, with behavior close to the similarly named flags listed above.

Linux

The Linux implementation requires that the address *addr* be page-aligned, and allows *length* to be zero. If there are some parts of the specified address range that are not mapped, the Linux version of **madvise()** ignores them and applies the call to the rest (but returns **ENOMEM** from the system call, as it should).

madvise(0, 0, advice) will return zero iff *advice* is supported by the kernel and can be relied on to probe for support.

STANDARDS [top](#)

None.

HISTORY [top](#)

First appeared in 4.4BSD.

Since Linux 3.18, support for this system call is optional, depending on the setting of the **CONFIG_ADVICE_SYSCALLS** configuration option.

SEE ALSO [top](#)



[getrlimit\(2\)](#), [memfd_secret\(2\)](#), [mincore\(2\)](#), [mmap\(2\)](#), [mprotect\(2\)](#),
[msync\(2\)](#), [munmap\(2\)](#), [prctl\(2\)](#), [process_madvise\(2\)](#),
[posix_madvise\(3\)](#), [core\(5\)](#)

Linux man-pages (unreleased) (date)

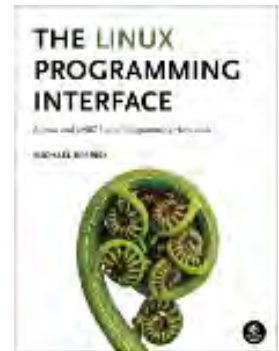
madvise(2)

Pages that refer to this page: [fork\(2\)](#), [getrlimit\(2\)](#), [ioctl_userfaultfd\(2\)](#),
[io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [mincore\(2\)](#), [open\(2\)](#), [prctl\(2\)](#),
[process_madvise\(2\)](#), [readahead\(2\)](#), [syscalls\(2\)](#), [userfaultfd\(2\)](#),
[io_uring_prep_madvise\(3\)](#), [malloc_trim\(3\)](#), [posix_madvise\(3\)](#), [core\(5\)](#), [proc\(5\)](#),
[tmpfs\(5\)](#), [capabilities\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of
[The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



Materials for Topic 6: Vectored I/O & Memory-Mapping

Full C Programs

- [writev_example.c](#) - a C program demonstrating the use of the `writev()` vectored I/O system call to write 3 lines at once into a file.
- [readv_example.c](#) - a C program demonstrating the use of the `readv()` vectored I/O system call to read 3 lines at once from a file.
- [mmap_example.c](#) - a C program that maps a file into main memory using `mmap()`, thereby letting the program access it as an array of characters.

Runnable Linux Commands

- The command:

```
gcc -Wall -Wextra -O2 -g -o program program.c
```

compiles the C source code located inside the file `program.c`. See more details [here](#).

- The command:

```
./short_prompt
```

executes code inside a file named `short_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).

- The command:

```
. ./long_prompt
```

executes code inside a file named `long_prompt` and sources it (applies all the changes to the current session.)

See more details [here](#).



This website by [Miriam Briskman](#) is licensed under [CC BY-NC 4.0](#).




```
1  /* This program exemplifies the vectored I/O
2  *   writev() function. It writes 3 lines of text
3  *   into a file using the single system call
4  *   writev(). Note that writev(), being a system
5  *   call, works on file descriptors (as covered
6  *   in Topic 4) and not on file pointers.
7  *
8  *   Miriam Briskman, 3/8/2023
9  *   CISC 3350, Brooklyn College
10 *   Licensed under CC BY-NC 4.0
11 */
12
13 #include <stdio.h>      // printf(), perror()
14 #include <stdlib.h>    // EXIT_SUCCESS, EXIT_FAILURE
15 #include <sys/types.h> // ssize_t type
16 #include <fcntl.h>     // creat()
17 #include <string.h>    // strlen()
18 #include <unistd.h>    // close()
19 #include <sys/uio.h>   // writev(), iovec structure
20
21 int main ()
22 {
23     struct iovec arr[3]; // Array of structures.
24     ssize_t num; // ssize_t is usually implemented
25                 // as an integer. We need one
26                 // because writev() returns a
27                 // ssize_t variable, so we need
28                 // it to store this value.
29     int file, index;
30
31     // An array containing 3 string literals:
32     char *buf[] = {
33         "Winter ended a couple of months ago.\n",
34         "It's spring now!\n",
35         "Summer is behind the corner!\n"
36     };
37
38     // Create a new file:
39     if ((file = creat ("seasons.txt", 0644)) == -1)
40     {
41         perror ("open");
42         exit (EXIT_FAILURE);
```



```
43     }
44
45     // To prepare for writing the 3 strings
46     //     above at once into a file using
47     //     vectored-i/o, we first need to
48     //     fill out the three iovec structures
49     //     that we created at the top of the
50     //     program:
51     for (index = 0; index < 3; index++)
52     {
53         arr[index].iov_base = buf[index];
54         arr[index].iov_len = strlen (buf[index]) + 1;
55     }
56
57     // Now, in one shot, we write out all the
58     //     three strings into the file!
59     if ((num = writev (file, arr, 3)) == -1)
60     {
61         perror ("writev");
62         exit (EXIT_FAILURE);
63     }
64
65     printf ("We wrote out %d bytes!\n", (int) num);
66
67     if (close (file) == -1)
68     {
69         perror ("close");
70         exit (EXIT_FAILURE);
71     }
72
73     return EXIT_SUCCESS;
74 }
75
```



```
1  /* This program exemplifies the vectored I/O
2  *   readv() function. It assumes the existence
3  *   of the file 'seasons.txt' within the current
4  *   directory and that it has 3 lines of text in
5  *   it. All 3 lines of text will be read in with
6  *   the single system call readv(). Note that
7  *   readv(), being a system call, works on file
8  *   descriptors (as covered in Topic 4) and not
9  *   on file pointers.
10 *
11 *   Miriam Briskman, 3/8/2023
12 *   CISC 3350, Brooklyn College
13 *   Licensed under CC BY-NC 4.0
14 */
15
16 #include <stdio.h>      // printf(), perror()
17 #include <stdlib.h>    // EXIT_SUCCESS, EXIT_FAILURE
18 #include <fcntl.h>     // open(), O_RDONLY flag
19 #include <unistd.h>    // close()
20 #include <sys/uio.h>   // readv(), iovec structure
21
22 int main ()
23 {
24     // To be able to use readv(), we first need
25     //   to know how many characters we will be
26     //   reading from each line:
27     int sizes [3] = {38, 18, 30};
28
29     int file, index;
30
31     // An array of 3 strings. We will store the
32     //   the lines that we read from the file
33     //   in these strings:
34     char * seasons [3];
35
36     // Create those char arrays on the heap:
37     for (index = 0; index < 3; index++)
38         seasons[index] = malloc (sizes[index]
39                                 * sizeof(char));
40
41     struct iovec arr[3]; // Array of structures.
42
```



```
43 // Open the file "seasons.txt" for reading:
44 if ((file = open ("seasons.txt", O_RDONLY)) == -1)
45 {
46     perror ("open");
47     exit (EXIT_FAILURE);
48 }
49
50 // To prepare for writing the 3 strings
51 // that we read from the file using
52 // vectored-i/o, we first need to
53 // fill out the three iovec structures
54 // that we created at the top of the
55 // program:
56 for (index = 0; index < 3; index++)
57 {
58     arr[index].iov_base = seasons[index];
59     arr[index].iov_len = sizes[index];
60 }
61
62 // Now, in one shot, we read in all the
63 // three strings from the file!
64 if (readv (file, arr, 3) == -1)
65 {
66     perror ("readv");
67     exit (EXIT_FAILURE);
68 }
69
70 // Print all the lines that we read in:
71 for (index = 1; index < 4; index++)
72     printf ("%d: %s", index, seasons[index - 1]);
73
74 // Free the allocated 'seasons' arrays:
75 for (index = 0; index < 3; index++)
76     free (seasons[index]);
77
78 // Close the file:
79 if (close (file) == -1)
80 {
81     perror ("close");
82     exit (EXIT_FAILURE);
83 }
84
85 return EXIT_SUCCESS;
```



86 | }
87 |

```
1 // The following program exemplifies the usage of
2 // mmap(), which is used to memory-map a file
3 // into main memory to use it as a variable
4 // and munmap() which cancels this mapping.
5
6 // This program is taken from Linux System Programming:
7 // Talking Directly to the Kernel and C Library,
8 // 2nd Edition, by Love. ISBN: 978-1-44933953-1,
9 // pages 109-110.
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <sys/types.h>
14 #include <sys/stat.h>
15 #include <fcntl.h>
16 #include <unistd.h>
17 #include <sys/mman.h>
18
19 int main (int argc, char *argv[])
20 {
21     struct stat file_info;
22     off_t len;
23     char *p;
24     int file;
25
26     // Check that the right number of arguments was
27     // passed:
28     if (argc < 2)
29     {
30         fprintf (stderr,
31                 "usage: %s <file>\n",
32                 argv[0]);
33         exit (EXIT_FAILURE);
34     }
35
36     // Open the file whose name was provided
37     // to the program in the command-line
38     // argument argv[1]:
39     if ((file = open (argv[1], O_RDONLY)) == -1)
40     {
41         perror ("open");
42         exit (EXIT_FAILURE);
```



```
43     }
44
45     // fstat() returns information about the file:
46     if (fstat (file, &file_info) == -1)
47     {
48         perror ("fstat");
49         exit (EXIT_FAILURE);
50     }
51
52     // The S_ISREG() macro checks if the given
53     //   file is a regular file (and not a
54     //   disk, socket, or other special file:)
55     if (!S_ISREG (file_info.st_mode))
56     {
57         fprintf (stderr,
58                 "%s is not a regular file\n",
59                 argv[1]);
60         exit (EXIT_FAILURE);
61     }
62
63     // sb.st_size contains the size of the file
64     //   in bytes:
65     p = mmap (0,
66              file_info.st_size,
67              PROT_READ,
68              MAP_SHARED,
69              file,
70              0);
71     if (p == MAP_FAILED)
72     {
73         perror ("mmap");
74         exit (EXIT_FAILURE);
75     }
76
77     // Since the file was mapped to memory, we
78     //   can start using the file as a variable,
79     //   so we can close the file now.
80     if (close (file) == -1)
81     {
82         perror ("close");
83         exit (EXIT_FAILURE);
84     }
85
```



```
86 // We refer to the bytes of the file by
87 // using array notation p[len]. That is,
88 // the content of the file can now be
89 // referred to using array notation! It's
90 // both fast and convenient to use:
91 for (len = 0; len < file_info.st_size; len++)
92     putchar (p[len]); // Prints a char to screen.
93
94 // Remove the mapping of the file from memory:
95 if (munmap (p, file_info.st_size) == -1)
96 {
97     perror ("munmap");
98     exit (EXIT_FAILURE);
99 }
100
101 return EXIT_FAILURE;
102 }
103
```



Topic 7: Processes

Note: A copy of each one of the sources listed below is included in this packet. To jump to the location of a source copy, click the page link under “Page”.

In case a source isn’t included (e.g., a YouTube video,) click the [↗](#) (external link) symbol under “Page” to open the external source in a browser window.

| # | Citation & Source Link | Page |
|----|--|------|
| 1 | “getpid(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/getpid.2.html | 771 |
| 2 | “exec(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/exec.3.html | 774 |
| 3 | “execve(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/execve.2.html | 780 |
| 4 | “fork(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/fork.2.html | 794 |
| 5 | “exit(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/exit.3.html | 801 |
| 6 | “_exit(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/_exit.2.html | 805 |
| 7 | “atexit(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/atexit.3.html | 809 |
| 8 | “wait(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/wait.2.html | 813 |
| 9 | “system(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/system.3.html | 824 |
| 10 | “sh(1p) - Linux manual page”, <i>man7.org</i> , 2017. URL: https://man7.org/linux/man-pages/man1/system.1p.html | 829 |
| 11 | “passwd(1) - Linux manual page”, <i>man7.org</i> , 22 Dec. 2023. URL: https://man7.org/linux/man-pages/man1/passwd.1.html | 858 |
| 12 | “setuid(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/setuid.2.html | 878 |
| 13 | “setgid(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/setgid.2.html | 869 |
| 14 | “seteuid(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/seteuid.2.html | 872 |
| 15 | “setegid(3p) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/setegid.3p.html | 875 |
| 16 | “getuid(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/getuid.2.html | 882 |

| # | Citation & Source Link | Page |
|----|--|------|
| 17 | “getgid(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/getgid.2.html | 885 |
| 18 | “setuid(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/setuid.2.html | 888 |
| 19 | “getuid(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/getuid.2.html | 891 |
| 20 | “setpgid(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/setpgid.2.html | 894 |
| 21 | “dup(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/dup.2.html | 899 |
| 22 | “daemon(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/daemon.3.html | 904 |
| 23 | Lankester, Branko, et al. “ps(1) - Linux manual page”, <i>man7.org</i> , 4 Oct. 2023. URL: https://man7.org/linux/man-pages/man1/ps.1.html | 907 |
| 24 | Briskman, Miriam. “Materials for Topic 7: Processes.” <i>Topic 7: Processes — CISC 3350 Materials</i> , 2023. URL: www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_07/ | 932 |
| 25 | Briskman, Miriam. “child_creation_example.c.” (C source code) 15 Mar. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_07/child_creation_example.c | 936 |
| 26 | Love, Robert. “wait_syscall_example.c.” (C source code), <i>Linux System Programming: Talking Directly to the Kernel and C Library</i> , O’Reilly Media, Inc., Sebastopol, CA, 2013, p. 153. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_07/wait_syscall_example.c | 940 |
| 27 | Briskman, Miriam. “system_syscall_implementation.c.” (C source code) 15 Mar. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_07/system_syscall_implementation.c | 942 |
| 28 | Love, Robert. “daemon_creation.c.” (C source code), <i>Linux System Programming: Talking Directly to the Kernel and C Library</i> , O’Reilly Media, Inc., Sebastopol, CA, 2013, pp. 173-174. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_07/daemon_creation.c | 944 |

getpid(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 getpid(2)

System Calls Manual

getpid(2)

NAME [top](#)

getpid, getppid - get process identification

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
pid_t getpid(void);  
pid_t getppid(void);
```

DESCRIPTION [top](#)

getpid() returns the process ID (PID) of the calling process. (This is often used by routines that generate unique temporary filenames.)

getppid() returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using **fork()**, or, if that process has already terminated, the ID of the process to which this process has been

reparented (either `init(1)` or a "subreaper" process defined via the `prctl(2)` `PR_SET_CHILD_SUBREAPER` operation).

ERRORS [top](#)

These functions are always successful.

VERSIONS [top](#)

On Alpha, instead of a pair of `getpid()` and `getppid()` system calls, a single `getxpid()` system call is provided, which returns a pair of PID and parent PID. The glibc `getpid()` and `getppid()` wrapper functions transparently deal with this. See `syscall(2)` for details regarding register mapping.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, 4.3BSD, SVr4.

C library/kernel differences

From glibc 2.3.4 up to and including glibc 2.24, the glibc wrapper function for `getpid()` cached PIDs, with the goal of avoiding additional system calls when a process calls `getpid()` repeatedly. Normally this caching was invisible, but its correct operation relied on support in the wrapper functions for `fork(2)`, `vfork(2)`, and `clone(2)`: if an application bypassed the glibc wrappers for these system calls by using `syscall(2)`, then a call to `getpid()` in the child would return the wrong value (to be precise: it would return the PID of the parent process). In addition, there were cases where `getpid()` could return the wrong value even when invoking `clone(2)` via the glibc wrapper function. (For a discussion of one such case, see BUGS in `clone(2)`.) Furthermore, the complexity of the caching code had been the source of a few bugs within glibc over the years.

Because of the aforementioned problems, since glibc 2.25, the PID



cache is removed: calls to **getpid()** always invoke the actual system call, rather than returning a cached value.

NOTES [top](#)

If the caller's parent is in a different PID namespace (see [pid_namespaces\(7\)](#)), **getppid()** returns 0.

From a kernel perspective, the PID (which is shared by all of the threads in a multithreaded process) is sometimes also known as the thread group ID (TGID). This contrasts with the kernel thread ID (TID), which is unique for each thread. For further details, see [gettid\(2\)](#) and the discussion of the **CLONE_THREAD** flag in [clone\(2\)](#).

SEE ALSO [top](#)

[clone\(2\)](#), [fork\(2\)](#), [gettid\(2\)](#), [kill\(2\)](#), [exec\(3\)](#), [mkstemp\(3\)](#), [tempnam\(3\)](#), [tmpfile\(3\)](#), [tmpnam\(3\)](#), [credentials\(7\)](#), [pid_namespaces\(7\)](#)

Linux man-pages (unreleased) (date)

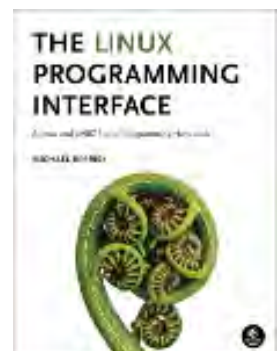
[getpid\(2\)](#)

Pages that refer to this page: [strace\(1\)](#), [capget\(2\)](#), [clone\(2\)](#), [fcntl\(2\)](#), [gettid\(2\)](#), [prctl\(2\)](#), [sched_setaffinity\(2\)](#), [sched_setscheduler\(2\)](#), [syscalls\(2\)](#), [id_t\(3type\)](#), [libcap\(3\)](#), [pmnotifyerr\(3\)](#), [pmwebtimerregister\(3\)](#), [raise\(3\)](#), [lload.conf\(5\)](#), [slapd.conf\(5\)](#), [slapd-config\(5\)](#), [credentials\(7\)](#), [fanotify\(7\)](#), [pid_namespaces\(7\)](#), [pthreads\(7\)](#), [signal-safety\(7\)](#), [lload\(8\)](#), [slapd\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



exec(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [BUGS](#) | [SEE ALSO](#)

 exec(3)

Library Functions Manual

exec(3)**NAME** [top](#)

execl, execlp, execl, execv, execvp, execvpe - execute a file

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
extern char **environ;
```

```
int execl(const char *pathname, const char *arg, ...  
          /*, (char *) NULL */);
```

```
int execlp(const char *file, const char *arg, ...  
          /*, (char *) NULL */);
```

```
int execl_e(const char *pathname, const char *arg, ...  
            /*, (char *) NULL, char *const envp[] */);
```

```
int execv(const char *pathname, char *const argv[]);
```

```
int execvp(const char *file, char *const argv[]);
```

```
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
execvpe():  
    _GNU_SOURCE
```

DESCRIPTION [top](#)

The `exec()` family of functions replaces the current process image with a new process image. The functions described in this manual page are layered on top of `execve(2)`. (See the manual page for `execve(2)` for further details about the replacement of the current process image.)

The initial argument for these functions is the name of a file that is to be executed.

The functions can be grouped based on the letters following the "exec" prefix.

l - `execl()`, `execlp()`, `execle()`

The `const char *arg` and subsequent ellipses can be thought of as `arg0, arg1, ..., argn`. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments *must* be terminated by a null pointer, and, since these are variadic functions, this pointer must be cast `(char *) NULL`.

By contrast with the 'l' functions, the 'v' functions (below) specify the command-line arguments of the executed program as a vector.

v - `execv()`, `execvp()`, `execvpe()`

The `char *const argv[]` argument is an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a null pointer.

e - `execle()`, `execvpe()`

The environment of the new process image is specified via the argument `envp`. The `envp` argument is an array of pointers to null-terminated strings and *must* be terminated by a null pointer.

All other `exec()` functions (which do not include 'e' in the suffix) take the environment for the new process image from the external variable `environ` in the calling process.

p - `execlp()`, `execvp()`, `execvpe()`

These functions duplicate the actions of the shell in searching for an executable file if the specified filename does not contain

a slash (/) character. The file is sought in the colon-separated list of directory pathnames specified in the **PATH** environment variable. If this variable isn't defined, the path list defaults to a list that includes the directories returned by `confstr(_CS_PATH)` (which typically returns the value `"/bin:/usr/bin"`) and possibly also the current working directory; see NOTES for further details.

`execvpe()` searches for the program using the value of **PATH** from the caller's environment, not from the `envp` argument.

If the specified filename includes a slash character, then **PATH** is ignored, and the file at the specified pathname is executed.

In addition, certain errors are treated specially.

If permission is denied for a file (the attempted `execve(2)` failed with the error **EACCES**), these functions will continue searching the rest of the search path. If no other file is found, however, they will return with `errno` set to **EACCES**.

If the header of a file isn't recognized (the attempted `execve(2)` failed with the error **ENOEXEC**), these functions will execute the shell (`/bin/sh`) with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

All other `exec()` functions (which do not include 'p' in the suffix) take as their first argument a (relative or absolute) pathname that identifies the program to be executed.

RETURN VALUE [top](#)

The `exec()` functions return only if an error has occurred. The return value is -1, and `errno` is set to indicate the error.

ERRORS [top](#)

All of these functions may fail and set `errno` for any of the errors specified for `execve(2)`.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|--|---------------|-------------|
| <code>execl()</code> , <code>execle()</code> , <code>execv()</code> | Thread safety | MT-Safe |
| <code>execlp()</code> , <code>execvp()</code> , <code>execvpe()</code> | Thread safety | MT-Safe env |

VERSIONS [top](#)

The default search path (used when the environment does not contain the variable **PATH**) shows some variation across systems. It generally includes `/bin` and `/usr/bin` (in that order) and may also include the current working directory. On some other systems, the current working is included after `/bin` and `/usr/bin`, as an anti-Trojan-horse measure. The glibc implementation long followed the traditional default where the current working directory is included at the start of the search path. However, some code refactoring during the development of glibc 2.24 caused the current working directory to be dropped altogether from the default search path. This accidental behavior change is considered mildly beneficial, and won't be reverted.

The behavior of `execlp()` and `execvp()` when errors occur while attempting to execute the file is historic practice, but has not traditionally been documented and is not specified by the POSIX standard. BSD (and possibly other systems) do an automatic sleep and retry if **ETXTBSY** is encountered. Linux treats it as a hard error and returns immediately.

Traditionally, the functions `execlp()` and `execvp()` ignored all errors except for the ones described above and **ENOMEM** and **E2BIG**, upon which they returned. They now return if any error other than the ones described above occurs.

STANDARDS [top](#)

environ

`execl()`

`execlp()`

`execle()`

`execv()`

`execvp()`

POSIX.1-2008.

execvpe()
GNU.

HISTORY [top](#)

environ
execl()
execlp()
execle()
execv()
execvp()
POSIX.1-2001.

execvpe()
glibc 2.11.

BUGS [top](#)

Before glibc 2.24, **execl()** and **execle()** employed [realloc\(3\)](#) internally and were consequently not async-signal-safe, in violation of the requirements of POSIX.1. This was fixed in glibc 2.24.

Architecture-specific details

On sparc and sparc64, **execv()** is provided as a system call by the kernel (with the prototype shown above) for compatibility with SunOS. This function is *not* employed by the **execv()** wrapper function on those architectures.

SEE ALSO [top](#)

[sh\(1\)](#), [execve\(2\)](#), [execveat\(2\)](#), [fork\(2\)](#), [ptrace\(2\)](#), [fexecve\(3\)](#), [system\(3\)](#), [environ\(7\)](#)

Linux man-pages (unreleased) (date) [exec\(3\)](#)

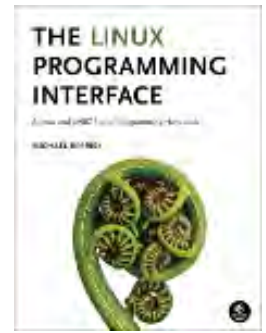
Pages that refer to this page: [pmllogger\(1\)](#), [watch\(1\)](#), [xargs\(1\)](#), [execve\(2\)](#), [getpid\(2\)](#), [ptrace\(2\)](#), [seccomp\(2\)](#), [staffs\(2\)](#), [vfork\(2\)](#), [atexit\(3\)](#), [clearenv\(3\)](#), [confstr\(3\)](#), [glob\(3\)](#), [libexpect\(3\)](#), [ltnng-ust\(3\)](#), [on_exit\(3\)](#), [pam_getenvlist\(3\)](#), [posix_spawn\(3\)](#), [statvfs\(3\)](#), [stdin\(3\)](#), [sysconf\(3\)](#), [system\(3\)](#), [systemd.exec\(5\)](#), [environ\(7\)](#), [signal-safety\(7\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



execve(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [EXAMPLES](#) | [SEE ALSO](#)

 execve(2)

System Calls Manual

execve(2)

NAME [top](#)

execve - execute program

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int execve(const char *pathname, char *const _Nullable argv[],  
           char *const _Nullable envp[]);
```

DESCRIPTION [top](#)

execve() executes the program referred to by *pathname*. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments.

pathname must be either a binary executable, or a script starting with a line of the form:

```
#!interpreter [optional-arg]
```

For details of the latter case, see "Interpreter scripts" below.

argv is an array of pointers to strings passed to the new program as its command-line arguments. By convention, the first of these strings (i.e., *argv[0]*) should contain the filename associated with the file being executed. The *argv* array must be terminated by a NULL pointer. (Thus, in the new program, *argv[argc]* will be NULL.)

envp is an array of pointers to strings, conventionally of the form **key=value**, which are passed as the environment of the new program. The *envp* array must be terminated by a NULL pointer.

This manual page describes the Linux system call in detail; for an overview of the nomenclature and the many, often preferable, standardised variants of this function provided by libc, including ones that search the **PATH** environment variable, see [exec\(3\)](#).

The argument vector and environment can be accessed by the new program's main function, when it is defined as:

```
int main(int argc, char *argv[], char *envp[])
```

Note, however, that the use of a third argument to the main function is not specified in POSIX.1; according to POSIX.1, the environment should be accessed via the external variable [environ\(7\)](#).

execve() does not return on success, and the text, initialized data, uninitialized data (bss), and stack of the calling process are overwritten according to the contents of the newly loaded program.

If the current program is being ptraced, a **SIGTRAP** signal is sent to it after a successful **execve()**.

If the set-user-ID bit is set on the program file referred to by *pathname*, then the effective user ID of the calling process is changed to that of the owner of the program file. Similarly, if the set-group-ID bit is set on the program file, then the effective group ID of the calling process is set to the group of the program file.

The aforementioned transformations of the effective IDs are *not* performed (i.e., the set-user-ID and set-group-ID bits are



ignored) if any of the following is true:

- the `no_new_privs` attribute is set for the calling thread (see [prctl\(2\)](#));
- the underlying filesystem is mounted `nosuid` (the `MS_NOSUID` flag for [mount\(2\)](#)); or
- the calling process is being ptraced.

The capabilities of the program file (see [capabilities\(7\)](#)) are also ignored if any of the above are true.

The effective user ID of the process is copied to the saved set-user-ID; similarly, the effective group ID is copied to the saved set-group-ID. This copying takes place after any effective ID changes that occur because of the set-user-ID and set-group-ID mode bits.

The process's real UID and real GID, as well as its supplementary group IDs, are unchanged by a call to `execve()`.

If the executable is an a.out dynamically linked binary executable containing shared-library stubs, the Linux dynamic linker [ld.so\(8\)](#) is called at the start of execution to bring needed shared objects into memory and link the executable with them.

If the executable is a dynamically linked ELF executable, the interpreter named in the PT_INTERP segment is used to load the needed shared objects. This interpreter is typically `/lib/ld-linux.so.2` for binaries linked with glibc (see [ld-linux.so\(8\)](#)).

Effect on process attributes

All process attributes are preserved during an `execve()`, except the following:

- The dispositions of any signals that are being caught are reset to the default ([signal\(7\)](#)).
- Any alternate signal stack is not preserved ([sigaltstack\(2\)](#)).
- Memory mappings are not preserved ([mmap\(2\)](#)).

- Attached System V shared memory segments are detached ([shmat\(2\)](#)).
- POSIX shared memory regions are unmapped ([shm_open\(3\)](#)).
- Open POSIX message queue descriptors are closed ([mq_overview\(7\)](#)).
- Any open POSIX named semaphores are closed ([sem_overview\(7\)](#)).
- POSIX timers are not preserved ([timer_create\(2\)](#)).
- Any open directory streams are closed ([opendir\(3\)](#)).
- Memory locks are not preserved ([mlock\(2\)](#), [mlockall\(2\)](#)).
- Exit handlers are not preserved ([atexit\(3\)](#), [on_exit\(3\)](#)).
- The floating-point environment is reset to the default (see [fenv\(3\)](#)).

The process attributes in the preceding list are all specified in POSIX.1. The following Linux-specific process attributes are also not preserved during an **execve()**:

- The process's "dumpable" attribute is set to the value 1, unless a set-user-ID program, a set-group-ID program, or a program with capabilities is being executed, in which case the dumpable flag may instead be reset to the value in [/proc/sys/fs/suid_dumpable](#), in the circumstances described under **PR_SET_DUMPABLE** in [prctl\(2\)](#). Note that changes to the "dumpable" attribute may cause ownership of files in the process's [/proc/pid](#) directory to change to [root:root](#), as described in [proc\(5\)](#).
- The [prctl\(2\)](#) **PR_SET_KEEPCAPS** flag is cleared.
- (Since Linux 2.4.36 / 2.6.23) If a set-user-ID or set-group-ID program is being executed, then the parent death signal set by [prctl\(2\)](#) **PR_SET_PDEATHSIG** flag is cleared.
- The process name, as set by [prctl\(2\)](#) **PR_SET_NAME** (and displayed by [ps -o comm](#)), is reset to the name of the new executable file.



- The **SECBIT_KEEP_CAPS** *securebits* flag is cleared. See [capabilities\(7\)](#).
- The termination signal is reset to **SIGCHLD** (see [clone\(2\)](#)).
- The file descriptor table is unshared, undoing the effect of the **CLONE_FILES** flag of [clone\(2\)](#).

Note the following further points:

- All threads other than the calling thread are destroyed during an **execve()**. Mutexes, condition variables, and other pthreads objects are not preserved.
- The equivalent of *setlocale(LC_ALL, "C")* is executed at program start-up.
- POSIX.1 specifies that the dispositions of any signals that are ignored or set to the default are left unchanged. POSIX.1 specifies one exception: if **SIGCHLD** is being ignored, then an implementation may leave the disposition unchanged or reset it to the default; Linux does the former.
- Any outstanding asynchronous I/O operations are canceled ([aio_read\(3\)](#), [aio_write\(3\)](#)).
- For the handling of capabilities during **execve()**, see [capabilities\(7\)](#).
- By default, file descriptors remain open across an **execve()**. File descriptors that are marked close-on-exec are closed; see the description of **FD_CLOEXEC** in [fcntl\(2\)](#). (If a file descriptor is closed, this will cause the release of all record locks obtained on the underlying file by this process. See [fcntl\(2\)](#) for details.) POSIX.1 says that if file descriptors 0, 1, and 2 would otherwise be closed after a successful **execve()**, and the process would gain privilege because the set-user-ID or set-group-ID mode bit was set on the executed file, then the system may open an unspecified file for each of these file descriptors. As a general principle, no portable program, whether privileged or not, can assume that these three file descriptors will remain closed across an **execve()**.

Interpreter scripts

An interpreter script is a text file that has execute permission enabled and whose first line is of the form:

```
#!interpreter [optional-arg]
```

The *interpreter* must be a valid pathname for an executable file.

If the *pathname* argument of **execve()** specifies an interpreter script, then *interpreter* will be invoked with the following arguments:

```
interpreter [optional-arg] pathname arg...
```

where *pathname* is the pathname of the file specified as the first argument of **execve()**, and *arg...* is the series of words pointed to by the *argv* argument of **execve()**, starting at *argv[1]*. Note that there is no way to get the *argv[0]* that was passed to the **execve()** call.

For portable use, *optional-arg* should either be absent, or be specified as a single word (i.e., it should not contain white space); see NOTES below.

Since Linux 2.6.28, the kernel permits the interpreter of a script to itself be a script. This permission is recursive, up to a limit of four recursions, so that the interpreter may be a script which is interpreted by a script, and so on.

Limits on size of arguments and environment

Most UNIX implementations impose some limit on the total size of the command-line argument (*argv*) and environment (*envp*) strings that may be passed to a new program. POSIX.1 allows an implementation to advertise this limit using the **ARG_MAX** constant (either defined in *<limits.h>* or available at run time using the call *sysconf(_SC_ARG_MAX)*).

Before Linux 2.6.23, the memory used to store the environment and argument strings was limited to 32 pages (defined by the kernel constant **MAX_ARG_PAGES**). On architectures with a 4-kB page size, this yields a maximum size of 128 kB.

On Linux 2.6.23 and later, most architectures support a size limit derived from the soft **RLIMIT_STACK** resource limit (see [getrlimit\(2\)](#)) that is in force at the time of the **execve()** call. (Architectures with no memory management unit are excepted: they



maintain the limit that was in effect before Linux 2.6.23.) This change allows programs to have a much larger argument and/or environment list. For these architectures, the total size is limited to 1/4 of the allowed stack size. (Imposing the 1/4-limit ensures that the new program always has some stack space.) Additionally, the total size is limited to 3/4 of the value of the kernel constant `_STK_LIM` (8 MiB). Since Linux 2.6.25, the kernel also places a floor of 32 pages on this size limit, so that, even when `RLIMIT_STACK` is set very low, applications are guaranteed to have at least as much argument and environment space as was provided by Linux 2.6.22 and earlier. (This guarantee was not provided in Linux 2.6.23 and 2.6.24.) Additionally, the limit per string is 32 pages (the kernel constant `MAX_ARG_STRLEN`), and the maximum number of strings is `0x7FFFFFFF`.

RETURN VALUE [top](#)

On success, `execve()` does not return, on error `-1` is returned, and `errno` is set to indicate the error.

ERRORS [top](#)

- E2BIG** The total number of bytes in the environment (`envp`) and argument list (`argv`) is too large.
- EACCES** Search permission is denied on a component of the path prefix of `pathname` or the name of a script interpreter. (See also [path_resolution\(7\)](#).)
- EACCES** The file or a script interpreter is not a regular file.
- EACCES** Execute permission is denied for the file or a script or ELF interpreter.
- EACCES** The filesystem is mounted `noexec`.
- EAGAIN** (since Linux 3.1)
Having changed its real UID using one of the `set*uid()` calls, the caller was—and is now still—above its `RLIMIT_NPROC` resource limit (see [setrlimit\(2\)](#)). For a more detailed explanation of this error, see NOTES.
- EFAULT** `pathname` or one of the pointers in the vectors `argv` or

envp points outside your accessible address space.

EINVAL An ELF executable had more than one PT_INTERP segment (i.e., tried to name more than one interpreter).

EIO An I/O error occurred.

EISDIR An ELF interpreter was a directory.

ELIBBAD

An ELF interpreter was not in a recognized format.

ELOOP Too many symbolic links were encountered in resolving *pathname* or the name of a script or ELF interpreter.

ELOOP The maximum recursion limit was reached during recursive script interpretation (see "Interpreter scripts", above). Before Linux 3.8, the error produced for this case was **ENOEXEC**.

EMFILE The per-process limit on the number of open file descriptors has been reached.

ENAMETOOLONG

pathname is too long.

ENFILE The system-wide limit on the total number of open files has been reached.

ENOENT The file *pathname* or a script or ELF interpreter does not exist.

ENOEXEC

An executable is not in a recognized format, is for the wrong architecture, or has some other format error that means it cannot be executed.

ENOMEM Insufficient kernel memory was available.

ENOTDIR

A component of the path prefix of *pathname* or a script or ELF interpreter is not a directory.

EPERM The filesystem is mounted *nosuid*, the user is not the superuser, and the file has the set-user-ID or set-group-



ID bit set.

EPERM The process is being traced, the user is not the superuser and the file has the set-user-ID or set-group-ID bit set.

EPERM A "capability-dumb" applications would not obtain the full set of permitted capabilities granted by the executable file. See [capabilities\(7\)](#).

ETXTBSY

The specified executable was open for writing by one or more processes.

VERSIONS [top](#)

POSIX does not document the `#!` behavior, but it exists (with some variations) on other UNIX systems.

On Linux, `argv` and `envp` can be specified as NULL. In both cases, this has the same effect as specifying the argument as a pointer to a list containing a single null pointer. **Do not take advantage of this nonstandard and nonportable misfeature!** On many other UNIX systems, specifying `argv` as NULL will result in an error (**EFAULT**). *Some* other UNIX systems treat the `envp==NULL` case the same as Linux.

POSIX.1 says that values returned by [sysconf\(3\)](#) should be invariant over the lifetime of a process. However, since Linux 2.6.23, if the **RLIMIT_STACK** resource limit changes, then the value reported by **_SC_ARG_MAX** will also change, to reflect the fact that the limit on space for holding command-line arguments and environment variables has changed.

Interpreter scripts

The kernel imposes a maximum length on the text that follows the `#!` characters at the start of a script; characters beyond the limit are ignored. Before Linux 5.1, the limit is 127 characters. Since Linux 5.1, the limit is 255 characters.

The semantics of the `optional-arg` argument of an interpreter script vary across implementations. On Linux, the entire string following the `interpreter` name is passed as a single argument to the interpreter, and this string can include white space. However, behavior differs on some other systems. Some systems use the first white space to terminate `optional-arg`. On some



systems, an interpreter script can have multiple arguments, and white spaces in *optional-arg* are used to delimit the arguments.

Linux (like most other modern UNIX systems) ignores the set-user-ID and set-group-ID bits on scripts.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.3BSD.

With UNIX V6, the argument list of an `exec()` call was ended by `0`, while the argument list of `main` was ended by `-1`. Thus, this argument list was not directly usable in a further `exec()` call. Since UNIX V7, both are `NULL`.

NOTES [top](#)

One sometimes sees `execve()` (and the related functions described in `exec(3)`) described as "executing a *new* process" (or similar). This is a highly misleading description: there is no new process; many attributes of the calling process remain unchanged (in particular, its PID). All that `execve()` does is arrange for an existing process (the calling process) to execute a new program.

Set-user-ID and set-group-ID processes can not be `ptrace(2)`d.

The result of mounting a filesystem *nosuid* varies across Linux kernel versions: some will refuse execution of set-user-ID and set-group-ID executables when this would give the user powers they did not have already (and return `EPERM`), some will just ignore the set-user-ID and set-group-ID bits and `exec()` successfully.

In most cases where `execve()` fails, control returns to the original executable image, and the caller of `execve()` can then handle the error. However, in (rare) cases (typically caused by resource exhaustion), failure may occur past the point of no return: the original executable image has been torn down, but the new image could not be completely built. In such cases, the



kernel kills the process with a **SIGSEGV** (**SIGKILL** until Linux 3.17) signal.

execve() and **EAGAIN**

A more detailed explanation of the **EAGAIN** error that can occur (since Linux 3.1) when calling **execve()** is as follows.

The **EAGAIN** error can occur when a *preceding* call to **setuid(2)**, **setreuid(2)**, or **setresuid(2)** caused the real user ID of the process to change, and that change caused the process to exceed its **RLIMIT_NPROC** resource limit (i.e., the number of processes belonging to the new real UID exceeds the resource limit). From Linux 2.6.0 to Linux 3.0, this caused the **set*uid()** call to fail. (Before Linux 2.6, the resource limit was not imposed on processes that changed their user IDs.)

Since Linux 3.1, the scenario just described no longer causes the **set*uid()** call to fail, because it too often led to security holes where buggy applications didn't check the return status and assumed that—if the caller had root privileges—the call would always succeed. Instead, the **set*uid()** calls now successfully change the real UID, but the kernel sets an internal flag, named **PF_NPROC_EXCEEDED**, to note that the **RLIMIT_NPROC** resource limit has been exceeded. If the **PF_NPROC_EXCEEDED** flag is set and the resource limit is still exceeded at the time of a subsequent **execve()** call, that call fails with the error **EAGAIN**. This kernel logic ensures that the **RLIMIT_NPROC** resource limit is still enforced for the common privileged daemon workflow—namely, **fork(2)** + **set*uid()** + **execve()**.

If the resource limit was not still exceeded at the time of the **execve()** call (because other processes belonging to this real UID terminated between the **set*uid()** call and the **execve()** call), then the **execve()** call succeeds and the kernel clears the **PF_NPROC_EXCEEDED** process flag. The flag is also cleared if a subsequent call to **fork(2)** by this process succeeds.

EXAMPLES [top](#)

The following program is designed to be execed by the second program below. It just echoes its command-line arguments, one per line.

```
/* myecho.c */
```



```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    for (size_t j = 0; j < argc; j++)
        printf("argv[%zu]: %s\n", j, argv[j]);

    exit(EXIT_SUCCESS);
}
```

This program can be used to exec the program named in its command-line argument:

```
/* execve.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    static char *newargv[] = { NULL, "hello", "world", NULL };
    static char *newenviron[] = { NULL };

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file-to-exec>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    newargv[0] = argv[1];

    execve(argv[1], newargv, newenviron);
    perror("execve"); /* execve() returns only on error */
    exit(EXIT_FAILURE);
}
```

We can use the second program to exec the first as follows:

```
$ cc myecho.c -o myecho
$ cc execve.c -o execve
$ ./execve ./myecho
argv[0]: ./myecho
```



```
argv[1]: hello
argv[2]: world
```

We can also use these programs to demonstrate the use of a script interpreter. To do this we create a script whose "interpreter" is our *myecho* program:

```
$ cat > script
#!/myecho script-arg
^D
$ chmod +x script
```

We can then use our program to exec the script:

```
$ ./execve ./script
argv[0]: ./myecho
argv[1]: script-arg
argv[2]: ./script
argv[3]: hello
argv[4]: world
```

SEE ALSO [top](#)

[chmod\(2\)](#), [execveat\(2\)](#), [fork\(2\)](#), [get_robust_list\(2\)](#), [ptrace\(2\)](#), [exec\(3\)](#), [fexecve\(3\)](#), [getauxval\(3\)](#), [getopt\(3\)](#), [system\(3\)](#), [capabilities\(7\)](#), [credentials\(7\)](#), [environ\(7\)](#), [path_resolution\(7\)](#), [ld.so\(8\)](#)

Linux man-pages (unreleased) (date) [execve\(2\)](#)

Pages that refer to this page: [pmcd\(1\)](#), [setpriv\(1\)](#), [strace\(1\)](#), [access\(2\)](#), [alarm\(2\)](#), [arch_prctl\(2\)](#), [brk\(2\)](#), [chdir\(2\)](#), [chmod\(2\)](#), [chroot\(2\)](#), [clone\(2\)](#), [close\(2\)](#), [eventfd\(2\)](#), [execveat\(2\)](#), [_exit\(2\)](#), [fanotify_mark\(2\)](#), [fcntl\(2\)](#), [flock\(2\)](#), [fork\(2\)](#), [getgroups\(2\)](#), [getitimer\(2\)](#), [getpriority\(2\)](#), [getrlimit\(2\)](#), [get_robust_list\(2\)](#), [getrusage\(2\)](#), [ioctl\(2\)](#), [ioctl_console\(2\)](#), [ioperm\(2\)](#), [iopl\(2\)](#), [keyctl\(2\)](#), [madvise\(2\)](#), [memfd_create\(2\)](#), [memfd_secret\(2\)](#), [mlock\(2\)](#), [mount\(2\)](#), [open\(2\)](#), [perf_event_open\(2\)](#), [personality\(2\)](#), [prctl\(2\)](#), [ptrace\(2\)](#), [sched_setaffinity\(2\)](#), [seccomp\(2\)](#), [semop\(2\)](#), [set_mempolicy\(2\)](#), [setpgid\(2\)](#), [setresuid\(2\)](#), [setreuid\(2\)](#), [setsid\(2\)](#), [setuid\(2\)](#), [shmop\(2\)](#), [sigaction\(2\)](#), [sigaltstack\(2\)](#), [signalfd\(2\)](#), [sigpending\(2\)](#), [sigprocmask\(2\)](#), [syscalls\(2\)](#), [timer_create\(2\)](#), [timerfd_create\(2\)](#), [umask\(2\)](#), [vfork\(2\)](#), [cap_get_file\(3\)](#), [cap_iab\(3\)](#), [cap_launch\(3\)](#), [catopen\(3\)](#), [exec\(3\)](#), [exit\(3\)](#), [fexecve\(3\)](#), [getauxval\(3\)](#), [getexeccon\(3\)](#), [getfscreatecon\(3\)](#), [getkeycreatecon\(3\)](#), [getsockcreatecon\(3\)](#), [libexpect\(3\)](#), [mq_close\(3\)](#), [posix_spawn\(3\)](#), [pthread_atfork\(3\)](#), [pthread_kill_other_threads_np\(3\)](#), [pthread_mutexattr_setrobust\(3\)](#), [sd_bus_creds_get_pid\(3\)](#), [sem_close\(3\)](#), [sigvec\(3\)](#), [system\(3\)](#), [auditd-plugins\(5\)](#), [core\(5\)](#), [elf\(5\)](#), [proc\(5\)](#), [systemd.exec\(5\)](#), [systemd-](#)

[system.conf\(5\)](#), [capabilities\(7\)](#), [cgroups\(7\)](#), [credentials\(7\)](#), [environ\(7\)](#), [inode\(7\)](#), [inotify\(7\)](#), [persistent-keyring\(7\)](#), [process-keyring\(7\)](#), [pthreads\(7\)](#), [sched\(7\)](#), [session-keyring\(7\)](#), [signal\(7\)](#), [signal-safety\(7\)](#), [thread-keyring\(7\)](#), [user-keyring\(7\)](#), [user_namespaces\(7\)](#), [user-session-keyring\(7\)](#), [vdso\(7\)](#), [pam_selinux\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *[The Linux Programming Interface](#)*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



fork(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [EXAMPLES](#) | [SEE ALSO](#)

fork(2)

System Calls Manual

fork(2)**NAME** [top](#)

fork - create a child process

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
pid_t fork(void);
```

DESCRIPTION [top](#)

`fork()` creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process and the parent process run in separate memory spaces. At the time of `fork()` both memory spaces have the same content. Memory writes, file mappings (`mmap(2)`), and unmappings (`munmap(2)`) performed by one of the processes do not affect the other.

The child process is an exact duplicate of the parent process except for the following points:

- The child has its own unique process ID, and this PID does not match the ID of any existing process group ([setpgid\(2\)](#)) or session.
- The child's parent process ID is the same as the parent's process ID.
- The child does not inherit its parent's memory locks ([mlock\(2\)](#), [mlockall\(2\)](#)).
- Process resource utilizations ([getrusage\(2\)](#)) and CPU time counters ([times\(2\)](#)) are reset to zero in the child.
- The child's set of pending signals is initially empty ([sigpending\(2\)](#)).
- The child does not inherit semaphore adjustments from its parent ([semop\(2\)](#)).
- The child does not inherit process-associated record locks from its parent ([fcntl\(2\)](#)). (On the other hand, it does inherit [fcntl\(2\)](#) open file description locks and [flock\(2\)](#) locks from its parent.)
- The child does not inherit timers from its parent ([setitimer\(2\)](#), [alarm\(2\)](#), [timer_create\(2\)](#)).
- The child does not inherit outstanding asynchronous I/O operations from its parent ([aio_read\(3\)](#), [aio_write\(3\)](#)), nor does it inherit any asynchronous I/O contexts from its parent (see [io_setup\(2\)](#)).

The process attributes in the preceding list are all specified in POSIX.1. The parent and child also differ with respect to the following Linux-specific process attributes:

- The child does not inherit directory change notifications ([dnotify](#)) from its parent (see the description of **F_NOTIFY** in [fcntl\(2\)](#)).
- The [prctl\(2\)](#) **PR_SET_PDEATHSIG** setting is reset so that the



child does not receive a signal when its parent terminates.

- The default timer slack value is set to the parent's current timer slack value. See the description of **PR_SET_TIMERSLACK** in [prctl\(2\)](#).
- Memory mappings that have been marked with the [madvise\(2\)](#) **MADV_DONTFORK** flag are not inherited across a **fork()**.
- Memory in address ranges that have been marked with the [madvise\(2\)](#) **MADV_WIPEONFORK** flag is zeroed in the child after a **fork()**. (The **MADV_WIPEONFORK** setting remains in place for those address ranges in the child.)
- The termination signal of the child is always **SIGCHLD** (see [clone\(2\)](#)).
- The port access permission bits set by [ioperm\(2\)](#) are not inherited by the child; the child must turn on any bits that it requires using [ioperm\(2\)](#).

Note the following further points:

- The child process is created with a single thread—the one that called **fork()**. The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of [pthread_atfork\(3\)](#) may be helpful for dealing with problems that this can cause.
- After a **fork()** in a multithreaded program, the child can safely call only async-signal-safe functions (see [signal-safety\(7\)](#)) until such time as it calls [execve\(2\)](#).
- The child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description (see [open\(2\)](#)) as the corresponding file descriptor in the parent. This means that the two file descriptors share open file status flags, file offset, and signal-driven I/O attributes (see the description of **F_SETOWN** and **F_SETSIG** in [fcntl\(2\)](#)).
- The child inherits copies of the parent's set of open message queue descriptors (see [mq_overview\(7\)](#)). Each file descriptor



in the child refers to the same open message queue description as the corresponding file descriptor in the parent. This means that the two file descriptors share the same flags (*mq_flags*).

- The child inherits copies of the parent's set of open directory streams (see [opendir\(3\)](#)). POSIX.1 says that the corresponding directory streams in the parent and child *may* share the directory stream positioning; on Linux/glibc they do not.

RETURN VALUE [top](#)

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and *errno* is set to indicate the error.

ERRORS [top](#)

EAGAIN A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error:

- the **RLIMIT_NPROC** soft resource limit (set via [setrlimit\(2\)](#)), which limits the number of processes and threads for a real user ID, was reached;
- the kernel's system-wide limit on the number of processes and threads, */proc/sys/kernel/threads-max*, was reached (see [proc\(5\)](#));
- the maximum number of PIDs, */proc/sys/kernel/pid_max*, was reached (see [proc\(5\)](#)); or
- the PID limit (*pids.max*) imposed by the cgroup "process number" (PIDs) controller was reached.

EAGAIN The caller is operating under the **SCHED_DEADLINE** scheduling policy and does not have the reset-on-fork flag set. See [sched\(7\)](#).



ENOMEM `fork()` failed to allocate the necessary kernel structures because memory is tight.

ENOMEM An attempt was made to create a child process in a PID namespace whose "init" process has terminated. See [pid_namespaces\(7\)](#).

ENOSYS `fork()` is not supported on this platform (for example, hardware without a Memory-Management Unit).

ERESTARTNOINTR (since Linux 2.6.17)
System call was interrupted by a signal and will be restarted. (This can be seen only during a trace.)

VERSIONS [top](#)

C library/kernel differences

Since glibc 2.3.3, rather than invoking the kernel's `fork()` system call, the glibc `fork()` wrapper that is provided as part of the NPTL threading implementation invokes [clone\(2\)](#) with flags that provide the same effect as the traditional system call. (A call to `fork()` is equivalent to a call to [clone\(2\)](#) specifying *flags* as just **SIGCHLD**.) The glibc wrapper invokes any fork handlers that have been established using [pthread_atfork\(3\)](#).

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.3BSD.

NOTES [top](#)

Under Linux, `fork()` is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

EXAMPLES [top](#)

See [pipe\(2\)](#) and [wait\(2\)](#) for more examples.

```
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(void)
{
    pid_t pid;

    if (signal(SIGCHLD, SIG_IGN) == SIG_ERR) {
        perror("signal");
        exit(EXIT_FAILURE);
    }
    pid = fork();
    switch (pid) {
    case -1:
        perror("fork");
        exit(EXIT_FAILURE);
    case 0:
        puts("Child exiting.");
        exit(EXIT_SUCCESS);
    default:
        printf("Child is PID %jd\n", (intmax_t) pid);
        puts("Parent exiting.");
        exit(EXIT_SUCCESS);
    }
}
```

SEE ALSO [top](#)

[clone\(2\)](#), [execve\(2\)](#), [exit\(2\)](#), [setrlimit\(2\)](#), [unshare\(2\)](#), [vfork\(2\)](#), [wait\(2\)](#), [daemon\(3\)](#), [pthread_atfork\(3\)](#), [capabilities\(7\)](#), [credentials\(7\)](#)

Linux man-pages (unreleased) (date)

[fork\(2\)](#)



Pages that refer to this page: [chrt\(1\)](#), [dbpmda\(1\)](#), [pmcd\(1\)](#), [setsid\(1\)](#), [strace\(1\)](#), [xargs\(1\)](#), [alarm\(2\)](#), [arch_prctl\(2\)](#), [bpf\(2\)](#), [chdir\(2\)](#), [chroot\(2\)](#), [clone\(2\)](#), [eventfd\(2\)](#), [execve\(2\)](#), [_exit\(2\)](#), [fcntl\(2\)](#), [flock\(2\)](#), [getitimer\(2\)](#), [getpid\(2\)](#), [getpriority\(2\)](#), [getrlimit\(2\)](#), [gettid\(2\)](#), [ioctl_userfaultfd\(2\)](#), [ioperm\(2\)](#), [iopl\(2\)](#), [kcmp\(2\)](#), [keyctl\(2\)](#), [lseek\(2\)](#), [madvise\(2\)](#), [memfd_create\(2\)](#), [memfd_secret\(2\)](#), [mlock\(2\)](#), [mmap\(2\)](#), [mount\(2\)](#), [nice\(2\)](#), [open\(2\)](#), [perf_event_open\(2\)](#), [pidfd_open\(2\)](#), [pipe\(2\)](#), [prctl\(2\)](#), [ptrace\(2\)](#), [sched_setaffinity\(2\)](#), [sched_setattr\(2\)](#), [sched_setscheduler\(2\)](#), [seccomp\(2\)](#), [select_tut\(2\)](#), [semop\(2\)](#), [set_mempolicy\(2\)](#), [setns\(2\)](#), [setpgid\(2\)](#), [setsid\(2\)](#), [shmop\(2\)](#), [sigaction\(2\)](#), [sigaltstack\(2\)](#), [signalfd\(2\)](#), [sigpending\(2\)](#), [sigprocmask\(2\)](#), [syscalls\(2\)](#), [timer_create\(2\)](#), [timerfd_create\(2\)](#), [umask\(2\)](#), [unshare\(2\)](#), [userfaultfd\(2\)](#), [vfork\(2\)](#), [wait\(2\)](#), [wait4\(2\)](#), [atexit\(3\)](#), [cap_launch\(3\)](#), [daemon\(3\)](#), [exec\(3\)](#), [id_t\(3type\)](#), [ltng-ust\(3\)](#), [on_exit\(3\)](#), [openpty\(3\)](#), [pam_end\(3\)](#), [__pmprocessexec\(3\)](#), [__pmprocesspipe\(3\)](#), [popen\(3\)](#), [posix_spawn\(3\)](#), [pthread_atfork\(3\)](#), [sd_bus_creds_get_pid\(3\)](#), [sem_init\(3\)](#), [system\(3\)](#), [core\(5\)](#), [proc\(5\)](#), [systemd.exec\(5\)](#), [capabilities\(7\)](#), [cgroups\(7\)](#), [cpuset\(7\)](#), [credentials\(7\)](#), [environ\(7\)](#), [epoll\(7\)](#), [mq_overview\(7\)](#), [persistent-keyring\(7\)](#), [pid_namespaces\(7\)](#), [pipe\(7\)](#), [pthreads\(7\)](#), [sched\(7\)](#), [session-keyring\(7\)](#), [signal\(7\)](#), [signal-safety\(7\)](#), [thread-keyring\(7\)](#), [user-keyring\(7\)](#), [user_namespaces\(7\)](#), [user-session-keyring\(7\)](#), [btrfs-balance\(8\)](#), [lslocks\(8\)](#), [trafgen\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



exit(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

exit(3)

Library Functions Manual

exit(3)

NAME [top](#)

exit - cause normal process termination

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdlib.h>
```

```
[[noreturn]] void exit(int status);
```

DESCRIPTION [top](#)

The `exit()` function causes normal process termination and the least significant byte of *status* (i.e., *status* & `0xFF`) is returned to the parent (see [wait\(2\)](#)).

All functions registered with [atexit\(3\)](#) and [on_exit\(3\)](#) are called, in the reverse order of their registration. (It is possible for one of these functions to use [atexit\(3\)](#) or [on_exit\(3\)](#) to register an additional function to be executed during exit processing; the new registration is added to the front of the list of functions that remain to be called.) If one



of these functions does not return (e.g., it calls `_exit(2)`, or kills itself with a signal), then none of the remaining functions is called, and further exit processing (in particular, flushing of `stdio(3)` streams) is abandoned. If a function has been registered multiple times using `atexit(3)` or `on_exit(3)`, then it is called as many times as it was registered.

All open `stdio(3)` streams are flushed and closed. Files created by `tmpfile(3)` are removed.

The C standard specifies two constants, **EXIT_SUCCESS** and **EXIT_FAILURE**, that may be passed to `exit()` to indicate successful or unsuccessful termination, respectively.

RETURN VALUE [top](#)

The `exit()` function does not return.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|---------------------|---------------|---------------------|
| <code>exit()</code> | Thread safety | MT-Unsafe race:exit |

The `exit()` function uses a global variable that is not protected, so it is not thread-safe.

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

C89, POSIX.1-2001, SVr4, 4.3BSD.

NOTES [top](#)

The behavior is undefined if one of the functions registered using `atexit(3)` and `on_exit(3)` calls either `exit()` or `longjmp(3)`. Note that a call to `execve(2)` removes registrations created using `atexit(3)` and `on_exit(3)`.

The use of `EXIT_SUCCESS` and `EXIT_FAILURE` is slightly more portable (to non-UNIX environments) than the use of `0` and some nonzero value like `1` or `-1`. In particular, VMS uses a different convention.

BSD has attempted to standardize exit codes (which some C libraries such as the GNU C library have also adopted); see the file `<syssexits.h>`.

After `exit()`, the exit status must be transmitted to the parent process. There are three cases:

- If the parent has set `SA_NOCLDWAIT`, or has set the `SIGCHLD` handler to `SIG_IGN`, the status is discarded and the child dies immediately.
- If the parent was waiting on the child, it is notified of the exit status and the child dies immediately.
- Otherwise, the child becomes a "zombie" process: most of the process resources are recycled, but a slot containing minimal information about the child process (termination status, resource usage statistics) is retained in process table. This allows the parent to subsequently use `waitpid(2)` (or similar) to learn the termination status of the child; at that point the zombie process slot is released.

If the implementation supports the `SIGCHLD` signal, this signal is sent to the parent. If the parent has set `SA_NOCLDWAIT`, it is undefined whether a `SIGCHLD` signal is sent.

Signals sent to other processes

If the exiting process is a session leader and its controlling terminal is the controlling terminal of the session, then each process in the foreground process group of this controlling terminal is sent a `SIGHUP` signal, and the terminal is disassociated from this session, allowing it to be acquired by a new controlling process.

If the exit of the process causes a process group to become orphaned, and if any member of the newly orphaned process group is stopped, then a **SIGHUP** signal followed by a **SIGCONT** signal will be sent to each process in this process group. See [setpgid\(2\)](#) for an explanation of orphaned process groups.

Except in the above cases, where the signalled processes may be children of the terminating process, termination of a process does *not* in general cause a signal to be sent to children of that process. However, a process can use the [prctl\(2\)](#) **PR_SET_PDEATHSIG** operation to arrange that it receives a signal if its parent terminates.

SEE ALSO [top](#)

[_exit\(2\)](#), [get_robust_list\(2\)](#), [setpgid\(2\)](#), [wait\(2\)](#), [atexit\(3\)](#), [on_exit\(3\)](#), [tmpfile\(3\)](#)

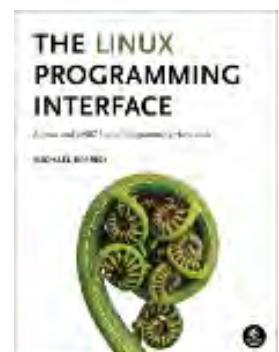
Linux man-pages (unreleased) (date) [exit\(3\)](#)

Pages that refer to this page: [man\(1\)](#), [_exit\(2\)](#), [kill\(2\)](#), [vfork\(2\)](#), [wait\(2\)](#), [abort\(3\)](#), [assert\(3\)](#), [assert_perror\(3\)](#), [atexit\(3\)](#), [err\(3\)](#), [error\(3\)](#), [EXIT_SUCCESS\(3const\)](#), [on_exit\(3\)](#), [pthread_create\(3\)](#), [pthread_detach\(3\)](#), [pthread_exit\(3\)](#), [sd_bus_set_exit_on_disconnect\(3\)](#), [setjmp\(3\)](#), [stdin\(3\)](#), [stdio\(3\)](#), [sysexits.h\(3head\)](#), [tmpfile\(3\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



`_exit(2)` — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

[_exit\(2\)](#)

System Calls Manual

[_exit\(2\)](#)

NAME [top](#)

`_exit`, `_Exit` - terminate the calling process

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
[[noreturn]] void _exit(int status);
```

```
#include <stdlib.h>
```

```
[[noreturn]] void _Exit(int status);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
_Exit():  
    _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
```

DESCRIPTION [top](#)

`_exit()` terminates the calling process "immediately". Any open file descriptors belonging to the process are closed. Any children of the process are inherited by `init(1)` (or by the nearest "subreaper" process as defined through the use of the `prctl(2)` `PR_SET_CHILD_SUBREAPER` operation). The process's parent is sent a `SIGCHLD` signal.

The value `status & 0xFF` is returned to the parent process as the process's exit status, and can be collected by the parent using one of the `wait(2)` family of calls.

The function `_Exit()` is equivalent to `_exit()`.

RETURN VALUE [top](#)

These functions do not return.

STANDARDS [top](#)

`_exit()`
POSIX.1-2008.

`_Exit()`
C11, POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.3BSD.

`_Exit()` was introduced by C99.

NOTES [top](#)

For a discussion on the effects of an exit, the transmission of exit status, zombie processes, signals sent, and so on, see `exit(3)`.

The function `_exit()` is like `exit(3)`, but does not call any functions registered with `atexit(3)` or `on_exit(3)`. Open `stdio(3)` streams are not flushed. On the other hand, `_exit()` does close open file descriptors, and this may cause an unknown delay,

waiting for pending output to finish. If the delay is undesired, it may be useful to call functions like [tcflush\(3\)](#) before calling `_exit()`. Whether any pending I/O is canceled, and which pending I/O may be canceled upon `_exit()`, is implementation-dependent.

C library/kernel differences

The text above in DESCRIPTION describes the traditional effect of `_exit()`, which is to terminate a process, and these are the semantics specified by POSIX.1 and implemented by the C library wrapper function. On modern systems, this means termination of all threads in the process.

By contrast with the C library wrapper function, the raw Linux `_exit()` system call terminates only the calling thread, and actions such as reparenting child processes or sending **SIGCHLD** to the parent process are performed only if this is the last thread in the thread group.

Up to glibc 2.3, the `_exit()` wrapper function invoked the kernel system call of the same name. Since glibc 2.3, the wrapper function invokes [exit_group\(2\)](#), in order to terminate all of the threads in a process.

SEE ALSO [top](#)

[execve\(2\)](#), [exit_group\(2\)](#), [fork\(2\)](#), [kill\(2\)](#), [wait\(2\)](#), [wait4\(2\)](#), [waitpid\(2\)](#), [atexit\(3\)](#), [exit\(3\)](#), [on_exit\(3\)](#), [termios\(3\)](#)

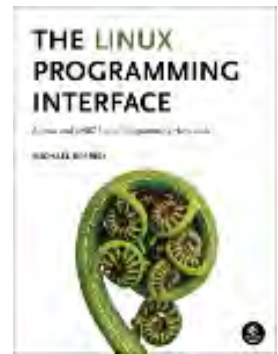
Linux man-pages (unreleased) (date) [_exit\(2\)](#)

Pages that refer to this page: [clone\(2\)](#), [exit_group\(2\)](#), [fork\(2\)](#), [kill\(2\)](#), [ptrace\(2\)](#), [seccomp\(2\)](#), [setsid\(2\)](#), [shmop\(2\)](#), [syscalls\(2\)](#), [vfork\(2\)](#), [wait\(2\)](#), [atexit\(3\)](#), [daemon\(3\)](#), [exit\(3\)](#), [on_exit\(3\)](#), [pmgetconfig\(3\)](#), [pmmomem\(3\)](#), [system\(3\)](#), [persistent-keyring\(7\)](#), [signal-safety\(7\)](#), [socket\(7\)](#), [user-keyring\(7\)](#), [user-session-keyring\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



atexit(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [EXAMPLES](#) | [SEE ALSO](#)

 atexit(3)

Library Functions Manual

atexit(3)

NAME [top](#)

atexit - register a function to be called at normal process termination

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdlib.h>
```

```
int atexit(void (*function)(void));
```

DESCRIPTION [top](#)

The **atexit()** function registers the given *function* to be called at normal process termination, either via **exit(3)** or via return from the program's *main()*. Functions so registered are called in the reverse order of their registration; no arguments are passed.

The same function may be registered multiple times: it is called once for each registration.

POSIX.1 requires that an implementation allow at least **ATEXIT_MAX**

(32) such functions to be registered. The actual limit supported by an implementation can be obtained using [sysconf\(3\)](#).

When a child process is created via [fork\(2\)](#), it inherits copies of its parent's registrations. Upon a successful call to one of the [exec\(3\)](#) functions, all registrations are removed.

RETURN VALUE [top](#)

The **atexit()** function returns the value 0 if successful; otherwise it returns a nonzero value.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------------|---------------|---------|
| atexit() | Thread safety | MT-Safe |

VERSIONS [top](#)

POSIX.1 says that the result of calling [exit\(3\)](#) more than once (i.e., calling [exit\(3\)](#) within a function registered using **atexit()**) is undefined. On some systems (but not Linux), this can result in an infinite recursion; portable programs should not invoke [exit\(3\)](#) inside a function registered using **atexit()**.

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, C89, C99, SVr4, 4.3BSD.

NOTES [top](#)

Functions registered using **atexit()** (and **on_exit(3)**) are not called if a process terminates abnormally because of the delivery of a signal.

If one of the registered functions calls **_exit(2)**, then any remaining functions are not invoked, and the other process termination steps performed by **exit(3)** are not performed.

The **atexit()** and **on_exit(3)** functions register functions on the same list: at normal process termination, the registered functions are invoked in reverse order of their registration by these two functions.

According to POSIX.1, the result is undefined if **longjmp(3)** is used to terminate execution of one of the functions registered using **atexit()**.

Linux notes

Since glibc 2.2.3, **atexit()** (and **on_exit(3)**) can be used within a shared library to establish functions that are called when the shared library is unloaded.

EXAMPLES [top](#)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
bye(void)
{
    printf("That was all, folks\n");
}

int
main(void)
{
    long a;
    int i;

    a = sysconf(_SC_ATEXIT_MAX);
    printf("ATEXIT_MAX = %ld\n", a);
}
```

```
i = atexit(bye);
if (i != 0) {
    fprintf(stderr, "cannot set exit function\n");
    exit(EXIT_FAILURE);
}

exit(EXIT_SUCCESS);
}
```

SEE ALSO [top](#)

[_exit\(2\)](#), [dlopen\(3\)](#), [exit\(3\)](#), [on_exit\(3\)](#)

Linux man-pages (unreleased) (date)

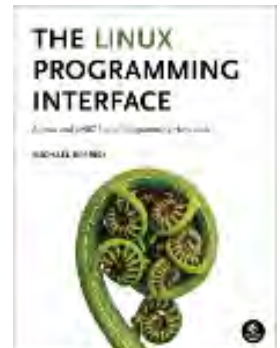
atexit(3)

Pages that refer to this page: [execve\(2\)](#), [_exit\(2\)](#), [abort\(3\)](#), [dlopen\(3\)](#), [exit\(3\)](#), [on_exit\(3\)](#), [pmdaopenlog\(3\)](#), [pmfault\(3\)](#), [pmopenlog\(3\)](#), [pthread_atfork\(3\)](#), [pthread_exit\(3\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



wait(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [EXAMPLES](#) | [SEE ALSO](#)

 wait(2)

System Calls Manual

wait(2)**NAME** [top](#)

wait, waitpid, waitid - wait for process to change state

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/wait.h>
```

```
pid_t wait(int *_Nullable wstatus);
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);
```

```
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
/* This is the glibc and POSIX interface; see
   NOTES for information on the raw system call. */
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

waitid():

Since glibc 2.26:

```
    _XOPEN_SOURCE >= 500 || _POSIX_C_SOURCE >= 200809L
```

glibc 2.25 and earlier:

```
    _XOPEN_SOURCE
```

```
    || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
```

```
    || /* glibc <= 2.19: */ _BSD_SOURCE
```



DESCRIPTION [top](#)

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).

If a child has already changed state, then these calls return immediately. Otherwise, they block until either a child changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the **SA_RESTART** flag of [sigaction\(2\)](#)). In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed *waitable*.

wait() and **waitpid()**

The **wait()** system call suspends execution of the calling thread until one of its children terminates. The call *wait(&wstatus)* is equivalent to:

```
waitpid(-1, &wstatus, 0);
```

The **waitpid()** system call suspends execution of the calling thread until a child specified by *pid* argument has changed state. By default, **waitpid()** waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below.

The value of *pid* can be:

- < **-1** meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.
- 1** meaning wait for any child process.
- 0** meaning wait for any child process whose process group ID is equal to that of the calling process at the time of the call to **waitpid()**.
- > **0** meaning wait for the child whose process ID is equal to

the value of *pid*.

The value of *options* is an OR of zero or more of the following constants:

WNOHANG

return immediately if no child has exited.

WUNTRACED

also return if a child has stopped (but not traced via `ptrace(2)`). Status for *traced* children which have stopped is provided even if this option is not specified.

WCONTINUED (since Linux 2.6.10)

also return if a stopped child has been resumed by delivery of **SIGCONT**.

(For Linux-only options, see below.)

If *wstatus* is not NULL, `wait()` and `waitpid()` store status information in the *int* to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in `wait()` and `waitpid(!)`):

WIFEXITED(*wstatus*)

returns true if the child terminated normally, that is, by calling `exit(3)` or `_exit(2)`, or by returning from `main()`.

WEXITSTATUS(*wstatus*)

returns the exit status of the child. This consists of the least significant 8 bits of the *status* argument that the child specified in a call to `exit(3)` or `_exit(2)` or as the argument for a return statement in `main()`. This macro should be employed only if **WIFEXITED** returned true.

WIFSIGNALED(*wstatus*)

returns true if the child process was terminated by a signal.

WTERMSIG(*wstatus*)

returns the number of the signal that caused the child process to terminate. This macro should be employed only if **WIFSIGNALED** returned true.

WCOREDUMP(*wstatus*)

returns true if the child produced a core dump (see [core\(5\)](#)). This macro should be employed only if **WIFSIGNALED** returned true.

This macro is not specified in POSIX.1-2001 and is not available on some UNIX implementations (e.g., AIX, SunOS). Therefore, enclose its use inside `#ifdef WCOREDUMP ... #endif`.

WIFSTOPPED(*wstatus*)

returns true if the child process was stopped by delivery of a signal; this is possible only if the call was done using **WUNTRACED** or when the child is being traced (see [ptrace\(2\)](#)).

WSTOPSIG(*wstatus*)

returns the number of the signal which caused the child to stop. This macro should be employed only if **WIFSTOPPED** returned true.

WIFCONTINUED(*wstatus*)

(since Linux 2.6.10) returns true if the child process was resumed by delivery of **SIGCONT**.

waitid()

The **waitid()** system call (available since Linux 2.6.9) provides more precise control over which child state changes to wait for.

The *idtype* and *id* arguments select the child(ren) to wait for, as follows:

idtype == **P_PID**

Wait for the child whose process ID matches *id*.

idtype == **P_PIDFD** (since Linux 5.4)

Wait for the child referred to by the PID file descriptor specified in *id*. (See [pidfd_open\(2\)](#) for further information on PID file descriptors.)

idtype == **P_PGID**

Wait for any child whose process group ID matches *id*. Since Linux 5.4, if *id* is zero, then wait for any child that is in the same process group as the caller's process group at the time of the call.

idtype == **P_ALL**



Wait for any child; *id* is ignored.

The child state changes to wait for are specified by ORing one or more of the following flags in *options*:

WEXITED

Wait for children that have terminated.

WSTOPPED

Wait for children that have been stopped by delivery of a signal.

WCONTINUED

Wait for (previously stopped) children that have been resumed by delivery of **SIGCONT**.

The following flags may additionally be ORed in *options*:

WNOHANG

As for **waitpid()**.

WNOWAIT

Leave the child in a waitable state; a later wait call can be used to again retrieve the child status information.

Upon successful return, **waitid()** fills in the following fields of the *siginfo_t* structure pointed to by *infop*:

si_pid The process ID of the child.

si_uid The real user ID of the child. (This field is not set on most other implementations.)

si_signo
Always set to **SIGCHLD**.

si_status
Either the exit status of the child, as given to **_exit(2)** (or **exit(3)**), or the signal that caused the child to terminate, stop, or continue. The *si_code* field can be used to determine how to interpret this field.

si_code
Set to one of: **CLD_EXITED** (child called **_exit(2)**); **CLD_KILLED** (child killed by signal); **CLD_DUMPED** (child killed by signal, and dumped core); **CLD_STOPPED** (child



stopped by signal); **CLD_TRAPPED** (traced child has trapped); or **CLD_CONTINUED** (child continued by **SIGCONT**).

If **WNOHANG** was specified in *options* and there were no children in a waitable state, then **waitid()** returns 0 immediately and the state of the *siginfo_t* structure pointed to by *infop* depends on the implementation. To (portably) distinguish this case from that where a child was in a waitable state, zero out the *si_pid* field before the call and check for a nonzero value in this field after the call returns.

POSIX.1-2008 Technical Corrigendum 1 (2013) adds the requirement that when **WNOHANG** is specified in *options* and there were no children in a waitable state, then **waitid()** should zero out the *si_pid* and *si_signo* fields of the structure. On Linux and other implementations that adhere to this requirement, it is not necessary to zero out the *si_pid* field before calling **waitid()**. However, not all implementations follow the POSIX.1 specification on this point.

RETURN VALUE [top](#)

wait(): on success, returns the process ID of the terminated child; on failure, -1 is returned.

waitpid(): on success, returns the process ID of the child whose state has changed; if **WNOHANG** was specified and one or more child(ren) specified by *pid* exist, but have not yet changed state, then 0 is returned. On failure, -1 is returned.

waitid(): returns 0 on success or if **WNOHANG** was specified and no child(ren) specified by *id* has yet changed state; on failure, -1 is returned.

On failure, each of these calls sets *errno* to indicate the error.

ERRORS [top](#)

EAGAIN The PID file descriptor specified in *id* is nonblocking and the process that it refers to has not terminated.

ECHILD (for **wait()**) The calling process does not have any unwaited-for children.

ECHILD (for `waitpid()` or `waitid()`) The process specified by *pid* (`waitpid()`) or *idtype* and *id* (`waitid()`) does not exist or is not a child of the calling process. (This can happen for one's own child if the action for **SIGCHLD** is set to **SIG_IGN**. See also the *Linux Notes* section about threads.)

EINTR **WNOHANG** was not set and an unblocked signal or a **SIGCHLD** was caught; see [signal\(7\)](#).

EINVAL The *options* argument was invalid.

ESRCH (for `wait()` or `waitpid()`) *pid* is equal to **INT_MIN**.

VERSIONS [top](#)

C library/kernel differences

`wait()` is actually a library function that (in glibc) is implemented as a call to [wait4\(2\)](#).

On some architectures, there is no `waitpid()` system call; instead, this interface is implemented via a C library wrapper function that calls [wait4\(2\)](#).

The raw `waitid()` system call takes a fifth argument, of type *struct rusage **. If this argument is non-NULL, then it is used to return resource usage information about the child, in the same manner as [wait4\(2\)](#). See [getrusage\(2\)](#) for details.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

SVr4, 4.3BSD, POSIX.1-2001.

NOTES [top](#)

A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait

to obtain information about the child. As long as a zombie is not removed from the system via a `wait`, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by `init(1)`, (or by the nearest "subreaper" process as defined through the use of the `prctl(2)` `PR_SET_CHILD_SUBREAPER` operation); `init(1)` automatically performs a `wait` to remove the zombies.

POSIX.1-2001 specifies that if the disposition of `SIGCHLD` is set to `SIG_IGN` or the `SA_NOCLDWAIT` flag is set for `SIGCHLD` (see `sigaction(2)`), then children that terminate do not become zombies and a call to `wait()` or `waitpid()` will block until all children have terminated, and then fail with `errno` set to `ECHILD`. (The original POSIX standard left the behavior of setting `SIGCHLD` to `SIG_IGN` unspecified. Note that even though the default disposition of `SIGCHLD` is "ignore", explicitly setting the disposition to `SIG_IGN` results in different treatment of zombie process children.)

Linux 2.6 conforms to the POSIX requirements. However, Linux 2.4 (and earlier) does not: if a `wait()` or `waitpid()` call is made while `SIGCHLD` is being ignored, the call behaves just as though `SIGCHLD` were not being ignored, that is, the call blocks until the next child terminates and then returns the process ID and status of that child.

Linux notes

In the Linux kernel, a kernel-scheduled thread is not a distinct construct from a process. Instead, a thread is simply a process that is created using the Linux-unique `clone(2)` system call; other routines such as the portable `pthread_create(3)` call are implemented using `clone(2)`. Before Linux 2.4, a thread was just a special case of a process, and as a consequence one thread could not wait on the children of another thread, even when the latter belongs to the same thread group. However, POSIX prescribes such functionality, and since Linux 2.4 a thread can, and by default will, wait on children of other threads in the same thread group.

The following Linux-specific *options* are for use with children created using `clone(2)`; they can also, since Linux 4.7, be used with `waitid()`:

__WCLONE



Wait for "clone" children only. If omitted, then wait for "non-clone" children only. (A "clone" child is one which delivers no signal, or a signal other than **SIGCHLD** to its parent upon termination.) This option is ignored if **__WALL** is also specified.

__WALL (since Linux 2.4)

Wait for all children, regardless of type ("clone" or "non-clone").

__WNOTHREAD (since Linux 2.4)

Do not wait for children of other threads in the same thread group. This was the default before Linux 2.4.

Since Linux 4.7, the **__WALL** flag is automatically implied if the child is being ptraced.

BUGS

[top](#)

According to POSIX.1-2008, an application calling **waitid()** must ensure that *infop* points to a *siginfo_t* structure (i.e., that it is a non-null pointer). On Linux, if *infop* is NULL, **waitid()** succeeds, and returns the process ID of the waited-for child. Applications should avoid relying on this inconsistent, nonstandard, and unnecessary feature.

EXAMPLES

[top](#)

The following program demonstrates the use of **fork(2)** and **waitpid()**. The program creates a child process. If no command-line argument is supplied to the program, then the child suspends its execution using **pause(2)**, to allow the user to send signals to the child. Otherwise, if a command-line argument is supplied, then the child exits immediately, using the integer supplied on the command line as the exit status. The parent process executes a loop that monitors the child using **waitpid()**, and uses the **W*()** macros described above to analyze the wait status value.

The following shell session demonstrates the use of the program:

```
$ ./a.out &  
Child PID is 32360  
[1] 32359  
$ kill -STOP 32360
```



```
stopped by signal 19
$ kill -CONT 32360
continued
$ kill -TERM 32360
killed by signal 15
[1]+  Done                    ./a.out
$
```

Program source

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int    wstatus;
    pid_t  cpid, w;

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {          /* Code executed by child */
        printf("Child PID is %jd\n", (intmax_t) getpid());
        if (argc == 1)
            pause();          /* Wait for signals */
        _exit(atoi(argv[1]));
    } else {                  /* Code executed by parent */
        do {
            w = waitpid(cpid, &wstatus, WUNTRACED | WCONTINUED);
            if (w == -1) {
                perror("waitpid");
                exit(EXIT_FAILURE);
            }

            if (WIFEXITED(wstatus)) {
                printf("exited, status=%d\n", WEXITSTATUS(wstatus));
            } else if (WIFSIGNALED(wstatus)) {
                printf("killed by signal %d\n", WTERMSIG(wstatus));
            }
        } while (w != 0);
    }
}
```



```
    } else if (WIFSTOPPED(wstatus)) {
        printf("stopped by signal %d\n", WSTOPSIG(wstatus));
    } else if (WIFCONTINUED(wstatus)) {
        printf("continued\n");
    }
} while (!WIFEXITED(wstatus) && !WIFSIGNALED(wstatus));
exit(EXIT_SUCCESS);
}
```

SEE ALSO [top](#)

[_exit\(2\)](#), [clone\(2\)](#), [fork\(2\)](#), [kill\(2\)](#), [ptrace\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [wait4\(2\)](#), [pthread_create\(3\)](#), [core\(5\)](#), [credentials\(7\)](#), [signal\(7\)](#)

Linux man-pages (unreleased) (date) [wait\(2\)](#)

Pages that refer to this page: [intro\(1\)](#), [waitpid\(1\)](#), [clone\(2\)](#), [_exit\(2\)](#), [fork\(2\)](#), [getrusage\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [kill\(2\)](#), [pidfd_open\(2\)](#), [prctl\(2\)](#), [ptrace\(2\)](#), [reboot\(2\)](#), [seccomp\(2\)](#), [seccomp_unotify\(2\)](#), [sigaction\(2\)](#), [syscalls\(2\)](#), [times\(2\)](#), [vfork\(2\)](#), [wait4\(2\)](#), [clock\(3\)](#), [exit\(3\)](#), [id_t\(3type\)](#), [io_uring_prep_waitid\(3\)](#), [__pmprocessexec\(3\)](#), [__pmprocesspipe\(3\)](#), [pmrecord\(3\)](#), [posix_spawn\(3\)](#), [pthread_exit\(3\)](#), [sd-event\(3\)](#), [sd_event_add_child\(3\)](#), [sd_event_add_inotify\(3\)](#), [system\(3\)](#), [proc\(5\)](#), [systemd.exec\(5\)](#), [credentials\(7\)](#), [man-pages\(7\)](#), [threads\(7\)](#), [signal\(7\)](#), [signal-safety\(7\)](#), [user_namespaces\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



system(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

 system(3)

Library Functions Manual

system(3)**NAME** [top](#)

system - execute a shell command

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdlib.h>
```

```
int system(const char *command);
```

DESCRIPTION [top](#)

The **system()** library function behaves as if it used [fork\(2\)](#) to create a child process that executed the shell command specified in *command* using [execl\(3\)](#) as follows:

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

system() returns after the command has been completed.

During execution of the command, **SIGCHLD** will be blocked, and **SIGINT** and **SIGQUIT** will be ignored, in the process that calls **system()**. (These signals will be handled according to their

defaults inside the child process that executes *command*.)

If *command* is NULL, then **system()** returns a status indicating whether a shell is available on the system.

RETURN VALUE [top](#)

The return value of **system()** is one of the following:

- If *command* is NULL, then a nonzero value if a shell is available, or 0 if no shell is available.
- If a child process could not be created, or its status could not be retrieved, the return value is -1 and *errno* is set to indicate the error.
- If a shell could not be executed in the child process, then the return value is as though the child shell terminated by calling **_exit(2)** with the status 127.
- If all system calls succeed, then the return value is the termination status of the child shell used to execute *command*. (The termination status of a shell is the termination status of the last command it executes.)

In the last two cases, the return value is a "wait status" that can be examined using the macros described in **waitpid(2)**. (i.e., **WIFEXITED()**, **WEXITSTATUS()**, and so on).

system() does not affect the wait status of any other children.

ERRORS [top](#)

system() can fail with any of the same errors as **fork(2)**.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see **attributes(7)**.

| Interface | Attribute | Value |
|-----------|-----------|-------|
| | | |



| | | |
|-----------------------|---------------|---------|
| <code>system()</code> | Thread safety | MT-Safe |
|-----------------------|---------------|---------|

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, C89.

NOTES [top](#)

`system()` provides simplicity and convenience: it handles all of the details of calling `fork(2)`, `execl(3)`, and `waitpid(2)`, as well as the necessary manipulations of signals; in addition, the shell performs the usual substitutions and I/O redirections for *command*. The main cost of `system()` is inefficiency: additional system calls are required to create the process that runs the shell and to execute the shell.

If the `_XOPEN_SOURCE` feature test macro is defined (before including *any* header files), then the macros described in `waitpid(2)` (`WEXITSTATUS()`, etc.) are made available when including `<stdlib.h>`.

As mentioned, `system()` ignores `SIGINT` and `SIGQUIT`. This may make programs that call it from a loop uninterruptible, unless they take care themselves to check the exit status of the child. For example:

```
while (something) {
    int ret = system("foo");

    if (WIFSIGNALED(ret) &&
        (WTERMSIG(ret) == SIGINT || WTERMSIG(ret) == SIGQUIT))
        break;
}
```

According to POSIX.1, it is unspecified whether handlers registered using `pthread_atfork(3)` are called during the execution of `system()`. In the glibc implementation, such

handlers are not called.

Before glibc 2.1.3, the check for the availability of `/bin/sh` was not actually performed if `command` was NULL; instead it was always assumed to be available, and `system()` always returned 1 in this case. Since glibc 2.1.3, this check is performed because, even though POSIX.1-2001 requires a conforming implementation to provide a shell, that shell may not be available or executable if the calling program has previously called `chroot(2)` (which is not specified by POSIX.1-2001).

It is possible for the shell command to terminate with a status of 127, which yields a `system()` return value that is indistinguishable from the case where a shell could not be executed in the child process.

Caveats

Do not use `system()` from a privileged program (a set-user-ID or set-group-ID program, or a program with capabilities) because strange values for some environment variables might be used to subvert system integrity. For example, `PATH` could be manipulated so that an arbitrary program is executed with privilege. Use the `exec(3)` family of functions instead, but not `execlp(3)` or `execvp(3)` (which also use the `PATH` environment variable to search for an executable).

`system()` will not, in fact, work properly from programs with set-user-ID or set-group-ID privileges on systems on which `/bin/sh` is bash version 2: as a security measure, bash 2 drops privileges on startup. (Debian uses a different shell, `dash(1)`, which does not do this when invoked as `sh`.)

Any user input that is employed as part of `command` should be *carefully* sanitized, to ensure that unexpected shell commands or command options are not executed. Such risks are especially grave when using `system()` from a privileged program.

BUGS

[top](#)

If the command name starts with a hyphen, `sh(1)` interprets the command name as an option, and the behavior is undefined. (See the `-c` option to `sh(1)`.) To work around this problem, prepend the command with a space as in the following call:



```
system(" -unfortunate-command-name");
```

SEE ALSO [top](#)

[sh\(1\)](#), [execve\(2\)](#), [fork\(2\)](#), [sigaction\(2\)](#), [sigprocmask\(2\)](#), [wait\(2\)](#),
[exec\(3\)](#), [signal\(7\)](#)

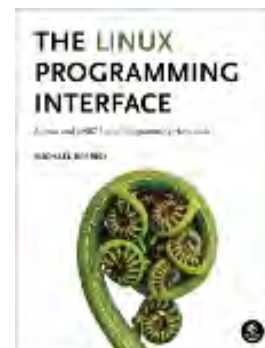
Linux man-pages (unreleased) (date) [system\(3\)](#)

Pages that refer to this page: [execve\(2\)](#), [confstr\(3\)](#), [curs_scr_dump\(3x\)](#), [exec\(3\)](#),
[__pmprocessexec\(3\)](#), [popen\(3\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sh(1p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [OPERANDS](#) | [STDIN](#) | [INPUT FILES](#) | [ENVIRONMENT VARIABLES](#) | [ASYNCHRONOUS EVENTS](#) | [STDOUT](#) | [STDERR](#) | [OUTPUT FILES](#) | [EXTENDED DESCRIPTION](#) | [EXIT STATUS](#) | [CONSEQUENCES OF ERRORS](#) | [APPLICATION USAGE](#) | [EXAMPLES](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 SH(1P)

POSIX Programmer's Manual

SH(1P)

PROLOG [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

sh – shell, the standard command language interpreter

SYNOPSIS [top](#)

```
sh [-abCefhimnuvx] [-o option]... [+abCefhimnuvx] [+o option]...  
    [command_file [argument...]]
```

```
sh -c [-abCefhimnuvx] [-o option]... [+abCefhimnuvx] [+o option]...  
    command_string [command_name [argument...]]
```

```
sh -s [-abCefhimnuvx] [-o option]... [+abCefhimnuvx] [+o option]...  
    [argument...]
```

DESCRIPTION [top](#)

The *sh* utility is a command language interpreter that shall execute commands read from a command line string, the standard input, or a specified file. The application shall ensure that the



commands to be executed are expressed in the language described in *Chapter 2, Shell Command Language*.

Pathname expansion shall not fail due to the size of a file.

Shell input and output redirections have an implementation-defined offset maximum that is established in the open file description.

OPTIONS [top](#)

The *sh* utility shall conform to the Base Definitions volume of POSIX.1-2017, *Section 12.2, Utility Syntax Guidelines*, with an extension for support of a leading `<plus-sign>` ('+') as noted below.

The `-a`, `-b`, `-C`, `-e`, `-f`, `-m`, `-n`, `-o` option, `-u`, `-v`, and `-x` options are described as part of the *set* utility in *Section 2.14, Special Built-In Utilities*. The option letters derived from the *set* special built-in shall also be accepted with a leading `<plus-sign>` ('+') instead of a leading `<hyphen-minus>` (meaning the reverse case of the option as described in this volume of POSIX.1-2017).

The following additional options shall be supported:

- `-c` Read commands from the *command_string* operand. Set the value of special parameter 0 (see *Section 2.5.2, Special Parameters*) from the value of the *command_name* operand and the positional parameters (\$1, \$2, and so on) in sequence from the remaining *argument* operands. No commands shall be read from the standard input.
- `-i` Specify that the shell is *interactive*; see below. An implementation may treat specifying the `-i` option as an error if the real user ID of the calling process does not equal the effective user ID or if the real group ID does not equal the effective group ID.
- `-s` Read commands from the standard input.

If there are no operands and the `-c` option is not specified, the `-s` option shall be assumed.

If the `-i` option is present, or if there are no operands and the

shell's standard input and standard error are attached to a terminal, the shell is considered to be *interactive*.

OPERANDS [top](#)

The following operands shall be supported:

- A single <hyphen-minus> shall be treated as the first operand and then ignored. If both '-' and "--" are given as arguments, or if other operands precede the single <hyphen-minus>, the results are undefined.

argument The positional parameters (\$1, \$2, and so on) shall be set to *arguments*, if any.

command_file

The pathname of a file containing commands. If the pathname contains one or more <slash> characters, the implementation attempts to read that file; the file need not be executable. If the pathname does not contain a <slash> character:

- * The implementation shall attempt to read that file from the current working directory; the file need not be executable.
- * If the file is not in the current working directory, the implementation may perform a search for an executable file using the value of *PATH*, as described in *Section 2.9.1.1, Command Search and Execution*.

Special parameter 0 (see *Section 2.5.2, Special Parameters*) shall be set to the value of *command_file*. If *sh* is called using a synopsis form that omits *command_file*, special parameter 0 shall be set to the value of the first argument passed to *sh* from its parent (for example, *argv[0]* for a C program), which is normally a pathname used to execute the *sh* utility.

command_name

A string assigned to special parameter 0 when executing the commands in *command_string*. If *command_name* is not specified, special parameter 0 shall be set to the value of the first argument passed to *sh* from its parent (for example, *argv[0]* for a C program), which is

normally a pathname used to execute the `sh` utility.

command_string

A string that shall be interpreted by the shell as one or more commands, as if the string were the argument to the `system()` function defined in the System Interfaces volume of POSIX.1-2017. If the *command_string* operand is an empty string, `sh` shall exit with a zero exit status.

STDIN

[top](#)

The standard input shall be used only if one of the following is true:

- * The `-s` option is specified.
- * The `-c` option is not specified and no operands are specified.
- * The script executes one or more commands that require input from standard input (such as a `read` command that does not redirect its input).

See the INPUT FILES section.

When the shell is using standard input and it invokes a command that also uses standard input, the shell shall ensure that the standard input file pointer points directly after the command it has read when the command begins execution. It shall not read ahead in such a manner that any characters intended to be read by the invoked command are consumed by the shell (whether interpreted by the shell or not) or that characters that are not read by the invoked command are not seen by the shell. When the command expecting to read standard input is started asynchronously by an interactive shell, it is unspecified whether characters are read by the command or interpreted by the shell.

If the standard input to `sh` is a FIFO or terminal device and is set to non-blocking reads, then `sh` shall enable blocking reads on standard input. This shall remain in effect when the command completes.

INPUT FILES

[top](#)



The input file shall be a text file, except that line lengths shall be unlimited. If the input file consists solely of zero or more blank lines and comments, *sh* shall exit with a zero exit status.

ENVIRONMENT VARIABLES [top](#)

The following environment variables shall affect the execution of *sh*:

ENV This variable, when and only when an interactive shell is invoked, shall be subjected to parameter expansion (see [Section 2.6.2, Parameter Expansion](#)) by the shell, and the resulting value shall be used as a pathname of a file containing shell commands to execute in the current environment. The file need not be executable. If the expanded value of *ENV* is not an absolute pathname, the results are unspecified. *ENV* shall be ignored if the real and effective user IDs or real and effective group IDs of the process are different.

FCEDIT This variable, when expanded by the shell, shall determine the default value for the *-e editor* option's *editor* option-argument. If *FCEDIT* is null or unset, *ed* shall be used as the editor.

HISTFILE Determine a pathname naming a command history file. If the *HISTFILE* variable is not set, the shell may attempt to access or create a file **.sh_history** in the directory referred to by the *HOME* environment variable. If the shell cannot obtain both read and write access to, or create, the history file, it shall use an unspecified mechanism that allows the history to operate properly. (References to history ``file'' in this section shall be understood to mean this unspecified mechanism in such cases.) An implementation may choose to access this variable only when initializing the history file; this initialization shall occur when *fc* or *sh* first attempt to retrieve entries from, or add entries to, the file, as the result of commands issued by the user, the file named by the *ENV* variable, or implementation-defined system start-up files. Implementations may choose to disable the history list mechanism for users with appropriate privileges who do not set *HISTFILE*; the specific circumstances under which this occurs are implementation-defined. If more than one instance of



the shell is using the same history file, it is unspecified how updates to the history file from those shells interact. As entries are deleted from the history file, they shall be deleted oldest first. It is unspecified when history file entries are physically removed from the history file.

HISTSIZE Determine a decimal number representing the limit to the number of previous commands that are accessible. If this variable is unset, an unspecified default greater than or equal to 128 shall be used. The maximum number of commands in the history list is unspecified, but shall be at least 128. An implementation may choose to access this variable only when initializing the history file, as described under **HISTFILE**. Therefore, it is unspecified whether changes made to **HISTSIZE** after the history file has been initialized are effective.

HOME Determine the pathname of the user's home directory. The contents of **HOME** are used in tilde expansion as described in *Section 2.6.1, Tilde Expansion*.

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of POSIX.1-2017, *Section 8.2, Internationalization Variables* for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_COLLATE Determine the behavior of range expressions, equivalence classes, and multi-character collating elements within pattern matching.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), which characters are defined as letters (character class **alpha**), and the behavior of character classes within pattern matching.

LC_MESSAGES Determine the locale that should be used to affect the



format and contents of diagnostic messages written to standard error.

MAIL Determine a pathname of the user's mailbox file for purposes of incoming mail notification. If this variable is set, the shell shall inform the user if the file named by the variable is created or if its modification time has changed. Informing the user shall be accomplished by writing a string of unspecified format to standard error prior to the writing of the next primary prompt string. Such check shall be performed only after the completion of the interval defined by the **MAILCHECK** variable after the last such check. The user shall be informed only if **MAIL** is set and **MAILPATH** is not set.

MAILCHECK Establish a decimal integer value that specifies how often (in seconds) the shell shall check for the arrival of mail in the files specified by the **MAILPATH** or **MAIL** variables. The default value shall be 600 seconds. If set to zero, the shell shall check before issuing each primary prompt.

MAILPATH Provide a list of pathnames and optional messages separated by <colon> characters. If this variable is set, the shell shall inform the user if any of the files named by the variable are created or if any of their modification times change. (See the preceding entry for **MAIL** for descriptions of mail arrival and user informing.) Each pathname can be followed by '%' and a string that shall be subjected to parameter expansion and written to standard error when the modification time changes. If a '%' character in the pathname is preceded by a <backslash>, it shall be treated as a literal '%' in the pathname. The default message is unspecified.

The **MAILPATH** environment variable takes precedence over the **MAIL** variable.

NLSPATH Determine the location of message catalogs for the processing of **LC_MESSAGES**.

PATH Establish a string formatted as described in the Base Definitions volume of POSIX.1-2017, *Chapter 8, Environment Variables*, used to effect command

interpretation; see [Section 2.9.1.1, Command Search and Execution](#).

PWD This variable shall represent an absolute pathname of the current working directory. Assignments to this variable may be ignored.

ASYNCHRONOUS EVENTS [top](#)

The *sh* utility shall take the standard action for all signals (see [Section 1.4, Utility Description Defaults](#)) with the following exceptions.

If the shell is interactive, SIGINT signals received during command line editing shall be handled as described in the EXTENDED DESCRIPTION, and SIGINT signals received at other times shall be caught but no action performed.

If the shell is interactive:

- * SIGQUIT and SIGTERM signals shall be ignored.
- * If the **-m** option is in effect, SIGTTIN, SIGTTOU, and SIGTSTP signals shall be ignored.
- * If the **-m** option is not in effect, it is unspecified whether SIGTTIN, SIGTTOU, and SIGTSTP signals are ignored, set to the default action, or caught. If they are caught, the shell shall, in the signal-catching function, set the signal to the default action and raise the signal (after taking any appropriate steps, such as restoring terminal settings).

The standard actions, and the actions described above for interactive shells, can be overridden by use of the *trap* special built-in utility (see [trap\(1p\)](#) and [Section 2.11, Signals and Error Handling](#)).

STDOUT [top](#)

See the STDERR section.

STDERR [top](#)

Except as otherwise stated (by the descriptions of any invoked utilities or in interactive mode), standard error shall be used



only for diagnostic messages.

OUTPUT FILES [top](#)

None.

EXTENDED DESCRIPTION [top](#)

See *Chapter 2, Shell Command Language*. The functionality described in the rest of the EXTENDED DESCRIPTION section shall be provided on implementations that support the User Portability Utilities option (and the rest of this section is not further shaded for this option).

Command History List

When the *sh* utility is being used interactively, it shall maintain a list of commands previously entered from the terminal in the file named by the *HISTFILE* environment variable. The type, size, and internal format of this file are unspecified. Multiple *sh* processes can share access to the file for a user, if file access permissions allow this; see the description of the *HISTFILE* environment variable.

Command Line Editing

When *sh* is being used interactively from a terminal, the current command and the command history (see [fc\(1p\)](#)) can be edited using *vi*-mode command line editing. This mode uses commands, described below, similar to a subset of those described in the *vi* utility. Implementations may offer other command line editing modes corresponding to other editing utilities.

The command *set -o vi* shall enable *vi*-mode editing and place *sh* into *vi* insert mode (see *Command Line Editing (vi-mode)*). This command also shall disable any other editing mode that the implementation may provide. The command *set +o vi* disables *vi*-mode editing.

Certain block-mode terminals may be unable to support shell command line editing. If a terminal is unable to provide either edit mode, it need not be possible to *set -o vi* when using the shell on this terminal.

In the following sections, the characters *erase*, *interrupt*, *kill*, and *end-of-file* are those set by the *stty* utility.



Command Line Editing (vi-mode)

In *vi* editing mode, there shall be a distinguished line, the edit line. All the editing operations which modify a line affect the edit line. The edit line is always the newest line in the command history buffer.

With *vi*-mode enabled, *sh* can be switched between insert mode and command mode.

When in insert mode, an entered character shall be inserted into the command line, except as noted in *vi Line Editing Insert Mode*. Upon entering *sh* and after termination of the previous command, *sh* shall be in insert mode.

Typing an escape character shall switch *sh* into command mode (see *vi Line Editing Command Mode*). In command mode, an entered character shall either invoke a defined operation, be used as part of a multi-character operation, or be treated as an error. A character that is not recognized as part of an editing command shall terminate any specific editing command and shall alert the terminal. If *sh* receives a SIGINT signal in command mode (whether generated by typing the *interrupt* character or by other means), it shall terminate command line editing on the current command line, reissue the prompt on the next line of the terminal, and reset the command history (see [fc\(1p\)](#)) so that the most recently executed command is the previous command (that is, the command that was being edited when it was interrupted is not re-entered into the history).

In the following sections, the phrase ``move the cursor to the beginning of the word'' shall mean ``move the cursor to the first character of the current word'' and the phrase ``move the cursor to the end of the word'' shall mean ``move the cursor to the last character of the current word''. The phrase ``beginning of the command line'' indicates the point between the end of the prompt string issued by the shell (or the beginning of the terminal line, if there is no prompt string) and the first character of the command text.

vi Line Editing Insert Mode

While in insert mode, any character typed shall be inserted in the current command line, unless it is from the following set.

<newline> Execute the current command line. If the current command line is not empty, this line shall be entered into the command history (see [fc\(1p\)](#)).



erase Delete the character previous to the current cursor position and move the current cursor position back one character. In insert mode, characters shall be erased from both the screen and the buffer when backspacing.

interrupt If *sh* receives a SIGINT signal in insert mode (whether generated by typing the *interrupt* character or by other means), it shall terminate command line editing with the same effects as described for interrupting command mode; see *Command Line Editing (vi-mode)*.

kill Clear all the characters from the input line.

<control>-V

Insert the next character input, even if the character is otherwise a special insert mode character.

<control>-W

Delete the characters from the one preceding the cursor to the preceding word boundary. The word boundary in this case is the closer to the cursor of either the beginning of the line or a character that is in neither the **blank** nor **punct** character classification of the current locale.

end-of-file

Interpreted as the end of input in *sh*. This interpretation shall occur only at the beginning of an input line. If *end-of-file* is entered other than at the beginning of the line, the results are unspecified.

<ESC> Place *sh* into command mode.

vi Line Editing Command Mode

In command mode for the command line editing feature, decimal digits not beginning with 0 that precede a command letter shall be remembered. Some commands use these decimal digits as a count number that affects the operation.

The term *motion command* represents one of the commands:

<space> 0 b F l W ^ \$; E f T w | , B e h t

If the current line is not the edit line, any command that modifies the current line shall cause the content of the current line to replace the content of the edit line, and the current line shall become the edit line. This replacement cannot be



undone (see the **u** and **U** commands below). The modification requested shall then be performed to the edit line. When the current line is the edit line, the modification shall be done directly to the edit line.

Any command that is preceded by *count* shall take a count (the numeric value of any preceding decimal digits). Unless otherwise noted, this count shall cause the specified operation to repeat by the number of times specified by the count. Also unless otherwise noted, a *count* that is out of range is considered an error condition and shall alert the terminal, but neither the cursor position, nor the command line, shall change.

The terms *word* and *bigword* are used as defined in the *vi* description. The term *save buffer* corresponds to the term *unnamed buffer* in *vi*.

The following commands shall be recognized in command mode:

<newline> Execute the current command line. If the current command line is not empty, this line shall be entered into the command history (see [fc\(1p\)](#)).

<control>-L
Redraw the current command line. Position the cursor at the same location on the redrawn line.

Insert the character '#' at the beginning of the current command line and treat the resulting edit line as a comment. This line shall be entered into the command history; see [fc\(1p\)](#).

= Display the possible shell word expansions (see [Section 2.6, Word Expansions](#)) of the bigword at the current command line position.

Note: This does not modify the content of the current line, and therefore does not cause the current line to become the edit line.

These expansions shall be displayed on subsequent terminal lines. If the bigword contains none of the characters '?', '*', or '[', an <asterisk> ('*') shall be implicitly assumed at the end. If any directories are matched, these expansions shall have a '/' character appended. After the expansion, the line shall be redrawn, the cursor repositioned at the current



cursor position, and *sh* shall be placed in command mode.

- ** Perform pathname expansion (see [Section 2.6.6, Pathname Expansion](#)) on the current bigword, up to the largest set of characters that can be matched uniquely. If the bigword contains none of the characters '?', '*', or '[', an <asterisk> ('*') shall be implicitly assumed at the end. This maximal expansion then shall replace the original bigword in the command line, and the cursor shall be placed after this expansion. If the resulting bigword completely and uniquely matches a directory, a '/' character shall be inserted directly after the bigword. If some other file is completely matched, a single <space> shall be inserted after the bigword. After this operation, *sh* shall be placed in insert mode.
- *** Perform pathname expansion on the current bigword and insert all expansions into the command to replace the current bigword, with each expansion separated by a single <space>. If at the end of the line, the current cursor position shall be moved to the first column position following the expansions and *sh* shall be placed in insert mode. Otherwise, the current cursor position shall be the last column position of the first character after the expansions and *sh* shall be placed in insert mode. If the current bigword contains none of the characters '?', '*', or '[', before the operation, an <asterisk> ('*') shall be implicitly assumed at the end.
- @*Letter*** Insert the value of the alias named *_Letter*. The symbol *Letter* represents a single alphabetic character from the portable character set; implementations may support additional characters as an extension. If the alias *_Letter* contains other editing commands, these commands shall be performed as part of the insertion. If no alias *_Letter* is enabled, this command shall have no effect.
- [*count*]~** Convert, if the current character is a lowercase letter, to the equivalent uppercase letter and *vice versa*, as prescribed by the current locale. The current cursor position then shall be advanced by one character. If the cursor was positioned on the last character of the line, the case conversion shall occur,



but the cursor shall not advance. If the '~' command is preceded by a *count*, that number of characters shall be converted, and the cursor shall be advanced to the character position after the last character converted. If the *count* is larger than the number of characters after the cursor, this shall not be considered an error; the cursor shall advance to the last character on the line.

[count]. Repeat the most recent non-motion command, even if it was executed on an earlier command line. If the previous command was preceded by a *count*, and no count is given on the '.' command, the count from the previous command shall be included as part of the repeated command. If the '.' command is preceded by a *count*, this shall override any *count* argument to the previous command. The *count* specified in the '.' command shall become the count for subsequent '.' commands issued without a count.

[number]v Invoke the *vi* editor to edit the current command line in a temporary file. When the editor exits, the commands in the temporary file shall be executed and placed in the command history. If a *number* is included, it specifies the command number in the command history to be edited, rather than the current command line.

[count]l (ell)

[count]<space>

Move the current cursor position to the next character position. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be advanced. If the *count* is larger than the number of characters after the cursor, this shall not be considered an error; the cursor shall advance to the last character on the line.

[count]h Move the current cursor position to the *count*th (default 1) previous character position. If the cursor was positioned on the first character of the line, the terminal shall be alerted and the cursor shall not be moved. If the count is larger than the number of characters before the cursor, this shall not be considered an error; the cursor shall move to the first character on the line.



- [count]w** Move to the start of the next word. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be advanced. If the *count* is larger than the number of words after the cursor, this shall not be considered an error; the cursor shall advance to the last character on the line.
- [count]W** Move to the start of the next bigword. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be advanced. If the *count* is larger than the number of bigwords after the cursor, this shall not be considered an error; the cursor shall advance to the last character on the line.
- [count]e** Move to the end of the current word. If at the end of a word, move to the end of the next word. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be advanced. If the *count* is larger than the number of words after the cursor, this shall not be considered an error; the cursor shall advance to the last character on the line.
- [count]E** Move to the end of the current bigword. If at the end of a bigword, move to the end of the next bigword. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be advanced. If the *count* is larger than the number of bigwords after the cursor, this shall not be considered an error; the cursor shall advance to the last character on the line.
- [count]b** Move to the beginning of the current word. If at the beginning of a word, move to the beginning of the previous word. If the cursor was positioned on the first character of the line, the terminal shall be alerted and the cursor shall not be moved. If the *count* is larger than the number of words preceding the cursor, this shall not be considered an error; the cursor shall return to the first character on the line.
- [count]B** Move to the beginning of the current bigword. If at the beginning of a bigword, move to the beginning of the previous bigword. If the cursor was positioned on the first character of the line, the terminal shall be



alerted and the cursor shall not be moved. If the *count* is larger than the number of bigwords preceding the cursor, this shall not be considered an error; the cursor shall return to the first character on the line.

- ^** Move the current cursor position to the first character on the input line that is not a <blank>.
- \$** Move to the last character position on the current command line.
- 0** (Zero.) Move to the first character position on the current command line.
- [count]** Move to the *count*th character position on the current command line. If no number is specified, move to the first position. The first character position shall be numbered 1. If the count is larger than the number of characters on the line, this shall not be considered an error; the cursor shall be placed on the last character on the line.
- [count]fc** Move to the first occurrence of the character '**c**' that occurs after the current cursor position. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be advanced. If the character '**c**' does not occur in the line after the current cursor position, the terminal shall be alerted and the cursor shall not be moved.
- [count]Fc** Move to the first occurrence of the character '**c**' that occurs before the current cursor position. If the cursor was positioned on the first character of the line, the terminal shall be alerted and the cursor shall not be moved. If the character '**c**' does not occur in the line before the current cursor position, the terminal shall be alerted and the cursor shall not be moved.
- [count]tc** Move to the character before the first occurrence of the character '**c**' that occurs after the current cursor position. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be advanced. If the character '**c**' does not occur in the line after the current cursor position, the terminal shall be alerted and the cursor shall not be moved.



[count]Tc Move to the character after the first occurrence of the character '**c**' that occurs before the current cursor position. If the cursor was positioned on the first character of the line, the terminal shall be alerted and the cursor shall not be moved. If the character '**c**' does not occur in the line before the current cursor position, the terminal shall be alerted and the cursor shall not be moved.

[count]; Repeat the most recent **f**, **F**, **t**, or **T** command. Any number argument on that previous command shall be ignored. Errors are those described for the repeated command.

[count], Repeat the most recent **f**, **F**, **t**, or **T** command. Any number argument on that previous command shall be ignored. However, reverse the direction of that command.

a Enter insert mode after the current cursor position. Characters that are entered shall be inserted before the next character.

A Enter insert mode after the end of the current command line.

i Enter insert mode at the current cursor position. Characters that are entered shall be inserted before the current character.

I Enter insert mode at the beginning of the current command line.

R Enter insert mode, replacing characters from the command line beginning at the current cursor position.

[count]cmotion

Delete the characters between the current cursor position and the cursor position that would result from the specified motion command. Then enter insert mode before the first character following any deleted characters. If *count* is specified, it shall be applied to the motion command. A *count* shall be ignored for the following motion commands:

0 ^ \$ c



If the motion command is the character `'c'`, the current command line shall be cleared and insert mode shall be entered. If the motion command would move the current cursor position toward the beginning of the command line, the character under the current cursor position shall not be deleted. If the motion command would move the current cursor position toward the end of the command line, the character under the current cursor position shall be deleted. If the *count* is larger than the number of characters between the current cursor position and the end of the command line toward which the motion command would move the cursor, this shall not be considered an error; all of the remaining characters in the aforementioned range shall be deleted and insert mode shall be entered. If the motion command is invalid, the terminal shall be alerted, the cursor shall not be moved, and no text shall be deleted.

C Delete from the current character to the end of the line and enter insert mode at the new end-of-line.

S Clear the entire edit line and enter insert mode.

[count]rc Replace the current character with the character `'c'`. With a number *count*, replace the current and the following *count*-1 characters. After this command, the current cursor position shall be on the last character that was changed. If the *count* is larger than the number of characters after the cursor, this shall not be considered an error; all of the remaining characters shall be changed.

[count]_ Append a `<space>` after the current character position and then append the last bigword in the previous input line after the `<space>`. Then enter insert mode after the last character just appended. With a number *count*, append the *count*th bigword in the previous line.

[count]x Delete the character at the current cursor position and place the deleted characters in the save buffer. If the cursor was positioned on the last character of the line, the character shall be deleted and the cursor position shall be moved to the previous character (the new last character). If the *count* is larger than the number of characters after the cursor, this shall not be considered an error; all the characters from the



cursor to the end of the line shall be deleted.

[count]X Delete the character before the current cursor position and place the deleted characters in the save buffer. The character under the current cursor position shall not change. If the cursor was positioned on the first character of the line, the terminal shall be alerted, and the **X** command shall have no effect. If the line contained a single character, the **X** command shall have no effect. If the line contained no characters, the terminal shall be alerted and the cursor shall not be moved. If the *count* is larger than the number of characters before the cursor, this shall not be considered an error; all the characters from before the cursor to the beginning of the line shall be deleted.

[count]dmotion

Delete the characters between the current cursor position and the character position that would result from the motion command. A number *count* repeats the motion command *count* times. If the motion command would move toward the beginning of the command line, the character under the current cursor position shall not be deleted. If the motion command is **d**, the entire current command line shall be cleared. If the *count* is larger than the number of characters between the current cursor position and the end of the command line toward which the motion command would move the cursor, this shall not be considered an error; all of the remaining characters in the aforementioned range shall be deleted. The deleted characters shall be placed in the save buffer.

D Delete all characters from the current cursor position to the end of the line. The deleted characters shall be placed in the save buffer.

[count]ymotion

Yank (that is, copy) the characters from the current cursor position to the position resulting from the motion command into the save buffer. A number *count* shall be applied to the motion command. If the motion command would move toward the beginning of the command line, the character under the current cursor position shall not be included in the set of yanked characters. If the motion command is **y**, the entire current command line shall be yanked into the save buffer. The current



cursor position shall be unchanged. If the *count* is larger than the number of characters between the current cursor position and the end of the command line toward which the motion command would move the cursor, this shall not be considered an error; all of the remaining characters in the aforementioned range shall be yanked.

Y Yank the characters from the current cursor position to the end of the line into the save buffer. The current character position shall be unchanged.

[count]p Put a copy of the current contents of the save buffer after the current cursor position. The current cursor position shall be advanced to the last character put from the save buffer. A *count* shall indicate how many copies of the save buffer shall be put.

[count]P Put a copy of the current contents of the save buffer before the current cursor position. The current cursor position shall be moved to the last character put from the save buffer. A *count* shall indicate how many copies of the save buffer shall be put.

u Undo the last command that changed the edit line. This operation shall not undo the copy of any command line to the edit line.

U Undo all changes made to the edit line. This operation shall not undo the copy of any command line to the edit line.

[count]k

[count]- Set the current command line to be the *count*th previous command line in the shell command history. If *count* is not specified, it shall default to 1. The cursor shall be positioned on the first character of the new command. If a **k** or **-** command would retreat past the maximum number of commands in effect for this shell (affected by the *HISTSIZE* environment variable), the terminal shall be alerted, and the command shall have no effect.

[count]j

[count]+ Set the current command line to be the *count*th next



command line in the shell command history. If *count* is not specified, it shall default to 1. The cursor shall be positioned on the first character of the new command. If a **j** or **+** command advances past the edit line, the current command line shall be restored to the edit line and the terminal shall be alerted.

[*number*]G Set the current command line to be the oldest command line stored in the shell command history. With a number *number*, set the current command line to be the command line *number* in the history. If command line *number* does not exist, the terminal shall be alerted and the command line shall not be changed.

/*pattern*<newline>

Move backwards through the command history, searching for the specified pattern, beginning with the previous command line. Patterns use the pattern matching notation described in *Section 2.13, Pattern Matching Notation*, except that the '^' character shall have special meaning when it appears as the first character of *pattern*. In this case, the '^' is discarded and the characters after the '^' shall be matched only at the beginning of a line. Commands in the command history shall be treated as strings, not as filenames. If the pattern is not found, the current command line shall be unchanged and the terminal shall be alerted. If it is found in a previous line, the current command line shall be set to that line and the cursor shall be set to the first character of the new command line.

If *pattern* is empty, the last non-empty pattern provided to **/** or **?** shall be used. If there is no previous non-empty pattern, the terminal shall be alerted and the current command line shall remain unchanged.

?*pattern*<newline>

Move forwards through the command history, searching for the specified pattern, beginning with the next command line. Patterns use the pattern matching notation described in *Section 2.13, Pattern Matching Notation*, except that the '^' character shall have special meaning when it appears as the first character of *pattern*. In this case, the '^' is discarded and the characters after the '^' shall be matched only at the beginning of a line. Commands in the command history



shall be treated as strings, not as filenames. If the pattern is not found, the current command line shall be unchanged and the terminal shall be alerted. If it is found in a following line, the current command line shall be set to that line and the cursor shall be set to the first character of the new command line.

If *pattern* is empty, the last non-empty pattern provided to */* or *?* shall be used. If there is no previous non-empty pattern, the terminal shall be alerted and the current command line shall remain unchanged.

- n** Repeat the most recent */* or *?* command. If there is no previous */* or *?*, the terminal shall be alerted and the current command line shall remain unchanged.
- N** Repeat the most recent */* or *?* command, reversing the direction of the search. If there is no previous */* or *?*, the terminal shall be alerted and the current command line shall remain unchanged.

EXIT STATUS [top](#)

The following exit values shall be returned:

- 0 The script to be executed consisted solely of zero or more blank lines or comments, or both.
- 1-125 A non-interactive shell detected an error other than *command_file* not found or executable, including but not limited to syntax, redirection, or variable assignment errors.
- 126 A specified *command_file* could not be executed due to an **[ENOEXEC]** error (see *Section 2.9.1.1, Command Search and Execution*, item 2).
- 127 A specified *command_file* could not be found by a non-interactive shell.

Otherwise, the shell shall return the exit status of the last command it invoked or attempted to invoke (see also the *exit* utility in *Section 2.14, Special Built-In Utilities*).

CONSEQUENCES OF ERRORS [top](#)

See *Section 2.8.1, Consequences of Shell Errors*.

The following sections are informative.

APPLICATION USAGE [top](#)

Standard input and standard error are the files that determine whether a shell is interactive when `-i` is not specified. For example:

```
sh > file
```

and:

```
sh 2> file
```

create interactive and non-interactive shells, respectively. Although both accept terminal input, the results of error conditions are different, as described in *Section 2.8.1, Consequences of Shell Errors*; in the second example a redirection error encountered by a special built-in utility aborts the shell.

A conforming application must protect its first operand, if it starts with a <plus-sign>, by preceding it with the "--" argument that denotes the end of the options.

Applications should note that the standard *PATH* to the shell cannot be assumed to be either `/bin/sh` or `/usr/bin/sh`, and should be determined by interrogation of the *PATH* returned by `getconf PATH`, ensuring that the returned pathname is an absolute pathname and not a shell built-in.

For example, to determine the location of the standard `sh` utility:

```
command -v sh
```

On some implementations this might return:

```
/usr/xpg4/bin/sh
```

Furthermore, on systems that support executable scripts (the "#!" construct), it is recommended that applications using executable scripts install them using `getconf PATH` to determine the shell



pathname and update the "#!" script appropriately as it is being installed (for example, with *sed*). For example:

```
#
# Installation time script to install correct POSIX shell pathname
#
# Get list of paths to check
#
Sifs=$IFS
Sifs_set=${IFS+y}
IFS=:
set -- $(getconf PATH)
if [ "$Sifs_set" = y ]
then
    IFS=$Sifs
else
    unset IFS
fi
#
# Check each path for 'sh'
#
for i
do
    if [ -x "${i}/sh ]
    then
        Pshell=${i}/sh
    fi
done
#
# This is the list of scripts to update. They should be of the
# form '${name}.source' and will be transformed to '${name}'.
# Each script should begin:
#
# #!INSTALLSHELLPATH
#
scripts="a b c"
#
# Transform each script
#
for i in ${scripts}
do
    sed -e "s|INSTALLSHELLPATH|${Pshell}|" < ${i}.source > ${i}
done
```

EXAMPLES

[top](#)



1. Execute a shell command from a string:

```
sh -c "cat myfile"
```

2. Execute a shell script from a file in the current directory:

```
sh my_shell_cmds
```

RATIONALE [top](#)

The *sh* utility and the *set* special built-in utility share a common set of options.

The name *IFS* was originally an abbreviation of ``Input Field Separators''; however, this name is misleading as the *IFS* characters are actually used as field terminators. One justification for ignoring the contents of *IFS* upon entry to the script, beyond security considerations, is to assist possible future shell compilers. Allowing *IFS* to be imported from the environment prevents many optimizations that might otherwise be performed via dataflow analysis of the script itself.

The text in the STDIN section about non-blocking reads concerns an instance of *sh* that has been invoked, probably by a C-language program, with standard input that has been opened using the `O_NONBLOCK` flag; see *open()* in the System Interfaces volume of POSIX.1-2017. If the shell did not reset this flag, it would immediately terminate because no input data would be available yet and that would be considered the same as end-of-file.

The options associated with a *restricted shell* (command name *rsh* and the `-r` option) were excluded because the standard developers considered that the implied level of security could not be achieved and they did not want to raise false expectations.

On systems that support set-user-ID scripts, a historical trapdoor has been to link a script to the name `-i`. When it is called by a sequence such as:

```
sh -
```

or by:

```
#! usr/bin/sh -
```

the historical systems have assumed that no option letters



follow. Thus, this volume of POSIX.1-2017 allows the single `<hyphen-minus>` to mark the end of the options, in addition to the use of the regular `"--"` argument, because it was considered that the older practice was so pervasive. An alternative approach is taken by the KornShell, where real and effective user/group IDs must match for an interactive shell; this behavior is specifically allowed by this volume of POSIX.1-2017.

Note: There are other problems with set-user-ID scripts that the two approaches described here do not resolve.

The initialization process for the history file can be dependent on the system start-up files, in that they may contain commands that effectively preempt the user's settings of `HISTFILE` and `HISTSIZE`. For example, function definition commands are recorded in the history file, unless the `set -o noLog` option is set. If the system administrator includes function definitions in some system start-up file called before the `ENV` file, the history file is initialized before the user gets a chance to influence its characteristics. In some historical shells, the history file is initialized just after the `ENV` file has been processed. Therefore, it is implementation-defined whether changes made to `HISTFILE` after the history file has been initialized are effective.

The default messages for the various `MAIL`-related messages are unspecified because they vary across implementations. Typical messages are:

```
"you have mail\n"
```

or:

```
"you have new mail\n"
```

It is important that the descriptions of command line editing refer to the same shell as that in POSIX.1-2008 so that interactive users can also be application programmers without having to deal with programmatic differences in their two environments. It is also essential that the utility name `sh` be specified because this explicit utility name is too firmly rooted in historical practice of application programs for it to change.

Consideration was given to mandating a diagnostic message when attempting to set `vi`-mode on terminals that do not support command line editing. However, it is not historical practice for the shell to be cognizant of all terminal types and thus be able



to detect inappropriate terminals in all cases. Implementations are encouraged to supply diagnostics in this case whenever possible, rather than leaving the user in a state where editing commands work incorrectly.

In early proposals, the KornShell-derived *emacs* mode of command line editing was included, even though the *emacs* editor itself was not. The community of *emacs* proponents was adamant that the full *emacs* editor not be standardized because they were concerned that an attempt to standardize this very powerful environment would encourage vendors to ship strictly conforming versions lacking the extensibility required by the community. The author of the original *emacs* program also expressed his desire to omit the program. Furthermore, there were a number of historical systems that did not include *emacs*, or included it without supporting it, but there were very few that did not include and support *vi*. The shell *emacs* command line editing mode was finally omitted because it became apparent that the KornShell version and the editor being distributed with the GNU system had diverged in some respects. The author of *emacs* requested that the POSIX *emacs* mode either be deleted or have a significant number of unspecified conditions. Although the KornShell author agreed to consider changes to bring the shell into alignment, the standard developers decided to defer specification at that time. At the time, it was assumed that convergence on an acceptable definition would occur for a subsequent draft, but that has not happened, and there appears to be no impetus to do so. In any case, implementations are free to offer additional command line editing modes based on the exact models of editors their users are most comfortable with.

Early proposals had the following list entry in *vi Line Editing Insert Mode*:

```
\      If followed by the erase or kill character, that character
      shall be inserted into the input line.  Otherwise, the
      <backslash> itself shall be inserted into the input line.
```

However, this is not actually a feature of *sh* command line editing insert mode, but one of some historical terminal line drivers. Some conforming implementations continue to do this when the *stty iexten* flag is set.

In interactive shells, SIGTERM is ignored so that *kill 0* does not kill the shell, and SIGINT is caught so that *wait* is interruptible. If the shell does not ignore SIGTTIN, SIGTTOU, and SIGTSTP signals when it is interactive and the *-m* option is not



in effect, these signals suspend the shell if it is not a session leader. If it is a session leader, the signals are discarded if they would stop the process, as required by the System Interfaces volume of POSIX.1-2017, *Section 2.4.3, Signal Actions* for orphaned process groups.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

Section 2.9.1.1, Command Search and Execution, Chapter 2, Shell Command Language, [cd\(1p\)](#), [echo\(1p\)](#), [exit\(1p\)](#), [fc\(1p\)](#), [pwd\(1p\)](#), [invalid](#), [set\(1p\)](#), [stty\(1p\)](#), [test\(1p\)](#), [trap\(1p\)](#), [umask\(1p\)](#), [vi\(1p\)](#)

The Base Definitions volume of POSIX.1-2017, *Chapter 8, Environment Variables, Section 12.2, Utility Syntax Guidelines*

The System Interfaces volume of POSIX.1-2017, [dup\(3p\)](#), [exec\(1p\)](#), [exit\(3p\)](#), [fork\(3p\)](#), [open\(3p\)](#), [pipe\(3p\)](#), [signal\(3p\)](#), [system\(3p\)](#), [ulimit\(3p\)](#), [umask\(3p\)](#), [wait\(3p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .



Pages that refer to this page: [command\(1p\)](#), [ed\(1p\)](#), [ex\(1p\)](#), [fc\(1p\)](#), [find\(1p\)](#), [make\(1p\)](#), [newgrp\(1p\)](#), [nohup\(1p\)](#), [script\(1\)](#), [time\(1p\)](#), [wait\(1p\)](#), [popen\(3p\)](#), [system\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



passwd(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [CAVEATS](#) | [FILES](#) | [EXIT VALUES](#) | [SEE ALSO](#) | [COLOPHON](#)

 PASSWD(1)

User Commands

PASSWD(1)

NAME [top](#)

passwd - change user password

SYNOPSIS [top](#)

```
passwd [options] [LOGIN]
```

DESCRIPTION [top](#)

The **passwd** command changes passwords for user accounts. A normal user may only change the password for their own account, while the superuser may change the password for any account. **passwd** also changes the account or associated password validity period.

Password Changes

The user is first prompted for their old password, if one is present. This password is then encrypted and compared against the stored password. The user has only one chance to enter the correct password. The superuser is permitted to bypass this step so that forgotten passwords may be changed.

After the password has been entered, password aging information is checked to see if the user is permitted to change the password at this time. If not, **passwd** refuses to change the password and exits.

The user is then prompted twice for a replacement password. The second entry is compared against the first and both are required to match in order for the password to be changed.

Then, the password is tested for complexity. **passwd** will reject any password which is not suitably complex. Care must be taken not to include the system default erase or kill characters.

Hints for user passwords

The security of a password depends upon the strength of the encryption algorithm and the size of the key space. The legacy **UNIX** System encryption method is based on the NBS DES algorithm. More recent methods are now recommended (see **ENCRYPT_METHOD**). The size of the key space depends upon the randomness of the password which is selected.

Compromises in password security normally result from careless password selection or handling. For this reason, you should not select a password which appears in a dictionary or which must be written down. The password should also not be a proper name, your license number, birth date, or street address. Any of these may be used as guesses to violate system security.

As a general guideline, passwords should be long and random. It's fine to use simple character sets, such as passwords consisting only of lowercase letters, if that helps memorizing longer passwords. For a password consisting only of lowercase English letters randomly chosen, and a length of 32, there are 26^{32} (approximately 2^{150}) different possible combinations. Being an exponential equation, it's apparent that the exponent (the length) is more important than the base (the size of the character set).

You can find advice on how to choose a strong password on http://en.wikipedia.org/wiki/Password_strength

OPTIONS [top](#)

The options which apply to the **passwd** command are:

-a, --all

This option can be used only with **-S** and causes show status

for all users.

-d, --delete

Delete a user's password (make it empty). This is a quick way to disable a password for an account. It will set the named account passwordless.

-e, --expire

Immediately expire an account's password. This in effect can force a user to change their password at the user's next login.

-h, --help

Display help message and exit.

-i, --inactive *INACTIVE*

This option is used to disable an account after the password has been expired for a number of days. After a user account has had an expired password for *INACTIVE* days, the user may no longer sign on to the account.

-k, --keep-tokens

Indicate password change should be performed only for expired authentication tokens (passwords). The user wishes to keep their non-expired tokens as before.

-l, --lock

Lock the password of the named account. This option disables a password by changing it to a value which matches no possible encrypted value (it adds a `'!`' at the beginning of the password).

Note that this does not disable the account. The user may still be able to login using another authentication token (e.g. an SSH key). To disable the account, administrators should use **usermod --expiredate 1** (this set the account's expire date to Jan 2, 1970).

Users with a locked password are not allowed to change their password.

-n, --mindays *MIN_DAYS*

Set the minimum number of days between password changes to *MIN_DAYS*. A value of zero for this field indicates that the

user may change their password at any time.

- q, --quiet**
Quiet mode.

- r, --repository *REPOSITORY***
change password in *REPOSITORY* repository

- R, --root *CHROOT_DIR***
Apply changes in the *CHROOT_DIR* directory and use the configuration files from the *CHROOT_DIR* directory. Only absolute paths are supported.

- P, --prefix *PREFIX_DIR***
Apply changes to configuration files under the root filesystem found under the directory *PREFIX_DIR*. This option does not chroot and is intended for preparing a cross-compilation target. Some limitations: NIS and LDAP users/groups are not verified. PAM authentication is using the host files. No SELINUX support.

- S, --status**
Display account status information. The status information consists of 7 fields. The first field is the user's login name. The second field indicates if the user account has a locked password (L), has no password (NP), or has a usable password (P). The third field gives the date of the last password change. The next four fields are the minimum age, maximum age, warning period, and inactivity period for the password. These ages are expressed in days.

- u, --unlock**
Unlock the password of the named account. This option re-enables a password by changing the password back to its previous value (to the value before using the **-l** option).

- w, --warndays *WARN_DAYS***
Set the number of days of warning before a password change is required. The *WARN_DAYS* option is the number of days prior to the password expiring that a user will be warned that their password is about to expire.

- x, --maxdays *MAX_DAYS***
Set the maximum number of days a password remains valid.



After `MAX_DAYS`, the password is required to be changed.

Passing the number `-1` as `MAX_DAYS` will remove checking a password's validity.

CAVEATS [top](#)

Password complexity checking may vary from site to site. The user is urged to select a password as complex as he or she feels comfortable with.

Users may not be able to change their password on a system if NIS is enabled and they are not logged into the NIS server.

`passwd` uses PAM to authenticate users and to change their passwords.

FILES [top](#)

`/etc/passwd`
User account information.

`/etc/shadow`
Secure user account information.

`/etc/pam.d/passwd`
PAM configuration for `passwd`.

EXIT VALUES [top](#)

The `passwd` command exits with the following values:

- `0`
success
- `1`
permission denied
- `2`
invalid combination of options

- 3 unexpected failure, nothing done
- 4 unexpected failure, passwd file missing
- 5 passwd file busy, try again
- 6 invalid argument to option

SEE ALSO [top](#)

[chpasswd\(8\)](#), [makepasswd\(1\)](#), [passwd\(5\)](#), [shadow\(5\)](#), [usermod\(8\)](#).

The following web page comically (yet correctly) compares the strength of two different methods for choosing a password:
"https://xkcd.com/936/"

COLOPHON [top](#)

This page is part of the *shadow-utils* (utilities for managing accounts and shadow password files) project. Information about the project can be found at <https://github.com/shadow-maint/shadow>. If you have a bug report for this manual page, send it to pkg-shadow-devel@alioth-lists.debian.net. This page was obtained from the project's upstream Git repository <https://github.com/shadow-maint/shadow> on 2023-12-22. (At that time, the date of the most recent commit that was found in the repository was 2023-12-15.) If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

shadow-utils 4.11.1

12/22/2023

PASSWD(1)

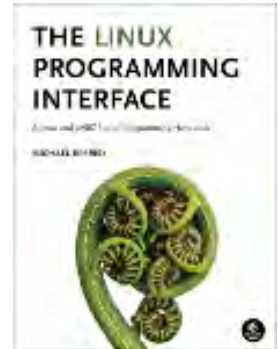


Pages that refer to this page: [ldappasswd\(1\)](#), [login\(1\)](#), [login\(1@@shadow-utils\)](#), [crypt\(3\)](#), [pts\(4\)](#), [login.defs\(5\)](#), [passwd\(5\)](#), [passwd\(5@@shadow-utils\)](#), [shadow\(5\)](#), [chpasswd\(8\)](#), [groupadd\(8\)](#), [groupdel\(8\)](#), [groupmems\(8\)](#), [groupmod\(8\)](#), [newusers\(8\)](#), [useradd\(8\)](#), [userdel\(8\)](#), [usermod\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *[The Linux Programming Interface](#)*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



setuid(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 setuid(2)

System Calls Manual

setuid(2)

NAME [top](#)

setuid - set user identity

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int setuid(uid_t uid);
```

DESCRIPTION [top](#)

setuid() sets the effective user ID of the calling process. If the calling process is privileged (more precisely: if the process has the **CAP_SETUID** capability in its user namespace), the real UID and saved set-user-ID are also set.

Under Linux, **setuid()** is implemented like the POSIX version with the **_POSIX_SAVED_IDS** feature. This allows a set-user-ID (other than root) program to drop all of its user privileges, do some un-privileged work, and then reengage the original effective user ID in a secure manner.

If the user is root or the program is set-user-ID-root, special care must be taken: **setuid()** checks the effective user ID of the caller and if it is the superuser, all process-related user ID's are set to *uid*. After this has occurred, it is impossible for the program to regain root privileges.

Thus, a set-user-ID-root program wishing to temporarily drop root privileges, assume the identity of an unprivileged user, and then regain root privileges afterward cannot use **setuid()**. You can accomplish this with [seteuid\(2\)](#).

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

Note: there are cases where **setuid()** can fail even when the caller is UID 0; it is a grave security error to omit checking for a failure return from **setuid()**.

ERRORS [top](#)

EAGAIN The call would change the caller's real UID (i.e., *uid* does not match the caller's real UID), but there was a temporary failure allocating the necessary kernel data structures.

EAGAIN *uid* does not match the real user ID of the caller and this call would bring the number of processes belonging to the real user ID *uid* over the caller's **RLIMIT_NPROC** resource limit. Since Linux 3.1, this error case no longer occurs (but robust applications should check for this error); see the description of **EAGAIN** in [execve\(2\)](#).

EINVAL The user ID specified in *uid* is not valid in this user namespace.

EPERM The user is not privileged (Linux: does not have the **CAP_SETUID** capability in its user namespace) and *uid* does not match the real UID or saved set-user-ID of the calling process.



VERSIONS [top](#)

C library/kernel differences

At the kernel level, user IDs and group IDs are a per-thread attribute. However, POSIX requires that all threads in a process share the same credentials. The NPTL threading implementation handles the POSIX requirements by providing wrapper functions for the various system calls that change process UIDs and GIDs. These wrapper functions (including the one for **setuid()**) employ a signal-based technique to ensure that when one thread changes credentials, all of the other threads in the process also change their credentials. For details, see [nptl\(7\)](#).

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4.

Not quite compatible with the 4.4BSD call, which sets all of the real, saved, and effective user IDs.

The original Linux **setuid()** system call supported only 16-bit user IDs. Subsequently, Linux 2.4 added **setuid32()** supporting 32-bit IDs. The glibc **setuid()** wrapper function transparently deals with the variation across kernel versions.

NOTES [top](#)

Linux has the concept of the filesystem user ID, normally equal to the effective user ID. The **setuid()** call also sets the filesystem user ID of the calling process. See [setfsuid\(2\)](#).

If *uid* is different from the old effective UID, the process will be forbidden from leaving core dumps.

SEE ALSO [top](#)

[getuid\(2\)](#), [seteuid\(2\)](#), [setfsuid\(2\)](#), [setreuid\(2\)](#), [capabilities\(7\)](#),
[credentials\(7\)](#), [user_namespaces\(7\)](#)

Linux man-pages (unreleased) (date)

setuid(2)

Pages that refer to this page: [capsh\(1\)](#), [access\(2\)](#), [execve\(2\)](#), [getresuid\(2\)](#), [getuid\(2\)](#),
[seccomp\(2\)](#), [seteuid\(2\)](#), [setresuid\(2\)](#), [setreuid\(2\)](#), [syscalls\(2\)](#), [vfork\(2\)](#),
[cap_get_proc\(3\)](#), [euidaccess\(3\)](#), [posix_spawn\(3\)](#), [systemd.exec\(5\)](#), [capabilities\(7\)](#),
[credentials\(7\)](#), [nptl\(7\)](#), [signal-safety\(7\)](#), [user_namespaces\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of
[The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



setgid(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 [setgid\(2\)](#)

System Calls Manual

[setgid\(2\)](#)

NAME [top](#)

setgid - set group identity

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int setgid(gid_t gid);
```

DESCRIPTION [top](#)

`setgid()` sets the effective group ID of the calling process. If the calling process is privileged (more precisely: has the **CAP_SETGID** capability in its user namespace), the real GID and saved set-group-ID are also set.

Under Linux, `setgid()` is implemented like the POSIX version with the **_POSIX_SAVED_IDS** feature. This allows a set-group-ID program that is not set-user-ID-root to drop all of its group privileges, do some un-privileged work, and then reengage the original effective group ID in a secure manner.

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

- EINVAL** The group ID specified in *gid* is not valid in this user namespace.
- EPERM** The calling process is not privileged (does not have the **CAP_SETGID** capability in its user namespace), and *gid* does not match the real group ID or saved set-group-ID of the calling process.

VERSIONS [top](#)

C library/kernel differences

At the kernel level, user IDs and group IDs are a per-thread attribute. However, POSIX requires that all threads in a process share the same credentials. The NPTL threading implementation handles the POSIX requirements by providing wrapper functions for the various system calls that change process UIDs and GIDs. These wrapper functions (including the one for **setgid()**) employ a signal-based technique to ensure that when one thread changes credentials, all of the other threads in the process also change their credentials. For details, see [nptl\(7\)](#).

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4.

The original Linux **setgid()** system call supported only 16-bit group IDs. Subsequently, Linux 2.4 added **setgid32()** supporting 32-bit IDs. The glibc **setgid()** wrapper function transparently deals with the variation across kernel versions.

SEE ALSO [top](#)

[getgid\(2\)](#), [setegid\(2\)](#), [setregid\(2\)](#), [capabilities\(7\)](#),
[credentials\(7\)](#), [user_namespaces\(7\)](#)

Linux man-pages (unreleased) (date)

setgid(2)

Pages that refer to this page: [capsh\(1\)](#), [pmdammv\(1\)](#), [access\(2\)](#), [getgid\(2\)](#),
[getgroups\(2\)](#), [setreuid\(2\)](#), [syscalls\(2\)](#), [cap_get_proc\(3\)](#), [euidaccess\(3\)](#), [proc\(5\)](#),
[systemd.exec\(5\)](#), [credentials\(7\)](#), [nptl\(7\)](#), [signal-safety\(7\)](#), [user_namespaces\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of
[The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



seteuid(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

seteuid(2)

System Calls Manual

seteuid(2)

NAME [top](#)

seteuid, setegid - set effective user or group ID

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int seteuid(uid_t euid);
int setegid(gid_t egid);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
seteuid(), setegid():
    _POSIX_C_SOURCE >= 200112L
    || /* glibc <= 2.19: */ _BSD_SOURCE
```

DESCRIPTION [top](#)

seteuid() sets the effective user ID of the calling process. Unprivileged processes may only set the effective user ID to the

real user ID, the effective user ID or the saved set-user-ID.

Precisely the same holds for **setegid()** with "group" instead of "user".

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

Note: there are cases where **seteuid()** can fail even when the caller is UID 0; it is a grave security error to omit checking for a failure return from **seteuid()**.

ERRORS [top](#)

EINVAL The target user or group ID is not valid in this user namespace.

EPERM In the case of **seteuid()**: the calling process is not privileged (does not have the **CAP_SETUID** capability in its user namespace) and *euid* does not match the current real user ID, current effective user ID, or current saved set-user-ID.

In the case of **setegid()**: the calling process is not privileged (does not have the **CAP_SETGID** capability in its user namespace) and *egid* does not match the current real group ID, current effective group ID, or current saved set-group-ID.

VERSIONS [top](#)

Setting the effective user (group) ID to the saved set-user-ID (saved set-group-ID) is possible since Linux 1.1.37 (1.1.38). On an arbitrary system one should check **_POSIX_SAVED_IDS**.

Under glibc 2.0, **seteuid(*euid*)** is equivalent to **setreuid(-1, *euid*)** and hence may change the saved set-user-ID. Under glibc 2.1 and later, it is equivalent to **setresuid(-1, *euid*, -1)** and hence does not change the saved set-user-ID. Analogous remarks



hold for **setegid()**, with the difference that the change in implementation from **setregid(-1, egid)** to **setresgid(-1, egid, -1)** occurred in glibc 2.2 or 2.3 (depending on the hardware architecture).

According to POSIX.1, **seteuid()** (**setegid()**) need not permit *egid* (*egid*) to be the same value as the current effective user (group) ID, and some implementations do not permit this.

C library/kernel differences

On Linux, **seteuid()** and **setegid()** are implemented as library functions that call, respectively, [setreuid\(2\)](#) and [setregid\(2\)](#).

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, 4.3BSD.

SEE ALSO [top](#)

[geteuid\(2\)](#), [setresuid\(2\)](#), [setreuid\(2\)](#), [setuid\(2\)](#), [capabilities\(7\)](#), [credentials\(7\)](#), [user_namespaces\(7\)](#)

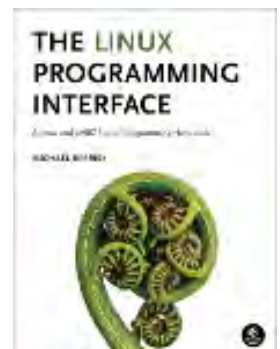
Linux man-pages (unreleased) (date) [seteuid\(2\)](#)

Pages that refer to this page: [pmdaproc\(1\)](#), [setgid\(2\)](#), [setreuid\(2\)](#), [setuid\(2\)](#), [proc\(5\)](#), [credentials\(7\)](#), [nptl\(7\)](#), [pthreads\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



setegid(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 SETEGID(3P)

POSIX Programmer's Manual

SETEGID(3P)**PROLOG** [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

setegid – set the effective group ID

SYNOPSIS [top](#)

```
#include <unistd.h>

int setegid(gid_t gid);
```

DESCRIPTION [top](#)

If *gid* is equal to the real group ID or the saved set-group-ID, or if the process has appropriate privileges, *setegid()* shall set the effective group ID of the calling process to *gid*; the real group ID, saved set-group-ID, and any supplementary group IDs shall remain unchanged.

The `setegid()` function shall not affect the supplementary group list in any way.

RETURN VALUE [top](#)

Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and `errno` set to indicate the error.

ERRORS [top](#)

The `setegid()` function shall fail if:

EINVAL The value of the `gid` argument is invalid and is not supported by the implementation.

EPERM The process does not have appropriate privileges and `gid` does not match the real group ID or the saved set-group-ID.

The following sections are informative.

EXAMPLES [top](#)

None.

APPLICATION USAGE [top](#)

None.

RATIONALE [top](#)

Refer to the RATIONALE section in `setuid(3p)`.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

[exec\(1p\)](#), [getegid\(3p\)](#), [geteuid\(3p\)](#), [getgid\(3p\)](#), [getuid\(3p\)](#),
[seteuid\(3p\)](#), [setgid\(3p\)](#), [setregid\(3p\)](#), [setreuid\(3p\)](#), [setuid\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [sys_types.h\(0p\)](#),
[unistd.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

IEEE/The Open Group

2017

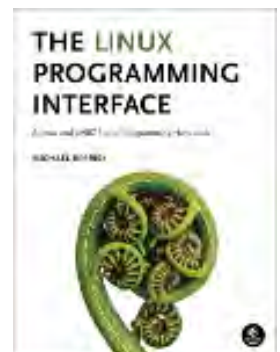
SETEGID(3P)

Pages that refer to this page: [unistd.h\(0p\)](#), [getegid\(3p\)](#), [geteuid\(3p\)](#), [getgid\(3p\)](#),
[getuid\(3p\)](#), [seteuid\(3p\)](#), [setgid\(3p\)](#), [setregid\(3p\)](#), [setreuid\(3p\)](#), [setuid\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



setuid(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 setuid(2)

System Calls Manual

setuid(2)

NAME [top](#)

setuid - set user identity

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>

int setuid(uid_t uid);
```

DESCRIPTION [top](#)

`setuid()` sets the effective user ID of the calling process. If the calling process is privileged (more precisely: if the process has the **CAP_SETUID** capability in its user namespace), the real UID and saved set-user-ID are also set.

Under Linux, `setuid()` is implemented like the POSIX version with the **_POSIX_SAVED_IDS** feature. This allows a set-user-ID (other than root) program to drop all of its user privileges, do some un-privileged work, and then reengage the original effective user ID in a secure manner.

If the user is root or the program is set-user-ID-root, special care must be taken: **setuid()** checks the effective user ID of the caller and if it is the superuser, all process-related user ID's are set to *uid*. After this has occurred, it is impossible for the program to regain root privileges.

Thus, a set-user-ID-root program wishing to temporarily drop root privileges, assume the identity of an unprivileged user, and then regain root privileges afterward cannot use **setuid()**. You can accomplish this with [seteuid\(2\)](#).

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

Note: there are cases where **setuid()** can fail even when the caller is UID 0; it is a grave security error to omit checking for a failure return from **setuid()**.

ERRORS [top](#)

EAGAIN The call would change the caller's real UID (i.e., *uid* does not match the caller's real UID), but there was a temporary failure allocating the necessary kernel data structures.

EAGAIN *uid* does not match the real user ID of the caller and this call would bring the number of processes belonging to the real user ID *uid* over the caller's **RLIMIT_NPROC** resource limit. Since Linux 3.1, this error case no longer occurs (but robust applications should check for this error); see the description of **EAGAIN** in [execve\(2\)](#).

EINVAL The user ID specified in *uid* is not valid in this user namespace.

EPERM The user is not privileged (Linux: does not have the **CAP_SETUID** capability in its user namespace) and *uid* does not match the real UID or saved set-user-ID of the calling process.



VERSIONS [top](#)

C library/kernel differences

At the kernel level, user IDs and group IDs are a per-thread attribute. However, POSIX requires that all threads in a process share the same credentials. The NPTL threading implementation handles the POSIX requirements by providing wrapper functions for the various system calls that change process UIDs and GIDs. These wrapper functions (including the one for **setuid()**) employ a signal-based technique to ensure that when one thread changes credentials, all of the other threads in the process also change their credentials. For details, see [nptl\(7\)](#).

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4.

Not quite compatible with the 4.4BSD call, which sets all of the real, saved, and effective user IDs.

The original Linux **setuid()** system call supported only 16-bit user IDs. Subsequently, Linux 2.4 added **setuid32()** supporting 32-bit IDs. The glibc **setuid()** wrapper function transparently deals with the variation across kernel versions.

NOTES [top](#)

Linux has the concept of the filesystem user ID, normally equal to the effective user ID. The **setuid()** call also sets the filesystem user ID of the calling process. See [setfsuid\(2\)](#).

If *uid* is different from the old effective UID, the process will be forbidden from leaving core dumps.

SEE ALSO [top](#)

[getuid\(2\)](#), [seteuid\(2\)](#), [setfsuid\(2\)](#), [setreuid\(2\)](#), [capabilities\(7\)](#),
[credentials\(7\)](#), [user_namespaces\(7\)](#)

Linux man-pages (unreleased)

(date)

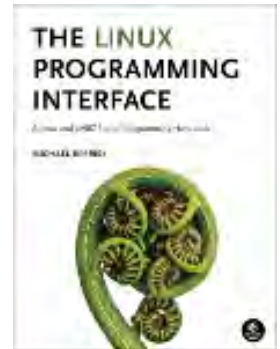
setuid(2)

Pages that refer to this page: [capsh\(1\)](#), [access\(2\)](#), [execve\(2\)](#), [getresuid\(2\)](#), [getuid\(2\)](#),
[seccomp\(2\)](#), [seteuid\(2\)](#), [setresuid\(2\)](#), [setreuid\(2\)](#), [syscalls\(2\)](#), [vfork\(2\)](#),
[cap_get_proc\(3\)](#), [euidaccess\(3\)](#), [posix_spawn\(3\)](#), [systemd.exec\(5\)](#), [capabilities\(7\)](#),
[credentials\(7\)](#), [nptl\(7\)](#), [signal-safety\(7\)](#), [user_namespaces\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of
[The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



getuid(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 getuid(2)

System Calls Manual

getuid(2)

NAME [top](#)

getuid, geteuid - get user identity

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
uid_t getuid(void);  
uid_t geteuid(void);
```

DESCRIPTION [top](#)

`getuid()` returns the real user ID of the calling process.

`geteuid()` returns the effective user ID of the calling process.

ERRORS [top](#)

These functions are always successful and never modify *errno*.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, 4.3BSD.

In UNIX V6 the `getuid()` call returned $(\text{euid} \ll 8) + \text{uid}$. UNIX V7 introduced separate calls `getuid()` and `geteuid()`.

The original Linux `getuid()` and `geteuid()` system calls supported only 16-bit user IDs. Subsequently, Linux 2.4 added `getuid32()` and `geteuid32()`, supporting 32-bit IDs. The glibc `getuid()` and `geteuid()` wrapper functions transparently deal with the variations across kernel versions.

On Alpha, instead of a pair of `getuid()` and `geteuid()` system calls, a single `getxuid()` system call is provided, which returns a pair of real and effective UIDs. The glibc `getuid()` and `geteuid()` wrapper functions transparently deal with this. See [syscall\(2\)](#) for details regarding register mapping.

SEE ALSO [top](#)

[getresuid\(2\)](#), [setreuid\(2\)](#), [setuid\(2\)](#), [credentials\(7\)](#)

Linux man-pages (unreleased) (date) [getuid\(2\)](#)

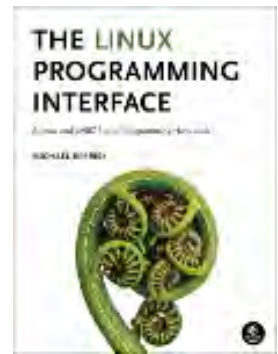
Pages that refer to this page: [groups\(1@@@shadow-utils\)](#), [homectl\(1\)](#), [journalctl\(1\)](#), [localectl\(1\)](#), [loginctl\(1\)](#), [machinectl\(1\)](#), [portablectl\(1\)](#), [ps\(1\)](#), [strace\(1\)](#), [systemctl\(1\)](#), [systemd\(1\)](#), [systemd-analyze\(1\)](#), [systemd-inhibit\(1\)](#), [systemd-nspawn\(1\)](#), [systemd-vmspawn\(1\)](#), [timedatectl\(1\)](#), [userdbctl\(1\)](#), [getresuid\(2\)](#), [seteuid\(2\)](#), [setpgid\(2\)](#), [setresuid\(2\)](#), [setreuid\(2\)](#), [setuid\(2\)](#), [syscalls\(2\)](#), [getlogin\(3\)](#), [id_t\(3type\)](#), [pam_close_session\(3\)](#), [pam_open_session\(3\)](#), [sysexits.h\(3head\)](#), [credentials\(7\)](#), [signal-safety\(7\)](#), [user_namespaces\(7\)](#), [systemd-tmpfiles\(8\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



getgid(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

getgid(2)

System Calls Manual

getgid(2)**NAME** [top](#)

getgid, getegid - get group identity

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
gid_t getgid(void);  
gid_t getegid(void);
```

DESCRIPTION [top](#)

`getgid()` returns the real group ID of the calling process.

`getegid()` returns the effective group ID of the calling process.

ERRORS [top](#)

These functions are always successful and never modify *errno*.



VERSIONS [top](#)

On Alpha, instead of a pair of `getgid()` and `getegid()` system calls, a single `getxgid()` system call is provided, which returns a pair of real and effective GIDs. The glibc `getgid()` and `getegid()` wrapper functions transparently deal with this. See [syscall\(2\)](#) for details regarding register mapping.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, 4.3BSD.

The original Linux `getgid()` and `getegid()` system calls supported only 16-bit group IDs. Subsequently, Linux 2.4 added `getgid32()` and `getegid32()`, supporting 32-bit IDs. The glibc `getgid()` and `getegid()` wrapper functions transparently deal with the variations across kernel versions.

SEE ALSO [top](#)

[getresgid\(2\)](#), [setgid\(2\)](#), [setregid\(2\)](#), [credentials\(7\)](#)

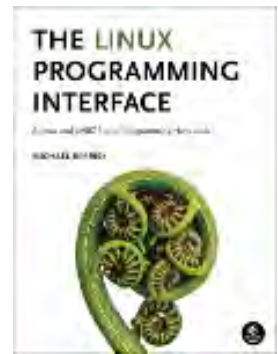
Linux man-pages (unreleased) [\(date\)](#) [getgid\(2\)](#)

Pages that refer to this page: [groups\(1@@shadow-utils\)](#), [ps\(1\)](#), [strace\(1\)](#), [getgroups\(2\)](#), [setgid\(2\)](#), [setreuid\(2\)](#), [syscalls\(2\)](#), [group_member\(3\)](#), [id_t\(3type\)](#), [credentials\(7\)](#), [signal-safety\(7\)](#), [user_namespaces\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



setsid(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 setsid(2)

System Calls Manual

setsid(2)

NAME [top](#)

setsid - creates a session and sets the process group ID

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
pid_t setsid(void);
```

DESCRIPTION [top](#)

setsid() creates a new session if the calling process is not a process group leader. The calling process is the leader of the new session (i.e., its session ID is made the same as its process ID). The calling process also becomes the process group leader of a new process group in the session (i.e., its process group ID is made the same as its process ID).

The calling process will be the only process in the new process group and in the new session.

Initially, the new session has no controlling terminal. For details of how a session acquires a controlling terminal, see [credentials\(7\)](#).

RETURN VALUE [top](#)

On success, the (new) session ID of the calling process is returned. On error, `(pid_t) -1` is returned, and `errno` is set to indicate the error.

ERRORS [top](#)

EPERM The process group ID of any process equals the PID of the calling process. Thus, in particular, `setuid()` fails if the calling process is already a process group leader.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4.

NOTES [top](#)

A child created via `fork(2)` inherits its parent's session ID. The session ID is preserved across an `execve(2)`.

A process group leader is a process whose process group ID equals its PID. Disallowing a process group leader from calling `setuid()` prevents the possibility that a process group leader places itself in a new session while other processes in the process group remain in the original session; such a scenario would break the strict two-level hierarchy of sessions and process groups. In order to be sure that `setuid()` will succeed, call `fork(2)` and have the parent `_exit(2)`, while the child (which by definition can't be a process group leader) calls `setuid()`.

If a session has a controlling terminal, and the **CLOCAL** flag for

that terminal is not set, and a terminal hangup occurs, then the session leader is sent a **SIGHUP** signal.

If a process that is a session leader terminates, then a **SIGHUP** signal is sent to each process in the foreground process group of the controlling terminal.

SEE ALSO [top](#)

[setuid\(1\)](#), [getuid\(2\)](#), [setpgid\(2\)](#), [setpgrp\(2\)](#), [tcgetuid\(3\)](#), [credentials\(7\)](#), [sched\(7\)](#)

Linux man-pages (unreleased) (date)

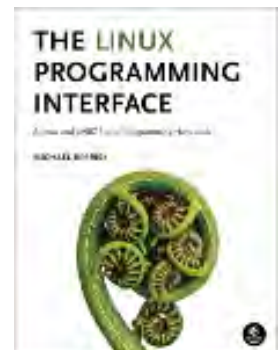
[setuid\(2\)](#)

Pages that refer to this page: [setuid\(1\)](#), [getuid\(2\)](#), [setpgid\(2\)](#), [syscalls\(2\)](#), [daemon\(3\)](#), [posix_spawn\(3\)](#), [tcgetpgrp\(3\)](#), [credentials\(7\)](#), [pthreads\(7\)](#), [pty\(7\)](#), [sched\(7\)](#), [signal-safety\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



getsid(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 getsid(2)

System Calls Manual

getsid(2)

NAME [top](#)

getsid - get session ID

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
pid_t getsid(pid_t pid);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
getsid():
    _XOPEN_SOURCE >= 500
    || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
```

DESCRIPTION [top](#)

getsid() returns the session ID of the process with process ID *pid*. If *pid* is 0, **getsid()** returns the session ID of the calling process.

RETURN VALUE [top](#)

On success, a session ID is returned. On error, `(pid_t) -1` is returned, and `errno` is set to indicate the error.

ERRORS [top](#)

EPERM A process with process ID `pid` exists, but it is not in the same session as the calling process, and the implementation considers this an error.

ESRCH No process with process ID `pid` was found.

VERSIONS [top](#)

Linux does not return **EPERM**.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4. Linux 2.0.

NOTES [top](#)

See [credentials\(7\)](#) for a description of sessions and session IDs.

SEE ALSO [top](#)

[getpgid\(2\)](#), [setsid\(2\)](#), [credentials\(7\)](#)

Linux man-pages (unreleased) [\(date\)](#) [getsid\(2\)](#)

Pages that refer to this page: [ps\(1\)](#), [setsid\(2\)](#), [syscalls\(2\)](#), [id_t\(3type\)](#), [sd_pid_get_owner_uid\(3\)](#), [tcgetsid\(3\)](#), [utmp\(5\)](#), [credentials\(7\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



setpgid(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

setpgid(2)

System Calls Manual

setpgid(2)

NAME [top](#)

setpgid, getpgid, setpgrp, getpgrp - set/get process group

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);
```

```
pid_t getpgrp(void); /* POSIX.1 version */
[[deprecated]] pid_t getpgrp(pid_t pid); /* BSD version */
```

```
int setpgrp(void); /* System V version */
[[deprecated]] int setpgrp(pid_t pid, pid_t pgid); /* BSD version */
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
getpgid():
    _XOPEN_SOURCE >= 500
    || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
```

```
setpgrp() (POSIX.1):
    _XOPEN_SOURCE >= 500
```

```

|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE

```

setpgrp() (BSD), **getpgrp()** (BSD):

[These are available only before glibc 2.19]

```

_BSD_SOURCE &&
! (_POSIX_SOURCE || _POSIX_C_SOURCE || _XOPEN_SOURCE
  || _GNU_SOURCE || _SVID_SOURCE)

```

DESCRIPTION [top](#)

All of these interfaces are available on Linux, and are used for getting and setting the process group ID (PGID) of a process. The preferred, POSIX.1-specified ways of doing this are: **getpgrp**(void), for retrieving the calling process's PGID; and **setpgid**(), for setting a process's PGID.

setpgid() sets the PGID of the process specified by *pid* to *pgid*. If *pid* is zero, then the process ID of the calling process is used. If *pgid* is zero, then the PGID of the process specified by *pid* is made the same as its process ID. If **setpgid**() is used to move a process from one process group to another (as is done by some shells when creating pipelines), both process groups must be part of the same session (see [setsid\(2\)](#) and [credentials\(7\)](#)). In this case, the *pgid* specifies an existing process group to be joined and the session ID of that group must match the session ID of the joining process.

The POSIX.1 version of **getpgrp**(), which takes no arguments, returns the PGID of the calling process.

getpgid() returns the PGID of the process specified by *pid*. If *pid* is zero, the process ID of the calling process is used. (Retrieving the PGID of a process other than the caller is rarely necessary, and the POSIX.1 **getpgrp**() is preferred for that task.)

The System V-style **setpgrp**(), which takes no arguments, is equivalent to **setpgid**(0, 0).

The BSD-specific **setpgrp**() call, which takes arguments *pid* and *pgid*, is a wrapper function that calls

```
setpgid(pid, pgid)
```

Since glibc 2.19, the BSD-specific **setpgrp**() function is no longer exposed by [<unistd.h>](#); calls should be replaced with the



setpgid() call shown above.

The BSD-specific **getpgrp()** call, which takes a single *pid* argument, is a wrapper function that calls

```
getpgid(pid)
```

Since glibc 2.19, the BSD-specific **getpgrp()** function is no longer exposed by `<unistd.h>`; calls should be replaced with calls to the POSIX.1 **getpgrp()** which takes no arguments (if the intent is to obtain the caller's PGID), or with the **getpgid()** call shown above.

RETURN VALUE [top](#)

On success, **setpgid()** and **setpgrp()** return zero. On error, -1 is returned, and *errno* is set to indicate the error.

The POSIX.1 **getpgrp()** always returns the PGID of the caller.

getpgid(), and the BSD-specific **getpgrp()** return a process group on success. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EACCES An attempt was made to change the process group ID of one of the children of the calling process and the child had already performed an `execve(2)` (**setpgid()**, **setpgrp()**).

EINVAL *pgid* is less than 0 (**setpgid()**, **setpgrp()**).

EPERM An attempt was made to move a process into a process group in a different session, or to change the process group ID of one of the children of the calling process and the child was in a different session, or to change the process group ID of a session leader (**setpgid()**, **setpgrp()**).

EPERM The target process group does not exist. (**setpgid()**, **setpgrp()**).

ESRCH For **getpgid()**: *pid* does not match any process. For **setpgid()**: *pid* is not the calling process and not a child of the calling process.

STANDARDS [top](#)

`getpgid()`
`setpgid()`
`getpgrp()` (no args)
`setpgrp()` (no args)
POSIX.1-2008 (but see HISTORY).

`setpgrp()` (2 args)
`getpgrp()` (1 arg)
None.

HISTORY [top](#)

`getpgid()`
`setpgid()`
`getpgrp()` (no args)
POSIX.1-2001.

`setpgrp()` (no args)
POSIX.1-2001. POSIX.1-2008 marks it as obsolete.

`setpgrp()` (2 args)
`getpgrp()` (1 arg)
4.2BSD.

NOTES [top](#)

A child created via `fork(2)` inherits its parent's process group ID. The PGID is preserved across an `execve(2)`.

Each process group is a member of a session and each process is a member of the session of which its process group is a member. (See `credentials(7)`.)

A session can have a controlling terminal. At any time, one (and only one) of the process groups in the session can be the foreground process group for the terminal; the remaining process groups are in the background. If a signal is generated from the terminal (e.g., typing the interrupt key to generate **SIGINT**), that signal is sent to the foreground process group. (See `termios(3)` for a description of the characters that generate signals.) Only the foreground process group may `read(2)` from the terminal; if a background process group tries to `read(2)` from the terminal, then the group is sent a **SIGTTIN** signal, which suspends

it. The [tcgetpgrp\(3\)](#) and [tcsetpgrp\(3\)](#) functions are used to get/set the foreground process group of the controlling terminal.

The [setpgid\(\)](#) and [getpgrp\(\)](#) calls are used by programs such as [bash\(1\)](#) to create process groups in order to implement shell job control.

If the termination of a process causes a process group to become orphaned, and if any member of the newly orphaned process group is stopped, then a **SIGHUP** signal followed by a **SIGCONT** signal will be sent to each process in the newly orphaned process group. An orphaned process group is one in which the parent of every member of process group is either itself also a member of the process group or is a member of a process group in a different session (see also [credentials\(7\)](#)).

SEE ALSO [top](#)

[getuid\(2\)](#), [setsid\(2\)](#), [tcgetpgrp\(3\)](#), [tcsetpgrp\(3\)](#), [termios\(3\)](#), [credentials\(7\)](#)

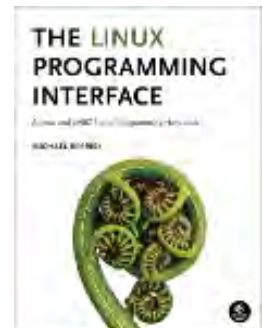
Linux man-pages (unreleased) (date) [setpgid\(2\)](#)

Pages that refer to this page: [setpgid\(1\)](#), [strace\(1\)](#), [fork\(2\)](#), [getsid\(2\)](#), [setsid\(2\)](#), [syscalls\(2\)](#), [exit\(3\)](#), [id_t\(3type\)](#), [killpg\(3\)](#), [posix_spawn\(3\)](#), [tcgetpgrp\(3\)](#), [credentials\(7\)](#), [pthread\(7\)](#), [signal-safety\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



dup(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 dup(2)

System Calls Manual

dup(2)**NAME** [top](#)

dup, dup2, dup3 - duplicate a file descriptor

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);

#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <fcntl.h> /* Definition of O_* constants */
#include <unistd.h>

int dup3(int oldfd, int newfd, int flags);
```

DESCRIPTION [top](#)

The **dup()** system call allocates a new file descriptor that refers to the same open file description as the descriptor *oldfd*. (For an explanation of open file descriptions, see [open\(2\)](#).) The new

file descriptor number is guaranteed to be the lowest-numbered file descriptor that was unused in the calling process.

After a successful return, the old and new file descriptors may be used interchangeably. Since the two file descriptors refer to the same open file description, they share file offset and file status flags; for example, if the file offset is modified by using `lseek(2)` on one of the file descriptors, the offset is also changed for the other file descriptor.

The two file descriptors do not share file descriptor flags (the close-on-exec flag). The close-on-exec flag (`FD_CLOEXEC`; see `fcntl(2)`) for the duplicate descriptor is off.

dup2()

The `dup2()` system call performs the same task as `dup()`, but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in `newfd`. In other words, the file descriptor `newfd` is adjusted so that it now refers to the same open file description as `oldfd`.

If the file descriptor `newfd` was previously open, it is closed before being reused; the close is performed silently (i.e., any errors during the close are not reported by `dup2()`).

The steps of closing and reusing the file descriptor `newfd` are performed *atomically*. This is important, because trying to implement equivalent functionality using `close(2)` and `dup()` would be subject to race conditions, whereby `newfd` might be reused between the two steps. Such reuse could happen because the main program is interrupted by a signal handler that allocates a file descriptor, or because a parallel thread allocates a file descriptor.

Note the following points:

- If `oldfd` is not a valid file descriptor, then the call fails, and `newfd` is not closed.
- If `oldfd` is a valid file descriptor, and `newfd` has the same value as `oldfd`, then `dup2()` does nothing, and returns `newfd`.

dup3()

`dup3()` is the same as `dup2()`, except that:



- The caller can force the close-on-exec flag to be set for the new file descriptor by specifying **O_CLOEXEC** in *flags*. See the description of the same flag in [open\(2\)](#) for reasons why this may be useful.
- If *oldfd* equals *newfd*, then **dup3()** fails with the error **EINVAL**.

RETURN VALUE [top](#)

On success, these system calls return the new file descriptor. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EBADF *oldfd* isn't an open file descriptor.

EBADF *newfd* is out of the allowed range for file descriptors (see the discussion of **RLIMIT_NOFILE** in [getrlimit\(2\)](#)).

EBUSY (Linux only) This may be returned by **dup2()** or **dup3()** during a race condition with [open\(2\)](#) and **dup()**.

EINTR The **dup2()** or **dup3()** call was interrupted by a signal; see [signal\(7\)](#).

EINVAL (**dup3()**) *flags* contain an invalid value.

EINVAL (**dup3()**) *oldfd* was equal to *newfd*.

EMFILE The per-process limit on the number of open file descriptors has been reached (see the discussion of **RLIMIT_NOFILE** in [getrlimit\(2\)](#)).

STANDARDS [top](#)

dup()
dup2() POSIX.1-2008.

dup3() Linux.

HISTORY [top](#)

dup()
dup2() POSIX.1-2001, SVr4, 4.3BSD.
dup3() Linux 2.6.27, glibc 2.9.

NOTES [top](#)

The error returned by **dup2()** is different from that returned by **fcntl(..., F_DUPFD, ...)** when *newfd* is out of range. On some systems, **dup2()** also sometimes returns **EINVAL** like **F_DUPFD**.

If *newfd* was open, any errors that would have been reported at **close(2)** time are lost. If this is of concern, then—unless the program is single-threaded and does not allocate file descriptors in signal handlers—the correct approach is *not* to close *newfd* before calling **dup2()**, because of the race condition described above. Instead, code something like the following could be used:

```
/* Obtain a duplicate of 'newfd' that can subsequently
   be used to check for close() errors; an EBADF error
   means that 'newfd' was not open. */

tmpfd = dup(newfd);
if (tmpfd == -1 && errno != EBADF) {
    /* Handle unexpected dup() error. */
}

/* Atomically duplicate 'oldfd' on 'newfd'. */

if (dup2(oldfd, newfd) == -1) {
    /* Handle dup2() error. */
}

/* Now check for close() errors on the file originally
   referred to by 'newfd'. */

if (tmpfd != -1) {
    if (close(tmpfd) == -1) {
        /* Handle errors from close. */
    }
}
```



SEE ALSO [top](#)

[close\(2\)](#), [fcntl\(2\)](#), [open\(2\)](#), [pidfd_getfd\(2\)](#)

Linux man-pages (unreleased) **(date)**

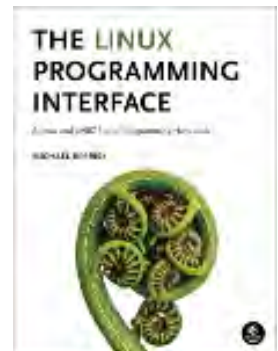
dup(2)

Pages that refer to this page: [bpf\(2\)](#), [fcntl\(2\)](#), [flock\(2\)](#), [getrlimit\(2\)](#), [kcmp\(2\)](#), [lseek\(2\)](#), [open\(2\)](#), [pidfd_getfd\(2\)](#), [syscalls\(2\)](#), [fileno\(3\)](#), [getdtablesize\(3\)](#), [posix_spawn\(3\)](#), [epoll\(7\)](#), [pipe\(7\)](#), [signal-safety\(7\)](#), [unix\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



daemon(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [BUGS](#) | [SEE ALSO](#)

 daemon(3)

Library Functions Manual

daemon(3)

NAME [top](#)

daemon - run in the background

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int daemon(int nochdir, int noclose);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

daemon():

Since glibc 2.21:

```
    _DEFAULT_SOURCE
```

In glibc 2.19 and 2.20:

```
    _DEFAULT_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
```

Up to and including glibc 2.19:

```
    _BSD_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
```

DESCRIPTION [top](#)



The `daemon()` function is for programs wishing to detach themselves from the controlling terminal and run in the background as system daemons.

If `nochdir` is zero, `daemon()` changes the process's current working directory to the root directory ("`/`"); otherwise, the current working directory is left unchanged.

If `noclose` is zero, `daemon()` redirects standard input, standard output, and standard error to `/dev/null`; otherwise, no changes are made to these file descriptors.

RETURN VALUE [top](#)

(This function forks, and if the `fork(2)` succeeds, the parent calls `_exit(2)`, so that further errors are seen by the child only.) On success `daemon()` returns zero. If an error occurs, `daemon()` returns -1 and sets `errno` to any of the errors specified for the `fork(2)` and `setsid(2)`.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------------------|---------------|---------|
| <code>daemon()</code> | Thread safety | MT-Safe |

VERSIONS [top](#)

A similar function appears on the BSDs.

The glibc implementation can also return -1 when `/dev/null` exists but is not a character device with the expected major and minor numbers. In this case, `errno` need not be set.

STANDARDS [top](#)

None.

HISTORY [top](#)

4.4BSD.

BUGS [top](#)

The GNU C library implementation of this function was taken from BSD, and does not employ the double-fork technique (i.e., `fork(2)`, `setsid(2)`, `fork(2)`) that is necessary to ensure that the resulting daemon process is not a session leader. Instead, the resulting daemon *is* a session leader. On systems that follow System V semantics (e.g., Linux), this means that if the daemon opens a terminal that is not already a controlling terminal for another session, then that terminal will inadvertently become the controlling terminal for the daemon.

SEE ALSO [top](#)

`fork(2)`, `setsid(2)`, `daemon(7)`, `logrotate(8)`

Linux man-pages (unreleased) [\(date\)](#)

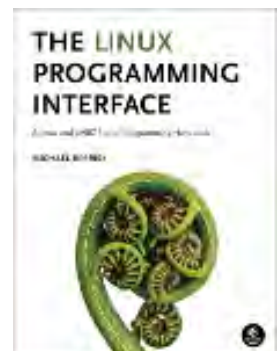
[daemon\(3\)](#)

Pages that refer to this page: `fork(2)`, `daemon(7)`

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



ps(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [EXAMPLES](#) | [SIMPLE PROCESS SELECTION](#) | [PROCESS SELECTION BY LIST](#) | [OUTPUT FORMAT CONTROL](#) | [OUTPUT MODIFIERS](#) | [THREAD DISPLAY](#) | [OTHER INFORMATION](#) | [NOTES](#) | [PROCESS FLAGS](#) | [PROCESS STATE CODES](#) | [OBSOLETE SORT KEYS](#) | [AIX FORMAT DESCRIPTORS](#) | [STANDARD FORMAT SPECIFIERS](#) | [ENVIRONMENT VARIABLES](#) | [PERSONALITY](#) | [BUGS](#) | [SEE ALSO](#) | [STANDARDS](#) | [AUTHOR](#) | [COLOPHON](#)

 PS(1)

User Commands

PS(1)**NAME** [top](#)

`ps` - report a snapshot of the current processes.

SYNOPSIS [top](#)

`ps` [*options*]

DESCRIPTION [top](#)

`ps` displays information about a selection of the active processes. If you want a repetitive update of the selection and the displayed information, use **top** instead.

This version of `ps` accepts several kinds of options:

- 1 UNIX options, which may be grouped and must be preceded by a dash.
- 2 BSD options, which may be grouped and must not be used with a dash.
- 3 GNU long options, which are preceded by two dashes.

Options of different types may be freely mixed, but conflicts can appear. There are some synonymous options, which are functionally identical, due to the many standards and `ps` implementations that this `ps` is compatible with.

By default, `ps` selects all processes with the same effective user ID (`euid=EUID`) as the current user and associated with the same terminal as the invoker. It displays the process ID (`pid=PID`), the terminal associated with the process (`tname=TTY`), the cumulated CPU time in `[DD-]hh:mm:ss` format (`time=TIME`), and the executable name (`ucmd=CMD`). Output is unsorted by default.

The use of BSD-style options will add process state (`stat=STAT`) to the default display and show the command args (`args=COMMAND`) instead of the executable name. You can override this with the **PS_FORMAT** environment variable. The use of BSD-style options

will also change the process selection to include processes on other terminals (TTYs) that are owned by you; alternately, this may be described as setting the selection to be the set of all processes filtered to exclude processes owned by other users or not on a terminal. These effects are not considered when options are described as being "identical" below, so **-M** will be considered identical to **Z** and so on.

Except as described below, process selection options are additive. The default selection is discarded, and then the selected processes are added to the set of processes to be displayed. A process will thus be shown if it meets any of the given selection criteria.

EXAMPLES [top](#)

To see every process on the system using standard syntax:

```
ps -e
ps -ef
ps -eF
ps -ely
```

To see every process on the system using BSD syntax:

```
ps ax
ps axu
```

To print a process tree:

```
ps -ejH
ps axjf
```

To get info about threads:

```
ps -eLf
ps axms
```

To get security info:

```
ps -eo euser,ruser,suser,fuser,f,comm,label
ps axZ
ps -eM
```

To see every process running as root (real & effective ID) in user format:

```
ps -U root -u root u
```

To see every process with a user-defined format:

```
ps -eo pid,tid,class,rtprio,ni,pri,psr,pcpu,stat,wchan:14,comm
ps axo stat,euid,ruid,tty,tpgid,sess,pgrp,ppid,pid,pcpu,comm
ps -Ao pid,tt,user,fname,tmout,f,wchan
```

Print only the process IDs of syslogd:

```
ps -C syslogd -o pid=
```

Print only the name of PID 42:

```
ps -q 42 -o comm=
```

SIMPLE PROCESS SELECTION [top](#)



- a** Lift the BSD-style "only yourself" restriction, which is imposed upon the set of all processes when some BSD-style (without "-") options are used or when the **ps** personality setting is BSD-like. The set of processes selected in this manner is in addition to the set of processes selected by other means. An alternate description is that this option causes **ps** to list all processes with a terminal (tty), or to list all processes when used together with the **x** option.

- A** Select all processes. Identical to **-e**.

- a** Select all processes except both session leaders (see [getsid\(2\)](#)) and processes not associated with a terminal.

- d** Select all processes except session leaders.

- deselect**
Select all processes except those that fulfill the specified conditions (negates the selection). Identical to **-N**.

- e** Select all processes. Identical to **-A**.

- g** Really all, even session leaders. This flag is obsolete and may be discontinued in a future release. It is normally implied by the **a** flag, and is only useful when operating in the sunos4 personality.

- N** Select all processes except those that fulfill the specified conditions (negates the selection). Identical to **--deselect**.

- T** Select all processes associated with this terminal. Identical to the **t** option without any argument.

- r** Restrict the selection to only running processes.

- x** Lift the BSD-style "must have a tty" restriction, which is imposed upon the set of all processes when some BSD-style (without "-") options are used or when the **ps** personality setting is BSD-like. The set of processes selected in this manner is in addition to the set of processes selected by other means. An alternate description is that this option causes **ps** to list all processes owned by you (same EUID as **ps**), or to list all processes when used together with the **a** option.

PROCESS SELECTION BY LIST [top](#)

These options accept a single argument in the form of a blank-separated or comma-separated list. They can be used multiple times. For example: **ps -p "1 2" -p 3,4**

123 Identical to **--pid 123**.

+123 Identical to **--sid 123**.

-123 Select by process group ID (PGID).

-C *cmdlist*

Select by command name. This selects the processes whose executable name is given in *cmdlist*. NOTE: The command name is not the same as the command line. Previous versions of *procs* and the kernel truncated this command name to 15 characters. This limitation is no longer present in both. If you depended on matching only 15 characters, you may no longer get a match.

-G *grplist*

Select by real group ID (RGID) or name. This selects the processes whose real group name or ID is in the *grplist* list. The real group ID identifies the group of the user who created the process, see [getgid\(2\)](#).

-g *grplist*

Select by session OR by effective group name. Selection by session is specified by many standards, but selection by effective group is the logical behavior that several other operating systems use. This **ps** will select by session when the list is completely numeric (as sessions are). Group ID numbers will work only when some group names are also specified. See the **-s** and **--group** options.

--Group *grplist*

Select by real group ID (RGID) or name. Identical to **-G**.

--group *grplist*

Select by effective group ID (EGID) or name. This selects the processes whose effective group name or ID is in *grplist*. The effective group ID describes the group whose file access permissions are used by the process (see [getegid\(2\)](#)). The **-g** option is often an alternative to **--group**.

p *pidlist*

Select by process ID. Identical to **-p** and **--pid**.

-p *pidlist*

Select by PID. This selects the processes whose process ID numbers appear in *pidlist*. Identical to **p** and **--pid**.

--pid *pidlist*

Select by process ID. Identical to **-p** and **p**.

--ppid *pidlist*

Select by parent process ID. This selects the processes with a parent process ID in *pidlist*. That is, it selects processes that are children of those listed in *pidlist*.

q *pidlist*

Select by process ID (quick mode). Identical to **-q** and **--quick-pid**.

-q *pidlist*

Select by PID (quick mode). This selects the processes whose process ID numbers appear in *pidlist*. With this



option **ps** reads the necessary info only for the pids listed in the *pidlist* and doesn't apply additional filtering rules. The order of pids is unsorted and preserved. No additional selection options, sorting and forest type listings are allowed in this mode. Identical to **q** and **--quick-pid**.

--quick-pid *pidlist*

Select by process ID (quick mode). Identical to **-q** and **q**.

-s *sesslist*

Select by session ID. This selects the processes with a session ID specified in *sesslist*.

--sid *sesslist*

Select by session ID. Identical to **-s**.

t *ttylist*

Select by tty. Nearly identical to **-t** and **--tty**, but can also be used with an empty *ttylist* to indicate the terminal associated with **ps**. Using the **T** option is considered cleaner than using **t** with an empty *ttylist*.

-t *ttylist*

Select by tty. This selects the processes associated with the terminals given in *ttylist*. Terminals (ttys, or screens for text output) can be specified in several forms: `/dev/ttyS1`, `ttyS1`, `S1`. A plain "-" may be used to select processes not attached to any terminal.

--tty *ttylist*

Select by terminal. Identical to **-t** and **t**.

U *userlist*

Select by effective user ID (EUID) or name. This selects the processes whose effective user name or ID is in *userlist*. The effective user ID describes the user whose file access permissions are used by the process (see `geteuid(2)`). Identical to **-u** and **--user**.

-U *userlist*

Select by real user ID (RUID) or name. It selects the processes whose real user name or ID is in the *userlist* list. The real user ID identifies the user who created the process, see `getuid(2)`.

-u *userlist*

Select by effective user ID (EUID) or name. This selects the processes whose effective user name or ID is in *userlist*.

The effective user ID describes the user whose file access permissions are used by the process (see `geteuid(2)`). Identical to **U** and **--user**.

--User *userlist*

Select by real user ID (RUID) or name. Identical to **-U**.

--user *userlist*



Select by effective user ID (EUID) or name. Identical to **-u** and **U**.

OUTPUT FORMAT CONTROL [top](#)

These options are used to choose the information displayed by **ps**. The output may differ by personality.

- c** Show different scheduler information for the **-l** option.
- context**
Display security context format (for SELinux).
- f** Do full-format listing. This option can be combined with many other UNIX-style options to add additional columns. It also causes the command arguments to be printed. When used with **-L**, the NLWP (number of threads) and LWP (thread ID) columns will be added. See the **c** option, the format keyword **args**, and the format keyword **comm**.
- F** Extra full format. See the **-f** option, which **-F** implies.
- format *format***
user-defined format. Identical to **-o** and **o**.
- j** BSD job control format.
- j** Jobs format.
- l** Display BSD long format.
- l** Long format. The **-y** option is often useful with this.
- M** Add a column of security data. Identical to **Z** (for SELinux).
- 0 *format***
is preloaded **o** (overloaded). The BSD **0** option can act like **-0** (user-defined output format with some common fields predefined) or can be used to specify sort order. Heuristics are used to determine the behavior of this option. To ensure that the desired behavior is obtained (sorting or formatting), specify the option in some other way (e.g. with **-0** or **--sort**). When used as a formatting option, it is identical to **-0**, with the BSD personality.
- 0 *format***
Like **-o**, but preloaded with some default columns. Identical to **-o pid,format,state,tname,time,command** or **-o pid,format,tname,time,cmd**, see **-o** below.
- o *format***
Specify user-defined format. Identical to **-o** and **--format**.
- o *format***
User-defined format. *format* is a single argument in the form of a blank-separated or comma-separated list, which



offers a way to specify individual output columns. The recognized keywords are described in the **STANDARD FORMAT SPECIFIERS** section below. Headers may be renamed (**ps -o pid,ruser=RealUser -o comm=Command**) as desired. If all column headers are empty (**ps -o pid= -o comm=**) then the header line will not be output. Column width will increase as needed for wide headers; this may be used to widen up columns such as WCHAN (**ps -o pid,wchan=WIDE-WCHAN-COLUMN -o comm**). Explicit width control (**ps opid,wchan:42,cmd**) is offered too. The behavior of **ps -o pid=X,comm=Y** varies with personality; output may be one column named "X,comm=Y" or two columns named "X" and "Y". Use multiple **-o** options when in doubt. Use the **PS_FORMAT** environment variable to specify a default as desired; DefSysV and DefBSD are macros that may be used to choose the default UNIX or BSD columns.

- P** Add a column showing **psr**.
- s** Display signal format.
- u** Display user-oriented format.
- v** Display virtual memory format.
- X** Register format.
- y** Do not show flags; show rss in place of addr. This option can only be used with **-l**.
- Z** Add a column of security data. Identical to **-M** (for SELinux).

OUTPUT MODIFIERS [top](#)

- c** Show the true command name. This is derived from the name of the executable file, rather than from the argv value. Command arguments and any modifications to them are thus not shown. This option effectively turns the **args** format keyword into the **comm** format keyword; it is useful with the **-f** format option and with the various BSD-style format options, which all normally display the command arguments. See the **-f** option, the format keyword **args**, and the format keyword **comm**.
- cols *n***
Set screen width.
- columns *n***
Set screen width.
- cumulative**
Include some dead child process data (as a sum with the parent).
- D *format***
Set the date format of the **lstart** field to *format*. This format is parsed by [strftime\(3\)](#) and should be a maximum of



24 characters to not mis-align columns.

--date-format *format*

Identical to **-D**.

e Show the environment after the command.

f ASCII art process hierarchy (forest).

--forest

ASCII art process tree.

h No header. (or, one header per screen in the BSD personality). The **h** option is problematic. Standard BSD **ps** uses this option to print a header on each page of output, but older Linux **ps** uses this option to totally disable the header. This version of **ps** follows the Linux usage of not printing the header unless the BSD personality has been selected, in which case it prints a header on each page of output. Regardless of the current personality, you can use the long options **--headers** and **--no-headers** to enable printing headers each page or disable headers entirely, respectively.

-H Show process hierarchy (forest).

--headers

Repeat header lines, one per page of output.

k *spec* Specify sorting order. Sorting syntax is `[+|-]key[, [+|-]key[, ...]]`. Choose a multi-letter key from the **STANDARD FORMAT SPECIFIERS** section. The "+" is optional since default direction is increasing numerical or lexicographic order. Identical to **--sort**.

Examples:

```
ps jaxkuid, -ppid, +pid
ps axk comm o comm, args
ps kstart_time -ef
```

--lines *n*

Set screen height.

n Numeric output for WCHAN and USER (including all types of UID and GID).

--no-headers

Print no header line at all. **--no-heading** is an alias for this option.

O *order*

Sorting order (overloaded). The BSD **O** option can act like **-O** (user-defined output format with some common fields predefined) or can be used to specify sort order. Heuristics are used to determine the behavior of this option. To ensure that the desired behavior is obtained (sorting or formatting), specify the option in some other way (e.g. with **-O** or **--sort**).



For sorting, obsolete BSD **O** option syntax is **O**[+|-]*k1*[, [+|-]*k2*[, ...]]. It orders the processes listing according to the multilevel sort specified by the sequence of one-letter short keys *k1*, *k2*, ... described in the **OBSOLETE SORT KEYS** section below. The "+" is currently optional, merely re-iterating the default direction on a key, but may help to distinguish an **O** sort from an **O** format. The "-" reverses direction only on the key it precedes.

--rows *n*

Set screen height.

S

Sum up some information, such as CPU usage, from dead child processes into their parent. This is useful for examining a system where a parent process repeatedly forks off short-lived children to do work.

--sort *spec*

Specify sorting order. Sorting syntax is [+|-]*key*[, [+|-]*key*[, ...]]. Choose a multi-letter key from the **STANDARD FORMAT SPECIFIERS** section. The "+" is optional since default direction is increasing numerical or lexicographic order. Identical to **k**. For example: **ps jax --sort=uid,-ppid,+pid**

--signames

Show signal masks using abbreviated signal names and expands the column. If the column width cannot show all signals, the column will end with a plus "+". Columns with only a hyphen have no signals.

w

Wide output. Use this option twice for unlimited width.

-w

Wide output. Use this option twice for unlimited width.

--width *n*

Set screen width.

THREAD DISPLAY [top](#)**H**

Show threads as if they were processes.

-L

Show threads, possibly with LWP and NLWP columns.

m

Show threads after processes.

-m

Show threads after processes.

-T

Show threads, possibly with SPID column.

OTHER INFORMATION [top](#)**--help** *section*

Print a help message. The *section* argument can be one of *simple*, *list*, *output*, *threads*, *misc*, or *all*. The argument can be shortened to one of the underlined letters as in:



s|l|o|t|m|a.

- info** Print debugging info.
- L** List all format specifiers.
- V** Print the procs-ng version.
- V** Print the procs-ng version.
- version**
Print the procs-ng version.

NOTES [top](#)

This **ps** works by reading the virtual files in /proc. This **ps** does not need to be setuid kmem or have any privileges to run. Do not give this **ps** any special permissions.

CPU usage is currently expressed as the percentage of time spent running during the entire lifetime of a process. This is not ideal, and it does not conform to the standards that **ps** otherwise conforms to. CPU usage is unlikely to add up to exactly 100%.

The SIZE and RSS fields don't count some parts of a process including the page tables, kernel stack, struct thread_info, and struct task_struct. This is usually at least 20 KiB of memory that is always resident. SIZE is the virtual size of the process (code+data+stack).

Processes marked <defunct> are dead processes (so-called "zombies") that remain because their parent has not destroyed them properly. These processes will be destroyed by **init(8)** if the parent process exits.

If the length of the username is greater than the width of the display column, the username will be truncated. See the **-o** and **-O** formatting options to customize length.

Commands options such as **ps -aux** are not recommended as it is a confusion of two different standards. According to the POSIX and UNIX standards, the above command asks to display all processes with a TTY (generally the commands users are running) plus all processes owned by a user named **x**. If that user doesn't exist, then **ps** will assume you really meant "**ps aux**".

PROCESS FLAGS [top](#)

The sum of these values is displayed in the "F" column, which is provided by the **flags** output specifier:

- 1 forked but didn't exec
- 4 used super-user privileges

PROCESS STATE CODES [top](#)

Here are the different values that the **s**, **stat** and **state** output specifiers (header "STAT" or "S") will display to describe the state of a process:

```

D    uninterruptible sleep (usually IO)
I    Idle kernel thread
R    running or runnable (on run queue)
S    interruptible sleep (waiting for an event to
    complete)
T    stopped by job control signal
t    stopped by debugger during the tracing
W    paging (not valid since the 2.6.xx kernel)
X    dead (should never be seen)
Z    defunct ("zombie") process, terminated but not
    reaped by its parent

```

For BSD formats and when the **stat** keyword is used, additional characters may be displayed:

```

<    high-priority (not nice to other users)
N    low-priority (nice to other users)
L    has pages locked into memory (for real-time and
    custom IO)
s    is a session leader
l    is multi-threaded (using CLONE_THREAD, like NPTL
    pthreads do)
+    is in the foreground process group

```

OBSOLETE SORT KEYS [top](#)

These keys are used by the BSD **O** option (when it is used for sorting). The GNU **--sort** option doesn't use these keys, but the specifiers described below in the **STANDARD FORMAT SPECIFIERS** section. Note that the values used in sorting are the internal values **ps** uses and not the "cooked" values used in some of the output format fields (e.g. sorting on tty will sort into device number, not according to the terminal name displayed). Pipe **ps** output into the [sort\(1\)](#) command if you want to sort the cooked values.

| KEY | LONG | DESCRIPTION |
|-----|----------|----------------------------------|
| c | cmd | simple name of executable |
| C | pcpu | cpu utilization |
| f | flags | flags as in long format F field |
| g | pgrp | process group ID |
| G | tpgid | controlling tty process group ID |
| j | cutime | cumulative user time |
| J | cstime | cumulative system time |
| k | utime | user time |
| m | minflt | number of minor page faults |
| M | majflt | number of major page faults |
| n | cminflt | cumulative minor page faults |
| N | cmajflt | cumulative major page faults |
| o | session | session ID |
| p | pid | process ID |
| P | ppid | parent process ID |
| r | rss | resident set size |
| R | resident | resident pages |
| s | size | memory size in kilobytes |



| | | |
|---|------------|--|
| S | share | amount of shared pages |
| t | tty | the device number of the controlling tty |
| T | start_time | time process was started |
| U | uid | user ID number |
| u | user | user name |
| v | vsize | total VM size in KiB |
| y | priority | kernel scheduling priority |

AIX FORMAT DESCRIPTORS [top](#)

This **ps** supports AIX format descriptors, which work somewhat like the formatting codes of `printf(1)` and `printf(3)`. The **NORMAL** codes are described in the next section.

| CODE | NORMAL | HEADER |
|------|--------|---------|
| %C | pcpu | %CPU |
| %G | group | GROUP |
| %P | ppid | PPID |
| %U | user | USER |
| %a | args | COMMAND |
| %c | comm | COMMAND |
| %g | rgroup | RGROUP |
| %n | nice | NI |
| %p | pid | PID |
| %r | pgid | PGID |
| %t | etime | ELAPSED |
| %u | ruser | RUSER |
| %x | time | TIME |
| %y | tty | TTY |
| %z | vsz | VSZ |

STANDARD FORMAT SPECIFIERS [top](#)

Here are the different keywords that may be used to control the output format (e.g., with option **-o**) or to sort the selected processes with the GNU-style **--sort** option.

For example: **ps -eo pid,user,args --sort user**

This version of **ps** tries to recognize most of the keywords used in other implementations of **ps**.

The following user-defined format specifiers may contain spaces: **args**, **cmd**, **comm**, **command**, **fname**, **ucmd**, **ucomm**, **lstart**, **bsdstart**, **start**.

Some keywords may not be available for sorting.

| CODE | HEADER | DESCRIPTION |
|-------------|--------|--|
| %cpu | %CPU | cpu utilization of the process in "##.#" format. Currently, it is the CPU time used divided by the time the process has been running (cputime/realtime ratio), expressed as a percentage. It will not add up to 100% unless you are lucky. (alias pcpu). |
| %mem | %MEM | ratio of the process's resident set size |

| | | |
|-----------------|---------|--|
| | | to the physical memory on the machine, expressed as a percentage. (alias pmem). |
| ag_id | AGID | The autogroup identifier associated with a process which operates in conjunction with the CFS scheduler to improve interactive desktop performance. |
| ag_nice | AGNI | The autogroup nice value which affects scheduling of all processes in that group. |
| args | COMMAND | command with all its arguments as a string. Modifications to the arguments may be shown. The output in this column may contain spaces. A process marked <defunct> is partly dead, waiting to be fully destroyed by its parent. Sometimes the process args will be unavailable; when this happens, ps will instead print the executable name in brackets. (alias cmd , command). See also the comm format keyword, the -f option, and the c option. When specified last, this column will extend to the edge of the display. If ps can not determine display width, as when output is redirected (piped) into a file or another command, the output width is undefined (it may be 80, unlimited, determined by the TERM variable, and so on). The COLUMNS environment variable or --cols option may be used to exactly determine the width in this case. The w or -w option may be also be used to adjust width. |
| blocked | BLOCKED | mask of the blocked signals, see signal(7) . According to the width of the field, a 32 or 64-bit mask in hexadecimal format is displayed, unless the --signames option is used. (alias sig_block , sigmask). |
| bsdstart | START | time the command started. If the process was started less than 24 hours ago, the output format is "HH:MM", else it is "Mmm:SS" (where Mmm is the three letters of the month). See also lstart , start , start_time , and stime . |
| bsdtime | TIME | accumulated cpu time, user + system. The display format is usually "MMM:SS", but can be shifted to the right if the process used more than 999 minutes of cpu time. |
| c | C | processor utilization. Currently, this is the integer value of the percent usage over the lifetime of the process. (see %cpu). |
| caught | CAUGHT | mask of the caught signals, see signal(7) . According to the width of the field, a 32 |



or 64 bits mask in hexadecimal format is displayed, unless the **--signames** option is used. (alias **sig_catch**, **sigcatch**).

| | | |
|-------------------|----------|---|
| cgroupname | CGNAME | display name of control groups to which the process belongs. |
| cgroup | CGROUP | display control groups to which the process belongs. |
| cgroupns | CGROUPNS | Unique inode number describing the namespace the process belongs to. See namespaces(7) . |
| class | CLS | scheduling class of the process. (alias policy , cls). Field's possible values are: <ul style="list-style-type: none"> - not reported TS SCHED_OTHER FF SCHED_FIFO RR SCHED_RR B SCHED_BATCH ISO SCHED_ISO IDL SCHED_IDLE DLN SCHED_DEADLINE ? unknown value |
| cls | CLS | scheduling class of the process. (alias policy , cls). Field's possible values are: <ul style="list-style-type: none"> - not reported TS SCHED_OTHER FF SCHED_FIFO RR SCHED_RR B SCHED_BATCH ISO SCHED_ISO IDL SCHED_IDLE DLN SCHED_DEADLINE ? unknown value |
| cmd | CMD | see args . (alias args , command). |
| comm | COMMAND | command name (only the executable name). The output in this column may contain spaces. (alias ucmd , ucomm). See also the args format keyword, the -f option, and the c option. When specified last, this column will extend to the edge of the display. If ps can not determine display width, as when output is redirected (piped) into a file or another command, the output width is undefined (it may be 80, unlimited, determined by the TERM variable, and so on). The COLUMNS environment variable or --cols option may be used to exactly determine the width in this case. The w or -w option may be also be used to adjust width. |



| | | |
|-----------------|---------|---|
| command | COMMAND | See args . (alias args , command). |
| cp | CP | per-mill (tenths of a percent) CPU usage. (see %cpu). |
| cputime | TIME | cumulative CPU time, "[DD-]hh:mm:ss" format. (alias time). |
| cputimes | TIME | cumulative CPU time in seconds (alias times). |
| cuc | %CUC | The CPU utilization of a process, including dead children, in an extended "##.###" format. (see also %cpu , c , cp , cuu , pcpu). |
| cuu | %CUU | The CPU utilization of a process in an extended "##.###" format. (see also %cpu , c , cp , cuc , pcpu). |
| drs | DRS | data resident set size, the amount of private memory <i>reserved</i> by a process. It is also known as DATA. Such memory may not yet be mapped to rss but will always be included included in the vsz amount. |
| egid | EGID | effective group ID number of the process as a decimal integer. (alias gid). |
| egroup | EGROUP | effective group ID of the process. This will be the textual group ID, if it can be obtained and the field width permits, or a decimal representation otherwise. (alias group). |
| eip | EIP | instruction pointer. As of kernel 4.9.xx will be zeroed out unless task is exiting or being core dumped. |
| esp | ESP | stack pointer. As of kernel 4.9.xx will be zeroed out unless task is exiting or being core dumped. |
| etime | ELAPSED | elapsed time since the process was started, in the form [[DD-]hh:]mm:ss. |
| etimes | ELAPSED | elapsed time since the process was started, in seconds. |
| environ | ENVIRON | T{ environment variables for the process. T} |
| euid | EUID | effective user ID (alias uid). |
| euser | EUSER | effective user name. This will be the textual user ID, if it can be obtained and the field width permits, or a decimal representation otherwise. The n option can be used to force the decimal |



| | | |
|-----------------|---------|--|
| | | representation. (alias uname , user). |
| exe | EXE | path to the executable. Useful if path cannot be printed via cmd , comm or args format options. |
| f | F | flags associated with the process, see the PROCESS FLAGS section. (alias flag , flags). |
| fgid | FGID | filesystem access group ID. (alias fsgid). |
| fgroup | FGROUP | filesystem access group ID. This will be the textual group ID, if it can be obtained and the field width permits, or a decimal representation otherwise. (alias fsgroup). |
| flag | F | see f . (alias f , flags). |
| flags | F | see f . (alias f , flag). |
| fname | COMMAND | first 8 bytes of the base name of the process's executable file. The output in this column may contain spaces. |
| fuid | FUID | filesystem access user ID. (alias fsuid). |
| fuser | FUSER | filesystem access user ID. This will be the textual user ID, if it can be obtained and the field width permits, or a decimal representation otherwise. |
| gid | GID | see egid . (alias egid). |
| group | GROUP | see egroup . (alias egroup). |
| ignored | IGNORED | mask of the ignored signals, see signal(7) . According to the width of the field, a 32 or 64 bits mask in hexadecimal format is displayed, unless the --signames option is used. (alias sig_ignore , sigignore). |
| ipcns | IPCNS | Unique inode number describing the namespace the process belongs to. See namespaces(7) . |
| label | LABEL | security label, most commonly used for SELinux context data. This is for the <i>Mandatory Access Control</i> ("MAC") found on high-security systems. |
| lstart | STARTED | time the command started. This will be in the form "DDD mmm HH:MM:SS YYYY" unless changed by the -D option. |
| lsession | SESSION | displays the login session identifier of a process, if systemd support has been included. |



| | | |
|----------------|---------|---|
| luid | LUID | displays Login ID associated with a process. |
| lwp | LWP | light weight process (thread) ID of the dispatchable entity (alias spid , tid). See tid for additional information. |
| lxc | LXC | The name of the lxc container within which a task is running. If a process is not running inside a container, a dash ('-') will be shown. |
| machine | MACHINE | displays the machine name for processes assigned to VM or container, if systemd support has been included. |
| majflt | MAJFLT | The number of major page faults that have occurred with this process. |
| minflt | MINFLT | The number of minor page faults that have occurred with this process. |
| mntns | MNTNS | Unique inode number describing the namespace the process belongs to. See namespaces(7) . |
| netns | NETNS | Unique inode number describing the namespace the process belongs to. See namespaces(7) . |
| ni | NI | nice value. This ranges from 19 (nicest) to -20 (not nice to others), see nice(1) . (alias nice). |
| nice | NI | see ni .(alias ni). |
| nlwp | NLWP | number of lwps (threads) in the process. (alias thcount). |
| numa | NUMA | The node associated with the most recently used processor. A -1 means that NUMA information is unavailable. |
| nwchan | WCHAN | address of the kernel function where the process is sleeping (use wchan if you want the kernel function name). |
| oom | OOM | Out of Memory Score. The value, ranging from 0 to +1000, used to select task(s) to kill when memory is exhausted. |
| oomadj | OOMADJ | Out of Memory Adjustment Factor. The value is added to the current out of memory score which is then used to determine which task to kill when memory is exhausted. |
| oid | OWNER | displays the Unix user identifier of the owner of the session of a process, if systemd support has been included. |



| | | |
|----------------|---------|---|
| pcpu | %CPU | see %cpu . (alias %cpu). |
| pending | PENDING | mask of the pending signals. See signal(7) . Signals pending on the process are distinct from signals pending on individual threads. Use the m option or the -m option to see both. According to the width of the field, a 32 or 64 bits mask in hexadecimal format is displayed, unless the --signames option is used. (alias sig). |
| pgid | PGID | process group ID or, equivalently, the process ID of the process group leader. (alias pgrp). |
| pgrp | PGRP | see pgid . (alias pgid). |
| pid | PID | a number representing the process ID (alias tgid). |
| pidns | PIDNS | Unique inode number describing the namespace the process belongs to. See namespaces(7) . |
| pmem | %MEM | see %mem . (alias %mem). |
| policy | POL | scheduling class of the process. (alias class , cls). Possible values are: <ul style="list-style-type: none"> - not reported TS SCHED_OTHER FF SCHED_FIFO RR SCHED_RR B SCHED_BATCH ISO SCHED_ISO IDL SCHED_IDLE DLN SCHED_DEADLINE ? unknown value |
| ppid | PPID | parent process ID. |
| pri | PRI | priority of the process. Higher number means higher priority. |
| psr | PSR | processor that process last executed on. |
| pss | PSS | Proportional share size, the non-swapped physical memory, with shared memory proportionally accounted to all tasks mapping it. |
| rbytes | RBYTES | Number of bytes which this process really did cause to be fetched from the storage layer. |
| rchars | RCHARS | Number of bytes which this task has caused to be read from storage. |



| | | |
|---------------|--------|---|
| rgid | RGID | real group ID. |
| rgroup | RGROUP | real group name. This will be the textual group ID, if it can be obtained and the field width permits, or a decimal representation otherwise. |
| rops | ROPS | Number of read I/O operations—that is, system calls such as <code>read(2)</code> and <code>pread(2)</code> . |
| rss | RSS | resident set size, the non-swapped physical memory that a task has used (in kilobytes). (alias rssize , rsz). |
| rssize | RSS | see rss . (alias rss , rsz). |
| rsz | RSZ | see rss . (alias rss , rssize). |
| rtprio | RTPRIO | realtime priority. |
| ruid | RUID | real user ID. |
| ruser | RUSER | real user ID. This will be the textual user ID, if it can be obtained and the field width permits, or a decimal representation otherwise. |
| s | S | minimal state display (one character). See section PROCESS STATE CODES for the different values. See also stat if you want additional information displayed. (alias state). |
| sched | SCH | scheduling policy of the process. The policies <code>SCHED_OTHER</code> (<code>SCHED_NORMAL</code>), <code>SCHED_FIFO</code> , <code>SCHED_RR</code> , <code>SCHED_BATCH</code> , <code>SCHED_ISO</code> , <code>SCHED_IDLE</code> and <code>SCHED_DEADLINE</code> are respectively displayed as <code>0</code> , <code>1</code> , <code>2</code> , <code>3</code> , <code>4</code> , <code>5</code> and <code>6</code> . |
| seat | SEAT | displays the identifier associated with all hardware devices assigned to a specific workplace, if systemd support has been included. |
| sess | SESS | session ID or, equivalently, the process ID of the session leader. (alias session , sid). |
| sgi_p | P | processor that the process is currently executing on. Displays "*" if the process is not currently running or runnable. |
| sgid | SGID | saved group ID. (alias svgid). |
| sgroup | SGROUP | saved group name. This will be the textual group ID, if it can be obtained and the field width permits, or a decimal representation otherwise. |



| | | |
|-------------------|---------|---|
| sid | SID | see sess. (alias sess , session). |
| sig | PENDING | see pending. (alias pending , sig_pend). |
| sigcatch | CAUGHT | see caught. (alias caught , sig_catch). |
| sigignore | IGNORED | see ignored. (alias ignored , sig_ignore). |
| sigmask | BLOCKED | see blocked. (alias blocked , sig_block). |
| size | SIZE | approximate amount of swap space that would be required if the process were to dirty all writable pages and then be swapped out. This number is very rough! |
| slice | SLICE | displays the slice unit which a process belongs to, if systemd support has been included. |
| spid | SPID | see lwp. (alias lwp , tid). |
| stackp | STACKP | address of the bottom (start) of stack for the process. |
| start | STARTED | time the command started. If the process was started less than 24 hours ago, the output format is "HH:MM:SS", else it is " Mmm dd" (where Mmm is a three-letter month name). See also bsdstart , start , start_time , and stime . |
| start_time | START | starting time or date of the process. Only the year will be displayed if the process was not started the same year ps was invoked, or "MmmDD" if it was not started the same day, or "HH:MM" otherwise. See also bsdstart , start , lstart , and stime . |
| stat | STAT | multi-character process state. See section PROCESS STATE CODES for the different values meaning. See also s and state if you just want the first character displayed. |
| state | S | see s. (alias s). |
| stime | STIME | see start_time. (alias start_time). |
| suid | SUID | saved user ID. (alias svuid). |
| supgid | SUPGID | group ids of supplementary groups, if any. See getgroups(2) . |
| supgrp | SUPGRP | group names of supplementary groups, if any. See getgroups(2) . |
| suser | SUSER | saved user name. This will be the textual user ID, if it can be obtained and the |



| | | |
|----------------|---------|--|
| | | field width permits, or a decimal representation otherwise. (alias svuser). |
| svgid | SVGID | see sgid . (alias sgid). |
| svuid | SVUID | see suid . (alias suid). |
| sz | SZ | size in physical pages of the core image of the process. This includes text, data, and stack space. Device mappings are currently excluded; this is subject to change. See vsz and rss . |
| tgid | TGID | a number representing the thread group to which a task belongs (alias pid). It is the process ID of the thread group leader. |
| thcount | THCNT | see nlwp . (alias nlwp). number of kernel threads owned by the process. |
| tid | TID | the unique number representing a dispatchable entity (alias spid , tid). This value may also appear as: a process ID (pid); a process group ID (pgrp); a session ID for the session leader (sid); a thread group ID for the thread group leader (tgid); and a tty process group ID for the process group leader (tpgid). |
| time | TIME | cumulative CPU time, "[DD-]HH:MM:SS" format. (alias cpuetime). |
| timens | TIMENS | Unique inode number describing the namespace the process belongs to. See namespaces(7) . |
| times | TIME | cumulative CPU time in seconds (alias cputimes). |
| tname | TTY | controlling tty (terminal). (alias tt , tty). |
| tpgid | TPGID | ID of the foreground process group on the tty (terminal) that the process is connected to, or -1 if the process is not connected to a tty. |
| trs | TRS | text resident set size, the amount of physical memory devoted to executable code. |
| tt | TT | controlling tty (terminal). (alias tname , tty). |
| tty | TT | controlling tty (terminal). (alias tname , tt). |
| ucmd | CMD | see comm . (alias comm , ucomm). |
| ucomm | COMMAND | see comm . (alias comm , ucmd). |



| | | |
|----------------|---------|--|
| uid | UID | see eid . (alias eid). |
| uname | USER | see esuser . (alias esuser , user). |
| unit | UNIT | displays unit which a process belongs to, if systemd support has been included. |
| user | USER | see esuser . (alias esuser , uname). |
| userns | USERNS | Unique inode number describing the namespace the process belongs to. See namespaces(7) . |
| uss | USS | Unique set size, the non-swapped physical memory, which is not shared with an another task. |
| utsns | UTSNS | Unique inode number describing the namespace the process belongs to. See namespaces(7) . |
| uunit | UUNIT | displays user unit which a process belongs to, if systemd support has been included. |
| vsz | VSZ | virtual memory size of the process in KiB (1024-byte units). Device mappings are currently excluded; this is subject to change. (alias vsz). |
| wbytes | WBYTES | Number of bytes which this process caused to be sent to the storage layer. |
| wcbytes | WCBYTES | Number of cancelled write bytes. |
| wchan | WCHAN | name of the kernel function in which the process is sleeping. |
| wchars | WCHARS | Number of bytes which this task has caused, or shall cause to be written to disk. |
| wops | WOPS | Number of write I/O operations—that is, system calls such as write(2) and pwrite(2) . |

ENVIRONMENT VARIABLES [top](#)

The following environment variables could affect **ps**:

COLUMNS

Override default display width.

LINES

Override default display height.

PS_PERSONALITY



Set to one of `posix`, `old`, `linux`, `bsd`, `sun`, `digital`... (see section **PERSONALITY** below).

CMD_ENV

Set to one of `posix`, `old`, `linux`, `bsd`, `sun`, `digital`... (see section **PERSONALITY** below).

I_WANT_A_BROKEN_PS

Force obsolete command line interpretation.

LC_TIME

Date format.

LIBPROC_HIDE_KERNEL

Set this to any value to hide kernel threads normally displayed with the `-e` option. This is equivalent to selecting `--ppid 2 -p 2 --deselect` instead. Also works in BSD mode.

PS_COLORS

Not currently supported.

PS_FORMAT

Default output format override. You may set this to a format string of the type used for the `-o` option. The **DefSysV** and **DefBSD** values are particularly useful.

POSIXLY_CORRECT

Don't find excuses to ignore bad "features".

POSIX2

When set to "on", acts as **POSIXLY_CORRECT**.

UNIX95

Don't find excuses to ignore bad "features".

_XPG

Cancel `CMD_ENV=irix` non-standard behavior.

In general, it is a bad idea to set these variables. The one exception is `CMD_ENV` or `PS_PERSONALITY`, which could be set to `linux` for normal systems. Without that setting, `ps` follows the useless and bad parts of the Unix98 standard.

PERSONALITY

[top](#)

| | |
|---------|--|
| 390 | like the OS/390 OpenEdition ps |
| aix | like AIX ps |
| bsd | like FreeBSD ps (totally non-standard) |
| compaq | like Digital Unix ps |
| debian | like the old Debian ps |
| digital | like Tru64 (was Digital Unix, was OSF/1) ps |
| gnu | like the old Debian ps |
| hp | like HP-UX ps |
| hpux | like HP-UX ps |
| irix | like Irix ps |
| linux | ***** recommended ***** |
| old | like the original Linux ps (totally non-standard) |
| os390 | like OS/390 Open Edition ps |



| | |
|----------|---|
| posix | standard |
| s390 | like OS/390 Open Edition ps |
| sco | like SCO ps |
| sgi | like Irix ps |
| solaris2 | like Solaris 2+ (SunOS 5) ps |
| sunos4 | like SunOS 4 (Solaris 1) ps (totally non-standard) |
| svr4 | standard |
| sysv | standard |
| tru64 | like Tru64 (was Digital Unix, was OSF/1) ps |
| unix | standard |
| unix95 | standard |
| unix98 | standard |

BUGS [top](#)

The fields **bsdstart** and **start** will only show the abbreviated month name in English. The fields **lstart** and **stime** will show the abbreviated month name in the configured locale but may exceed the column width due to the different lengths for abbreviated month and day names across languages.

SEE ALSO [top](#)

[pgrep\(1\)](#), [pstree\(1\)](#), [top\(1\)](#), [strftime\(3\)](#), [proc\(5\)](#).

STANDARDS [top](#)

This **ps** conforms to:

- 1 Version 2 of the Single Unix Specification
- 2 The Open Group Technical Standard Base Specifications, Issue 6
- 3 IEEE Std 1003.1, 2004 Edition
- 4 X/Open System Interfaces Extension [UP XSI]
- 5 ISO/IEC 9945:2003

AUTHOR [top](#)

ps was originally written by Branko Lankester (lankeste@fwi.uva.nl). Michael K. Johnson (johnsonm@redhat.com) re-wrote it significantly to use the `proc` filesystem, changing a few things in the process. Michael Shields (mjshield@nyx.cs.du.edu) added the `pid-list` feature. Charles Blake (cblake@bbn.com) added multi-level sorting, the `dirent`-style library, the device name-to-number `mmaped` database, the approximate binary search directly on `System.map`, and many code and documentation cleanups. David Mossberger-Tang wrote the generic BFD support for `psupdate`. Albert Cahalan (albert@users.sf.net) rewrote `ps` for full Unix98 and BSD support, along with some ugly hacks for obsolete and foreign syntax.

Please send bug reports to (procps@freelists.org). No subscription is required or suggested.

COLOPHON [top](#)

This page is part of the *procps-ng* (/proc filesystem utilities) project. Information about the project can be found at <https://gitlab.com/procps-ng/procps>. If you have a bug report for this manual page, see <https://gitlab.com/procps-ng/procps/blob/master/Documentation/bugs.md>. This page was obtained from the project's upstream Git repository <https://gitlab.com/procps-ng/procps.git> on 2023-12-22. (At that time, the date of the most recent commit that was found in the repository was 2023-10-16.) If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

procps-ng**2023-10-04****PS(1)**

Pages that refer to this page: [free\(1\)](#), [fuser\(1\)](#), [htop\(1\)](#), [killall\(1\)](#), [pcp-ps\(1\)](#), [pgrep\(1\)](#), [pidstat\(1\)](#), [pmap\(1\)](#), [pmsleep\(1\)](#), [pslog\(1\)](#), [pstree\(1\)](#), [pwdx\(1\)](#), [slabtop\(1\)](#), [systemd\(1\)](#), [systemd-cgls\(1\)](#), [systemd-firstboot\(1\)](#), [systemd-nspawn\(1\)](#), [tcpdump\(1\)](#), [tload\(1\)](#), [top\(1\)](#), [uptime\(1\)](#), [w\(1\)](#), [proc\(5\)](#), [credentials\(7\)](#), [pid_namespaces\(7\)](#), [pthreads\(7\)](#), [sched\(7\)](#), [lsof\(8\)](#), [systemd-machined.service\(8\)](#), [tcpdump\(8\)](#), [vmstat\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



Materials for Topic 7: Processes

Full C Programs

- [child_creation_example.c](#) - a C program that forks, executes, and waits upon a child process. It demonstrates the use of the `getpid()`, `getppid()`, `fork()`, `execlp()`, `waitpid()`, `perror()`, `fprintf()`, `atexit()`, and `exit()` syscalls/functions.
- [wait_syscall_example.c](#) - a C program that demonstrates how the `wait()` system call is issued on a running child process.
- [system_syscall_implementation.c](#) - a C program that implements the `system()` system call and attempts to use it to run the command whose name is passed as the command-line argument to `./system_syscall_implementation`.
For example, `./system_syscall_implementation "ls -al"` will cause the `ls -al` command to run in the terminal.
- [daemon_creation.c](#) - a C program that creates a daemon process and lets it sleep for 30 seconds. You can confirm that this daemon is running by typing: `ps -xj` in the terminal and seeing that the program `./daemon_creation` has `init` with the process id of 1 as its parent (`PPID = 1`) and that it is not connected to any terminal (`TTY = ?`).

Runnable Linux Commands

Quick Links:

- [gcc](#)
- [././short_prompt](#)
- [././long_prompt](#)
- [vi](#)
- [/bin/sh](#)
- [passwd](#)
- [cat myFile.txt | grep Hello | sort](#)
- [ps -xj](#)

- The command:

```
gcc -Wall -Wextra -O2 -g -o program program.c
```

compiles the C source code located inside the file `program.c`. See more details [here](#).

- The command:

```
././short_prompt
```

executes code inside a file named `.short_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).

- The command:

```
././long_prompt
```

executes code inside a file named `.long_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).



- The command:

```
vi /home/text.txt
```

opens the file at `/home/text.txt` inside the `vi` Linux text editor.

- The command:

```
/bin/sh -c someCommand
```

runs an instance of the `sh` shell interpreter and, within it, executes the Linux command provided by `someCommand`. After the command returns, the `sh` shell interpreter returns (terminates) as well, which returns control to the shell interpreter that was working in the terminal before this command was called (e.g., `bash`.) For instance, typing:

```
/bin/sh -c ls -al
```

 will run the `ls -al` listing command within that new instance of `sh`.

- The command:

```
passwd
```

will prompt you to change the password for your Linux user account. You will first type your current password, re-type it, and then type the new password. The `root` privileged user can also change the passwords of other users by typing: `passwd userName`, where `userName` is the username of one of the device's/machine's users.

- The command:

```
cat myFile.txt | grep Hello | sort
```

alphabetically sorts the search results of the word "Hello" found in the file `myFile.txt`. The way this command works,

just as `ls -al | less` does (see more about the `less` content viewer [here](#)), is by piping the output of one command as the input into the next command. That is, we 'project' the content of `myFile.txt` into the pattern-searching command `grep`, which then projects its search results into the `sort` command that sorts that input.

- The command:

```
ps -xj
```

outputs the list of all the processes running by your user, including daemons. Daemon processes will have the parent process ID (`ppid`) of 1 and will have a '?' under the TTY column, meaning that they aren't connected to any terminals (daemons are background processes that don't show their output in any terminal,) which is the definition of a daemon.



This website by [Miriam Briskman](#) is licensed under [CC BY-NC 4.0](#).

```
1  /* A C program that forks, executes, and waits upon
2  *   a child process. It demonstrates the use of
3  *   the getpid(), getppid(), fork(), execlp(),
4  *   waitpid(), perror(), fprintf(), atexit(), and
5  *   exit() syscalls/functions.
6  *
7  *   Miriam Briskman, 3/15/2023
8  *   CISC 3350, Brooklyn College
9  *   Licensed under CC BY-NC 4.0
10 */
11
12 #include <sys/types.h> // pid_t, gid_t, etc.
13 #include <unistd.h>    // Defines some system calls.
14 #include <sys/wait.h> // wait(), waitpid(), etc.
15 #include <stdio.h>    // perror(), printf(), fprintf().
16 #include <stdlib.h>   // EXIT_SUCCESS, EXIT_FAILURE.
17 #include <inttypes.h> // intmax_t.
18
19 // Definition of a function that we call
20 // during a process's termination:
21 void bye (void)
22 {
23     printf ("Parent: Signing off!\n");
24 }
25
26 // Definition of another function we call
27 // during a process's termination:
28 void out (void)
29 {
30     printf ("Parent: atexit() succeeded!\n");
31 }
32
33 // Follow-up question:
34 // Why don't we need to include a copy of
35 // the bye() and out() function declarations
36 // here?
37 // Hint: what is the difference between the
38 // bye() and out() functions here and the
39 // functions that we created in 'functions.c'
40 // that we covered several lectures ago?
41
42 int main ()
```




```
43 {
44     printf ("Parent: started executing now!\n");
45
46     // Fork (create) a child process:
47     pid_t childpid = fork();
48     if (childpid == -1)
49     {
50         perror ("fork");
51         exit (EXIT_FAILURE);
52     }
53
54     // We enter this 'if' only inside the parent
55     // process:
56     if (childpid)
57     {
58         // The function out() and then the function
59         // bye() will execute automatically right
60         // before the parent terminates. The
61         // execution of the functions that atexit()
62         // registers is in the reversed order of
63         // the registration (we first register
64         // bye() and then out()), but they are
65         // called in the opposite order: out()
66         // followed by bye().
67         if (atexit (bye) != 0)
68         {
69             fprintf (stderr,
70                     "atexit(): can't register bye().\n");
71             exit (EXIT_FAILURE);
72         }
73         if (atexit (out) != 0)
74         {
75             fprintf (stderr,
76                     "atexit(): can't register out().\n");
77             exit (EXIT_FAILURE);
78         }
79
80         printf ("Parent: My pid = %jd.\n",
81                (intmax_t) getpid ());
82         printf ("Parent: I just forked a child! "
83                "I'll wait for it to terminate.\n");
84
85         int status;
```



```
86
87 // Wait for the child with ID of 'childpid'
88 //   to return before continuing the
89 //   parent's execution. The parent will
90 //   pause its execution on the CPU while
91 //   waiting for the child, and the
92 //   waitpid() call will only return when
93 //   the child finished working.
94 pid_t temp = waitpid (childpid, &status, 0);
95 if (temp == -1)
96 {
97     perror ("waitpid");
98     // out() and bye() might be called here:
99     exit (EXIT_FAILURE);
100 }
101
102 printf ("Parent: The child that just returned "
103        "has the pid = %d.\n",
104        temp);
105
106 // Let's check what happened with the child:
107 if (WIFEXITED (status))
108     printf ("Parent: It terminated normally "
109            "with the exit status = %d.\n",
110            WEXITSTATUS (status));
111 if (WIFSIGNALED (status))
112     printf ("Parent: It was killed by the "
113            "signal = %d%s.\n",
114            WTERMSIG (status),
115            WCOREDUMP (status) ? " (dumped core)" : "");
116 if (WIFSTOPPED (status))
117     printf ("Parent: It was stopped by the "
118            "signal = %d.\n",
119            WSTOPSIG (status));
120 if (WIFCONTINUED (status))
121     printf ("Parent: Its execution "
122            "continued.\n");
123 }
124 // We enter this 'else' only inside the child
125 //   process:
126 else // Inside the child, 'childpid' is always 0.
127 {
128     printf ("Child: My pid = %jd.\n",
```



```
129     (intmax_t) getpid ());
130     printf ("Child: Parent's pid = %jd.\n",
131           (intmax_t) getppid ());
132
133     printf ("Child: I'll call execlp() now and "
134           "replace myself with the 'ls -al' "
135           "command:\n");
136
137     // Execute the command/program: ls -al as the
138     //   child!
139     if (execlp ("ls", "ls", "-al", NULL) == -1)
140     {
141         perror ("execlp");
142         exit (EXIT_FAILURE);
143     }
144
145     // When execlp() executes, the child's
146     //   process will be replaced completely
147     //   with the program "ls -al".
148 }
149
150 // Only the parent calls out() and bye() here:
151 return EXIT_SUCCESS;
152 }
153
```



```
1 // A program demonstrating the call to wait().
2 //
3 // This program is taken from Linux System Programming:
4 //     Talking Directly to the Kernel and C Library,
5 //     2nd Edition, by Love. ISBN: 978-1-44933953-1,
6 //     page 153.
7
8 #include <unistd.h>
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <sys/types.h>
12 #include <sys/wait.h>
13
14 int main (void)
15 {
16     int status;
17     pid_t pid;
18
19     // The statement below (1) forks a child process,
20     // (2) terminates it right away, and (3) makes
21     // the child return with the return value of 1.
22     // The reason that the child (and not the parent) is
23     // terminated is that the 'if' statement asks
24     // whether the returned value from fork() is 0,
25     // which is only true inside the child's process
26     // (since, inside the parent, fork() returns the
27     // pid of the child which is never 0,) so only
28     // the child enters this 'if' statement, which
29     // causes it execute 'exit (EXIT_FAILURE)'.
30     if (!fork ())
31     {
32         perror ("fork");
33         exit (EXIT_FAILURE);
34     }
35
36     // The wait() system call waits for any of this
37     // process's children to return. Since this
38     // process created only 1 child, we will
39     // necessarily wait for this one child.
40     if ((pid = wait (&status)) == -1)
41     {
42         perror ("wait");
```



```
43     exit (EXIT_FAILURE);
44 }
45
46 printf ("pid = %d\n", pid);
47
48 // The following 'if' statements are commonly used
49 // to check the status of the returned child.
50 // Each of the functions WEXITSTATUS(),
51 // WTERMSIG(), etc., will look into the 'status'
52 // integer to find whether the bits of 'status'
53 // indicate that the child returned in some
54 // either nominal or abnormal way.
55 // It is advised that you include these statements
56 // after any instance of calling wait(),
57 // waitpid(), etc., to find the details about
58 // the return of every child.
59 if (WIFEXITED (status))
60     printf ("Normal termination with the exit "
61            "status = %d.\n",
62            WEXITSTATUS (status));
63 if (WIFSIGNALED (status))
64     printf ("Killed by the signal = %d%s.\n",
65            WTERMSIG (status),
66            WCOREDUMP (status) ? " (dumped core)" : "");
67 if (WIFSTOPPED (status))
68     printf ("Stopped by the signal = %d.\n",
69            WSTOPSIG (status));
70 if (WIFCONTINUED (status))
71     printf ("Continued.\n");
72
73 return EXIT_SUCCESS;
74 }
75
```



```
1  /* A C program that implements the system() system
2  *   call and attempts to use it to run the
3  *   command whose name is passed as the
4  *   command-line argument to the program.
5  *
6  *
7  *   Miriam Briskman, 3/15/2023
8  *   CISC 3350, Brooklyn College
9  *   Licensed under CC BY-NC 4.0
10 */
11
12 #include <sys/types.h> // pid_t, gid_t, etc.
13 #include <unistd.h>    // Defines some system calls.
14 #include <sys/wait.h> // wait(), waitpid(), etc.
15 #include <stdio.h>    // perror(), printf(), fprintf().
16 #include <stdlib.h>   // EXIT_SUCCESS, EXIT_FAILURE.
17
18 /*
19 *   my_system - synchronously spawns and waits for
20 *   the command "/bin/sh -c <cmd>".
21 *
22 *   Returns -1 on error of any sort, or the exit
23 *   code from the launched process. Does not block
24 *   or ignore any signals.
25 *
26 *   This function is taken from Linux System Programming:
27 *   Talking Directly to the Kernel and C Library,
28 *   2nd Edition, by Love. ISBN: 978-1-44933953-1,
29 *   pages 55-56.
30 */
31 int my_system (char * const cmd)
32 {
33     int status;
34     pid_t pid;
35
36     pid = fork ();
37     if (pid == -1)
38         return -1;
39
40     else if (pid == 0)
41     {
42         char * argv[4];
```



```
43     argv[0] = "sh";
44     argv[1] = "-c";
45     argv[2] = cmd;
46     argv[3] = NULL;
47     execv ("/bin/sh", argv);
48     exit (-1);
49 }
50
51 if (waitpid (pid, &status, 0) == -1)
52     return -1;
53 else if (WIFEXITED (status))
54     return WEXITSTATUS (status);
55
56 return -1;
57 }
58
59 int main (int argc, char * argv[])
60 {
61     if (argc == 1)
62     {
63         fprintf (stderr,
64                 "Usage:\n\t%s someCommand\n"
65                 "where someCommand is replaced by "
66                 "some Linux command.\n",
67                 argv[0]);
68         return EXIT_FAILURE;
69     }
70
71     // Executing the command that was passed as
72     // the command-line argument argv[1]:
73     int ret = my_system (argv[1]);
74
75     printf ("The command returned the "
76            "status: %d.\n",
77            ret);
78
79     return EXIT_SUCCESS;
80 }
81
```



```
1 // A program that shows how a daemon process is
2 //   created in Linux.
3 //
4 // This program is taken from Linux System Programming:
5 //   Talking Directly to the Kernel and C Library,
6 //   2nd Edition, by Love. ISBN: 978-1-44933953-1,
7 //   pages 173-174.
8
9 #include <sys/types.h>
10 #include <sys/stat.h>
11 #include <stdlib.h>
12 #include <stdio.h>
13 #include <fcntl.h>
14 #include <unistd.h>
15
16 int main (void)
17 {
18     pid_t pid;
19     int i;
20
21     // Create new process:
22     pid = fork ();
23     if (pid == -1)
24         return -1;
25     else if (pid != 0)
26         exit (EXIT_SUCCESS);
27
28     // Create a new session and
29     //   a process group:
30     if (setsid () == -1)
31         return -1;
32
33     // Set the working directory to the
34     //   root directory:
35     if (chdir ("/") == -1)
36         return -1;
37
38     // OPEN_MAX contains the maximum possible
39     //   number of files that are allowed to
40     //   be opened by a single program in
41     //   Linux:
42     int OPEN_MAX = sysconf(_SC_OPEN_MAX);
```





```
43
44 // Close all open files:
45 for (i = 0; i < OPEN_MAX; i++)
46     close (i);
47
48 // Redirect fd's 0,1,2 (for stdin, stdout,
49 //     and stderr) to /dev/null. This
50 //     action disconnects the daemon from
51 //     terminal i/o:
52 open ("/dev/null", O_RDWR); /* stdin */
53 dup (0); /* stdout */
54 dup (0); /* stderr */
55
56 /* do its daemon thing... */
57
58 // Let's make the daemon sleep (exists without
59 //     doing anything) for 30 seconds before it
60 //     terminates:
61 sleep(30);
62
63 return EXIT_SUCCESS;
64 }
65
```



Topic 8: Threads

Note: A copy of each one of the sources listed below is included in this packet. To jump to the location of a source copy, click the page link under “Page”.

In case a source isn’t included (e.g., a YouTube video,) click the  (external link) symbol under “Page” to open the external source in a browser window.

| # | Citation & Source Link | Page |
|----|--|------|
| 1 | Morelli, R., and R. Walde. “Threads and Concurrent Programming: 14.1 Introduction.” <i>Saylor Academy</i> , Trinity College, 2020. URL: https://learn.saylor.org/mod/book/view.php?id=67488 | 948 |
| 2 | Multilingualjourney. “Linux Kernel Support for Threads (Light Weight Processes).” <i>Programming in Linux</i> , 3 Apr. 2012. URL: https://linuxprograms.wordpress.com/2007/12/19/linux-kernel-support-for-threads-light-weight-processe/ | 958 |
| 3 | Multilingualjourney. “Threads Programming in Linux: Examples.” <i>Programming in Linux</i> , 3 Apr. 2012. URL: https://linuxprograms.wordpress.com/2007/12/29/threads-programming-in-linux-examples/ | 962 |
| 4 | “pthread_create(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/pthread_create.3.html | 969 |
| 5 | “pthread_self(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/pthread_self.3.html | 978 |
| 6 | “pthread_equal(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/pthread_equal.3.html | 981 |
| 7 | “pthread_exit(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/pthread_exit.3.html | 984 |
| 8 | “pthread_cancel(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/pthread_cancel.3.html | 987 |
| 9 | “pthread_setcancelstate(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/pthread_setcancelstate.3.html | 992 |
| 10 | “pthread_join(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/pthread_join.3.html | 1001 |
| 11 | “pthread_detach(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/pthread_detach.3.html | 1005 |
| 12 | “pthread_mutex_lock(3p) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/pthread_mutex_lock.3p.html | 1008 |
| 13 | Briskman, Miriam. “Materials for Topic 8: Threads.” <i>Topic 8: Threads — CISC 3350 Materials</i> , 2023. URL: www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_08/ | 1015 |

| # | Citation & Source Link | Page |
|----|--|------|
| 14 | Briskman, Miriam. “unsynced_threads_experiment.c .” (C source code) 16 Apr. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_08/unsynced_threads_experiment.c | 1017 |
| 15 | Briskman, Miriam. “synced_threads_experiment.c .” (C source code) 16 Apr. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_08/synced_threads_experiment.c | 1020 |
| 16 | Briskman, Miriam. “unsynced_bank_withdraw.c .” (C source code) 16 Apr. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_08/unsynced_bank_withdraw.c | 1023 |
| 17 | Briskman, Miriam. “synced_bank_withdraw.c .” (C source code) 16 Apr. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_08/synced_bank_withdraw.c | 1027 |
| 18 | Briskman, Miriam. “threads_example.c .” (C source code) 16 Apr. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_08/threads_example.c | 1032 |

Threads and Concurrent Programming

14.1 Introduction

OBJECTIVES

After studying this chapter, you will

- Understand the concept of a thread.
- Know how to design and write multithreaded programs.
- Be able to use the Thread class and the Runnable interface.
- Understand the life cycle of a thread.
- Know how to synchronize threads.

OUTLINE

14.1 Introduction 1

4.2 What Is a Thread?

14.3 From the Java Library: java.lang.Thread

14.4 Thread States and Life Cycle

14.5 Using Threads to Improve Interface Responsiveness

14.6 Case Study: Cooperating Threads

14.7 Case Study: The Game of Pong Chapter Summary

Introduction

This chapter is about doing more than one thing at a time. Doing more than one thing at once is commonplace in our everyday lives. For example, let's say your breakfast today consists of cereal, toast, and a cup of java. You have to do three things at once to have breakfast: eat cereal, eat toast, and drink coffee.

Actually, you do these things "at the same time" by alternating among them: You take a spoonful of cereal, then a bite of toast, and then sip some coffee. Then you have another bite of toast, or another spoonful of cereal, more coffee, and so on, until breakfast is finished. If the phone rings while you're having breakfast, you will probably answer it—and continue to have breakfast, or at least to sip the coffee. This means you're doing even more "at the same time." Everyday life is full of examples where we do more than one task at the same time.

The computer programs we have written so far have performed one task at a time. But there are plenty of applications where a program needs to do several things at once, or **concurrently**. For example, if you wrote an Internet chat program, it would let several users take part in a discussion group. The program would have to read messages from several users at the same time and broadcast them to the other participants in the group. The reading and broadcasting tasks would have to take place concurrently. In Java, concurrent programming is handled by *threads*, the topic of this chapter.

Source: R. Morelli and R. Walde, Trinity College



This work is licensed under a [Creative Commons Attribution 4.0 License](https://creativecommons.org/licenses/by/4.0/).

Threads and Concurrent Programming

14.2 What Is a Thread?

A **thread** (or a *thread of execution* or a *thread of control*) is a single sequence of executable statements within a program. For Java applications, the flow of control begins at the first statement in `main()` and continues sequentially through the program statements. For Java applets, the flow of control begins with the first statement in `init()`. Loops within a program cause a certain block of statements to be repeated. If-else structures cause certain statements to be selected and others to be skipped. Method calls cause the flow of execution to jump to another part of the program, from which it returns after the method's statements are executed. Thus, within a single thread, you can trace the sequential flow of execution from one statement to the next.

One way to visualize a thread is to imagine that you could make a list of the program's statements as they are executed by the computer's central processing unit (CPU). Thus, for a particular execution of a program with loops, method calls, and selection statements, you could list each instruction that was executed, beginning at the first, and continuing until the program stopped, as a single sequence of executed statements. That's a thread!

Now imagine that we break a program up into two or more independent threads. Each thread will have its own sequence of instructions. Within a single thread, the statements are executed one after the other, as usual. However, by alternately executing the statements from one thread and another, the computer can run several threads *concurrently*. Even though the CPU executes one instruction at a time, it can run multiple threads concurrently by rapidly alternating among them. The main advantage of concurrency is that it allows the computer to do more than one task at a time. For example, the CPU could alternate between downloading an image from the Internet and running a spreadsheet calculation. This is the same way you ate toast and cereal and drank coffee in our earlier breakfast example. From our perspective, it might look as if the computer had several CPUs working in parallel, but that's just the illusion created by effectively scheduling threads.

 Annotation 2020-03-24 202645

Threads and Concurrent Programming

14.2 What Is a Thread?

Concurrent Execution of Threads

The technique of concurrently executing several tasks within a program is known as **multitasking**. A **task** in this sense is a computer operation of some sort, such as reading or saving a file, compiling a program, or displaying an image on the screen. Multitasking requires the use of a separate thread for each of the tasks. The methods available in the Java Thread class make it possible (and quite simple) to implement **multithreaded** programs.

Most computers, including personal computers, are *sequential* machines that consist of a single CPU, which is capable of executing one machine instruction at a time. In contrast, *parallel computers*, used primarily for large scale scientific and engineering applications, are made up of multiple CPUs working in tandem.

Today's personal computers, running at clock speeds over 1 gigahertz— 1 *gigahertz* equals 1 billion cycles per second—are capable of executing millions of machine instructions per second. Despite its great speed, however, a single CPU can process only one instruction at a time.

Each CPU uses a **fetch-execute cycle** to retrieve the next instruction from memory and execute it. Since CPUs can execute only one instruction at a time, multithreaded programs are made possible by dividing the CPU's time and sharing it among the threads. The CPU's schedule is managed by a **scheduling algorithm**, which is an algorithm that schedules threads for execution on the CPU. The choice of a scheduling algorithm depends on the platform on which the program is running. Thus, thread scheduling might be handled differently on Unix, Windows, and Macintosh systems.

One common scheduling technique is known as time slicing, in which each thread alternatively gets a slice of the CPU's time. For example, suppose we have a program that consists of two threads. Using this technique, the system would give each thread a small **quantum** of CPU time— say, one thousandth of a second (one *millisecond*)—to execute its instructions. When its quantum expires, the thread would be preempted and the other thread would be given a chance to run. The algorithm would then alternate in this **round-robin** fashion between one

thread and the other (Fig. 14.1). During each millisecond on a 300-megahertz CPU, a thread can execute 300,000 machine instructions. One **megahertz** equals 1 million cycles per second. Thus, within each second of real time, each thread will receive 500 time slices and will be able to execute something like 150 million machine instructions.

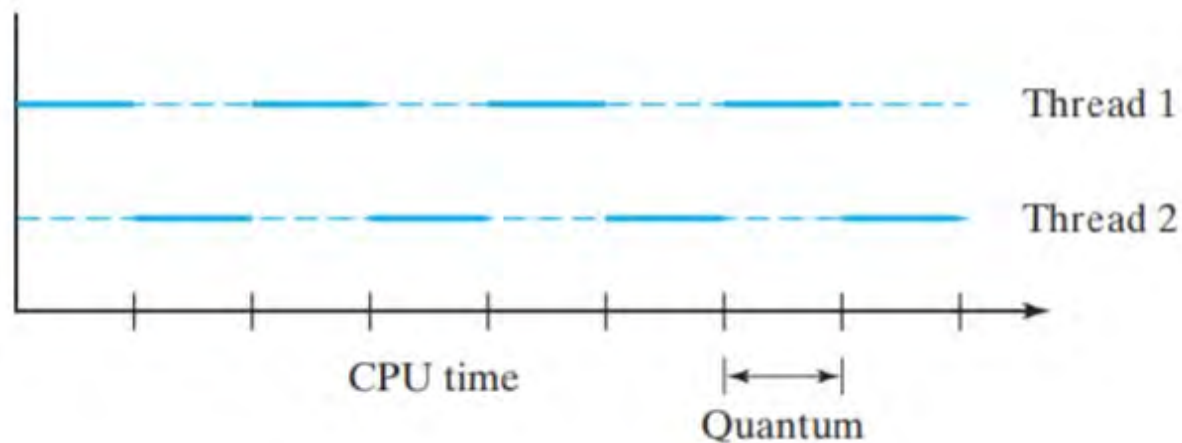


Figure 14.1: Each thread gets a slice of the CPU's time.

Under **priority scheduling**, threads of higher priority are allowed to run to completion before lower-priority threads are given a chance. An example of a high-priority thread would be one that is processing keyboard input or any other kind of interactive input from the user. If such tasks were given low priority, users would experience noticeable delays in their interaction, which would be quite unacceptable.

The only way a high-priority thread can be preempted is if a thread of still higher priority becomes available to run. In many cases, higher-priority threads are those that can complete their task within a few milliseconds, so they can be allowed to run to completion without starving the lower-priority threads. An example would be processing a user's keystroke, a task that can begin as soon as the key is struck and can be completed very quickly. Starvation occurs when one thread is repeatedly preempted by other threads.

JAVA LANGUAGE RULE **Thread Support.** Depending on the hardware platform, Java threads can be supported by assigning different threads to different processors, by time slicing a single processor, or by time slicing many hardware processors.

Threads and Concurrent Programming

14.2 What Is a Thread?

Multithreaded Numbers

Let's consider a simple example of a threaded program. Suppose we give every individual thread a unique ID number, and each time it runs, it prints its ID ten times. For example, when the thread with ID 1 runs the output produced would just be a sequence of ten 1's: 1111111111.

As shown in Figure 14.2, the `NumberThread` class is defined `+NumberThread(in n : int)` `+run()` `NumberThread` `+run()` `Thread` as a subclass of `Thread` and overrides the `run()` method. To set the thread's ID number, the constructor takes a single parameter that is use to set the thread's ID number. In the `run()` method, the thread simply executes a loop that prints its own number ten times:

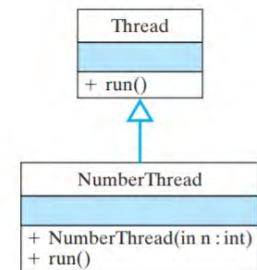


Figure 14.2: The `NumberThread` class overrides the inherited `run()` method.

```
public class NumberThread extends Thread {  
    int num;  
  
    public NumberThread(int n) {  
        num = n;  
    }  
    public void run() {  
        for (int k=0; k < 10; k++) {  
            System.out.print(num);  
        } // for  
    } // run()  
} // NumberThread
```

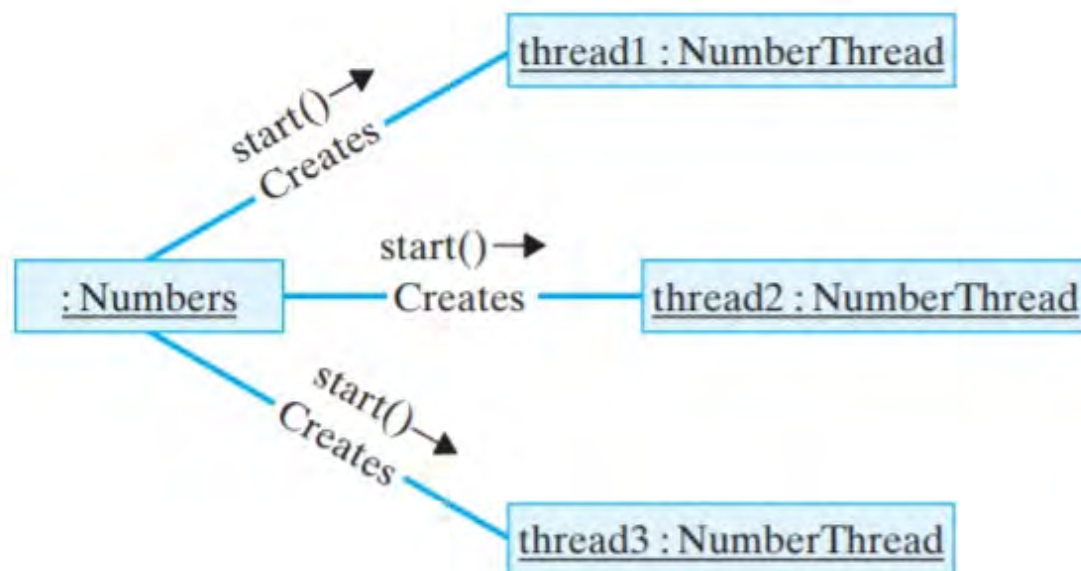


Figure 14.3: The Numbers object creates several instances of NumberThread and tells each one to start().

Now let's define another class whose task will be to create many `NumberThreads` and get them all running at the same time (Fig. 14.3). For each `NumberThread`, we want to call its constructor and then `start()` it:

```
public class Numbers {
    public static void main(String args[]) {
        // 5 threads
        NumberThread number1, number2, number3, number4, number5;

        // Create and start each thread
        number1 = new NumberThread(1); number1.start();
        number2 = new NumberThread(2); number2.start();
        number3 = new NumberThread(3); number3.start();
        number4 = new NumberThread(4); number4.start();
        number5 = new NumberThread(5); number5.start();
    } // main()
} // Numbers
```

When a thread is started by calling its `start()` method, it automatically calls its `run()` method. The output generated by this version of the `Numbers` application is as follows:

```
11111111112222222222333333333344444444445555555555
```

From this output, it appears that the individual threads were run in the order in which they were created. In this case, each thread was able to run to completion before the next thread started running.

What if we increase the number of iterations that each thread performs? Will each thread still run to completion? The following output was generated for 200 iterations per thread:

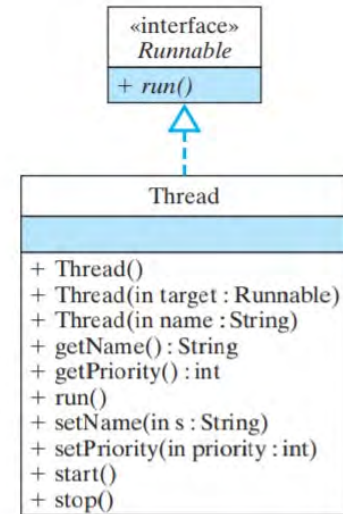


Figure 14.4: The `java.lang.Thread` class. NOTE: NEEDS REVISION TO ADD PRIORITY, `YIELD()` and `SLEEP()`.

Programming in Linux

Show me the code

Linux Kernel Support for Threads (Light Weight Processes)

Posted on [December 19, 2007](#)

Yes, Linux supports threads. Thread is also a context of execution like the processes. We can do programming with threads. The further discussion on this topic is with respect to the Linux 2.6 kernel. As of now, there is support for multiple threads in the Linux kernel and in the user space, we make utilize of the Threads library like POSIX threads. The way POSIX threads is implemented in Linux is different from other Operating Systems (Because they are no specific system calls for dealing with threads)

So the question arises if Linux does not have any system calls corresponding to threads creation, how are the POSIX threads created. And if POSIX thread implementation is done in the user space alone, there are many issues that has to be dealt with. Let's think that POSIX thread is implemented completely in the user space, so the scheduling of threads is to be done by the POSIX threads library itself.

Threads must be light weight in the sense that all the threads in the same process share the same address space. Implementing POSIX threads in the user space can solve this issue. Because threads are nothing but an abstraction to the programmer. He/she enjoys some of the benefits of threads. But if one of the threads is blocked on a particular system call like read, the whole process will be blocked because the very idea of the threads is transparent to the kernel. So all the benefits of threads can not be utilized.

So how is this issue solved in Linux? It is done by implementing the so called 'Light weight processes'. A `fork()` system call creates a new process. If the child process does not have any `execve()` like system calls, both the child and the parent process share the same address space for the text(program code). And the data address space is marked as 'Copy on write' that is in the beginning both the child and the parent process share the same data address space, but any attempt of changing the data by the child will result in creating a new data address space for the child.

Since now we got an idea about how the `fork()` system call works, we can now think about implementing threads. As we know that threads in the same process share the same address space for text and data, so we need not set any 'Copy on write'. The issues of data synchroniztion which comes up when two threads access the same data variable has to be worked upon by the programmer.

There is a system call called clone called `clone()` or `clone2()` which helps in creating a child process but unlike `fork()`, we have more control on deciding the behaviour of the child process whether we want to have the new child share the same filesystem information, file descriptor table, signal handler table or the memory space of the parent. In fact the threads library makes use of the `clone()` system call to create new threads.

So by this time, we got an idea that threads in Linux are nothing but processes or better to be called 'Light weight processes'(because of the sharing of data).

There is an advantage of this way of implementation. To the kernel everything is seen as processes and the scheduler has nothing to think about separate scheduling techniques for threads and processes. This makes the implementation as simple as possible and also solves the above blocking problem of the threads implementation in user space.

Let's check a small snippet of code

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void * function1(void *arg)
{
```

```
pthread_t tid=pthread_self();
printf("In thread %u and process %u\n",tid,getpid());
}

void * function2(void *arg)
{
pthread_t tid=pthread_self();
printf("In thread %u and process %u\n",tid,getpid());
}

int main()
{
void *status;
pthread_t tid1,tid2;
pthread_attr_t attr;

if(pthread_create(&tid1,NULL,function1,NULL)){
perror("Failure");
exit(1);
}

if(pthread_create(&tid2,NULL,function2,NULL)){
perror("Failure");
exit(2);
}

pthread_join(tid1,NULL);
pthread_join(tid2,NULL);
printf("In main thread %u and process %u\n",pthread_self(),getpid());
}
```


To compile this program

```
$gcc thread.c -lpthread
```

On Executing, the output is

```
In thread 3086625680 and process 5480  
In thread 3076135824 and process 5480  
In main thread 3086628544 and process 5480
```

How can this be possible? By our above discussion, we found that threads are nothing but light weight processes. But ***getpid()*** (which returns the Process ID) gives same PID for all the processes. This is because the POSIX standard says that the threads must return the same PID (based on the assumption that they all are running in the same process). To deal with this issue, Linux introduced a **tgid**(Thread group Leader ID) field. The **tgid** is same as the PID of the first light weight process in the threads group(Group of threads created by a process including itself). System calls like ***getpid()*** has been so designed that they return the **tgid** of the process instead of the PID, thus threads in the same thread group get the same value from ***getpid()***.

This entry was posted in [POSIX](#), [POSIX Threads](#), [Threads](#) and tagged [Kernel](#), [LWP](#), [pthread](#), [Threads](#) by [multilingualjourney](#). Bookmark the [permalink \[https://linuxprograms.wordpress.com/2007/12/19/linux-kernel-support-for-threads-light-weight-processe/\]](https://linuxprograms.wordpress.com/2007/12/19/linux-kernel-support-for-threads-light-weight-processe/) .

Programming in Linux

Show me the code

Threads Programming in Linux: Examples

Posted on [December 29, 2007](#)

One of the important purpose of threads is to achieve concurrency. There may be many independent tasks in a program which can be done in parallel without the influence of the other. One of the first step in using threads is to first recognize the fact whether the program needs threads or not, otherwise the very purpose of threads becomes futile. For example if you are designing a program which involves reading a file and display it, you may utilize two or more threads, one to read the file, the second to update the display and other threads to monitor the user inputs in the form of keyboard interrupts or mouse movements. There can be cases when threads are dependent on each other like in case of our above example, if the concerned application is an editor so a character key pressed should be soon informed to the other thread which updates display so that it may open the menu corresponding to that key shortcut or do some other appropriate action.

Example 1:

Two threads displaying two strings “Hello” and “How are you?” independent of each other.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

void * thread1()
{
    while(1){
        printf("Hello!!\n");
    }
}

void * thread2()
{
    while(1){
        printf("How are you?\n");
    }
}

int main()
{
    int status;
    pthread_t tid1,tid2;

    pthread_create(&tid1,NULL,thread1,NULL);
    pthread_create(&tid2,NULL,thread2,NULL);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    return 0;
}
```

Now compile this program (Note the -l option is to load the pthread library)

```
$gcc thread.c -lpthread
```

On running, you can see many interleaved “Hello!!” and “How are you?” messages

Example 2

This example involves a reader and a writer thread. The reader thread reads a string from the user and writer thread displays it.

This program uses semaphore so as to achieve synchronization

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>

char n[1024];
sem_t len;

void * read1()
{
    while(1){
        printf("Enter a string");
        scanf("%s",n);
        sem_post(&len);
    }
}

void * writel()
{
```

```
while(1) {
    sem_wait(&len);
    printf("The string entered is :");
    printf("==== %s\n",n);
}

}

int main()
{
    int status;
    pthread_t tr, tw;

    pthread_create(&tr,NULL,read1,NULL);
    pthread_create(&tw,NULL,writel,NULL);

    pthread_join(tr,NULL);
    pthread_join(tw,NULL);
    return 0;
}
```

On running, in most cases we may be able to achieve a serial read and write(Thread1reads a string and Thread2 displays the same string). But suppose we insert a sleep function() in write1 like

```
void * writel()
{
    while(1) {
        sleep(5);
        sem_wait(&len);
        printf("The string entered is :");
```

```
        printf("==== %s\n", n);  
    }  
}
```

The thread 1 may read one more string and thread2 displays the last read string. That is no serial read and write is achieved.

So we may need to use the condition variables to achieve serial read and write.

Example 3

This example involves a reader and a writer thread. The reader thread reads a string from the user and writer thread displays it. This program uses condition variables to achieve synchronization and achieve serial programming.

```
#include <stdio.h>  
#include <pthread.h>  
#include <semaphore.h>  
#include <stdlib.h>  
  
#define TRUE 1  
#define FALSE 0  
  
char n[1024];  
pthread_mutex_t lock= PTHREAD_MUTEX_INITIALIZER;  
int string_read=FALSE;  
  
pthread_cond_t cond;  
  
void * read1()
```

```
{
    while(1) {
        while(string_read);
        pthread_mutex_lock(&lock);
        printf("Enter a string: ");
        scanf("%s",n);
        string_read=TRUE;
        pthread_mutex_unlock(&lock);
        pthread_cond_signal(&cond);
    }
}

void * writel()
{
    while(1) {
        pthread_mutex_lock(&lock);
        while(!string_read)
            pthread_cond_wait(&cond,&lock);
        printf("The string entered is %s\n",n);

        string_read=FALSE;
        pthread_mutex_unlock(&lock);
    }
}

int main()
{
    int status;
    pthread_t tr, tw;

    pthread_create(&tr,NULL,read1,NULL);
    pthread_create(&tw,NULL,writel,NULL);
}
```

```
pthread_join(tr, NULL);  
pthread_join(tw, NULL);  
return 0;  
}
```

The output is serial read and write.

In the beginning, I started the discussion that threads are used to achieve concurrency. But the above examples can be easily done by simple scanf and printf i.e.,

```
scanf("%s",n);  
printf("%s",n);
```

But these examples were given to demonstrate the semaphores and condition variables. Example 3 can be further modified to design a reader/writer application. Example string_read boolean variable can be converted to a string_count variable.

This entry was posted in [POSIX](#), [POSIX Threads](#), [Threads](#) and tagged [POSIX Threads](#), [pthread](#) by [multilingualjourney](#). Bookmark the [permalink \[https://linuxprograms.wordpress.com/2007/12/29/threads-programming-in-linux-examples/\]](https://linuxprograms.wordpress.com/2007/12/29/threads-programming-in-linux-examples/) .

pthread_create(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [EXAMPLES](#) | [SEE ALSO](#)

pthread_create(3)

Library Functions Manual

pthread_create(3)

NAME [top](#)

pthread_create - create a new thread

LIBRARY [top](#)

POSIX threads library (*Libpthread*, *-Lpthread*)

SYNOPSIS [top](#)

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *restrict thread,  
                  const pthread_attr_t *restrict attr,  
                  void *(*start_routine)(void *),  
                  void *restrict arg);
```

DESCRIPTION [top](#)

The `pthread_create()` function starts a new thread in the calling process. The new thread starts execution by invoking `start_routine()`; `arg` is passed as the sole argument of `start_routine()`.

The new thread terminates in one of the following ways:

- It calls `pthread_exit(3)`, specifying an exit status value that is available to another thread in the same process that calls `pthread_join(3)`.

- It returns from `start_routine()`. This is equivalent to calling `pthread_exit(3)` with the value supplied in the `return` statement.
- It is canceled (see `pthread_cancel(3)`).
- Any of the threads in the process calls `exit(3)`, or the main thread performs a return from `main()`. This causes the termination of all threads in the process.

The `attr` argument points to a `pthread_attr_t` structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using `pthread_attr_init(3)` and related functions. If `attr` is NULL, then the thread is created with default attributes.

Before returning, a successful call to `pthread_create()` stores the ID of the new thread in the buffer pointed to by `thread`; this identifier is used to refer to the thread in subsequent calls to other pthreads functions.

The new thread inherits a copy of the creating thread's signal mask (`pthread_sigmask(3)`). The set of pending signals for the new thread is empty (`sigpending(2)`). The new thread does not inherit the creating thread's alternate signal stack (`sigaltstack(2)`).

The new thread inherits the calling thread's floating-point environment (`fenv(3)`).

The initial value of the new thread's CPU-time clock is 0 (see `pthread_getcpuclockid(3)`).

Linux-specific details

The new thread inherits copies of the calling thread's capability sets (see `capabilities(7)`) and CPU affinity mask (see `sched_setaffinity(2)`).

RETURN VALUE [top](#)

On success, `pthread_create()` returns 0; on error, it returns an error number, and the contents of `*thread` are undefined.

ERRORS [top](#)



EAGAIN Insufficient resources to create another thread.

EAGAIN A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error: the **RLIMIT_NPROC** soft resource limit (set via [setrlimit\(2\)](#)), which limits the number of processes and threads for a real user ID, was reached; the kernel's system-wide limit on the number of processes and threads, [/proc/sys/kernel/threads-max](#), was reached (see [proc\(5\)](#)); or the maximum number of PIDs, [/proc/sys/kernel/pid_max](#), was reached (see [proc\(5\)](#)).

EINVAL Invalid settings in *attr*.

EPERM No permission to set the scheduling policy and parameters specified in *attr*.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-------------------------------|---------------|---------|
| <code>pthread_create()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

NOTES [top](#)

See [pthread_self\(3\)](#) for further information on the thread ID returned in **thread* by `pthread_create()`. Unless real-time scheduling policies are being employed, after a call to `pthread_create()`, it is indeterminate which thread—the caller or the new thread—will next execute.

A thread may either be *joinable* or *detached*. If a thread is joinable, then another thread can call `pthread_join(3)` to wait for the thread to terminate and fetch its exit status. Only when a terminated joinable thread has been joined are the last of its resources released back to the system. When a detached thread terminates, its resources are automatically released back to the system: it is not possible to join with the thread in order to obtain its exit status. Making a thread detached is useful for some types of daemon threads whose exit status the application does not need to care about. By default, a new thread is created in a joinable state, unless *attr* was set to create the thread in a detached state (using `pthread_attr_setdetachstate(3)`).

Under the NPTL threading implementation, if the **RLIMIT_STACK** soft resource limit *at the time the program started* has any value other than "unlimited", then it determines the default stack size of new threads. Using `pthread_attr_setstacksize(3)`, the stack size attribute can be explicitly set in the *attr* argument used to create a thread, in order to obtain a stack size other than the default. If the **RLIMIT_STACK** resource limit is set to "unlimited", a per-architecture value is used for the stack size. Here is the value for a few architectures:

| Architecture | Default stack size |
|--------------|--------------------|
| i386 | 2 MB |
| IA-64 | 32 MB |
| PowerPC | 4 MB |
| S/390 | 2 MB |
| Sparc-32 | 2 MB |
| Sparc-64 | 4 MB |
| x86_64 | 2 MB |

BUGS

[top](#)

In the obsolete LinuxThreads implementation, each of the threads in a process has a different process ID. This is in violation of the POSIX threads specification, and is the source of many other nonconformances to the standard; see `pthreads(7)`.



EXAMPLES top

The program below demonstrates the use of `pthread_create()`, as well as a number of other functions in the pthreads API.

In the following run, on a system providing the NPTL threading implementation, the stack size defaults to the value given by the "stack size" resource limit:

```
$ ulimit -s
8192          # The stack size limit is 8 MB (0x800000 bytes)
$ ./a.out hola salut servus
Thread 1: top of stack near 0xb7dd03b8; argv_string=hola
Thread 2: top of stack near 0xb75cf3b8; argv_string=salut
Thread 3: top of stack near 0xb6dce3b8; argv_string=servus
Joined with thread 1; returned value was HOLA
Joined with thread 2; returned value was SALUT
Joined with thread 3; returned value was SERVUS
```

In the next run, the program explicitly sets a stack size of 1 MB (using `pthread_attr_setstacksize(3)`) for the created threads:

```
$ ./a.out -s 0x100000 hola salut servus
Thread 1: top of stack near 0xb7d723b8; argv_string=hola
Thread 2: top of stack near 0xb7c713b8; argv_string=salut
Thread 3: top of stack near 0xb7b703b8; argv_string=servus
Joined with thread 1; returned value was HOLA
Joined with thread 2; returned value was SALUT
Joined with thread 3; returned value was SERVUS
```

Program source

```
#include <ctype.h>
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define handle_error_en(en, msg) \
    do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)
```

```
struct thread_info { /* Used as argument to thread_start() */
    pthread_t thread_id; /* ID returned by pthread_create() */
    int thread_num; /* Application-defined thread # */
    char *argv_string; /* From command-line argument */
};
```

```
/* Thread start function: display address near top of our stack,
and return upper-cased copy of argv_string. */
```

```
static void *
thread_start(void *arg)
{
    struct thread_info *tinfo = arg;
    char *uargv;

    printf("Thread %d: top of stack near %p; argv_string=%s\n",
           tinfo->thread_num, (void *) &tinfo, tinfo->argv_string);

    uargv = strdup(tinfo->argv_string);
    if (uargv == NULL)
        handle_error("strdup");

    for (char *p = uargv; *p != '\0'; p++)
        *p = toupper(*p);

    return uargv;
}
```

```
int
main(int argc, char *argv[])
{
    int s, opt;
    void *res;
    size_t num_threads;
    ssize_t stack_size;
    pthread_attr_t attr;
    struct thread_info *tinfo;

    /* The "-s" option specifies a stack size for our threads. */

    stack_size = -1;
    while ((opt = getopt(argc, argv, "s:")) != -1) {
        switch (opt) {
            case 's':
                stack_size = strtoul(optarg, NULL, 0);
                break;
        }
    }
}
```



```
default:
    fprintf(stderr, "Usage: %s [-s stack-size] arg...\n",
            argv[0]);
    exit(EXIT_FAILURE);
}
}

num_threads = argc - optind;

/* Initialize thread creation attributes. */

s = pthread_attr_init(&attr);
if (s != 0)
    handle_error_en(s, "pthread_attr_init");

if (stack_size > 0) {
    s = pthread_attr_setstacksize(&attr, stack_size);
    if (s != 0)
        handle_error_en(s, "pthread_attr_setstacksize");
}

/* Allocate memory for pthread_create() arguments. */

tinfo = calloc(num_threads, sizeof(*tinfo));
if (tinfo == NULL)
    handle_error("calloc");

/* Create one thread for each command-line argument. */

for (size_t tnum = 0; tnum < num_threads; tnum++) {
    tinfo[tnum].thread_num = tnum + 1;
    tinfo[tnum].argv_string = argv[optind + tnum];

    /* The pthread_create() call stores the thread ID into
       corresponding element of tinfo[]. */

    s = pthread_create(&tinfo[tnum].thread_id, &attr,
                      &thread_start, &tinfo[tnum]);
    if (s != 0)
        handle_error_en(s, "pthread_create");
}

/* Destroy the thread attributes object, since it is no
   longer needed. */

s = pthread_attr_destroy(&attr);
if (s != 0)
```



```
    handle_error_en(s, "pthread_attr_destroy");

/* Now join with each thread, and display its returned value. */

for (size_t tnum = 0; tnum < num_threads; tnum++) {
    s = pthread_join(tinfo[tnum].thread_id, &res);
    if (s != 0)
        handle_error_en(s, "pthread_join");

    printf("Joined with thread %d; returned value was %s\n",
          tinfo[tnum].thread_num, (char *) res);
    free(res);      /* Free memory allocated by thread */
}

free(tinfo);
exit(EXIT_SUCCESS);
}
```

SEE ALSO [top](#)

[getrlimit\(2\)](#), [pthread_attr_init\(3\)](#), [pthread_cancel\(3\)](#),
[pthread_detach\(3\)](#), [pthread_equal\(3\)](#), [pthread_exit\(3\)](#),
[pthread_getattr_np\(3\)](#), [pthread_join\(3\)](#), [pthread_self\(3\)](#),
[pthread_setattr_default_np\(3\)](#), [pthreads\(7\)](#)

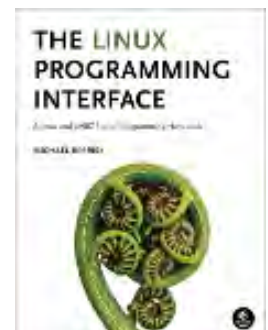
Linux man-pages (unreleased) (date) [pthread_create\(3\)](#)

Pages that refer to this page: [mmap\(2\)](#), [spu_run\(2\)](#), [wait\(2\)](#), [pthread_attr_init\(3\)](#),
[pthread_attr_setdetachstate\(3\)](#), [pthread_attr_setguardsize\(3\)](#),
[pthread_attr_setinheritsched\(3\)](#), [pthread_attr_setschedparam\(3\)](#),
[pthread_attr_setschedpolicy\(3\)](#), [pthread_attr_setscope\(3\)](#), [pthread_attr_setstack\(3\)](#),
[pthread_attr_setstackaddr\(3\)](#), [pthread_attr_setstacksize\(3\)](#), [pthread_cancel\(3\)](#),
[pthread_detach\(3\)](#), [pthread_equal\(3\)](#), [pthread_exit\(3\)](#), [pthread_getattr_default_np\(3\)](#),
[pthread_getattr_np\(3\)](#), [pthread_join\(3\)](#), [pthread_self\(3\)](#), [pthread_setaffinity_np\(3\)](#),
[pthread_setname_np\(3\)](#), [pthread_setschedparam\(3\)](#), [pthread_setschedprio\(3\)](#),
[pthread_sigmask\(3\)](#), [pthreads\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



pthread_self(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

pthread_self(3)

Library Functions Manual

pthread_self(3)

NAME [top](#)

`pthread_self` - obtain ID of the calling thread

LIBRARY [top](#)

POSIX threads library (*Libpthread*, *-lpthread*)

SYNOPSIS [top](#)

```
#include <pthread.h>

pthread_t pthread_self(void);
```

DESCRIPTION [top](#)

The `pthread_self()` function returns the ID of the calling thread. This is the same value that is returned in **thread* in the `pthread_create(3)` call that created this thread.

RETURN VALUE [top](#)

This function always succeeds, returning the calling thread's ID.

ERRORS [top](#)



This function always succeeds.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------------------------|---------------|---------|
| <code>pthread_self()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

NOTES [top](#)

POSIX.1 allows an implementation wide freedom in choosing the type used to represent a thread ID; for example, representation using either an arithmetic type or a structure is permitted. Therefore, variables of type `pthread_t` can't portably be compared using the C equality operator (`==`); use [pthread_equal\(3\)](#) instead.

Thread identifiers should be considered opaque: any attempt to use a thread ID other than in pthreads calls is nonportable and can lead to unspecified results.

Thread IDs are guaranteed to be unique only within a process. A thread ID may be reused after a terminated thread has been joined, or a detached thread has terminated.

The thread ID returned by `pthread_self()` is not the same thing as the kernel thread ID returned by a call to [gettid\(2\)](#).

SEE ALSO [top](#)



[pthread_create\(3\)](#), [pthread_equal\(3\)](#), [pthreads\(7\)](#)

Linux man-pages (unreleased) (date)

[pthread_self\(3\)](#)

Pages that refer to this page: [gettid\(2\)](#), [pthread_create\(3\)](#), [pthread_equal\(3\)](#), [pthread_getcpuclockid\(3\)](#), [pthread_kill\(3\)](#), [pthread_setaffinity_np\(3\)](#), [pthread_setschedparam\(3\)](#), [pthread_setschedprio\(3\)](#), [pthreads\(7\)](#), [signal-safety\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



pthread_equal(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 [pthread_equal\(3\)](#)

Library Functions Manual

[pthread_equal\(3\)](#)

NAME [top](#)

pthread_equal - compare thread IDs

LIBRARY [top](#)

POSIX threads library (*Libpthread*, *-lpthread*)

SYNOPSIS [top](#)

```
#include <pthread.h>
```

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

DESCRIPTION [top](#)

The `pthread_equal()` function compares two thread identifiers.

RETURN VALUE [top](#)

If the two thread IDs are equal, `pthread_equal()` returns a nonzero value; otherwise, it returns 0.

ERRORS [top](#)

This function always succeeds.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|------------------------------|---------------|---------|
| <code>pthread_equal()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

NOTES [top](#)

The `pthread_equal()` function is necessary because thread IDs should be considered opaque: there is no portable way for applications to directly compare two `pthread_t` values.

SEE ALSO [top](#)

[pthread_create\(3\)](#), [pthread_self\(3\)](#), [pthreads\(7\)](#)

Linux man-pages (unreleased) (date) [pthread_equal\(3\)](#)

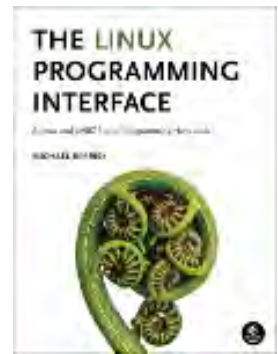
Pages that refer to this page: [pthread_create\(3\)](#), [pthread_self\(3\)](#), [pthreads\(7\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *[The Linux Programming Interface](#)*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



pthread_exit(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

pthread_exit(3)

Library Functions Manual

pthread_exit(3)

NAME [top](#)

pthread_exit - terminate calling thread

LIBRARY [top](#)

POSIX threads library (*Libpthread*, *-lpthread*)

SYNOPSIS [top](#)

```
#include <pthread.h>
```

```
[[noreturn]] void pthread_exit(void *retval);
```

DESCRIPTION [top](#)

The `pthread_exit()` function terminates the calling thread and returns a value via `retval` that (if the thread is joinable) is available to another thread in the same process that calls `pthread_join(3)`.

Any clean-up handlers established by `pthread_cleanup_push(3)` that have not yet been popped, are popped (in the reverse of the order in which they were pushed) and executed. If the thread has any thread-specific data, then, after the clean-up handlers have been executed, the corresponding destructor functions are called, in

an unspecified order.

When a thread terminates, process-shared resources (e.g., mutexes, condition variables, semaphores, and file descriptors) are not released, and functions registered using `atexit(3)` are not called.

After the last thread in a process terminates, the process terminates as by calling `exit(3)` with an exit status of zero; thus, process-shared resources are released and functions registered using `atexit(3)` are called.

RETURN VALUE [top](#)

This function does not return to the caller.

ERRORS [top](#)

This function always succeeds.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------------------------|---------------|---------|
| <code>pthread_exit()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

NOTES [top](#)

Performing a return from the start function of any thread other than the main thread results in an implicit call to **pthread_exit()**, using the function's return value as the thread's exit status.

To allow other threads to continue execution, the main thread should terminate by calling **pthread_exit()** rather than **exit(3)**.

The value pointed to by *retval* should not be located on the calling thread's stack, since the contents of that stack are undefined after the thread terminates.

BUGS [top](#)

Currently, there are limitations in the kernel implementation logic for **wait(2)**ing on a stopped thread group with a dead thread group leader. This can manifest in problems such as a locked terminal if a stop signal is sent to a foreground process whose thread group leader has already called **pthread_exit()**.

SEE ALSO [top](#)

[pthread_create\(3\)](#), [pthread_join\(3\)](#), [pthreads\(7\)](#)

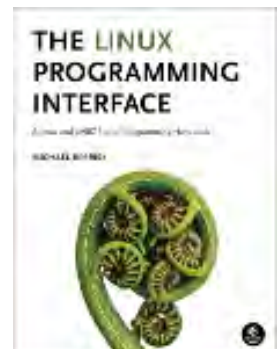
Linux man-pages (unreleased) (date) [pthread_exit\(3\)](#)

Pages that refer to this page: [prctl\(2\)](#), [pthread_cancel\(3\)](#), [pthread_cleanup_push\(3\)](#), [pthread_create\(3\)](#), [pthread_detach\(3\)](#), [pthread_join\(3\)](#), [pthread_tryjoin_np\(3\)](#), [proc\(5\)](#), [pthreads\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



pthread_cancel(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [EXAMPLES](#) | [SEE ALSO](#)

pthread_cancel(3)

Library Functions Manual

pthread_cancel(3)

NAME [top](#)

`pthread_cancel` - send a cancelation request to a thread

LIBRARY [top](#)

POSIX threads library (*Libpthread*, *-lpthread*)

SYNOPSIS [top](#)

```
#include <pthread.h>
```

```
int pthread_cancel(pthread_t thread);
```

DESCRIPTION [top](#)

The `pthread_cancel()` function sends a cancelation request to the thread *thread*. Whether and when the target thread reacts to the cancelation request depends on two attributes that are under the control of that thread: its cancelability *state* and *type*.

A thread's cancelability state, determined by `pthread_setcancelstate(3)`, can be *enabled* (the default for new threads) or *disabled*. If a thread has disabled cancelation, then a cancelation request remains queued until the thread enables cancelation. If a thread has enabled cancelation, then its cancelability type determines when cancelation occurs.

A thread's cancelation type, determined by

`pthread_setcanceltype(3)`, may be either *asynchronous* or *deferred* (the default for new threads). Asynchronous cancelability means that the thread can be canceled at any time (usually immediately, but the system does not guarantee this). Deferred cancelability means that cancelation will be delayed until the thread next calls a function that is a *cancelation point*. A list of functions that are or may be cancelation points is provided in `pthread(7)`.

When a cancelation requested is acted on, the following steps occur for *thread* (in this order):

- (1) Cancelation clean-up handlers are popped (in the reverse of the order in which they were pushed) and called. (See `pthread_cleanup_push(3)`.)
- (2) Thread-specific data destructors are called, in an unspecified order. (See `pthread_key_create(3)`.)
- (3) The thread is terminated. (See `pthread_exit(3)`.)

The above steps happen asynchronously with respect to the `pthread_cancel()` call; the return status of `pthread_cancel()` merely informs the caller whether the cancelation request was successfully queued.

After a canceled thread has terminated, a join with that thread using `pthread_join(3)` obtains `PTHREAD_CANCELED` as the thread's exit status. (Joining with a thread is the only way to know that cancelation has completed.)

RETURN VALUE [top](#)

On success, `pthread_cancel()` returns 0; on error, it returns a nonzero error number.

ERRORS [top](#)

ESRCH No thread with the ID *thread* could be found.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see `attributes(7)`.

| Interface | Attribute | Value |
|-------------------------------|---------------|---------|
| <code>pthread_cancel()</code> | Thread safety | MT-Safe |

VERSIONS [top](#)

On Linux, cancelation is implemented using signals. Under the NPTL threading implementation, the first real-time signal (i.e., signal 32) is used for this purpose. On LinuxThreads, the second real-time signal is used, if real-time signals are available, otherwise **SIGUSR2** is used.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

glibc 2.0 POSIX.1-2001.

EXAMPLES [top](#)

The program below creates a thread and then cancels it. The main thread joins with the canceled thread to check that its exit status was **PTHREAD_CANCELED**. The following shell session shows what happens when we run the program:

```
$ ./a.out
thread_func(): started; cancelation disabled
main(): sending cancelation request
thread_func(): about to enable cancelation
main(): thread was canceled
```

Program source

```
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#define handle_error_en(en, msg) \
    do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

static void *
thread_func(void *ignored_argument)
{
    int s;

    /* Disable cancelation for a while, so that we don't
       immediately react to a cancelation request. */

    s = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_setcancelstate");

    printf("%s(): started; cancelation disabled\n", __func__);
    sleep(5);
    printf("%s(): about to enable cancelation\n", __func__);

    s = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_setcancelstate");

    /* sleep() is a cancelation point. */

    sleep(1000);          /* Should get canceled while we sleep */

    /* Should never get here. */

    printf("%s(): not canceled!\n", __func__);
    return NULL;
}

int
main(void)
{
    pthread_t thr;
    void *res;
    int s;

    /* Start a thread and then send it a cancelation request. */

    s = pthread_create(&thr, NULL, &thread_func, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_create");

    sleep(2);            /* Give thread a chance to get started */
}
```



```
printf("%s(): sending cancelation request\n", __func__);
s = pthread_cancel(thr);
if (s != 0)
    handle_error_en(s, "pthread_cancel");

/* Join with thread to see what its exit status was. */

s = pthread_join(thr, &res);
if (s != 0)
    handle_error_en(s, "pthread_join");

if (res == PTHREAD_CANCELED)
    printf("%s(): thread was canceled\n", __func__);
else
    printf("%s(): thread wasn't canceled (shouldn't happen!)\n",
          __func__);
exit(EXIT_SUCCESS);
}
```

SEE ALSO [top](#)

[pthread_cleanup_push\(3\)](#), [pthread_create\(3\)](#), [pthread_exit\(3\)](#),
[pthread_join\(3\)](#), [pthread_key_create\(3\)](#),
[pthread_setcancelstate\(3\)](#), [pthread_setcanceltype\(3\)](#),
[pthread_testcancel\(3\)](#), [pthreads\(7\)](#)

Linux man-pages (unreleased) (date) [pthread_cancel\(3\)](#)

Pages that refer to this page: [pthread_cleanup_push\(3\)](#),
[pthread_cleanup_push_defer_np\(3\)](#), [pthread_create\(3\)](#), [pthread_detach\(3\)](#),
[pthread_join\(3\)](#), [pthread_kill_other_threads_np\(3\)](#), [pthread_setcancelstate\(3\)](#),
[pthread_testcancel\(3\)](#), [pthreads\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



pthread_setcancelstate(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [EXAMPLES](#) | [SEE ALSO](#)

pthread_...cancelstate(3) Library Functions Manual *pthread_...cancelstate(3)*

NAME [top](#)

pthread_setcancelstate, pthread_setcanceltype - set cancelability state and type

LIBRARY [top](#)

POSIX threads library (*Libpthread*, *-lpthread*)

SYNOPSIS [top](#)

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

DESCRIPTION [top](#)

The `pthread_setcancelstate()` sets the cancelability state of the calling thread to the value given in *state*. The previous cancelability state of the thread is returned in the buffer pointed to by *oldstate*. The *state* argument must have one of the following values:

PTHREAD_CANCEL_ENABLE

The thread is cancelable. This is the default cancelability state in all new threads, including the initial thread. The thread's cancelability type determines when a cancelable thread will respond to a cancellation request.

PTHREAD_CANCEL_DISABLE

The thread is not cancelable. If a cancellation request is received, it is blocked until cancelability is enabled.

The `pthread_setcanceltype()` sets the cancelability type of the calling thread to the value given in *type*. The previous cancelability type of the thread is returned in the buffer pointed to by *oldtype*. The *type* argument must have one of the following values:

PTHREAD_CANCEL_DEFERRED

A cancellation request is deferred until the thread next calls a function that is a cancellation point (see [pthreads\(7\)](#)). This is the default cancelability type in all new threads, including the initial thread.

Even with deferred cancellation, a cancellation point in an asynchronous signal handler may still be acted upon and the effect is as if it was an asynchronous cancellation.

PTHREAD_CANCEL_ASYNCIOUS

The thread can be canceled at any time. (Typically, it will be canceled immediately upon receiving a cancellation request, but the system doesn't guarantee this.)

The set-and-get operation performed by each of these functions is atomic with respect to other threads in the process calling the same function.

RETURN VALUE [top](#)

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS [top](#)

The `pthread_setcancelstate()` can fail with the following error:

EINVAL Invalid value for *state*.

The `pthread_setcanceltype()` can fail with the following error:

EINVAL Invalid value for *type*.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|---|---------------------|---------|
| <code>pthread_setcancelstate()</code> , <code>pthread_setcanceltype()</code> | Thread safety | MT-Safe |
| <code>pthread_setcancelstate()</code> , <code>pthread_setcanceltype()</code> | Async-cancel safety | AC-Safe |

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

glibc 2.0 POSIX.1-2001.

NOTES [top](#)

For details of what happens when a thread is canceled, see [pthread_cancel\(3\)](#).

Briefly disabling cancelability is useful if a thread performs some critical action that must not be interrupted by a cancellation request. Beware of disabling cancelability for long periods, or around operations that may block for long periods, since that will render the thread unresponsive to cancellation requests.

Asynchronous cancelability

Setting the cancelability type to **PTHREAD_CANCEL_ASYNCHRONOUS** is rarely useful. Since the thread could be canceled at *any* time, it cannot safely reserve resources (e.g., allocating memory with `malloc(3)`), acquire mutexes, semaphores, or locks, and so on. Reserving resources is unsafe because the application has no way of knowing what the state of these resources is when the thread is canceled; that is, did cancelation occur before the resources were reserved, while they were reserved, or after they were released? Furthermore, some internal data structures (e.g., the linked list of free blocks managed by the `malloc(3)` family of functions) may be left in an inconsistent state if cancelation occurs in the middle of the function call. Consequently, clean-up handlers cease to be useful.

Functions that can be safely asynchronously canceled are called *async-cancel-safe functions*. POSIX.1-2001 and POSIX.1-2008 require only that `pthread_cancel(3)`, `pthread_setcancelstate()`, and `pthread_setcanceltype()` be *async-cancel-safe*. In general, other library functions can't be safely called from an asynchronously cancelable thread.

One of the few circumstances in which asynchronous cancelability is useful is for cancelation of a thread that is in a pure compute-bound loop.

Portability notes

The Linux threading implementations permit the *oldstate* argument of `pthread_setcancelstate()` to be NULL, in which case the information about the previous cancelability state is not returned to the caller. Many other implementations also permit a NULL *oldstat* argument, but POSIX.1 does not specify this point, so portable applications should always specify a non-NULL value in *oldstate*. A precisely analogous set of statements applies for the *oldtype* argument of `pthread_setcanceltype()`.

EXAMPLES [top](#)

See `pthread_cancel(3)`.

SEE ALSO [top](#)

[pthread_cancel\(3\)](#), [pthread_cleanup_push\(3\)](#),
[pthread_testcancel\(3\)](#), [pthreads\(7\)](#)

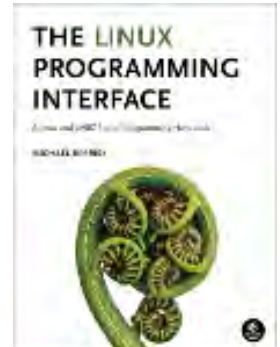
Linux man-pages (unreleased) **(date)** *[pthread_...cancelstate\(3\)](#)*

Pages that refer to this page: [pthread_cancel\(3\)](#), [pthread_cleanup_push\(3\)](#),
[pthread_cleanup_push_defer_np\(3\)](#), [pthread_kill_other_threads_np\(3\)](#),
[pthread_testcancel\(3\)](#), [pthreads\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of
[The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



pthread_join(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [EXAMPLES](#) | [SEE ALSO](#)

 [pthread_join\(3\)](#)

Library Functions Manual

[pthread_join\(3\)](#)

NAME [top](#)

pthread_join - join with a terminated thread

LIBRARY [top](#)

POSIX threads library (*Libpthread*, *-lpthread*)

SYNOPSIS [top](#)

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **retval);
```

DESCRIPTION [top](#)

The `pthread_join()` function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then `pthread_join()` returns immediately. The thread specified by *thread* must be joinable.

If *retval* is not NULL, then `pthread_join()` copies the exit status of the target thread (i.e., the value that the target thread supplied to `pthread_exit(3)`) into the location pointed to by *retval*. If the target thread was canceled, then `PTHREAD_CANCELED` is placed in the location pointed to by *retval*.

If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling `pthread_join()` is canceled, then the target thread will remain joinable (i.e., it will not be detached).

RETURN VALUE [top](#)

On success, `pthread_join()` returns 0; on error, it returns an error number.

ERRORS [top](#)

EDEADLK

A deadlock was detected (e.g., two threads tried to join with each other); or *thread* specifies the calling thread.

EINVAL *thread* is not a joinable thread.

EINVAL Another thread is already waiting to join with this thread.

ESRCH No thread with the ID *thread* could be found.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------------------------|---------------|---------|
| <code>pthread_join()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

NOTES [top](#)

After a successful call to `pthread_join()`, the caller is guaranteed that the target thread has terminated. The caller may then choose to do any clean-up that is required after termination of the thread (e.g., freeing memory or other resources that were allocated to the target thread).

Joining with a thread that has previously been joined results in undefined behavior.

Failure to join with a thread that is joinable (i.e., one that is not detached), produces a "zombie thread". Avoid doing this, since each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads (or processes).

There is no pthreads analog of `waitpid(-1, &status, 0)`, that is, "join with any terminated thread". If you believe you need this functionality, you probably need to rethink your application design.

All of the threads in a process are peers: any thread can join with any other thread in the process.

EXAMPLES [top](#)

See `pthread_create(3)`.

SEE ALSO [top](#)

`pthread_cancel(3)`, `pthread_create(3)`, `pthread_detach(3)`,
`pthread_exit(3)`, `pthread_tryjoin_np(3)`, `threads(7)`

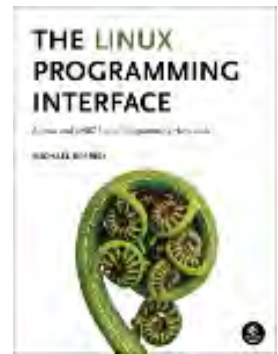
Linux man-pages (unreleased) (date) [pthread_join\(3\)](#)

Pages that refer to this page: [pthread_attr_setdetachstate\(3\)](#), [pthread_cancel\(3\)](#),
[pthread_create\(3\)](#), [pthread_detach\(3\)](#), [pthread_exit\(3\)](#), [pthread_tryjoin_np\(3\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



pthread_join(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [EXAMPLES](#) | [SEE ALSO](#)

 [pthread_join\(3\)](#)

Library Functions Manual

[pthread_join\(3\)](#)

NAME [top](#)

pthread_join - join with a terminated thread

LIBRARY [top](#)

POSIX threads library (*Libpthread*, *-lpthread*)

SYNOPSIS [top](#)

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **retval);
```

DESCRIPTION [top](#)

The `pthread_join()` function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then `pthread_join()` returns immediately. The thread specified by *thread* must be joinable.

If *retval* is not NULL, then `pthread_join()` copies the exit status of the target thread (i.e., the value that the target thread supplied to `pthread_exit(3)`) into the location pointed to by *retval*. If the target thread was canceled, then `PTHREAD_CANCELED` is placed in the location pointed to by *retval*.

If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling `pthread_join()` is canceled, then the target thread will remain joinable (i.e., it will not be detached).

RETURN VALUE [top](#)

On success, `pthread_join()` returns 0; on error, it returns an error number.

ERRORS [top](#)

EDEADLK

A deadlock was detected (e.g., two threads tried to join with each other); or *thread* specifies the calling thread.

EINVAL *thread* is not a joinable thread.

EINVAL Another thread is already waiting to join with this thread.

ESRCH No thread with the ID *thread* could be found.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------------------------|---------------|---------|
| <code>pthread_join()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

NOTES [top](#)

After a successful call to `pthread_join()`, the caller is guaranteed that the target thread has terminated. The caller may then choose to do any clean-up that is required after termination of the thread (e.g., freeing memory or other resources that were allocated to the target thread).

Joining with a thread that has previously been joined results in undefined behavior.

Failure to join with a thread that is joinable (i.e., one that is not detached), produces a "zombie thread". Avoid doing this, since each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads (or processes).

There is no pthreads analog of `waitpid(-1, &status, 0)`, that is, "join with any terminated thread". If you believe you need this functionality, you probably need to rethink your application design.

All of the threads in a process are peers: any thread can join with any other thread in the process.

EXAMPLES [top](#)

See `pthread_create(3)`.

SEE ALSO [top](#)

`pthread_cancel(3)`, `pthread_create(3)`, `pthread_detach(3)`, `pthread_exit(3)`, `pthread_tryjoin_np(3)`, `threads(7)`

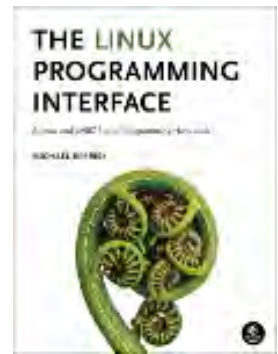
Linux man-pages (unreleased) (date) [pthread_join\(3\)](#)

Pages that refer to this page: [pthread_attr_setdetachstate\(3\)](#), [pthread_cancel\(3\)](#), [pthread_create\(3\)](#), [pthread_detach\(3\)](#), [pthread_exit\(3\)](#), [pthread_tryjoin_np\(3\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



pthread_detach(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [EXAMPLES](#) | [SEE ALSO](#)

[pthread_detach\(3\)](#)

Library Functions Manual

[pthread_detach\(3\)](#)

NAME [top](#)

pthread_detach - detach a thread

LIBRARY [top](#)

POSIX threads library (*Libpthread*, *-lpthread*)

SYNOPSIS [top](#)

```
#include <pthread.h>
```

```
int pthread_detach(pthread_t thread);
```

DESCRIPTION [top](#)

The `pthread_detach()` function marks the thread identified by *thread* as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

Attempting to detach an already detached thread results in unspecified behavior.

RETURN VALUE [top](#)



On success, **pthread_detach()** returns 0; on error, it returns an error number.

ERRORS [top](#)

EINVAL *thread* is not a joinable thread.

ESRCH No thread with the ID *thread* could be found.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-------------------------|---------------|---------|
| pthread_detach() | Thread safety | MT-Safe |

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

NOTES [top](#)

Once a thread has been detached, it can't be joined with [pthread_join\(3\)](#) or be made joinable again.

A new thread can be created in a detached state using [pthread_attr_setdetachstate\(3\)](#) to set the detached attribute of the *attr* argument of [pthread_create\(3\)](#).

The detached attribute merely determines the behavior of the system when the thread terminates; it does not prevent the thread from being terminated if the process terminates using [exit\(3\)](#) (or equivalently, if the main thread returns).



Either `pthread_join(3)` or `pthread_detach()` should be called for each thread that an application creates, so that system resources for the thread can be released. (But note that the resources of any threads for which one of these actions has not been done will be freed when the process terminates.)

EXAMPLES [top](#)

The following statement detaches the calling thread:

```
pthread_detach(pthread_self());
```

SEE ALSO [top](#)

[pthread_attr_setdetachstate\(3\)](#), [pthread_cancel\(3\)](#),
[pthread_create\(3\)](#), [pthread_exit\(3\)](#), [pthread_join\(3\)](#), [pthreads\(7\)](#)

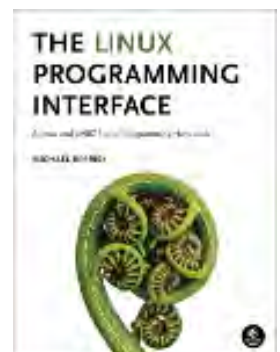
Linux man-pages (unreleased) (date) [pthread_detach\(3\)](#)

Pages that refer to this page: [pthread_attr_setdetachstate\(3\)](#), [pthread_create\(3\)](#),
[pthread_join\(3\)](#), [pthreads\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



pthread_mutex_lock(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

PTHREAD_MUTEX_LOCK(3P) POSIX Programmer's Manual **PTHREAD_MUTEX_LOCK(3P)**

PROLOG [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock – lock and unlock a mutex

SYNOPSIS [top](#)

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION [top](#)

The mutex object referenced by *mutex* shall be locked by a call to *pthread_mutex_lock()* that returns zero or **[EOWNERDEAD]**. If the

mutex is already locked by another thread, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner. If a thread attempts to relock a mutex that it has already locked, *pthread_mutex_lock()* shall behave as described in the **Relock** column of the following table. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, *pthread_mutex_unlock()* shall behave as described in the **Unlock When Not Owner** column of the following table.

| Mutex Type | Robustness | Relock | Unlock When Not Owner |
|------------|------------|--------------------------|-----------------------|
| NORMAL | non-robust | deadlock | undefined behavior |
| NORMAL | robust | deadlock | error returned |
| ERRORCHECK | either | error returned | error returned |
| RECURSIVE | either | recursive (see below) | error returned |
| DEFAULT | non-robust | undefined behavior† | undefined behavior† |
| DEFAULT | robust | undefined behavior† | error returned |

† If the mutex type is PTHREAD_MUTEX_DEFAULT, the behavior of *pthread_mutex_lock()* may correspond to one of the three other standard mutex types as described in the table above. If it does not correspond to one of those three, the behavior is undefined for the cases marked †.

Where the table indicates recursive behavior, the mutex shall maintain the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count shall be set to one. Every time a thread relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks the mutex, the lock count shall be decremented by one. When the lock count reaches zero, the mutex shall become available for other threads to acquire.

The *pthread_mutex_trylock()* function shall be equivalent to



`pthread_mutex_lock()`, except that if the mutex object referenced by `mutex` is currently locked (by any thread, including the current thread), the call shall return immediately. If the mutex type is `PTHREAD_MUTEX_RECURSIVE` and the mutex is currently owned by the calling thread, the mutex lock count shall be incremented by one and the `pthread_mutex_trylock()` function shall immediately return success.

The `pthread_mutex_unlock()` function shall release the mutex object referenced by `mutex`. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by `mutex` when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

(In the case of `PTHREAD_MUTEX_RECURSIVE` mutexes, the mutex shall become available when the count reaches zero and the calling thread no longer has any locks on this mutex.)

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread shall resume waiting for the mutex as if it was not interrupted.

If `mutex` is a robust mutex and the process containing the owning thread terminated while holding the mutex lock, a call to `pthread_mutex_lock()` shall return the error value **[EOWNERDEAD]**. If `mutex` is a robust mutex and the owning thread terminated while holding the mutex lock, a call to `pthread_mutex_lock()` may return the error value **[EOWNERDEAD]** even if the process in which the owning thread resides has not terminated. In these cases, the mutex is locked by the thread but the state it protects is marked as inconsistent. The application should ensure that the state is made consistent for reuse and when that is complete call `pthread_mutex_consistent()`. If the application is unable to recover the state, it should unlock the mutex without a prior call to `pthread_mutex_consistent()`, after which the mutex is marked permanently unusable.

If `mutex` does not refer to an initialized mutex object, the behavior of `pthread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` is undefined.

RETURN VALUE

[top](#)



If successful, the `pthread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS [top](#)

The `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions shall fail if:

EAGAIN The mutex could not be acquired because the maximum number of recursive locks for `mutex` has been exceeded.

EINVAL The `mutex` was created with the protocol attribute having the value `PTHREAD_PRIO_PROTECT` and the calling thread's priority is higher than the mutex's current priority ceiling.

ENOTRECOVERABLE

The state protected by the mutex is not recoverable.

EOWNERDEAD

The mutex is a robust mutex and the process containing the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

The `pthread_mutex_lock()` function shall fail if:

EDEADLK

The mutex type is `PTHREAD_MUTEX_ERRORCHECK` and the current thread already owns the mutex.

The `pthread_mutex_trylock()` function shall fail if:

EBUSY The `mutex` could not be acquired because it was already locked.

The `pthread_mutex_unlock()` function shall fail if:

EPERM The mutex type is `PTHREAD_MUTEX_ERRORCHECK` or `PTHREAD_MUTEX_RECURSIVE`, or the mutex is a robust mutex, and the current thread does not own the mutex.

The `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions may fail if:

EOWNERDEAD

The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

The `pthread_mutex_lock()` function may fail if:

EDEADLK

A deadlock condition was detected.

These functions shall not return an error code of **[EINTR]**.

The following sections are informative.

EXAMPLES

[top](#)

None.

APPLICATION USAGE

[top](#)

Applications that have assumed that non-zero return values are errors will need updating for use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting a currently inconsistent state is **[EOWNERDEAD]**. Applications that do not check the error returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If an application is supposed to work with normal and robust mutexes it should check all return values for error conditions and if necessary take appropriate action.

RATIONALE

[top](#)

Mutex objects are intended to serve as a low-level primitive from which other thread synchronization functions can be built. As such, the implementation of mutexes should be as efficient as possible, and this has ramifications on the features available at the interface.



The mutex functions and the particular default settings of the mutex attributes have been motivated by the desire to not preclude fast, inlined implementations of mutex locking and unlocking.

Since most attributes only need to be checked when a thread is going to be blocked, the use of attributes does not slow the (common) mutex-locking case.

Likewise, while being able to extract the thread ID of the owner of a mutex might be desirable, it would require storing the current thread ID when each mutex is locked, and this could incur unacceptable levels of overhead. Similar arguments apply to a *mutex_tryunlock* operation.

For further rationale on the extended mutex types, see the Rationale (Informative) volume of POSIX.1-2017, *Threads Extensions*.

If an implementation detects that the value specified by the *mutex* argument does not refer to an initialized mutex object, it is recommended that the function should fail and report an **[EINVAL]** error.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

[pthread_mutex_consistent\(3p\)](#), [pthread_mutex_destroy\(3p\)](#),
[pthread_mutex_timedlock\(3p\)](#), [pthread_mutexattr_getrobust\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, *Section 4.12, Memory Synchronization*, [pthread.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The



Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

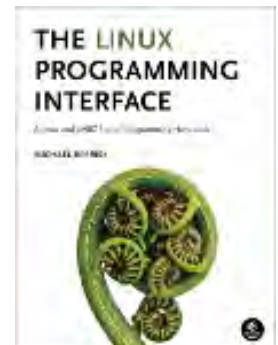
IEEE/The Open Group**2017*****PTHREAD_MUTEX_LOCK(3P)***

Pages that refer to this page: [pthread.h\(0p\)](#), [pthread_mutexattr_getrobust\(3p\)](#), [pthread_mutexattr_gettype\(3p\)](#), [pthread_mutex_consistent\(3p\)](#), [pthread_mutex_destroy\(3p\)](#), [pthread_mutex_getprioceiling\(3p\)](#), [pthread_mutex_timedlock\(3p\)](#), [pthread_mutex_trylock\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



Materials for Topic 8: Threads

Full C Programs

- [unsynced_threads_experiment.c](#) - a C program that demonstrates the effect of a race condition.
- [synced_threads_experiment.c](#) - a C program that solves the issue in the unsynced threads experiment.
- [unsynced_bank_withdraw.c](#) - a C program that demonstrates the effect of a race condition using the a bank withdrawal example.
- [synced_bank_withdraw.c](#) - a C program that solves the issue in the unsynced bank withdraw program.
- [threads_example.c](#) - a C program showing the use of `pthread_create()` and `pthread_join()`.

Runnable Linux Commands

- The command:

```
gcc -Wall -Wextra -O2 -g -pthread -o program program.c
```

compiles the C source code located inside the file `program.c` and links the `Pthread` library to the executable.

See more details [here](#).

- The command:

```
./short_prompt
```

executes code inside a file named `.short_prompt` and

sources it (applies all the changes to the current session.)

See more details [here](#).

- The command:

```
././long_prompt
```

executes code inside a file named `./long_prompt` and

sources it (applies all the changes to the current session.)

See more details [here](#).



This website by [Miriam Briskman](#) is licensed under [CC BY-NC 4.0](#).


```
1  /* This program demonstrates the results of
2  *   a race condition. While, in theory,
3  *   we expect a specific number to be
4  *   output, a different number may be
5  *   generated in practice due to an
6  *   existing race condition.
7  *
8  *   Miriam Briskman, 4/16/2023
9  *   CISC 3350, Brooklyn College
10 *   Licensed under CC BY-NC 4.0
11 */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <pthread.h>
16 #include <errno.h>
17
18 // The number of thread structures:
19 #define THREAD_NUM 1000
20
21 // An integer global variable (accessible by
22 //   all the threads:)
23 int number = THREAD_NUM;
24
25 // A function that increments the global
26 //   variable 'number':
27 void * add (void * arg)
28 {
29     number = number + 1; // <-- Race condition here!
30     return arg;
31 }
32
33 int main ()
34 {
35     pthread_t threads [THREAD_NUM];
36     int i = 0, j, ret;
37
38     printf ("In each round of this experiment, "
39            "each one of %d threads will attempt "
40            "to increment 0.\nTheoretically, "
41            "after %d increments, the result "
42            "should be %d.\nIn practice, this "
```



```
43     "isn't always the case!\n\n",
44     THREAD_NUM,
45     THREAD_NUM,
46     THREAD_NUM);
47
48     printf ("This experiment will keep running "
49            "until the result is other than %d:\n\n",
50            THREAD_NUM);
51
52     // Run the experiment until number is other than
53     //     THREAD_NUM:
54     while (number == THREAD_NUM)
55     {
56         // Initialize 'number' to 0:
57         number = 0;
58
59         // Let the threads start running:
60         for (j = 0; j < THREAD_NUM; j++)
61         {
62             ret = pthread_create (&(threads[j]),
63                                 NULL,
64                                 add,
65                                 NULL);
66
67             if (ret)
68             {
69                 errno = ret;
70                 perror ("pthread_create");
71                 exit (EXIT_FAILURE);
72             }
73
74             // Wait for the threads to return:
75             for (j = 0; j < THREAD_NUM; j++)
76             {
77                 ret = pthread_join (threads[j], NULL);
78                 if (ret)
79                 {
80                     errno = ret;
81                     perror ("pthread_join");
82                     exit (EXIT_FAILURE);
83                 }
84             }
85
```



```
86     i++;
87 }
88
89 printf ("-----\n"
90        "-----\n"
91        "In round %d: result = %d.\n"
92        "-----\n"
93        "-----\n\n",
94        (i + 1), number);
95
96 printf ("The reason for this situation is a "
97        "race condition that prevents additions"
98        "\nfrom being done in sequential order, "
99        "thus corrupting the result.\n\n");
100
101 printf ("Specifically, the machine instructions "
102        "of 2 or more of the threads \nwere "
103        "interleaved, which caused this "
104        "corruption to happen.\n");
105
106 return EXIT_SUCCESS;
107 }
108
```



```
1  /* This program demonstrates that, with
2  *   thread synchronization, no race
3  *   conditions happen, so data isn't
4  *   corrupted. This program includes
5  *   a solution to the issue demonstrated
6  *   in unsynced_threads_experiment.c
7  *
8  *   Miriam Briskman, 4/16/2023
9  *   CISC 3350, Brooklyn College
10 *   Licensed under CC BY-NC 4.0
11 */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <pthread.h>
16 #include <errno.h>
17
18 // The number of thread structures:
19 #define THREAD_NUM 1000
20
21 // An integer global variable (accessible by
22 //   all the threads:)
23 int number = THREAD_NUM;
24
25 // Definiting the mutex type:
26 pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
27
28 void * add (void * arg)
29 {
30     pthread_mutex_lock (&my_mutex); // Locking
31     number = number + 1;
32     pthread_mutex_unlock (&my_mutex); // Unlocking
33     return arg;
34 }
35
36 int main ()
37 {
38     pthread_t threads [THREAD_NUM];
39     int i = 0, j, ret;
40
41     printf ("In each round of this experiment, each "
42            "one of %d threads will attempt to "
```



```
43     "increment 0.\n"
44     "Since this program synchronizes threads, "
45     "the result is guaranteed to be "
46     "exactly %d.\n",
47     THREAD_NUM,
48     THREAD_NUM);
49
50 printf ("We run only 1000 rounds:\n\n");
51
52 // Run the experiment until number is other
53 // than THREAD_NUM:
54 while (number == THREAD_NUM && i < 1000)
55 {
56     // Initialize 'number' to 0:
57     number = 0;
58
59     // Let the threads start running:
60     for (j = 0; j < THREAD_NUM; j++)
61     {
62         ret = pthread_create (&(threads[j]),
63                               NULL,
64                               add,
65                               NULL);
66         if (ret)
67         {
68             errno = ret;
69             perror ("pthread_create");
70             exit (EXIT_FAILURE);
71         }
72     }
73
74     /* Wait for the threads to return: */
75     for (j = 0; j < THREAD_NUM; j++)
76     {
77         ret = pthread_join (threads[j], NULL);
78         if (ret)
79         {
80             errno = ret;
81             perror ("pthread_join");
82             exit (EXIT_FAILURE);
83         }
84     }
85
```



```
86     i++;
87 }
88
89 printf ("-----\n"
90        "-----\n"
91        "In round %d: result = %d.\n"
92        "-----\n"
93        "-----\n\n",
94        i, number);
95
96 if (number == THREAD_NUM)
97     printf ("Getting %d in each of the rounds "
98            "implies that we eliminated the "
99            "race condition!\n",
100           THREAD_NUM);
101 else
102 {
103     printf ("The reason for this situation is "
104            "a race condition that prevents "
105            "additions\nfrom being done in "
106            "sequential order, thus corrupting "
107            "the result.\n\n");
108
109     printf ("Specifically, the machine instructions "
110            "of 2 or more of the threads \nwere "
111            "interleaved, which caused this "
112            "corruption to happen.\n");
113 }
114
115 return EXIT_SUCCESS;
116 }
117
```



```
1  /* This program demonstrates the results of
2  *   a race condition. While, in theory,
3  *   we expect a specific number to be
4  *   output, a different number may be
5  *   generated in practice due to an
6  *   existing race condition.
7  *
8  *   Miriam Briskman, 4/16/2023
9  *   CISC 3350, Brooklyn College
10 *   Licensed under CC BY-NC 4.0
11 */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <pthread.h>
16 #include <errno.h>
17
18 /* Definiting the 'BankAccount' struct: */
19 struct BankAccount
20 {
21     double balance;
22 };
23
24 /* Definiting the 'Activity' struct: */
25 struct Activity
26 {
27     double amount;
28     int thread_num;
29     struct BankAccount * ptr;
30 };
31
32 // A function that performs a bank withdrawal:
33 void * withdraw (void * activity)
34 {
35     struct Activity * my_activity
36         = ((struct Activity *) activity);
37
38     /* Race condition starts here: */
39     const double amount = my_activity -> amount,
40         balance = (my_activity -> ptr) -> balance;
41     const int thread_num = my_activity -> thread_num;
42     printf ("--- Thread %d: [Balance = $%.2f] "
```



```
43     "Trying to withdraw $%.2f...\n",
44     thread_num,
45     balance,
46     amount);
47 if (balance < amount)
48     pthread_exit ((void *)-1);
49
50 (my_activity -> ptr) -> balance = balance - amount;
51 /* Race condition ends above. */
52
53 printf ("--- Thread %d: Success: withdrew $%.2f.\n",
54         thread_num,
55         amount);
56
57 pthread_exit ((void *)0);
58 }
59
60 int main ()
61 {
62     pthread_t threads [2];
63     int i = 0, ret;
64     long long result[1], // An array of 1 element.
65         temp = -1;
66     result[0] = -1;
67     struct BankAccount b1 = {500.0};
68     struct Activity a1 = {200.0, 1, &b1},
69         a2 = {400.0, 2, &b1};
70
71     printf ("In each round of this experiment, "
72            "each one of two threads will attempt "
73            "to withdraw from a balance of $500.\n"
74            "One thread will withdraw $200, while "
75            "another will $400.\n"
76            "Theoretically, before the 2nd withdrawal, "
77            "the bank should eliminate that other "
78            "withdrawal due to insufficient balance.\n"
79            "In practice, this isn't always the case!\n\n");
80
81     printf ("This experiment will keep running until "
82            "the result is other than an error message "
83            "from the bank. In other words, we will "
84            "show that, eventually, a race condition "
85            "will happen, and the bank will "
```




```
86         "miscalculate the account balance!\n");
87
88 // Run the experiment until number is other than
89 //   THREAD_NUM:
90 while (result[0] == -1 || temp == -1)
91 {
92     b1.balance = 500.0;
93
94     /* Let the threads start running: */
95     ret = pthread_create (&(threads[0]),
96                          NULL,
97                          withdraw,
98                          (void *) &a1);
99
100    if (ret)
101    {
102        errno = ret;
103        perror ("pthread_create");
104        exit (EXIT_FAILURE);
105    }
106
107    ret = pthread_create (&(threads[1]),
108                        NULL,
109                        withdraw,
110                        (void *) &a2);
111
112    if (ret)
113    {
114        errno = ret;
115        perror ("pthread_create");
116        exit (EXIT_FAILURE);
117    }
118
119    i++;
120
121 // Wait for the threads to return:
122 ret = pthread_join (threads[0], (void *) &result);
123 if (ret)
124 {
125     errno = ret;
126     perror ("pthread_join");
127     exit (EXIT_FAILURE);
128 }
129
130 temp = *result;
```



```
129
130     ret = pthread_join (threads[1], (void *) &result);
131     if (ret)
132     {
133         errno = ret;
134         perror ("pthread_join");
135         exit (EXIT_FAILURE);
136     }
137
138     if (result[0] == -1 || temp == -1)
139         printf ("Round %d: [Result = -1] Error on "
140                "2nd withdrawal due to an "
141                "insufficient balance of $%.2f.\n",
142                i,
143                b1.balance);
144
145 }
146
147 printf ("-----"
148        "-----\n"
149        "Round %d: [Result = 0] Balance = $%.2f "
150        "(no withdrawal error.)\n"
151        "-----"
152        "-----\n\n",
153        i, b1.balance);
154
155 printf ("The reason for this situation is a "
156        "race condition that prevents additions"
157        "\nfrom being done in sequential order, "
158        "thus corrupting the result.\n\n");
159
160 printf ("Specifically, the machine instructions "
161        "of 2 or more of the threads \nwere "
162        "interleaved, which caused this "
163        "corruption to happen.\n");
164
165 return EXIT_SUCCESS;
166 }
167
```



```
1  /* This program demonstrates the prevention
2  *   of the a race condition present in
3  *   unsynched_bank_withdraw.c
4  *
5  *   Miriam Briskman, 4/16/2023
6  *   CISC 3350, Brooklyn College
7  *   Licensed under CC BY-NC 4.0
8  */
9
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <pthread.h>
13 #include <errno.h>
14
15 /* Definiting the 'BankAccount' struct: */
16 struct BankAccount
17 {
18     double balance;
19 };
20
21 /* Definiting the 'Activity' struct: */
22 struct Activity
23 {
24     double amount;
25     int thread_num;
26     struct BankAccount * ptr;
27 };
28
29 // Definiting the mutex type:
30 pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
31
32 // A function that performs a bank withdrawal:
33 void * withdraw (void * activity)
34 {
35     struct Activity * my_activity
36         = ((struct Activity *) activity);
37
38     pthread_mutex_lock (&my_mutex); // Locking
39     const double amount = my_activity -> amount,
40         balance = (my_activity -> ptr) -> balance;
41     const int thread_num = my_activity -> thread_num;
42     printf ("--- Thread %d: [Balance = $%.2f] "
```



```
43     "Trying to withdraw $%.2f...\n",
44     thread_num,
45     balance,
46     amount);
47     if (balance < amount)
48     {
49         pthread_mutex_unlock (&my_mutex); // Unlocking
50         pthread_exit ((void *)-1);
51     }
52
53     (my_activity -> ptr) -> balance = balance - amount;
54     pthread_mutex_unlock (&my_mutex); // Unlocking
55
56     printf ("--- Thread %d: Success: withdrew $%.2f.\n",
57           thread_num,
58           amount);
59
60     pthread_exit ((void *)0);
61 }
62
63 int main ()
64 {
65     pthread_t threads [2];
66     int i = 0, ret;
67     long long result[1], // An array of 1 element.
68           temp = -1;
69     result[0] = -1;
70     struct BankAccount b1 = {500.0};
71     struct Activity a1 = {200.0, 1, &b1},
72           a2 = {400.0, 2, &b1};
73
74     printf ("In each round of this experiment, "
75           "each one of two threads will attempt "
76           "to withdraw from a balance of $500.\n"
77           "One thread will withdraw $200, while "
78           "another will $400.\n"
79           "Since this program synchronizes "
80           "threads, we are guaranteed to receive "
81           "a warning message from the bank due to "
82           "insufficient balance before the other "
83           "withdrawal.\n\n");
84
85     printf ("We run only 1000 rounds:\n\n");
```



```
86
87 // Run the experiment until number is other than
88 //   THREAD_NUM:
89 while ((result[0] == -1 || temp == -1) && i < 1000)
90 {
91     b1.balance = 500.0;
92
93     printf ("Round %d:\n", i + 1);
94
95     /* Let the threads start running: */
96     ret = pthread_create (&(threads[0]),
97                          NULL,
98                          withdraw,
99                          (void *) &a1);
100    if (ret)
101    {
102        errno = ret;
103        perror ("pthread_create");
104        exit (EXIT_FAILURE);
105    }
106
107    ret = pthread_create (&(threads[1]),
108                          NULL,
109                          withdraw,
110                          (void *) &a2);
111    if (ret)
112    {
113        errno = ret;
114        perror ("pthread_create");
115        exit (EXIT_FAILURE);
116    }
117
118    i++;
119
120    // Wait for the threads to return:
121    ret = pthread_join (threads[0], (void *) &result);
122    if (ret)
123    {
124        errno = ret;
125        perror ("pthread_join");
126        exit (EXIT_FAILURE);
127    }
128
```



```
129     temp = *result;
130
131     ret = pthread_join (threads[1], (void *) &result);
132     if (ret)
133     {
134         errno = ret;
135         perror ("pthread_join");
136         exit (EXIT_FAILURE);
137     }
138
139     if (result[0] == -1 || temp == -1)
140         printf ("Round %d: [Result = -1] Error "
141                "on 2nd withdrawal due to an "
142                "insufficient balance of $%.2f.\n",
143                i,
144                b1.balance);
145 }
146
147 if (result[0] == -1 || temp == -1)
148     printf ("-----\n"
149            "-----\n"
150            "Getting a withdrawal error in each "
151            "of the rounds implies that we "
152            "eliminated the race condition!\n");
153 else
154 {
155     printf ("-----\n"
156            "-----\n"
157            "Round %d: [Result = %lld] Balance = "
158            "$%.2f (no withdrawal error.)\n"
159            "-----\n"
160            "-----\n\n",
161            i,
162            *result,
163            b1.balance);
164     printf ("The reason for this situation is "
165            "a race condition that prevents "
166            "additions\nfrom being done in "
167            "sequential order, thus corrupting "
168            "the result.\n\n");
169
170     printf ("Specifically, the machine instructions "
171            "of 2 or more of the threads \nwere "
```



```
172         "interleaved, which caused this "  
173         "corruption to happen.\n");  
174     }  
175  
176     return EXIT_SUCCESS;  
177 }  
178
```



```
1  /* A C program using pthread_create() and
2   *   pthread_join() to create 3 threads and manage
3   *   their execution and termination.
4   *
5   *   Miriam Briskman, 4/16/2023
6   *   CISC 3350, Brooklyn College
7   *   Licensed under CC BY-NC 4.0
8   */
9
10 #include <stdlib.h>
11 #include <stdio.h>
12 #include <pthread.h>
13 #include <errno.h>
14
15 // Defining a function that will be run
16 //   by threads:
17 void * run (void * message)
18 {
19     printf ("%s", (char *) message);
20     return message;
21 }
22
23 int main (void)
24 {
25     pthread_t my_threads [3];
26     int ret, index;
27
28     char * messages [] = {"Hello from Thread 1!\n",
29                           "Thread 2 says hi!\n",
30                           "What's up? It's Thread 3!\n"};
31
32     // Create 3 threads, assigning each its own
33     //   message:
34     for (index = 0; index < 3; index++)
35     {
36         ret = pthread_create (&(my_threads[index]),
37                               NULL,
38                               run,
39                               (void *) (messages[index]));
40         if (ret)
41         {
42             errno = ret;
```





```
43         perror ("pthread_create");
44         exit (EXIT_FAILURE);
45     }
46 }
47
48 // Wait for the threads to exit. If we didn't
49 //   join here, we'd risk terminating this
50 //   main thread before the other 3 threads
51 //   finished working:
52 for (index = 0; index < 3; index++)
53 {
54     ret = pthread_join (my_threads[index], NULL);
55     if (ret)
56     {
57         errno = ret;
58         perror ("pthread_join");
59         exit (EXIT_FAILURE);
60     }
61 }
62
63 return EXIT_SUCCESS;
64 }
65
```



Topic 9: CPU Scheduling

Note: A copy of each one of the sources listed below is included in this packet. To jump to the location of a source copy, click the page link under “Page”.

In case a source isn’t included (e.g., a YouTube video,) click the  (external link) symbol under “Page” to open the external source in a browser window.

| # | Citation & Source Link | Page |
|----|--|------|
| 1 | Kalin, Marty. “CFS: Completely fair process scheduling in Linux?” <i>Opensource.Com</i> . URL: https://opensource.com/article/19/2/fair-scheduling-linux | 1035 |
| 2 | “sched_yield(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/sched_yield.2.html | 1045 |
| 3 | “nice(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/nice.2.html | 1048 |
| 4 | “getpriority(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/getpriority.2.html | 1051 |
| 5 | “sched_setaffinity(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html | 1055 |
| 6 | “CPU_SET(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/CPU_SET.3.html | 1063 |
| 7 | “sched(7) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man7/sched.7.html | 1070 |
| 8 | “sched_setscheduler(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/sched_setscheduler.2.html | 1087 |
| 9 | “sched_setparam(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/sched_setparam.2.html | 1092 |
| 10 | “sched_get_priority_max(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/sched_get_priority_max.2.html | 1095 |
| 11 | “sched_rr_get_interval(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/sched_rr_get_interval.2.html | 1098 |
| 12 | “getrlimit(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/getrlimit.2.html | 1101 |
| 13 | Briskman, Miriam. “Materials for Topic 9: CPU Scheduling.” <i>Topic 9: CPU Scheduling — CISC 3350 Materials</i> , 2023. URL: www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_09/ | 1114 |
| 14 | Briskman, Miriam. “scheduling_examples.c .” (C source code) 27 Mar. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_09/scheduling_examples.c | 1119 |



LOG IN



opensource.com

CFS: Completely fair process scheduling in Linux

CFS gives every task a fair share of processor resources in a low-fuss but highly efficient way.

By [Marty Kalin](#)

February 5, 2019 | [0 Comments](#) | 11 min read

 129 readers like this.



Image by: [Opensource.com](#)

Linux takes a modular approach to processor scheduling in that different algorithms can be used to schedule different process types. A scheduling class specifies which scheduling policy applies to which type of process. Completely fair scheduling (CFS),



which became part of the Linux 2.6.23 kernel in 2007, is the scheduling class for normal (as opposed to real-time) processes and therefore is named **SCHED_NORMAL**.

CFS is geared for the interactive applications typical in a desktop environment, but it can be configured as **SCHED_BATCH** to favor the batch workloads common, for example, on a high-volume web server. In any case, CFS breaks dramatically with what might be called "classic preemptive scheduling." Also, the "completely fair" claim has to be seen with a technical eye; otherwise, the claim might seem like an empty boast.

Let's dig into the details of what sets CFS apart from—indeed, above—other process schedulers. Let's start with a quick review of some core technical terms.

Some core concepts

Linux inherits the Unix view of a *process* as a program in execution. As such, a process must contend with other processes for shared system resources: *memory* to hold instructions and data, at least one *processor* to execute instructions, and *I/O devices* to interact with the external world. Process scheduling is how the operating system (OS) assigns tasks (e.g., crunching some numbers, copying a file) to processors—a running process then performs the task. A process has one or more *threads of execution*, which are sequences of machine-level instructions. To schedule a process is to schedule one of its threads on a processor.

The Linux Terminal

- ▶ [Top 7 terminal emulators for Linux](#)
- ▶ [10 command-line tools for data analysis in Linux](#)
- ▶ [Download Now: SSH cheat sheet](#)
- ▶ [Advanced Linux commands cheat sheet](#)
- ▶ [Linux command line tutorials](#)

In a simplifying move, Linux turns process scheduling into thread scheduling by treating a scheduled process as if it were single-threaded. If a process is multi-threaded with N threads, then N scheduling actions would be required to cover the threads. Threads

within a multi-threaded process remain related in that they share resources such as memory address space. Linux threads are sometimes described as lightweight processes, with the *lightweight* underscoring the sharing of resources among the threads within a process.

Although a process can be in various states, two are of particular interest in scheduling. A *blocked* process is awaiting the completion of some event such as an I/O event. The process can resume execution only after the event completes. A *runnable* process is one that is not currently blocked.

A process is *processor-bound* (aka *compute-bound*) if it consumes mostly processor as opposed to I/O resources, and *I/O-bound* in the opposite case; hence, a processor-bound process is mostly runnable, whereas an I/O-bound process is mostly blocked. As examples, crunching numbers is processor-bound, and accessing files is I/O-bound. Although an entire process might be characterized as either processor-bound or I/O-bound, a given process may be one or the other during different stages of its execution. Interactive desktop applications, such as browsers, tend to be I/O-bound.

A good process scheduler has to balance the needs of processor-bound and I/O-bound tasks, especially in an operating system such as Linux that thrives on so many hardware platforms: desktop machines, embedded devices, mobile devices, server clusters, supercomputers, and more.

Classic preemptive scheduling versus CFS

Unix popularized classic preemptive scheduling, which other operating systems including VAX/VMS, Windows NT, and Linux later adopted. At the center of this scheduling model is a *fixed timeslice*, the amount of time (e.g., 50ms) that a task is allowed to hold a processor until preempted in favor of some other task. If a preempted process has not finished its work, the process must be rescheduled. This model is powerful in that it supports multitasking (concurrency) through processor time-sharing, even on the single-CPU machines of yesteryear.

The classic model typically includes multiple scheduling queues, one per process priority: Every process in a higher-priority queue gets scheduled before any process in a lower-priority queue. As an example, VAX/VMS uses 32 priority queues for scheduling.

CFS dispenses with fixed timeslices and explicit priorities. The amount of time for a given task on a processor is computed dynamically as the scheduling context changes over the system's lifetime. Here is a sketch of the motivating ideas and technical details:

- Imagine a processor, P , which is idealized in that it can execute multiple tasks *simultaneously*. For example, tasks $T1$ and $T2$ can execute on P at the same time, with each receiving 50% of P 's magical processing power. This idealization describes *perfect multitasking*, which CFS strives to achieve on actual as opposed to idealized processors. CFS is designed to approximate perfect multitasking.
- The CFS scheduler has a *target latency*, which is the minimum amount of time—idealized to an infinitely small duration—required for every runnable task to get at least one turn on the processor. If such a duration could be infinitely small, then each runnable task would have had a turn on the processor during any given timespan, however small (e.g., 10ms, 5ns, etc.). Of course, an idealized infinitely small duration must be approximated in the real world, and the default approximation is 20ms. Each runnable task then gets a $1/N$ slice of the target latency, where N is the number of tasks. For example, if the target latency is 20ms and there are four contending tasks, then each task gets a timeslice of 5ms. By the way, if there is only a single task during a scheduling event, this lucky task gets the entire target latency as its slice. The *fair* in CFS comes to the fore in the $1/N$ slice given to each task contending for a processor.
- The $1/N$ slice is, indeed, a timeslice—but not a fixed one because such a slice depends on N , the number of tasks currently contending for the processor. The system changes over time. Some processes terminate and new ones are spawned; runnable processes block and blocked processes become runnable. The value of N is dynamic and so, therefore, is the $1/N$ timeslice computed for each runnable task contending for a processor. The traditional **nice** value is used to weight the $1/N$ slice: a low-priority **nice** value means that only some fraction of the $1/N$ slice is given to a task, whereas a high-priority **nice** value means that a proportionately greater fraction of the $1/N$ slice is given to a task. In summary, **nice** values do not determine the slice, but only modify the $1/N$ slice that represents fairness among the contending tasks.
- The operating system incurs overhead whenever a *context switch* occurs; that is, when one process is preempted in favor of another. To keep this overhead from

becoming unduly large, there is a minimum amount of time (with a typical setting of 1ms to 4ms) that any scheduled process must run before being preempted. This minimum is known as the *minimum granularity*. If many tasks (e.g., 20) are contending for the processor, then the minimum granularity (assume 4ms) might be *more* than the $1/N$ slice (in this case, 1ms). If the minimum granularity turns out to be larger than the $1/N$ slice, the system is overloaded because there are too many tasks contending for the processor—and fairness goes out the window.

- When does preemption occur? CFS tries to minimize context switches, given their overhead: time spent on a context switch is time unavailable for other tasks. Accordingly, once a task gets the processor, it runs for its entire weighted $1/N$ slice before being preempted in favor of some other task. Suppose task T1 has run for its weighted $1/N$ slice, and runnable task T2 currently has the lowest *virtual runtime* (vruntime) among the tasks contending for the processor. The vruntime records, in nanoseconds, how long a task has run on the processor. In this case, T1 would be preempted in favor of T2.
- The scheduler tracks the vruntime for all tasks, runnable and blocked. The lower a task's vruntime, the more deserving the task is for time on the processor. CFS accordingly moves low-vruntime tasks towards the front of the scheduling line. Details are forthcoming because the *line* is implemented as a tree, not a list.
- How often should the CFS scheduler reschedule? There is a simple way to determine the *scheduling period*. Suppose that the target latency (TL) is 20ms and the minimum granularity (MG) is 4ms:

$TL / MG = (20 / 4) = 5$ ## five or fewer tasks are ok

In this case, five or fewer tasks would allow each task a turn on the processor during the target latency. For example, if the task number is five, each runnable task has a $1/N$ slice of 4ms, which happens to equal the minimum granularity; if the task number is three, each task gets a $1/N$ slice of almost 7ms. In either case, the scheduler would reschedule in 20ms, the duration of the target latency.

Trouble occurs if the number of tasks (e.g., 10) exceeds TL / MG because now each task must get the minimum time of 4ms instead of the computed $1/N$ slice, which is 2ms. In this case, the scheduler would reschedule in 40ms:



```
(number of tasks) * MG = (10 * 4) = 40ms ## period = 40ms
```

Linux schedulers that predate CFS use heuristics to promote the fair treatment of interactive tasks with respect to scheduling. CFS takes a quite different approach by letting the vruntime facts speak mostly for themselves, which happens to support *sleepers fairness*. An interactive task, by its very nature, tends to sleep a lot in the sense that it awaits user inputs and so becomes I/O-bound; hence, such a task tends to have a relatively low vruntime, which tends to move the task towards the front of the scheduling line.

Special features

CFS supports symmetrical multiprocessing (SMP) in which any process (whether kernel or user) can execute on any processor. Yet configurable *scheduling domains* can be used to group processors for load balancing or even segregation. If several processors share the same scheduling policy, then load balancing among them is an option; if a particular processor has a scheduling policy different from the others, then this processor would be segregated from the others with respect to scheduling.

Configurable *scheduling groups* are another CFS feature. As an example, consider the Nginx web server that's running on my desktop machine. At startup, this server has a master process and four worker processes, which act as HTTP request handlers. For any HTTP request, the particular worker that handles the request is irrelevant; it matters only that the request is handled in a timely manner, and so the four workers together provide a pool from which to draw a task-handler as requests come in. It thus seems fair to treat the four Nginx workers as a group rather than as individuals for scheduling purposes, and a scheduling group can be used to do just that. The four Nginx workers could be configured to have a single vruntime among them rather than individual vruntimes. Configuration is done in the traditional Linux way, through files. For vruntime-sharing, a file named **cpu.shares**, with the details given through familiar shell commands, would be created.

As noted earlier, Linux supports *scheduling classes* so that different scheduling policies, together with their implementing algorithms, can coexist on the same platform. A scheduling class is implemented as a code module in C. CFS, the scheduling class described so far, is **SCHED_NORMAL**. There are also scheduling classes specifically for real-time tasks, **SCHED_FIFO** (first in, first out) and **SCHED_RR** (round robin). Under

SCHED_FIFO, tasks run to completion; under **SCHED_RR**, tasks run until they exhaust a fixed timeslice and are preempted.

CFS implementation

CFS requires efficient data structures to track task information and high-performance code to generate the schedules. Let's begin with a central term in scheduling, the *runqueue*. This is a data structure that represents a timeline for scheduled tasks. Despite the name, the runqueue need not be implemented in the traditional way, as a FIFO list. CFS breaks with tradition by using a time-ordered red-black tree as a runqueue. The data structure is well-suited for the job because it is a self-balancing binary search tree, with efficient **insert** and **remove** operations that execute in **$O(\log N)$** time, where N is the number of nodes in the tree. Also, a tree is an excellent data structure for organizing entities into a hierarchy based on a particular property, in this case a vruntime.

In CFS, the tree's internal nodes represent tasks to be scheduled, and the tree as a whole, like any runqueue, represents a timeline for task execution. Red-black trees are in wide use beyond scheduling; for example, Java uses this data structure to implement its **TreeMap**.

Under CFS, every processor has a specific runqueue of tasks, and no task occurs at the same time in more than one runqueue. Each runqueue is a red-black tree. The tree's internal nodes represent tasks or task groups, and these nodes are indexed by their vruntime values so that (in the tree as a whole or in any subtree) the internal nodes to the left have lower vruntime values than the ones to the right:

```

    25    ## 25 is a task vruntime
    /\
   17 29    ## 17 roots the left subtree, 29 the right one
  /\ ...
 5 19    ## and so on
... \
    nil    ## leaf nodes are nil

```

In summary, tasks with the lowest vruntime—and, therefore, the greatest need for a processor—reside somewhere in the left subtree; tasks with relatively high vruntimes congregate in the right subtree. A preempted task would go into the right subtree, thus

giving other tasks a chance to move leftwards in the tree. A task with the smallest vruntime winds up in the tree's leftmost (internal) node, which is thus the front of the runqueue.

The CFS scheduler has an instance, the C **task_struct**, to track detailed information about each task to be scheduled. This structure embeds a **sched_entity** structure, which in turn has scheduling-specific information, in particular, the vruntime per task or task group:

```
struct task_struct {          /** info on a task **/  
    ...  
    struct sched_entity se;  /** vruntime, etc. **/  
    ...  
};
```

The red-black tree is implemented in familiar C fashion, with a premium on pointers for efficiency. A **cfs_rq** structure instance embeds a **rb_root** field named **tasks_timeline**, which points to the root of a red-black tree. Each of the tree's internal nodes has pointers to the parent and the two child nodes; the leaf nodes have *nil* as their value.

CFS illustrates how a straightforward idea—give every task a fair share of processor resources—can be implemented in a low-fuss but highly efficient way. It's worth repeating that CFS achieves fair and efficient scheduling without traditional artifacts such as fixed timeslices and explicit task priorities. The pursuit of even better schedulers goes on, of course; for the moment, however, CFS is as good as it gets for general-purpose processor scheduling.

Tags:

LINUX

Marty Kalin

I'm an academic in computer science (College of Computing and Digital Media, DePaul University) with wide experience in





software development, mostly in production planning and scheduling (steel industry) and product configuration (truck and bus manufacturing). Details on books and other publications are available at

[More about me](#)

Comments are closed.

These comments are closed.

Related Content



[What's new in GNOME 44?](#)



[5 reasons virtual machines still matter](#)



[Remove the background from an image with this Linux command](#)



This work is licensed under a Creative Commons Attribution-Share Alike 4.0 International License.

ABOUT THIS SITE

The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.



Opensource.com aspires to publish all content under a **Creative Commons license** but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.

A note on advertising: Opensource.com does not sell advertising on the site or in any of its newsletters.

opensource.com

Copyright ©2024 Red Hat, Inc.

[Privacy Policy](#)

[Terms of use](#)

[Cookie preferences](#)



sched_yield(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [CAVEATS](#) | [SEE ALSO](#)

 [sched_yield\(2\)](#)

System Calls Manual

[sched_yield\(2\)](#)

NAME [top](#)

`sched_yield` - yield the processor

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sched.h>
```

```
int sched_yield(void);
```

DESCRIPTION [top](#)

`sched_yield()` causes the calling thread to relinquish the CPU. The thread is moved to the end of the queue for its static priority and a new thread gets to run.

RETURN VALUE [top](#)

On success, `sched_yield()` returns 0. On error, -1 is returned, and `errno` is set to indicate the error.



ERRORS [top](#)

In the Linux implementation, **sched_yield()** always succeeds.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001 (but optional). POSIX.1-2008.

Before POSIX.1-2008, systems on which **sched_yield()** is available defined **_POSIX_PRIORITY_SCHEDULING** in *<unistd.h>*.

CAVEATS [top](#)

sched_yield() is intended for use with real-time scheduling policies (i.e., **SCHED_FIFO** or **SCHED_RR**). Use of **sched_yield()** with nondeterministic scheduling policies such as **SCHED_OTHER** is unspecified and very likely means your application design is broken.

If the calling thread is the only thread in the highest priority list at that time, it will continue to run after a call to **sched_yield()**.

Avoid calling **sched_yield()** unnecessarily or inappropriately (e.g., when resources needed by other schedulable threads are still held by the caller), since doing so will result in unnecessary context switches, which will degrade system performance.

SEE ALSO [top](#)

[sched\(7\)](#)

Linux man-pages (unreleased) (date)

[sched_yield\(2\)](#)

Pages that refer to this page: [getrlimit\(2\)](#), [sched_setattr\(2\)](#), [sched_setscheduler\(2\)](#), [syscalls\(2\)](#), [pthread_yield\(3\)](#), [sched\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



nice(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

nice(2)

System Calls Manual

nice(2)

NAME [top](#)

nice - change process priority

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int nice(int inc);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
nice():
    _XOPEN_SOURCE
        || /* Since glibc 2.19: */ _DEFAULT_SOURCE
        || /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

DESCRIPTION [top](#)

nice() adds *inc* to the nice value for the calling thread. (A higher nice value means a lower priority.)

The range of the nice value is +19 (low priority) to -20 (high priority). Attempts to set a nice value outside the range are clamped to the range.

Traditionally, only a privileged process could lower the nice value (i.e., set a higher priority). However, since Linux 2.6.12, an unprivileged process can decrease the nice value of a target process that has a suitable **RLIMIT_NICE** soft limit; see [getrlimit\(2\)](#) for details.

RETURN VALUE [top](#)

On success, the new nice value is returned (but see NOTES below). On error, -1 is returned, and *errno* is set to indicate the error.

A successful call can legitimately return -1. To detect an error, set *errno* to 0 before the call, and check whether it is nonzero after **nice()** returns -1.

ERRORS [top](#)

EPERM The calling process attempted to increase its priority by supplying a negative *inc* but has insufficient privileges. Under Linux, the **CAP_SYS_NICE** capability is required. (But see the discussion of the **RLIMIT_NICE** resource limit in [setrlimit\(2\)](#).)

VERSIONS [top](#)

C library/kernel differences

POSIX.1 specifies that **nice()** should return the new nice value. However, the raw Linux system call returns 0 on success. Likewise, the **nice()** wrapper function provided in glibc 2.2.3 and earlier returns 0 on success.

Since glibc 2.2.4, the **nice()** wrapper function provided by glibc provides conformance to POSIX.1 by calling [getpriority\(2\)](#) to obtain the new nice value, which is then returned to the caller.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.3BSD.

NOTES [top](#)

For further details on the nice value, see [sched\(7\)](#).

Note: the addition of the "autogroup" feature in Linux 2.6.38 means that the nice value no longer has its traditional effect in many circumstances. For details, see [sched\(7\)](#).

SEE ALSO [top](#)

[nice\(1\)](#), [renice\(1\)](#), [fork\(2\)](#), [getpriority\(2\)](#), [getrlimit\(2\)](#), [setpriority\(2\)](#), [capabilities\(7\)](#), [sched\(7\)](#)

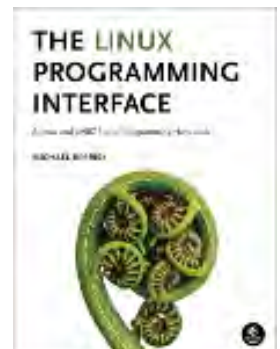
Linux man-pages (unreleased) (date) [nice\(2\)](#)

Pages that refer to this page: [nice\(1\)](#), [getpriority\(2\)](#), [getrlimit\(2\)](#), [sched_setaffinity\(2\)](#), [sched_setattr\(2\)](#), [sched_setparam\(2\)](#), [sched_setscheduler\(2\)](#), [syscalls\(2\)](#), [capabilities\(7\)](#), [sched\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



getpriority(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

 [getpriority\(2\)](#)

System Calls Manual

[getpriority\(2\)](#)

NAME [top](#)

getpriority, setpriority - get/set program scheduling priority

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/resource.h>
```

```
int getpriority(int which, id_t who);  
int setpriority(int which, id_t who, int prio);
```

DESCRIPTION [top](#)

The scheduling priority of the process, process group, or user, as indicated by *which* and *who* is obtained with the **getpriority()** call and set with the **setpriority()** call. The process attribute dealt with by these system calls is the same attribute (also known as the "nice" value) that is dealt with by [nice\(2\)](#).

The value *which* is one of **PRIO_PROCESS**, **PRIO_PGRP**, or **PRIO_USER**, and *who* is interpreted relative to *which* (a process identifier for **PRIO_PROCESS**, process group identifier for **PRIO_PGRP**, and a

user ID for **PRIO_USER**). A zero value for *who* denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process.

The *prio* argument is a value in the range -20 to 19 (but see NOTES below), with -20 being the highest priority and 19 being the lowest priority. Attempts to set a priority outside this range are silently clamped to the range. The default priority is 0; lower values give a process a higher scheduling priority.

The **getpriority()** call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The **setpriority()** call sets the priorities of all of the specified processes to the specified value.

Traditionally, only a privileged process could lower the nice value (i.e., set a higher priority). However, since Linux 2.6.12, an unprivileged process can decrease the nice value of a target process that has a suitable **RLIMIT_NICE** soft limit; see [getrlimit\(2\)](#) for details.

RETURN VALUE [top](#)

On success, **getpriority()** returns the calling thread's nice value, which may be a negative number. On error, it returns -1 and sets *errno* to indicate the error.

Since a successful call to **getpriority()** can legitimately return the value -1, it is necessary to clear *errno* prior to the call, then check *errno* afterward to determine if -1 is an error or a legitimate value.

setpriority() returns 0 on success. On failure, it returns -1 and sets *errno* to indicate the error.

ERRORS [top](#)

EACCES The caller attempted to set a lower nice value (i.e., a higher process priority), but did not have the required privilege (on Linux: did not have the **CAP_SYS_NICE** capability).

EINVAL *which* was not one of **PRIO_PROCESS**, **PRIO_PGRP**, or **PRIO_USER**.

EPERM A process was located, but its effective user ID did not match either the effective or the real user ID of the caller, and was not privileged (on Linux: did not have the **CAP_SYS_NICE** capability). But see NOTES below.

ESRCH No process was located using the *which* and *who* values specified.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.4BSD (these interfaces first appeared in 4.2BSD).

NOTES [top](#)

For further details on the nice value, see [sched\(7\)](#).

Note: the addition of the "autogroup" feature in Linux 2.6.38 means that the nice value no longer has its traditional effect in many circumstances. For details, see [sched\(7\)](#).

A child created by [fork\(2\)](#) inherits its parent's nice value. The nice value is preserved across [execve\(2\)](#).

The details on the condition for **EPERM** depend on the system. The above description is what POSIX.1-2001 says, and seems to be followed on all System V-like systems. Linux kernels before Linux 2.6.12 required the real or effective user ID of the caller to match the real user of the process *who* (instead of its effective user ID). Linux 2.6.12 and later require the effective user ID of the caller to match the real or effective user ID of the process *who*. All BSD-like systems (SunOS 4.1.3, Ultrix 4.2, 4.3BSD, FreeBSD 4.3, OpenBSD-2.5, ...) behave in the same manner as Linux 2.6.12 and later.

C library/kernel differences

The `getpriority` system call returns nice values translated to the range 40..1, since a negative return value would be interpreted as an error. The glibc wrapper function for `getpriority()` translates the value back according to the formula $unice = 20 - knice$ (thus, the 40..1 range returned by the kernel corresponds to the range -20..19 as seen by user space).

BUGS [top](#)

According to POSIX, the nice value is a per-process setting. However, under the current Linux/NPTL implementation of POSIX threads, the nice value is a per-thread attribute: different threads in the same process can have different nice values. Portable applications should avoid relying on the Linux behavior, which may be made standards conformant in the future.

SEE ALSO [top](#)

[nice\(1\)](#), [renice\(1\)](#), [fork\(2\)](#), [capabilities\(7\)](#), [sched\(7\)](#)

[Documentation/scheduler/sched-nice-design.txt](#) in the Linux kernel source tree (since Linux 2.6.23)

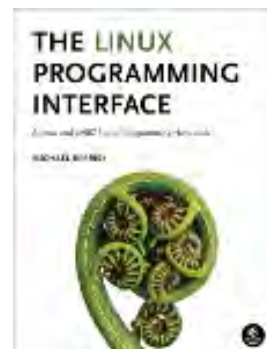
Linux man-pages (unreleased) (date) [getpriority\(2\)](#)

Pages that refer to this page: [renice\(1\)](#), [getrlimit\(2\)](#), [ioprio_set\(2\)](#), [nice\(2\)](#), [sched_rr_get_interval\(2\)](#), [sched_setaffinity\(2\)](#), [sched_setattr\(2\)](#), [sched_setparam\(2\)](#), [sched_setscheduler\(2\)](#), [syscalls\(2\)](#), [errno\(3\)](#), [id_t\(3type\)](#), [proc\(5\)](#), [systemd.exec\(5\)](#), [capabilities\(7\)](#), [credentials\(7\)](#), [pid_namespaces\(7\)](#), [pthreads\(7\)](#), [sched\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sched_setaffinity(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [EXAMPLES](#) | [SEE ALSO](#)

sched_setaffinity(2) System Calls Manual *sched_setaffinity(2)*

NAME [top](#)

`sched_setaffinity`, `sched_getaffinity` - set and get a thread's CPU affinity mask

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#define _GNU_SOURCE                /* See feature_test_macros(7) */
#include <sched.h>

int sched_setaffinity(pid_t pid, size_t cpusetsize,
                     const cpu_set_t *mask);
int sched_getaffinity(pid_t pid, size_t cpusetsize,
                     cpu_set_t *mask);
```

DESCRIPTION [top](#)

A thread's CPU affinity mask determines the set of CPUs on which it is eligible to run. On a multiprocessor system, setting the CPU affinity mask can be used to obtain performance benefits. For example, by dedicating one CPU to a particular thread (i.e., setting the affinity mask of that thread to specify a single CPU, and setting the affinity mask of all other threads to exclude that CPU), it is possible to ensure maximum execution speed for that thread. Restricting a thread to run on a single CPU also

avoids the performance cost caused by the cache invalidation that occurs when a thread ceases to execute on one CPU and then recommences execution on a different CPU.

A CPU affinity mask is represented by the `cpu_set_t` structure, a "CPU set", pointed to by `mask`. A set of macros for manipulating CPU sets is described in [CPU_SET\(3\)](#).

`sched_setaffinity()` sets the CPU affinity mask of the thread whose ID is `pid` to the value specified by `mask`. If `pid` is zero, then the calling thread is used. The argument `cpusetsize` is the length (in bytes) of the data pointed to by `mask`. Normally this argument would be specified as `sizeof(cpu_set_t)`.

If the thread specified by `pid` is not currently running on one of the CPUs specified in `mask`, then that thread is migrated to one of the CPUs specified in `mask`.

`sched_getaffinity()` writes the affinity mask of the thread whose ID is `pid` into the `cpu_set_t` structure pointed to by `mask`. The `cpusetsize` argument specifies the size (in bytes) of `mask`. If `pid` is zero, then the mask of the calling thread is returned.

RETURN VALUE [top](#)

On success, `sched_setaffinity()` and `sched_getaffinity()` return 0 (but see "C library/kernel differences" below, which notes that the underlying `sched_getaffinity()` differs in its return value). On failure, -1 is returned, and `errno` is set to indicate the error.

ERRORS [top](#)

EFAULT A supplied memory address was invalid.

EINVAL The affinity bit mask `mask` contains no processors that are currently physically on the system and permitted to the thread according to any restrictions that may be imposed by `cpuset` cgroups or the "cpuset" mechanism described in [cpuset\(7\)](#).

EINVAL (`sched_getaffinity()` and, before Linux 2.6.9, `sched_setaffinity()`) `cpusetsize` is smaller than the size of the affinity mask used by the kernel.



EPERM (`sched_setaffinity()`) The calling thread does not have appropriate privileges. The caller needs an effective user ID equal to the real user ID or effective user ID of the thread identified by *pid*, or it must possess the **CAP_SYS_NICE** capability in the user namespace of the thread *pid*.

ESRCH The thread whose ID is *pid* could not be found.

STANDARDS [top](#)

Linux.

HISTORY [top](#)

Linux 2.5.8, glibc 2.3.

Initially, the glibc interfaces included a *cpusetsize* argument, typed as *unsigned int*. In glibc 2.3.3, the *cpusetsize* argument was removed, but was then restored in glibc 2.3.4, with type *size_t*.

NOTES [top](#)

After a call to `sched_setaffinity()`, the set of CPUs on which the thread will actually run is the intersection of the set specified in the *mask* argument and the set of CPUs actually present on the system. The system may further restrict the set of CPUs on which the thread runs if the "cpuset" mechanism described in [cpuset\(7\)](#) is being used. These restrictions on the actual set of CPUs on which the thread will run are silently imposed by the kernel.

There are various ways of determining the number of CPUs available on the system, including: inspecting the contents of */proc/cpuinfo*; using [sysconf\(3\)](#) to obtain the values of the **_SC_NPROCESSORS_CONF** and **_SC_NPROCESSORS_ONLN** parameters; and inspecting the list of CPU directories under */sys/devices/system/cpu/*.

[sched\(7\)](#) has a description of the Linux scheduling scheme.

The affinity mask is a per-thread attribute that can be adjusted independently for each of the threads in a thread group. The value returned from a call to [gettid\(2\)](#) can be passed in the



argument *pid*. Specifying *pid* as 0 will set the attribute for the calling thread, and passing the value returned from a call to `getpid(2)` will set the attribute for the main thread of the thread group. (If you are using the POSIX threads API, then use `pthread_setaffinity_np(3)` instead of `sched_setaffinity()`.)

The *isolcpus* boot option can be used to isolate one or more CPUs at boot time, so that no processes are scheduled onto those CPUs. Following the use of this boot option, the only way to schedule processes onto the isolated CPUs is via `sched_setaffinity()` or the `cpuset(7)` mechanism. For further information, see the kernel source file *Documentation/admin-guide/kernel-parameters.txt*. As noted in that file, *isolcpus* is the preferred mechanism of isolating CPUs (versus the alternative of manually setting the CPU affinity of all processes on the system).

A child created via `fork(2)` inherits its parent's CPU affinity mask. The affinity mask is preserved across an `execve(2)`.

C library/kernel differences

This manual page describes the glibc interface for the CPU affinity calls. The actual system call interface is slightly different, with the *mask* being typed as *unsigned long **, reflecting the fact that the underlying implementation of CPU sets is a simple bit mask.

On success, the raw `sched_getaffinity()` system call returns the number of bytes placed copied into the *mask* buffer; this will be the minimum of *cpusetsize* and the size (in bytes) of the *cpumask_t* data type that is used internally by the kernel to represent the CPU set bit mask.

Handling systems with large CPU affinity masks

The underlying system calls (which represent CPU masks as bit masks of type *unsigned long **) impose no restriction on the size of the CPU mask. However, the *cpu_set_t* data type used by glibc has a fixed size of 128 bytes, meaning that the maximum CPU number that can be represented is 1023. If the kernel CPU affinity mask is larger than 1024, then calls of the form:

```
sched_getaffinity(pid, sizeof(cpu_set_t), &mask);
```

fail with the error **EINVAL**, the error produced by the underlying system call for the case where the *mask* size specified in *cpusetsize* is smaller than the size of the affinity mask used by the kernel. (Depending on the system CPU topology, the kernel affinity mask can be substantially larger than the number of



active CPUs in the system.)

When working on systems with large kernel CPU affinity masks, one must dynamically allocate the *mask* argument (see [CPU_ALLOC\(3\)](#)). Currently, the only way to do this is by probing for the size of the required mask using `sched_getaffinity()` calls with increasing mask sizes (until the call does not fail with the error **EINVAL**).

Be aware that [CPU_ALLOC\(3\)](#) may allocate a slightly larger CPU set than requested (because CPU sets are implemented as bit masks allocated in units of `sizeof(Long)`). Consequently, `sched_getaffinity()` can set bits beyond the requested allocation size, because the kernel sees a few additional bits. Therefore, the caller should iterate over the bits in the returned set, counting those which are set, and stop upon reaching the value returned by [CPU_COUNT\(3\)](#) (rather than iterating over the number of bits requested to be allocated).

EXAMPLES [top](#)

The program below creates a child process. The parent and child then each assign themselves to a specified CPU and execute identical loops that consume some CPU time. Before terminating, the parent waits for the child to complete. The program takes three command-line arguments: the CPU number for the parent, the CPU number for the child, and the number of loop iterations that both processes should perform.

As the sample runs below demonstrate, the amount of real and CPU time consumed when running the program will depend on intra-core caching effects and whether the processes are using the same CPU.

We first employ [lscpu\(1\)](#) to determine that this (x86) system has two cores, each with two CPUs:

```
$ lscpu | egrep -i 'core.*:|socket'  
Thread(s) per core:      2  
Core(s) per socket:     2  
Socket(s):                1
```

We then time the operation of the example program for three cases: both processes running on the same CPU; both processes running on different CPUs on the same core; and both processes running on different CPUs on different cores.

```
$ time -p ./a.out 0 0 10000000
```



```
real 14.75
user 3.02
sys 11.73
$ time -p ./a.out 0 1 100000000
real 11.52
user 3.98
sys 19.06
$ time -p ./a.out 0 3 100000000
real 7.89
user 3.29
sys 12.07
```

Program source

```
#define _GNU_SOURCE
#include <err.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int          parentCPU, childCPU;
    cpu_set_t    set;
    unsigned int nloops;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s parent-cpu child-cpu num-loops\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }

    parentCPU = atoi(argv[1]);
    childCPU = atoi(argv[2]);
    nloops = atoi(argv[3]);

    CPU_ZERO(&set);

    switch (fork()) {
    case -1:          /* Error */
        err(EXIT_FAILURE, "fork");

    case 0:          /* Child */
        CPU_SET(childCPU, &set);
```



```
    if (sched_setaffinity(getpid(), sizeof(set), &set) == -1)
        err(EXIT_FAILURE, "sched_setaffinity");

    for (unsigned int j = 0; j < nloops; j++)
        getppid();

    exit(EXIT_SUCCESS);

default:          /* Parent */
    CPU_SET(parentCPU, &set);

    if (sched_setaffinity(getpid(), sizeof(set), &set) == -1)
        err(EXIT_FAILURE, "sched_setaffinity");

    for (unsigned int j = 0; j < nloops; j++)
        getppid();

    wait(NULL);    /* Wait for child to terminate */
    exit(EXIT_SUCCESS);
}
}
```

SEE ALSO [top](#)

[lscpu\(1\)](#), [nproc\(1\)](#), [taskset\(1\)](#), [clone\(2\)](#), [getcpu\(2\)](#), [getpriority\(2\)](#), [gettid\(2\)](#), [nice\(2\)](#), [sched_get_priority_max\(2\)](#), [sched_get_priority_min\(2\)](#), [sched_getscheduler\(2\)](#), [sched_setscheduler\(2\)](#), [setpriority\(2\)](#), [CPU_SET\(3\)](#), [get_nprocs\(3\)](#), [pthread_setaffinity_np\(3\)](#), [sched_getcpu\(3\)](#), [capabilities\(7\)](#), [cpuset\(7\)](#), [sched\(7\)](#), [numactl\(8\)](#)

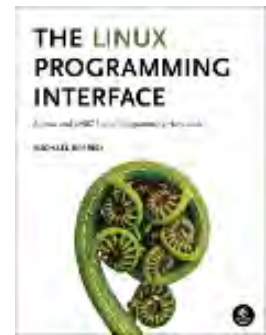
Linux man-pages (unreleased) **(date)** ***sched_setaffinity(2)***

Pages that refer to this page: [systemd-nspawn\(1\)](#), [taskset\(1\)](#), [getcpu\(2\)](#), [gettid\(2\)](#), [sched_get_priority_max\(2\)](#), [sched_setattr\(2\)](#), [sched_setparam\(2\)](#), [sched_setscheduler\(2\)](#), [syscalls\(2\)](#), [CPU_SET\(3\)](#), [numa\(3\)](#), [pthread_attr_setaffinity_np\(3\)](#), [pthread_create\(3\)](#), [pthread_setaffinity_np\(3\)](#), [systemd.exec\(5\)](#), [capabilities\(7\)](#), [cpuset\(7\)](#), [credentials\(7\)](#), [pthreads\(7\)](#), [sched\(7\)](#), [migratepages\(8\)](#), [numactl\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



CPU_SET(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [EXAMPLES](#) | [SEE ALSO](#)

 CPU_SET(3)

Library Functions Manual

CPU_SET(3)

NAME [top](#)

CPU_SET, CPU_CLR, CPU_ISSET, CPU_ZERO, CPU_COUNT, CPU_AND, CPU_OR, CPU_XOR, CPU_EQUAL, CPU_ALLOC, CPU_ALLOC_SIZE, CPU_FREE, CPU_SET_S, CPU_CLR_S, CPU_ISSET_S, CPU_ZERO_S, CPU_COUNT_S, CPU_AND_S, CPU_OR_S, CPU_XOR_S, CPU_EQUAL_S - macros for manipulating CPU sets

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <sched.h>

void CPU_ZERO(cpu_set_t *set);

void CPU_SET(int cpu, cpu_set_t *set);
void CPU_CLR(int cpu, cpu_set_t *set);
int CPU_ISSET(int cpu, cpu_set_t *set);

int CPU_COUNT(cpu_set_t *set);

void CPU_AND(cpu_set_t *destset,
             cpu_set_t *srcset1, cpu_set_t *srcset2);
void CPU_OR(cpu_set_t *destset,
            cpu_set_t *srcset1, cpu_set_t *srcset2);
void CPU_XOR(cpu_set_t *destset,
```

```
    cpu_set_t *srcset1, cpu_set_t *srcset2);

int CPU_EQUAL(cpu_set_t *set1, cpu_set_t *set2);

cpu_set_t *CPU_ALLOC(int num_cpus);
void CPU_FREE(cpu_set_t *set);
size_t CPU_ALLOC_SIZE(int num_cpus);

void CPU_ZERO_S(size_t setsize, cpu_set_t *set);

void CPU_SET_S(int cpu, size_t setsize, cpu_set_t *set);
void CPU_CLR_S(int cpu, size_t setsize, cpu_set_t *set);
int CPU_ISSET_S(int cpu, size_t setsize, cpu_set_t *set);

int CPU_COUNT_S(size_t setsize, cpu_set_t *set);

void CPU_AND_S(size_t setsize, cpu_set_t *destset,
               cpu_set_t *srcset1, cpu_set_t *srcset2);
void CPU_OR_S(size_t setsize, cpu_set_t *destset,
              cpu_set_t *srcset1, cpu_set_t *srcset2);
void CPU_XOR_S(size_t setsize, cpu_set_t *destset,
               cpu_set_t *srcset1, cpu_set_t *srcset2);

int CPU_EQUAL_S(size_t setsize, cpu_set_t *set1, cpu_set_t *set2);
```

DESCRIPTION [top](#)

The `cpu_set_t` data structure represents a set of CPUs. CPU sets are used by `sched_setaffinity(2)` and similar interfaces.

The `cpu_set_t` data type is implemented as a bit mask. However, the data structure should be treated as opaque: all manipulation of CPU sets should be done via the macros described in this page.

The following macros are provided to operate on the CPU set `set`:

CPU_ZERO()

Clears `set`, so that it contains no CPUs.

CPU_SET()

Add CPU `cpu` to `set`.

CPU_CLR()

Remove CPU `cpu` from `set`.

CPU_ISSET()



Test to see if CPU *cpu* is a member of *set*.

CPU_COUNT()

Return the number of CPUs in *set*.

Where a *cpu* argument is specified, it should not produce side effects, since the above macros may evaluate the argument more than once.

The first CPU on the system corresponds to a *cpu* value of 0, the next CPU corresponds to a *cpu* value of 1, and so on. No assumptions should be made about particular CPUs being available, or the set of CPUs being contiguous, since CPUs can be taken offline dynamically or be otherwise absent. The constant **CPU_SETSIZE** (currently 1024) specifies a value one greater than the maximum CPU number that can be stored in *cpu_set_t*.

The following macros perform logical operations on CPU sets:

CPU_AND()

Store the intersection of the sets *srcset1* and *srcset2* in *destset* (which may be one of the source sets).

CPU_OR()

Store the union of the sets *srcset1* and *srcset2* in *destset* (which may be one of the source sets).

CPU_XOR()

Store the XOR of the sets *srcset1* and *srcset2* in *destset* (which may be one of the source sets). The XOR means the set of CPUs that are in either *srcset1* or *srcset2*, but not both.

CPU_EQUAL()

Test whether two CPU set contain exactly the same CPUs.

Dynamically sized CPU sets

Because some applications may require the ability to dynamically size CPU sets (e.g., to allocate sets larger than that defined by the standard *cpu_set_t* data type), glibc nowadays provides a set of macros to support this.

The following macros are used to allocate and deallocate CPU sets:

CPU_ALLOC()

Allocate a CPU set large enough to hold CPUs in the range

0 to *num_cpus-1*.

CPU_ALLOC_SIZE()

Return the size in bytes of the CPU set that would be needed to hold CPUs in the range 0 to *num_cpus-1*. This macro provides the value that can be used for the *setsize* argument in the **CPU*_S()** macros described below.

CPU_FREE()

Free a CPU set previously allocated by **CPU_ALLOC()**.

The macros whose names end with "_S" are the analogs of the similarly named macros without the suffix. These macros perform the same tasks as their analogs, but operate on the dynamically allocated CPU set(s) whose size is *setsize* bytes.

RETURN VALUE [top](#)

CPU_ISSET() and **CPU_ISSET_S()** return nonzero if *cpu* is in *set*; otherwise, it returns 0.

CPU_COUNT() and **CPU_COUNT_S()** return the number of CPUs in *set*.

CPU_EQUAL() and **CPU_EQUAL_S()** return nonzero if the two CPU sets are equal; otherwise they return 0.

CPU_ALLOC() returns a pointer on success, or NULL on failure. (Errors are as for [malloc\(3\)](#).)

CPU_ALLOC_SIZE() returns the number of bytes required to store a CPU set of the specified cardinality.

The other functions do not return a value.

STANDARDS [top](#)

Linux.

HISTORY [top](#)

The **CPU_ZERO()**, **CPU_SET()**, **CPU_CLR()**, and **CPU_ISSET()** macros were added in glibc 2.3.3.

CPU_COUNT() first appeared in glibc 2.6.

CPU_AND(), **CPU_OR()**, **CPU_XOR()**, **CPU_EQUAL()**, **CPU_ALLOC()**, **CPU_ALLOC_SIZE()**, **CPU_FREE()**, **CPU_ZERO_S()**, **CPU_SET_S()**, **CPU_CLR_S()**, **CPU_ISSET_S()**, **CPU_AND_S()**, **CPU_OR_S()**, **CPU_XOR_S()**, and **CPU_EQUAL_S()** first appeared in glibc 2.7.

NOTES [top](#)

To duplicate a CPU set, use [memcpy\(3\)](#).

Since CPU sets are bit masks allocated in units of long words, the actual number of CPUs in a dynamically allocated CPU set will be rounded up to the next multiple of `sizeof(unsigned long)`. An application should consider the contents of these extra bits to be undefined.

Notwithstanding the similarity in the names, note that the constant **CPU_SETSIZE** indicates the number of CPUs in the `cpu_set_t` data type (thus, it is effectively a count of the bits in the bit mask), while the `setsize` argument of the **CPU*_S()** macros is a size in bytes.

The data types for arguments and return values shown in the SYNOPSIS are hints what about is expected in each case. However, since these interfaces are implemented as macros, the compiler won't necessarily catch all type errors if you violate the suggestions.

BUGS [top](#)

On 32-bit platforms with glibc 2.8 and earlier, **CPU_ALLOC()** allocates twice as much space as is required, and **CPU_ALLOC_SIZE()** returns a value twice as large as it should. This bug should not affect the semantics of a program, but does result in wasted memory and less efficient operation of the macros that operate on dynamically allocated CPU sets. These bugs are fixed in glibc 2.9.

EXAMPLES [top](#)

The following program demonstrates the use of some of the macros used for dynamically allocated CPU sets.

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>

#include <assert.h>

int
main(int argc, char *argv[])
{
    cpu_set_t *cpusetp;
    size_t size, num_cpus;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <num-cpus>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    num_cpus = atoi(argv[1]);

    cpusetp = CPU_ALLOC(num_cpus);
    if (cpusetp == NULL) {
        perror("CPU_ALLOC");
        exit(EXIT_FAILURE);
    }

    size = CPU_ALLOC_SIZE(num_cpus);

    CPU_ZERO_S(size, cpusetp);
    for (size_t cpu = 0; cpu < num_cpus; cpu += 2)
        CPU_SET_S(cpu, size, cpusetp);

    printf("CPU_COUNT() of set:    %d\n", CPU_COUNT_S(size, cpusetp));

    CPU_FREE(cpusetp);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO [top](#)

[sched_setaffinity\(2\)](#), [pthread_attr_setaffinity_np\(3\)](#),
[pthread_setaffinity_np\(3\)](#), [cpuset\(7\)](#)

Linux man-pages (unreleased) **(date)**

CPU_SET(3)

Pages that refer to this page: [sched_setaffinity\(2\)](#), [pthread_attr_setaffinity_np\(3\)](#),
[pthread_setaffinity_np\(3\)](#), [tracefs_instance_set_affinity\(3\)](#), [cpuset\(7\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sched(7) — Linux manual page

[NAME](#) | [DESCRIPTION](#) | [NOTES](#) | [SEE ALSO](#)

sched(7)

Miscellaneous Information Manual

sched(7)

NAME [top](#)

sched - overview of CPU scheduling

DESCRIPTION [top](#)

Since Linux 2.6.23, the default scheduler is CFS, the "Completely Fair Scheduler". The CFS scheduler replaced the earlier "O(1)" scheduler.

API summary

Linux provides the following system calls for controlling the CPU scheduling behavior, policy, and priority of processes (or, more precisely, threads).

[nice\(2\)](#)

Set a new nice value for the calling thread, and return the new nice value.

[getpriority\(2\)](#)

Return the nice value of a thread, a process group, or the set of threads owned by a specified user.

[setpriority\(2\)](#)

Set the nice value of a thread, a process group, or the set of threads owned by a specified user.

[sched_setscheduler\(2\)](#)

Set the scheduling policy and parameters of a specified

thread.

[sched_getscheduler\(2\)](#)

Return the scheduling policy of a specified thread.

[sched_setparam\(2\)](#)

Set the scheduling parameters of a specified thread.

[sched_getparam\(2\)](#)

Fetch the scheduling parameters of a specified thread.

[sched_get_priority_max\(2\)](#)

Return the maximum priority available in a specified scheduling policy.

[sched_get_priority_min\(2\)](#)

Return the minimum priority available in a specified scheduling policy.

[sched_rr_get_interval\(2\)](#)

Fetch the quantum used for threads that are scheduled under the "round-robin" scheduling policy.

[sched_yield\(2\)](#)

Cause the caller to relinquish the CPU, so that some other thread be executed.

[sched_setaffinity\(2\)](#)

(Linux-specific) Set the CPU affinity of a specified thread.

[sched_getaffinity\(2\)](#)

(Linux-specific) Get the CPU affinity of a specified thread.

[sched_setaattr\(2\)](#)

Set the scheduling policy and parameters of a specified thread. This (Linux-specific) system call provides a superset of the functionality of [sched_setscheduler\(2\)](#) and [sched_setparam\(2\)](#).

[sched_getattr\(2\)](#)

Fetch the scheduling policy and parameters of a specified thread. This (Linux-specific) system call provides a



superset of the functionality of `sched_getscheduler(2)` and `sched_getparam(2)`.

Scheduling policies

The scheduler is the kernel component that decides which runnable thread will be executed by the CPU next. Each thread has an associated scheduling policy and a *static* scheduling priority, *sched_priority*. The scheduler makes its decisions based on knowledge of the scheduling policy and static priority of all threads on the system.

For threads scheduled under one of the normal scheduling policies (`SCHED_OTHER`, `SCHED_IDLE`, `SCHED_BATCH`), *sched_priority* is not used in scheduling decisions (it must be specified as 0).

Processes scheduled under one of the real-time policies (`SCHED_FIFO`, `SCHED_RR`) have a *sched_priority* value in the range 1 (low) to 99 (high). (As the numbers imply, real-time threads always have higher priority than normal threads.) Note well: POSIX.1 requires an implementation to support only a minimum 32 distinct priority levels for the real-time policies, and some systems supply just this minimum. Portable programs should use `sched_get_priority_min(2)` and `sched_get_priority_max(2)` to find the range of priorities supported for a particular policy.

Conceptually, the scheduler maintains a list of runnable threads for each possible *sched_priority* value. In order to determine which thread runs next, the scheduler looks for the nonempty list with the highest static priority and selects the thread at the head of this list.

A thread's scheduling policy determines where it will be inserted into the list of threads with equal static priority and how it will move inside this list.

All scheduling is preemptive: if a thread with a higher static priority becomes ready to run, the currently running thread will be preempted and returned to the wait list for its static priority level. The scheduling policy determines the ordering only within the list of runnable threads with equal static priority.

SCHED_FIFO: First in-first out scheduling

SCHED_FIFO can be used only with static priorities higher than 0,



which means that when a **SCHED_FIFO** thread becomes runnable, it will always immediately preempt any currently running **SCHED_OTHER**, **SCHED_BATCH**, or **SCHED_IDLE** thread. **SCHED_FIFO** is a simple scheduling algorithm without time slicing. For threads scheduled under the **SCHED_FIFO** policy, the following rules apply:

- A running **SCHED_FIFO** thread that has been preempted by another thread of higher priority will stay at the head of the list for its priority and will resume execution as soon as all threads of higher priority are blocked again.
- When a blocked **SCHED_FIFO** thread becomes runnable, it will be inserted at the end of the list for its priority.
- If a call to `sched_setscheduler(2)`, `sched_setparam(2)`, `sched_setattr(2)`, `pthread_setschedparam(3)`, or `pthread_setschedprio(3)` changes the priority of the running or runnable **SCHED_FIFO** thread identified by *pid* the effect on the thread's position in the list depends on the direction of the change to threads priority:
 - (a) If the thread's priority is raised, it is placed at the end of the list for its new priority. As a consequence, it may preempt a currently running thread with the same priority.
 - (b) If the thread's priority is unchanged, its position in the run list is unchanged.
 - (c) If the thread's priority is lowered, it is placed at the front of the list for its new priority.

According to POSIX.1-2008, changes to a thread's priority (or policy) using any mechanism other than `pthread_setschedprio(3)` should result in the thread being placed at the end of the list for its priority.

- A thread calling `sched_yield(2)` will be put at the end of the list.

No other events will move a thread scheduled under the **SCHED_FIFO** policy in the wait list of runnable threads with equal static priority.



A **SCHED_FIFO** thread runs until either it is blocked by an I/O request, it is preempted by a higher priority thread, or it calls `sched_yield(2)`.

SCHED_RR: Round-robin scheduling

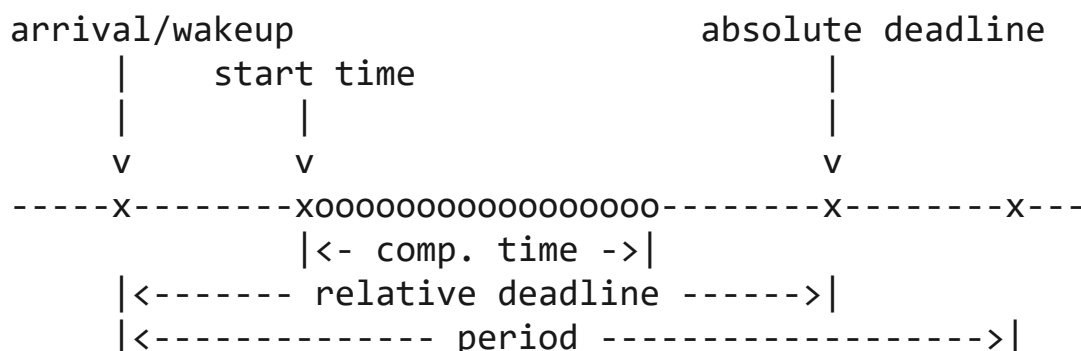
SCHED_RR is a simple enhancement of **SCHED_FIFO**. Everything described above for **SCHED_FIFO** also applies to **SCHED_RR**, except that each thread is allowed to run only for a maximum time quantum. If a **SCHED_RR** thread has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority. A **SCHED_RR** thread that has been preempted by a higher priority thread and subsequently resumes execution as a running thread will complete the unexpired portion of its round-robin time quantum. The length of the time quantum can be retrieved using `sched_rr_get_interval(2)`.

SCHED_DEADLINE: Sporadic task model deadline scheduling

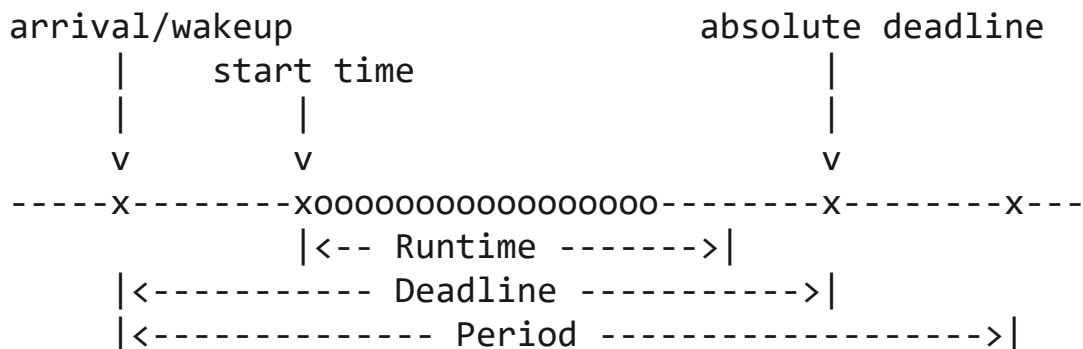
Since Linux 3.14, Linux provides a deadline scheduling policy (**SCHED_DEADLINE**). This policy is currently implemented using GEDF (Global Earliest Deadline First) in conjunction with CBS (Constant Bandwidth Server). To set and fetch this policy and associated attributes, one must use the Linux-specific `sched_setattr(2)` and `sched_getattr(2)` system calls.

A sporadic task is one that has a sequence of jobs, where each job is activated at most once per period. Each job also has a *relative deadline*, before which it should finish execution, and a *computation time*, which is the CPU time necessary for executing the job. The moment when a task wakes up because a new job has to be executed is called the *arrival time* (also referred to as the request time or release time). The *start time* is the time at which a task starts its execution. The *absolute deadline* is thus obtained by adding the relative deadline to the arrival time.

The following diagram clarifies these terms:



When setting a **SCHED_DEADLINE** policy for a thread using `sched_setattr(2)`, one can specify three parameters: *Runtime*, *Deadline*, and *Period*. These parameters do not necessarily correspond to the aforementioned terms: usual practice is to set *Runtime* to something bigger than the average computation time (or worst-case execution time for hard real-time tasks), *Deadline* to the relative deadline, and *Period* to the period of the task. Thus, for **SCHED_DEADLINE** scheduling, we have:



The three deadline-scheduling parameters correspond to the `sched_runtime`, `sched_deadline`, and `sched_period` fields of the `sched_attr` structure; see `sched_setattr(2)`. These fields express values in nanoseconds. If `sched_period` is specified as 0, then it is made the same as `sched_deadline`.

The kernel requires that:

$$\text{sched_runtime} \leq \text{sched_deadline} \leq \text{sched_period}$$

In addition, under the current implementation, all of the parameter values must be at least 1024 (i.e., just over one microsecond, which is the resolution of the implementation), and less than 2^{63} . If any of these checks fails, `sched_setattr(2)` fails with the error **EINVAL**.

The CBS guarantees non-interference between tasks, by throttling threads that attempt to over-run their specified *Runtime*.

To ensure deadline scheduling guarantees, the kernel must prevent situations where the set of **SCHED_DEADLINE** threads is not feasible (schedulable) within the given constraints. The kernel thus performs an admittance test when setting or changing **SCHED_DEADLINE** policy and attributes. This admission test calculates whether the change is feasible; if it is not,



`sched_setattr(2)` fails with the error **EBUSY**.

For example, it is required (but not necessarily sufficient) for the total utilization to be less than or equal to the total number of CPUs available, where, since each thread can maximally run for Runtime per Period, that thread's utilization is its Runtime divided by its Period.

In order to fulfill the guarantees that are made when a thread is admitted to the **SCHED_DEADLINE** policy, **SCHED_DEADLINE** threads are the highest priority (user controllable) threads in the system; if any **SCHED_DEADLINE** thread is runnable, it will preempt any thread scheduled under one of the other policies.

A call to `fork(2)` by a thread scheduled under the **SCHED_DEADLINE** policy fails with the error **EAGAIN**, unless the thread has its reset-on-fork flag set (see below).

A **SCHED_DEADLINE** thread that calls `sched_yield(2)` will yield the current job and wait for a new period to begin.

SCHED_OTHER: Default Linux time-sharing scheduling

SCHED_OTHER can be used at only static priority 0 (i.e., threads under real-time policies always have priority over **SCHED_OTHER** processes). **SCHED_OTHER** is the standard Linux time-sharing scheduler that is intended for all threads that do not require the special real-time mechanisms.

The thread to run is chosen from the static priority 0 list based on a *dynamic* priority that is determined only inside this list. The dynamic priority is based on the nice value (see below) and is increased for each time quantum the thread is ready to run, but denied to run by the scheduler. This ensures fair progress among all **SCHED_OTHER** threads.

In the Linux kernel source code, the **SCHED_OTHER** policy is actually named **SCHED_NORMAL**.

The nice value

The nice value is an attribute that can be used to influence the CPU scheduler to favor or disfavor a process in scheduling decisions. It affects the scheduling of **SCHED_OTHER** and **SCHED_BATCH** (see below) processes. The nice value can be modified using `nice(2)`, `setpriority(2)`, or `sched_setattr(2)`.

According to POSIX.1, the nice value is a per-process attribute; that is, the threads in a process should share a nice value. However, on Linux, the nice value is a per-thread attribute: different threads in the same process may have different nice values.

The range of the nice value varies across UNIX systems. On modern Linux, the range is -20 (high priority) to +19 (low priority). On some other systems, the range is -20..20. Very early Linux kernels (before Linux 2.0) had the range -infinity..15.

The degree to which the nice value affects the relative scheduling of **SCHED_OTHER** processes likewise varies across UNIX systems and across Linux kernel versions.

With the advent of the CFS scheduler in Linux 2.6.23, Linux adopted an algorithm that causes relative differences in nice values to have a much stronger effect. In the current implementation, each unit of difference in the nice values of two processes results in a factor of 1.25 in the degree to which the scheduler favors the higher priority process. This causes very low nice values (+19) to truly provide little CPU to a process whenever there is any other higher priority load on the system, and makes high nice values (-20) deliver most of the CPU to applications that require it (e.g., some audio applications).

On Linux, the **RLIMIT_NICE** resource limit can be used to define a limit to which an unprivileged process's nice value can be raised; see [setrlimit\(2\)](#) for details.

For further details on the nice value, see the subsections on the autogroup feature and group scheduling, below.

SCHED_BATCH: Scheduling batch processes

(Since Linux 2.6.16.) **SCHED_BATCH** can be used only at static priority 0. This policy is similar to **SCHED_OTHER** in that it schedules the thread according to its dynamic priority (based on the nice value). The difference is that this policy will cause the scheduler to always assume that the thread is CPU-intensive. Consequently, the scheduler will apply a small scheduling penalty with respect to wakeup behavior, so that this thread is mildly disfavored in scheduling decisions.



This policy is useful for workloads that are noninteractive, but do not want to lower their nice value, and for workloads that want a deterministic scheduling policy without interactivity causing extra preemptions (between the workload's tasks).

SCHED_IDLE: Scheduling very low priority jobs

(Since Linux 2.6.23.) **SCHED_IDLE** can be used only at static priority 0; the process nice value has no influence for this policy.

This policy is intended for running jobs at extremely low priority (lower even than a +19 nice value with the **SCHED_OTHER** or **SCHED_BATCH** policies).

Resetting scheduling policy for child processes

Each thread has a reset-on-fork scheduling flag. When this flag is set, children created by `fork(2)` do not inherit privileged scheduling policies. The reset-on-fork flag can be set by either:

- ORing the **SCHED_RESET_ON_FORK** flag into the *policy* argument when calling `sched_setscheduler(2)` (since Linux 2.6.32); or
- specifying the **SCHED_FLAG_RESET_ON_FORK** flag in *attr.sched_flags* when calling `sched_setattr(2)`.

Note that the constants used with these two APIs have different names. The state of the reset-on-fork flag can analogously be retrieved using `sched_getscheduler(2)` and `sched_getattr(2)`.

The reset-on-fork feature is intended for media-playback applications, and can be used to prevent applications evading the **RLIMIT_RTTIME** resource limit (see `getrlimit(2)`) by creating multiple child processes.

More precisely, if the reset-on-fork flag is set, the following rules apply for subsequently created children:

- If the calling thread has a scheduling policy of **SCHED_FIFO** or **SCHED_RR**, the policy is reset to **SCHED_OTHER** in child processes.
- If the calling process has a negative nice value, the nice



value is reset to zero in child processes.

After the reset-on-fork flag has been enabled, it can be reset only if the thread has the **CAP_SYS_NICE** capability. This flag is disabled in child processes created by `fork(2)`.

Privileges and resource limits

Before Linux 2.6.12, only privileged (**CAP_SYS_NICE**) threads can set a nonzero static priority (i.e., set a real-time scheduling policy). The only change that an unprivileged thread can make is to set the **SCHED_OTHER** policy, and this can be done only if the effective user ID of the caller matches the real or effective user ID of the target thread (i.e., the thread specified by *pid*) whose policy is being changed.

A thread must be privileged (**CAP_SYS_NICE**) in order to set or modify a **SCHED_DEADLINE** policy.

Since Linux 2.6.12, the **RLIMIT_RTPRIO** resource limit defines a ceiling on an unprivileged thread's static priority for the **SCHED_RR** and **SCHED_FIFO** policies. The rules for changing scheduling policy and priority are as follows:

- If an unprivileged thread has a nonzero **RLIMIT_RTPRIO** soft limit, then it can change its scheduling policy and priority, subject to the restriction that the priority cannot be set to a value higher than the maximum of its current priority and its **RLIMIT_RTPRIO** soft limit.
- If the **RLIMIT_RTPRIO** soft limit is 0, then the only permitted changes are to lower the priority, or to switch to a non-real-time policy.
- Subject to the same rules, another unprivileged thread can also make these changes, as long as the effective user ID of the thread making the change matches the real or effective user ID of the target thread.
- Special rules apply for the **SCHED_IDLE** policy. Before Linux 2.6.39, an unprivileged thread operating under this policy cannot change its policy, regardless of the value of its **RLIMIT_RTPRIO** resource limit. Since Linux 2.6.39, an unprivileged thread can switch to either the **SCHED_BATCH** or the **SCHED_OTHER** policy so long as its nice value falls within



the range permitted by its **RLIMIT_NICE** resource limit (see [getrlimit\(2\)](#)).

Privileged (**CAP_SYS_NICE**) threads ignore the **RLIMIT_RTPRIO** limit; as with older kernels, they can make arbitrary changes to scheduling policy and priority. See [getrlimit\(2\)](#) for further information on **RLIMIT_RTPRIO**.

Limiting the CPU usage of real-time and deadline processes

A nonblocking infinite loop in a thread scheduled under the **SCHED_FIFO**, **SCHED_RR**, or **SCHED_DEADLINE** policy can potentially block all other threads from accessing the CPU forever. Before Linux 2.6.25, the only way of preventing a runaway real-time process from freezing the system was to run (at the console) a shell scheduled under a higher static priority than the tested application. This allows an emergency kill of tested real-time applications that do not block or terminate as expected.

Since Linux 2.6.25, there are other techniques for dealing with runaway real-time and deadline processes. One of these is to use the **RLIMIT_RTTIME** resource limit to set a ceiling on the CPU time that a real-time process may consume. See [getrlimit\(2\)](#) for details.

Since Linux 2.6.25, Linux also provides two */proc* files that can be used to reserve a certain amount of CPU time to be used by non-real-time processes. Reserving CPU time in this fashion allows some CPU time to be allocated to (say) a root shell that can be used to kill a runaway process. Both of these files specify time values in microseconds:

/proc/sys/kernel/sched_rt_period_us

This file specifies a scheduling period that is equivalent to 100% CPU bandwidth. The value in this file can range from 1 to **INT_MAX**, giving an operating range of 1 microsecond to around 35 minutes. The default value in this file is 1,000,000 (1 second).

/proc/sys/kernel/sched_rt_runtime_us

The value in this file specifies how much of the "period" time can be used by all real-time and deadline scheduled processes on the system. The value in this file can range from -1 to **INT_MAX-1**. Specifying -1 makes the run time the same as the period; that is, no CPU time is set aside



for non-real-time processes (which was the behavior before Linux 2.6.25). The default value in this file is 950,000 (0.95 seconds), meaning that 5% of the CPU time is reserved for processes that don't run under a real-time or deadline scheduling policy.

Response time

A blocked high priority thread waiting for I/O has a certain response time before it is scheduled again. The device driver writer can greatly reduce this response time by using a "slow interrupt" interrupt handler.

Miscellaneous

Child processes inherit the scheduling policy and parameters across a `fork(2)`. The scheduling policy and parameters are preserved across `execve(2)`.

Memory locking is usually needed for real-time processes to avoid paging delays; this can be done with `mlock(2)` or `mlockall(2)`.

The autogroup feature

Since Linux 2.6.38, the kernel provides a feature known as autogrouping to improve interactive desktop performance in the face of multiprocess, CPU-intensive workloads such as building the Linux kernel with large numbers of parallel build processes (i.e., the `make(1) -j` flag).

This feature operates in conjunction with the CFS scheduler and requires a kernel that is configured with **CONFIG_SCHED_AUTOGROUP**. On a running system, this feature is enabled or disabled via the file `/proc/sys/kernel/sched_autogroup_enabled`; a value of 0 disables the feature, while a value of 1 enables it. The default value in this file is 1, unless the kernel was booted with the `noautogroup` parameter.

A new autogroup is created when a new session is created via `setsid(2)`; this happens, for example, when a new terminal window is started. A new process created by `fork(2)` inherits its parent's autogroup membership. Thus, all of the processes in a session are members of the same autogroup. An autogroup is automatically destroyed when the last process in the group terminates.

When autogrouping is enabled, all of the members of an autogroup



are placed in the same kernel scheduler "task group". The CFS scheduler employs an algorithm that equalizes the distribution of CPU cycles across task groups. The benefits of this for interactive desktop performance can be described via the following example.

Suppose that there are two autogroups competing for the same CPU (i.e., presume either a single CPU system or the use of `taskset(1)` to confine all the processes to the same CPU on an SMP system). The first group contains ten CPU-bound processes from a kernel build started with `make -j10`. The other contains a single CPU-bound process: a video player. The effect of autogrouping is that the two groups will each receive half of the CPU cycles. That is, the video player will receive 50% of the CPU cycles, rather than just 9% of the cycles, which would likely lead to degraded video playback. The situation on an SMP system is more complex, but the general effect is the same: the scheduler distributes CPU cycles across task groups such that an autogroup that contains a large number of CPU-bound processes does not end up hogging CPU cycles at the expense of the other jobs on the system.

A process's autogroup (task group) membership can be viewed via the file `/proc/pid/autogroup`:

```
$ cat /proc/1/autogroup
/autogroup-1 nice 0
```

This file can also be used to modify the CPU bandwidth allocated to an autogroup. This is done by writing a number in the "nice" range to the file to set the autogroup's nice value. The allowed range is from +19 (low priority) to -20 (high priority). (Writing values outside of this range causes `write(2)` to fail with the error **EINVAL**.)

The autogroup nice setting has the same meaning as the process nice value, but applies to distribution of CPU cycles to the autogroup as a whole, based on the relative nice values of other autogroups. For a process inside an autogroup, the CPU cycles that it receives will be a product of the autogroup's nice value (compared to other autogroups) and the process's nice value (compared to other processes in the same autogroup).

The use of the `cgroups(7)` CPU controller to place processes in



cgroups other than the root CPU cgroup overrides the effect of autogrouping.

The autogroup feature groups only processes scheduled under non-real-time policies (**SCHED_OTHER**, **SCHED_BATCH**, and **SCHED_IDLE**). It does not group processes scheduled under real-time and deadline policies. Those processes are scheduled according to the rules described earlier.

The nice value and group scheduling

When scheduling non-real-time processes (i.e., those scheduled under the **SCHED_OTHER**, **SCHED_BATCH**, and **SCHED_IDLE** policies), the CFS scheduler employs a technique known as "group scheduling", if the kernel was configured with the **CONFIG_FAIR_GROUP_SCHED** option (which is typical).

Under group scheduling, threads are scheduled in "task groups". Task groups have a hierarchical relationship, rooted under the initial task group on the system, known as the "root task group". Task groups are formed in the following circumstances:

- All of the threads in a CPU cgroup form a task group. The parent of this task group is the task group of the corresponding parent cgroup.
- If autogrouping is enabled, then all of the threads that are (implicitly) placed in an autogroup (i.e., the same session, as created by `setsid(2)`) form a task group. Each new autogroup is thus a separate task group. The root task group is the parent of all such autogroups.
- If autogrouping is enabled, then the root task group consists of all processes in the root CPU cgroup that were not otherwise implicitly placed into a new autogroup.
- If autogrouping is disabled, then the root task group consists of all processes in the root CPU cgroup.
- If group scheduling was disabled (i.e., the kernel was configured without **CONFIG_FAIR_GROUP_SCHED**), then all of the processes on the system are notionally placed in a single task group.

Under group scheduling, a thread's nice value has an effect for



scheduling decisions *only relative to other threads in the same task group*. This has some surprising consequences in terms of the traditional semantics of the nice value on UNIX systems. In particular, if autogrouping is enabled (which is the default in various distributions), then employing `setpriority(2)` or `nice(1)` on a process has an effect only for scheduling relative to other processes executed in the same session (typically: the same terminal window).

Conversely, for two processes that are (for example) the sole CPU-bound processes in different sessions (e.g., different terminal windows, each of whose jobs are tied to different autogroups), *modifying the nice value of the process in one of the sessions has no effect* in terms of the scheduler's decisions relative to the process in the other session. A possibly useful workaround here is to use a command such as the following to modify the autogroup nice value for *all* of the processes in a terminal session:

```
$ echo 10 > /proc/self/autogroup
```

Real-time features in the mainline Linux kernel

Since Linux 2.6.18, Linux is gradually becoming equipped with real-time capabilities, most of which are derived from the former *realtime-preempt* patch set. Until the patches have been completely merged into the mainline kernel, they must be installed to achieve the best real-time performance. These patches are named:

```
patch-kernelversion-rtpatchversion
```

and can be downloaded from
(<http://www.kernel.org/pub/linux/kernel/projects/rt/>).

Without the patches and prior to their full inclusion into the mainline kernel, the kernel configuration offers only the three preemption classes `CONFIG_PREEMPT_NONE`, `CONFIG_PREEMPT_VOLUNTARY`, and `CONFIG_PREEMPT_DESKTOP` which respectively provide no, some, and considerable reduction of the worst-case scheduling latency.

With the patches applied or after their full inclusion into the mainline kernel, the additional configuration item `CONFIG_PREEMPT_RT` becomes available. If this is selected, Linux is transformed into a regular real-time operating system. The



FIFO and RR scheduling policies are then used to run a thread with true real-time priority and a minimum worst-case scheduling latency.

NOTES [top](#)

The [cgroups\(7\)](#) CPU controller can be used to limit the CPU consumption of groups of processes.

Originally, Standard Linux was intended as a general-purpose operating system being able to handle background processes, interactive applications, and less demanding real-time applications (applications that need to usually meet timing deadlines). Although the Linux 2.6 allowed for kernel preemption and the newly introduced O(1) scheduler ensures that the time needed to schedule is fixed and deterministic irrespective of the number of active tasks, true real-time computing was not possible up to Linux 2.6.17.

SEE ALSO [top](#)

[chcpu\(1\)](#), [chrt\(1\)](#), [lscpu\(1\)](#), [ps\(1\)](#), [taskset\(1\)](#), [top\(1\)](#), [getpriority\(2\)](#), [mlock\(2\)](#), [mlockall\(2\)](#), [munlock\(2\)](#), [munlockall\(2\)](#), [nice\(2\)](#), [sched_get_priority_max\(2\)](#), [sched_get_priority_min\(2\)](#), [sched_getaffinity\(2\)](#), [sched_getparam\(2\)](#), [sched_getscheduler\(2\)](#), [sched_rr_get_interval\(2\)](#), [sched_setaffinity\(2\)](#), [sched_setparam\(2\)](#), [sched_setscheduler\(2\)](#), [sched_yield\(2\)](#), [setpriority\(2\)](#), [pthread_getaffinity_np\(3\)](#), [pthread_getschedparam\(3\)](#), [pthread_setaffinity_np\(3\)](#), [sched_getcpu\(3\)](#), [capabilities\(7\)](#), [cpuset\(7\)](#)

Programming for the real world - POSIX.4 by Bill O. Gallmeister, O'Reilly & Associates, Inc., ISBN 1-56592-074-0.

The Linux kernel source files [Documentation/scheduler/sched-deadline.txt](#), [Documentation/scheduler/sched-rt-group.txt](#), [Documentation/scheduler/sched-design-CFS.txt](#), and [Documentation/scheduler/sched-nice-design.txt](#)

Linux man-pages (unreleased) (date) [sched\(7\)](#)

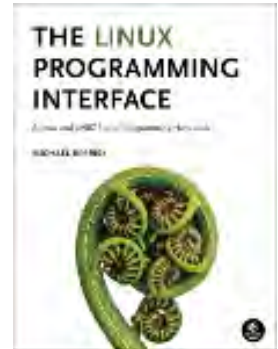


Pages that refer to this page: [chrt\(1\)](#), [renice\(1\)](#), [taskset\(1\)](#), [uclampset\(1\)](#), [fork\(2\)](#), [futex\(2\)](#), [getpriority\(2\)](#), [getrlimit\(2\)](#), [nice\(2\)](#), [sched_get_priority_max\(2\)](#), [sched_rr_get_interval\(2\)](#), [sched_setaffinity\(2\)](#), [sched_setattr\(2\)](#), [sched_setparam\(2\)](#), [sched_setscheduler\(2\)](#), [sched_yield\(2\)](#), [setsid\(2\)](#), [pthread_attr_setinheritsched\(3\)](#), [pthread_attr_setschedparam\(3\)](#), [pthread_attr_setschedpolicy\(3\)](#), [pthread_setaffinity_np\(3\)](#), [pthread_setschedparam\(3\)](#), [pthread_setschedprio\(3\)](#), [pthread_yield\(3\)](#), [sched_getcpu\(3\)](#), [proc\(5\)](#), [systemd.exec\(5\)](#), [systemd.resource-control\(5\)](#), [capabilities\(7\)](#), [cgroups\(7\)](#), [cpuset\(7\)](#), [threads\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sched_setscheduler(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

 [sched_setscheduler\(2\)](#)

System Calls Manual

[sched_setscheduler\(2\)](#)

NAME [top](#)

`sched_setscheduler`, `sched_getscheduler` - set and get scheduling policy/parameters

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sched.h>
```

```
int sched_setscheduler(pid_t pid, int policy,  
                      const struct sched_param *param);  
int sched_getscheduler(pid_t pid);
```

DESCRIPTION [top](#)

The `sched_setscheduler()` system call sets both the scheduling policy and parameters for the thread whose ID is specified in *pid*. If *pid* equals zero, the scheduling policy and parameters of the calling thread will be set.

The scheduling parameters are specified in the *param* argument,

which is a pointer to a structure of the following form:

```
struct sched_param {  
    ...  
    int sched_priority;  
    ...  
};
```

In the current implementation, the structure contains only one field, *sched_priority*. The interpretation of *param* depends on the selected policy.

Currently, Linux supports the following "normal" (i.e., non-real-time) scheduling policies as values that may be specified in *policy*:

SCHED_OTHER

the standard round-robin time-sharing policy;

SCHED_BATCH

for "batch" style execution of processes; and

SCHED_IDLE

for running *very* low priority background jobs.

For each of the above policies, *param->sched_priority* must be 0.

Various "real-time" policies are also supported, for special time-critical applications that need precise control over the way in which runnable threads are selected for execution. For the rules governing when a process may use these policies, see [sched\(7\)](#). The real-time policies that may be specified in *policy* are:

SCHED_FIFO

a first-in, first-out policy; and

SCHED_RR

a round-robin policy.

For each of the above policies, *param->sched_priority* specifies a scheduling priority for the thread. This is a number in the range returned by calling [sched_get_priority_min\(2\)](#) and [sched_get_priority_max\(2\)](#) with the specified *policy*. On Linux,



these system calls return, respectively, 1 and 99.

Since Linux 2.6.32, the **SCHED_RESET_ON_FORK** flag can be ORed in *policy* when calling **sched_setscheduler()**. As a result of including this flag, children created by **fork(2)** do not inherit privileged scheduling policies. See **sched(7)** for details.

sched_getscheduler() returns the current scheduling policy of the thread identified by *pid*. If *pid* equals zero, the policy of the calling thread will be retrieved.

RETURN VALUE [top](#)

On success, **sched_setscheduler()** returns zero. On success, **sched_getscheduler()** returns the policy for the thread (a nonnegative integer). On error, both calls return -1, and *errno* is set to indicate the error.

ERRORS [top](#)

EINVAL Invalid arguments: *pid* is negative or *param* is NULL.

EINVAL (**sched_setscheduler()**) *policy* is not one of the recognized policies.

EINVAL (**sched_setscheduler()**) *param* does not make sense for the specified *policy*.

EPERM The calling thread does not have appropriate privileges.

ESRCH The thread whose ID is *pid* could not be found.

VERSIONS [top](#)

POSIX.1 does not detail the permissions that an unprivileged thread requires in order to call **sched_setscheduler()**, and details vary across systems. For example, the Solaris 7 manual page says that the real or effective user ID of the caller must match the real user ID or the save set-user-ID of the target.

The scheduling policy and parameters are in fact per-thread

attributes on Linux. The value returned from a call to [gettid\(2\)](#) can be passed in the argument *pid*. Specifying *pid* as 0 will operate on the attributes of the calling thread, and passing the value returned from a call to [getpid\(2\)](#) will operate on the attributes of the main thread of the thread group. (If you are using the POSIX threads API, then use [pthread_setschedparam\(3\)](#), [pthread_getschedparam\(3\)](#), and [pthread_setschedprio\(3\)](#), instead of the `sched_*(2)` system calls.)

STANDARDS [top](#)

POSIX.1-2008 (but see BUGS below).

`SCHED_BATCH` and `SCHED_IDLE` are Linux-specific.

HISTORY [top](#)

POSIX.1-2001.

NOTES [top](#)

Further details of the semantics of all of the above "normal" and "real-time" scheduling policies can be found in the [sched\(7\)](#) manual page. That page also describes an additional policy, `SCHED_DEADLINE`, which is settable only via [sched_setattr\(2\)](#).

POSIX systems on which `sched_setscheduler()` and `sched_getscheduler()` are available define `_POSIX_PRIORITY_SCHEDULING` in `<unistd.h>`.

BUGS [top](#)

POSIX.1 says that on success, `sched_setscheduler()` should return the previous scheduling policy. Linux `sched_setscheduler()` does not conform to this requirement, since it always returns 0 on success.

SEE ALSO [top](#)

[chrt\(1\)](#), [nice\(2\)](#), [sched_get_priority_max\(2\)](#),
[sched_get_priority_min\(2\)](#), [sched_getaffinity\(2\)](#),
[sched_getattr\(2\)](#), [sched_getparam\(2\)](#), [sched_rr_get_interval\(2\)](#),
[sched_setaffinity\(2\)](#), [sched_setattr\(2\)](#), [sched_setparam\(2\)](#),
[sched_yield\(2\)](#), [setpriority\(2\)](#), [capabilities\(7\)](#), [cpuset\(7\)](#),
[sched\(7\)](#)

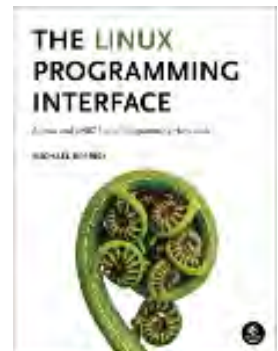
Linux man-pages (unreleased) **(date)** ***sched_setscheduler(2)***

Pages that refer to this page: [chrt\(1\)](#), [uclampset\(1\)](#), [getrlimit\(2\)](#), [gettid\(2\)](#), [mlock\(2\)](#),
[nanosleep\(2\)](#), [prctl\(2\)](#), [sched_get_priority_max\(2\)](#), [sched_setaffinity\(2\)](#),
[sched_setattr\(2\)](#), [sched_setparam\(2\)](#), [syscalls\(2\)](#), [id_t\(3type\)](#), [posix_spawn\(3\)](#),
[proc\(5\)](#), [systemd.exec\(5\)](#), [capabilities\(7\)](#), [cpuset\(7\)](#), [credentials\(7\)](#), [sched\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of
[The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sched_setparam(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 [sched_setparam\(2\)](#)

System Calls Manual

[sched_setparam\(2\)](#)

NAME [top](#)

`sched_setparam`, `sched_getparam` - set and get scheduling parameters

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sched.h>
```

```
int sched_setparam(pid_t pid, const struct sched_param *param);  
int sched_getparam(pid_t pid, struct sched_param *param);
```

```
struct sched_param {  
    ...  
    int sched_priority;  
    ...  
};
```

DESCRIPTION [top](#)

`sched_setparam()` sets the scheduling parameters associated with the scheduling policy for the thread whose thread ID is specified



in *pid*. If *pid* is zero, then the parameters of the calling thread are set. The interpretation of the argument *param* depends on the scheduling policy of the thread identified by *pid*. See [sched\(7\)](#) for a description of the scheduling policies supported under Linux.

sched_getparam() retrieves the scheduling parameters for the thread identified by *pid*. If *pid* is zero, then the parameters of the calling thread are retrieved.

sched_setparam() checks the validity of *param* for the scheduling policy of the thread. The value *param->sched_priority* must lie within the range given by [sched_get_priority_min\(2\)](#) and [sched_get_priority_max\(2\)](#).

For a discussion of the privileges and resource limits related to scheduling priority and policy, see [sched\(7\)](#).

POSIX systems on which **sched_setparam()** and **sched_getparam()** are available define **_POSIX_PRIORITY_SCHEDULING** in [<unistd.h>](#).

RETURN VALUE [top](#)

On success, **sched_setparam()** and **sched_getparam()** return 0. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EINVAL Invalid arguments: *param* is NULL or *pid* is negative

EINVAL (**sched_setparam()**) The argument *param* does not make sense for the current scheduling policy.

EPERM (**sched_setparam()**) The caller does not have appropriate privileges (Linux: does not have the **CAP_SYS_NICE** capability).

ESRCH The thread whose ID is *pid* could not be found.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

SEE ALSO [top](#)

[getpriority\(2\)](#), [gettid\(2\)](#), [nice\(2\)](#), [sched_get_priority_max\(2\)](#), [sched_get_priority_min\(2\)](#), [sched_getaffinity\(2\)](#), [sched_getscheduler\(2\)](#), [sched_setaffinity\(2\)](#), [sched_setattr\(2\)](#), [sched_setscheduler\(2\)](#), [setpriority\(2\)](#), [capabilities\(7\)](#), [sched\(7\)](#)

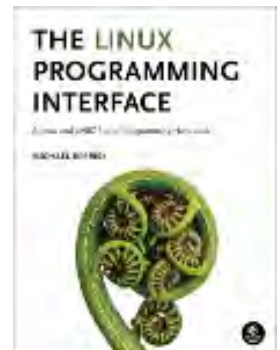
Linux man-pages (unreleased) (date) [sched_setparam\(2\)](#)

Pages that refer to this page: [getrlimit\(2\)](#), [gettid\(2\)](#), [sched_get_priority_max\(2\)](#), [sched_setattr\(2\)](#), [sched_setscheduler\(2\)](#), [syscalls\(2\)](#), [posix_spawn\(3\)](#), [capabilities\(7\)](#), [credentials\(7\)](#), [sched\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sched_get_priority_max(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

sched_get_priority_max(2) System Calls Manual *sched_get_priority_max(2)*

NAME [top](#)

`sched_get_priority_max`, `sched_get_priority_min` - get static priority range

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sched.h>
```

```
int sched_get_priority_max(int policy);  
int sched_get_priority_min(int policy);
```

DESCRIPTION [top](#)

`sched_get_priority_max()` returns the maximum priority value that can be used with the scheduling algorithm identified by *policy*. `sched_get_priority_min()` returns the minimum priority value that can be used with the scheduling algorithm identified by *policy*. Supported *policy* values are **SCHED_FIFO**, **SCHED_RR**, **SCHED_OTHER**, **SCHED_BATCH**, **SCHED_IDLE**, and **SCHED_DEADLINE**. Further details about these policies can be found in [sched\(7\)](#).

Processes with numerically higher priority values are scheduled before processes with numerically lower priority values. Thus, the value returned by `sched_get_priority_max()` will be greater than the value returned by `sched_get_priority_min()`.

Linux allows the static priority range 1 to 99 for the **SCHED_FIFO** and **SCHED_RR** policies, and the priority 0 for the remaining policies. Scheduling priority ranges for the various policies are not alterable.

The range of scheduling priorities may vary on other POSIX systems, thus it is a good idea for portable applications to use a virtual priority range and map it to the interval given by `sched_get_priority_max()` and `sched_get_priority_min()` POSIX.1 requires a spread of at least 32 between the maximum and the minimum values for **SCHED_FIFO** and **SCHED_RR**.

POSIX systems on which `sched_get_priority_max()` and `sched_get_priority_min()` are available define `_POSIX_PRIORITY_SCHEDULING` in `<unistd.h>`.

RETURN VALUE [top](#)

On success, `sched_get_priority_max()` and `sched_get_priority_min()` return the maximum/minimum priority value for the named scheduling policy. On error, -1 is returned, and `errno` is set to indicate the error.

ERRORS [top](#)

EINVAL The argument `policy` does not identify a defined scheduling policy.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

SEE ALSO [top](#)

[sched_getaffinity\(2\)](#), [sched_getparam\(2\)](#), [sched_getscheduler\(2\)](#),
[sched_setaffinity\(2\)](#), [sched_setparam\(2\)](#), [sched_setscheduler\(2\)](#),
[sched\(7\)](#)

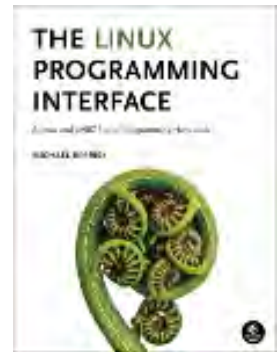
Linux man-pages (unreleased) **(date)** *[sched_get_priority_max\(2\)](#)*

Pages that refer to this page: [sched_setaffinity\(2\)](#), [sched_setattr\(2\)](#),
[sched_setparam\(2\)](#), [sched_setscheduler\(2\)](#), [syscalls\(2\)](#),
[pthread_attr_setschedparam\(3\)](#), [pthread_setschedparam\(3\)](#),
[pthread_setschedprio\(3\)](#), [sched\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of
[The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sched_rr_get_interval(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

[sched_rr_get_interval\(2\)](#) System Calls Manual [sched_rr_get_interval\(2\)](#)

NAME [top](#)

`sched_rr_get_interval` - get the SCHED_RR interval for the named process

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sched.h>
```

```
int sched_rr_get_interval(pid_t pid, struct timespec *tp);
```

DESCRIPTION [top](#)

`sched_rr_get_interval()` writes into the `timespec(3)` structure pointed to by `tp` the round-robin time quantum for the process identified by `pid`. The specified process should be running under the **SCHED_RR** scheduling policy.

If `pid` is zero, the time quantum for the calling process is written into `*tp`.

RETURN VALUE [top](#)

On success, `sched_rr_get_interval()` returns 0. On error, -1 is returned, and `errno` is set to indicate the error.

ERRORS [top](#)

EFAULT Problem with copying information to user space.

EINVAL Invalid pid.

ENOSYS The system call is not yet implemented (only on rather old kernels).

ESRCH Could not find a process with the ID *pid*.

VERSIONS [top](#)

Linux

Linux 3.9 added a new mechanism for adjusting (and viewing) the **SCHED_RR** quantum: the `/proc/sys/kernel/sched_rr_timeslice_ms` file exposes the quantum as a millisecond value, whose default is 100. Writing 0 to this file resets the quantum to the default value.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

Linux

POSIX does not specify any mechanism for controlling the size of the round-robin time quantum. Older Linux kernels provide a (nonportable) method of doing this. The quantum can be controlled by adjusting the process's nice value (see `setpriority(2)`). Assigning a negative (i.e., high) nice value results in a longer quantum; assigning a positive (i.e., low) nice value results in a shorter quantum. The default quantum is

0.1 seconds; the degree to which changing the nice value affects the quantum has varied somewhat across kernel versions. This method of adjusting the quantum was removed starting with Linux 2.6.24.

NOTES [top](#)

POSIX systems on which `sched_rr_get_interval()` is available define `_POSIX_PRIORITY_SCHEDULING` in `<unistd.h>`.

SEE ALSO [top](#)

`timespec(3)`, `sched(7)`

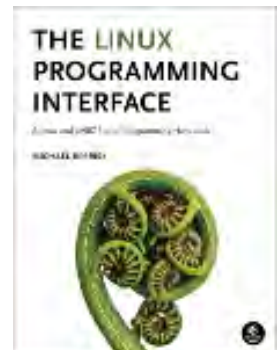
Linux man-pages (unreleased) (date) [sched_rr_get_interval\(2\)](#)

Pages that refer to this page: [sched_setattr\(2\)](#), [sched_setscheduler\(2\)](#), [syscalls\(2\)](#), [proc\(5\)](#), [sched\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



getrlimit(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [EXAMPLES](#) | [SEE ALSO](#)

 [getrlimit\(2\)](#)

System Calls Manual

[getrlimit\(2\)](#)

NAME [top](#)

getrlimit, setrlimit, prlimit - get/set resource limits

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);

int prlimit(pid_t pid, int resource,
            const struct rlimit *_Nullable new_limit,
            struct rlimit *_Nullable old_limit);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
prlimit():
    _GNU_SOURCE
```

DESCRIPTION [top](#)

The **getrlimit()** and **setrlimit()** system calls get and set resource limits. Each resource has an associated soft and hard limit, as defined by the *rlimit* structure:

```
struct rlimit {
    rlim_t rlim_cur; /* Soft limit */
    rlim_t rlim_max; /* Hard limit (ceiling for rlim_cur) */
};
```

The soft limit is the value that the kernel enforces for the corresponding resource. The hard limit acts as a ceiling for the soft limit: an unprivileged process may set only its soft limit to a value in the range from 0 up to the hard limit, and (irreversibly) lower its hard limit. A privileged process (under Linux: one with the **CAP_SYS_RESOURCE** capability in the initial user namespace) may make arbitrary changes to either limit value.

The value **RLIM_INFINITY** denotes no limit on a resource (both in the structure returned by **getrlimit()** and in the structure passed to **setrlimit()**).

The *resource* argument must be one of:

RLIMIT_AS

This is the maximum size of the process's virtual memory (address space). The limit is specified in bytes, and is rounded down to the system page size. This limit affects calls to **brk(2)**, **mmap(2)**, and **mremap(2)**, which fail with the error **ENOMEM** upon exceeding this limit. In addition, automatic stack expansion fails (and generates a **SIGSEGV** that kills the process if no alternate stack has been made available via **sigaltstack(2)**). Since the value is a *long*, on machines with a 32-bit *long* either this limit is at most 2 GiB, or this resource is unlimited.

RLIMIT_CORE

This is the maximum size of a *core* file (see **core(5)**) in bytes that the process may dump. When 0 no core dump files are created. When nonzero, larger dumps are truncated to this size.

RLIMIT_CPU

This is a limit, in seconds, on the amount of CPU time that the process can consume. When the process reaches the soft limit, it is sent a **SIGXCPU** signal. The default



action for this signal is to terminate the process. However, the signal can be caught, and the handler can return control to the main program. If the process continues to consume CPU time, it will be sent **SIGXCPU** once per second until the hard limit is reached, at which time it is sent **SIGKILL**. (This latter point describes Linux behavior. Implementations vary in how they treat processes which continue to consume CPU time after reaching the soft limit. Portable applications that need to catch this signal should perform an orderly termination upon first receipt of **SIGXCPU**.)

RLIMIT_DATA

This is the maximum size of the process's data segment (initialized data, uninitialized data, and heap). The limit is specified in bytes, and is rounded down to the system page size. This limit affects calls to `brk(2)`, `sbrk(2)`, and (since Linux 4.7) `mmap(2)`, which fail with the error **ENOMEM** upon encountering the soft limit of this resource.

RLIMIT_FSIZE

This is the maximum size in bytes of files that the process may create. Attempts to extend a file beyond this limit result in delivery of a **SIGXFSZ** signal. By default, this signal terminates a process, but a process can catch this signal instead, in which case the relevant system call (e.g., `write(2)`, `truncate(2)`) fails with the error **EFBIG**.

RLIMIT_LOCKS (Linux 2.4.0 to Linux 2.4.24)

This is a limit on the combined number of `flock(2)` locks and `fcntl(2)` leases that this process may establish.

RLIMIT_MEMLOCK

This is the maximum number of bytes of memory that may be locked into RAM. This limit is in effect rounded down to the nearest multiple of the system page size. This limit affects `mlock(2)`, `mlockall(2)`, and the `mmap(2)` **MAP_LOCKED** operation. Since Linux 2.6.9, it also affects the `shmctl(2)` **SHM_LOCK** operation, where it sets a maximum on the total bytes in shared memory segments (see `shmget(2)`) that may be locked by the real user ID of the calling process. The `shmctl(2)` **SHM_LOCK** locks are accounted for separately from the per-process memory locks established



by `mlock(2)`, `mlockall(2)`, and `mmap(2) MAP_LOCKED`; a process can lock bytes up to this limit in each of these two categories.

Before Linux 2.6.9, this limit controlled the amount of memory that could be locked by a privileged process. Since Linux 2.6.9, no limits are placed on the amount of memory that a privileged process may lock, and this limit instead governs the amount of memory that an unprivileged process may lock.

RLIMIT_MSGQUEUE (since Linux 2.6.8)

This is a limit on the number of bytes that can be allocated for POSIX message queues for the real user ID of the calling process. This limit is enforced for `mq_open(3)`. Each message queue that the user creates counts (until it is removed) against this limit according to the formula:

Since Linux 3.5:

```
bytes = attr.mq_maxmsg * sizeof(struct msg_msg) +
        MIN(attr.mq_maxmsg, MQ_PRIO_MAX) *
            sizeof(struct posix_msg_tree_node)+
            /* For overhead */
        attr.mq_maxmsg * attr.mq_msgsize;
            /* For message data */
```

Linux 3.4 and earlier:

```
bytes = attr.mq_maxmsg * sizeof(struct msg_msg *) +
            /* For overhead */
        attr.mq_maxmsg * attr.mq_msgsize;
            /* For message data */
```

where *attr* is the `mq_attr` structure specified as the fourth argument to `mq_open(3)`, and the `msg_msg` and `posix_msg_tree_node` structures are kernel-internal structures.

The "overhead" addend in the formula accounts for overhead bytes required by the implementation and ensures that the user cannot create an unlimited number of zero-length messages (such messages nevertheless each consume some system memory for bookkeeping overhead).



RLIMIT_NICE (since Linux 2.6.12, but see BUGS below)

This specifies a ceiling to which the process's nice value can be raised using `setpriority(2)` or `nice(2)`. The actual ceiling for the nice value is calculated as $20 - rlim_cur$. The useful range for this limit is thus from 1 (corresponding to a nice value of 19) to 40 (corresponding to a nice value of -20). This unusual choice of range was necessary because negative numbers cannot be specified as resource limit values, since they typically have special meanings. For example, **RLIM_INFINITY** typically is the same as -1. For more detail on the nice value, see [sched\(7\)](#).

RLIMIT_NOFILE

This specifies a value one greater than the maximum file descriptor number that can be opened by this process. Attempts (`open(2)`, `pipe(2)`, `dup(2)`, etc.) to exceed this limit yield the error **EMFILE**. (Historically, this limit was named **RLIMIT_OFILE** on BSD.)

Since Linux 4.5, this limit also defines the maximum number of file descriptors that an unprivileged process (one without the **CAP_SYS_RESOURCE** capability) may have "in flight" to other processes, by being passed across UNIX domain sockets. This limit applies to the `sendmsg(2)` system call. For further details, see [unix\(7\)](#).

RLIMIT_NPROC

This is a limit on the number of extant process (or, more precisely on Linux, threads) for the real user ID of the calling process. So long as the current number of processes belonging to this process's real user ID is greater than or equal to this limit, `fork(2)` fails with the error **EAGAIN**.

The **RLIMIT_NPROC** limit is not enforced for processes that have either the **CAP_SYS_ADMIN** or the **CAP_SYS_RESOURCE** capability, or run with real user ID 0.

RLIMIT_RSS

This is a limit (in bytes) on the process's resident set (the number of virtual pages resident in RAM). This limit has effect only in Linux 2.4.x, $x < 30$, and there affects only calls to `advise(2)` specifying **MADV_WILLNEED**.



RLIMIT_RTPRIO (since Linux 2.6.12, but see BUGS)

This specifies a ceiling on the real-time priority that may be set for this process using [sched_setscheduler\(2\)](#) and [sched_setparam\(2\)](#).

For further details on real-time scheduling policies, see [sched\(7\)](#)

RLIMIT_RTIME (since Linux 2.6.25)

This is a limit (in microseconds) on the amount of CPU time that a process scheduled under a real-time scheduling policy may consume without making a blocking system call. For the purpose of this limit, each time a process makes a blocking system call, the count of its consumed CPU time is reset to zero. The CPU time count is not reset if the process continues trying to use the CPU but is preempted, its time slice expires, or it calls [sched_yield\(2\)](#).

Upon reaching the soft limit, the process is sent a **SIGXCPU** signal. If the process catches or ignores this signal and continues consuming CPU time, then **SIGXCPU** will be generated once each second until the hard limit is reached, at which point the process is sent a **SIGKILL** signal.

The intended use of this limit is to stop a runaway real-time process from locking up the system.

For further details on real-time scheduling policies, see [sched\(7\)](#)

RLIMIT_SIGPENDING (since Linux 2.6.8)

This is a limit on the number of signals that may be queued for the real user ID of the calling process. Both standard and real-time signals are counted for the purpose of checking this limit. However, the limit is enforced only for [sigqueue\(3\)](#); it is always possible to use [kill\(2\)](#) to queue one instance of any of the signals that are not already queued to the process.

RLIMIT_STACK

This is the maximum size of the process stack, in bytes. Upon reaching this limit, a **SIGSEGV** signal is generated. To handle this signal, a process must employ an alternate



signal stack ([sigaltstack\(2\)](#)).

Since Linux 2.6.23, this limit also determines the amount of space used for the process's command-line arguments and environment variables; for details, see [execve\(2\)](#).

prlimit()

The Linux-specific **prlimit()** system call combines and extends the functionality of **setrlimit()** and **getrlimit()**. It can be used to both set and get the resource limits of an arbitrary process.

The *resource* argument has the same meaning as for **setrlimit()** and **getrlimit()**.

If the *new_limit* argument is not NULL, then the *rlimit* structure to which it points is used to set new values for the soft and hard limits for *resource*. If the *old_limit* argument is not NULL, then a successful call to **prlimit()** places the previous soft and hard limits for *resource* in the *rlimit* structure pointed to by *old_limit*.

The *pid* argument specifies the ID of the process on which the call is to operate. If *pid* is 0, then the call applies to the calling process. To set or get the resources of a process other than itself, the caller must have the **CAP_SYS_RESOURCE** capability in the user namespace of the process whose resource limits are being changed, or the real, effective, and saved set user IDs of the target process must match the real user ID of the caller *and* the real, effective, and saved set group IDs of the target process must match the real group ID of the caller.

RETURN VALUE [top](#)

On success, these system calls return 0. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EFAULT A pointer argument points to a location outside the accessible address space.

EINVAL The value specified in *resource* is not valid; or, for **setrlimit()** or **prlimit()**: *rlim->rlim_cur* was greater than



rlim → *rlim_max*.

- EPERM** An unprivileged process tried to raise the hard limit; the **CAP_SYS_RESOURCE** capability is required to do this.
- EPERM** The caller tried to increase the hard **RLIMIT_NOFILE** limit above the maximum defined by */proc/sys/fs/nr_open* (see [proc\(5\)](#))
- EPERM** (**prlimit()**) The calling process did not have permission to set limits for the process specified by *pid*.
- ESRCH** Could not find a process with the ID specified in *pid*.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|--|---------------|---------|
| getrlimit() , setrlimit() , prlimit() | Thread safety | MT-Safe |

STANDARDS [top](#)

getrlimit()
setrlimit()
 POSIX.1-2008.

prlimit()
 Linux.

RLIMIT_MEMLOCK and **RLIMIT_NPROC** derive from BSD and are not specified in POSIX.1; they are present on the BSDs and Linux, but on few other implementations. **RLIMIT_RSS** derives from BSD and is not specified in POSIX.1; it is nevertheless present on most implementations. **RLIMIT_MSGQUEUE**, **RLIMIT_NICE**, **RLIMIT_RTPRIO**, **RLIMIT_RTIME**, and **RLIMIT_SIGPENDING** are Linux-specific.

HISTORY [top](#)



getrlimit()

setrlimit()

POSIX.1-2001, SVr4, 4.3BSD.

prlimit()

Linux 2.6.36, glibc 2.13.

NOTES

[top](#)

A child process created via [fork\(2\)](#) inherits its parent's resource limits. Resource limits are preserved across [execve\(2\)](#).

Resource limits are per-process attributes that are shared by all of the threads in a process.

Lowering the soft limit for a resource below the process's current consumption of that resource will succeed (but will prevent the process from further increasing its consumption of the resource).

One can set the resource limits of the shell using the built-in [ulimit](#) command ([limit](#) in [csh\(1\)](#)). The shell's resource limits are inherited by the processes that it creates to execute commands.

Since Linux 2.6.24, the resource limits of any process can be inspected via [/proc/pid/limits](#); see [proc\(5\)](#).

Ancient systems provided a [vlimit\(\)](#) function with a similar purpose to [setrlimit\(\)](#). For backward compatibility, glibc also provides [vlimit\(\)](#). All new applications should be written using [setrlimit\(\)](#).

C library/kernel ABI differences

Since glibc 2.13, the glibc [getrlimit\(\)](#) and [setrlimit\(\)](#) wrapper functions no longer invoke the corresponding system calls, but instead employ [prlimit\(\)](#), for the reasons described in [BUGS](#).

The name of the glibc wrapper function is [prlimit\(\)](#); the underlying system call is [prlimit64\(\)](#).

BUGS

[top](#)



In older Linux kernels, the **SIGXCPU** and **SIGKILL** signals delivered when a process encountered the soft and hard **RLIMIT_CPU** limits were delivered one (CPU) second later than they should have been. This was fixed in Linux 2.6.8.

In Linux 2.6.x kernels before Linux 2.6.17, a **RLIMIT_CPU** limit of 0 is wrongly treated as "no limit" (like **RLIM_INFINITY**). Since Linux 2.6.17, setting a limit of 0 does have an effect, but is actually treated as a limit of 1 second.

A kernel bug means that **RLIMIT_RTPRIO** does not work in Linux 2.6.12; the problem is fixed in Linux 2.6.13.

In Linux 2.6.12, there was an off-by-one mismatch between the priority ranges returned by `getpriority(2)` and **RLIMIT_NICE**. This had the effect that the actual ceiling for the nice value was calculated as $19 - rlim_cur$. This was fixed in Linux 2.6.13.

Since Linux 2.6.12, if a process reaches its soft **RLIMIT_CPU** limit and has a handler installed for **SIGXCPU**, then, in addition to invoking the signal handler, the kernel increases the soft limit by one second. This behavior repeats if the process continues to consume CPU time, until the hard limit is reached, at which point the process is killed. Other implementations do not change the **RLIMIT_CPU** soft limit in this manner, and the Linux behavior is probably not standards conformant; portable applications should avoid relying on this Linux-specific behavior. The Linux-specific **RLIMIT_RTTIME** limit exhibits the same behavior when the soft limit is encountered.

Kernels before Linux 2.4.22 did not diagnose the error **EINVAL** for `setrlimit()` when $rlim->rlim_cur$ was greater than $rlim->rlim_max$.

Linux doesn't return an error when an attempt to set **RLIMIT_CPU** has failed, for compatibility reasons.

Representation of "large" resource limit values on 32-bit platforms

The glibc `getrlimit()` and `setrlimit()` wrapper functions use a 64-bit `rlim_t` data type, even on 32-bit platforms. However, the `rlim_t` data type used in the `getrlimit()` and `setrlimit()` system calls is a (32-bit) *unsigned long*. Furthermore, in Linux, the kernel represents resource limits on 32-bit platforms as *unsigned long*. However, a 32-bit data type is not wide enough. The most pertinent limit here is **RLIMIT_FSIZE**, which specifies the maximum size to which a file can grow: to be useful, this limit must be



represented using a type that is as wide as the type used to represent file offsets—that is, as wide as a 64-bit **off_t** (assuming a program compiled with `_FILE_OFFSET_BITS=64`).

To work around this kernel limitation, if a program tried to set a resource limit to a value larger than can be represented in a 32-bit *unsigned long*, then the glibc **setrlimit()** wrapper function silently converted the limit value to **RLIM_INFINITY**. In other words, the requested resource limit setting was silently ignored.

Since glibc 2.13, glibc works around the limitations of the **getrlimit()** and **setrlimit()** system calls by implementing **setrlimit()** and **getrlimit()** as wrapper functions that call **prlimit()**.

EXAMPLES [top](#)

The program below demonstrates the use of **prlimit()**.

```
#define _GNU_SOURCE
#define _FILE_OFFSET_BITS 64
#include <err.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>
#include <time.h>

int
main(int argc, char *argv[])
{
    pid_t          pid;
    struct rlimit  old, new;
    struct rlimit  *newp;

    if (!(argc == 2 || argc == 4)) {
        fprintf(stderr, "Usage: %s <pid> [<new-soft-limit> "
            "<new-hard-limit>]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    pid = atoi(argv[1]);          /* PID of target process */

    newp = NULL;
```



```
if (argc == 4) {
    new.rlim_cur = atoi(argv[2]);
    new.rlim_max = atoi(argv[3]);
    newp = &new;
}

/* Set CPU time limit of target process; retrieve and display
   previous limit */

if (prlimit(pid, RLIMIT_CPU, newp, &old) == -1)
    err(EXIT_FAILURE, "prlimit-1");
printf("Previous limits: soft=%jd; hard=%jd\n",
       (intmax_t) old.rlim_cur, (intmax_t) old.rlim_max);

/* Retrieve and display new CPU time limit */

if (prlimit(pid, RLIMIT_CPU, NULL, &old) == -1)
    err(EXIT_FAILURE, "prlimit-2");
printf("New limits: soft=%jd; hard=%jd\n",
       (intmax_t) old.rlim_cur, (intmax_t) old.rlim_max);

exit(EXIT_SUCCESS);
}
```

SEE ALSO [top](#)

[prlimit\(1\)](#), [dup\(2\)](#), [fcntl\(2\)](#), [fork\(2\)](#), [getrusage\(2\)](#), [mlock\(2\)](#), [mmap\(2\)](#), [open\(2\)](#), [quotactl\(2\)](#), [sbrk\(2\)](#), [shmctl\(2\)](#), [malloc\(3\)](#), [sigqueue\(3\)](#), [ulimit\(3\)](#), [core\(5\)](#), [capabilities\(7\)](#), [cgroups\(7\)](#), [credentials\(7\)](#), [signal\(7\)](#)

Linux man-pages (unreleased) (date) [getrlimit\(2\)](#)

Pages that refer to this page: [homectl\(1\)](#), [prlimit\(1\)](#), [renice\(1\)](#), [strace\(1\)](#), [systemd-nspawn\(1\)](#), [brk\(2\)](#), [dup\(2\)](#), [execve\(2\)](#), [fcntl\(2\)](#), [fork\(2\)](#), [getpriority\(2\)](#), [getrusage\(2\)](#), [io_uring_register\(2\)](#), [io_uring_setup\(2\)](#), [madvise\(2\)](#), [memfd_secret\(2\)](#), [mlock\(2\)](#), [mmap\(2\)](#), [mremap\(2\)](#), [nice\(2\)](#), [open\(2\)](#), [perf_event_open\(2\)](#), [pidfd_getfd\(2\)](#), [pidfd_open\(2\)](#), [prctl\(2\)](#), [quotactl\(2\)](#), [seccomp\(2\)](#), [seccomp_unotify\(2\)](#), [select\(2\)](#), [shmctl\(2\)](#), [sigaltstack\(2\)](#), [syscalls\(2\)](#), [timer_create\(2\)](#), [write\(2\)](#), [errno\(3\)](#), [getdtablesize\(3\)](#), [malloc\(3\)](#), [mq_open\(3\)](#), [pthread_attr_setstacksize\(3\)](#), [pthread_create\(3\)](#), [pthread_getattr_np\(3\)](#), [pthread_setschedparam\(3\)](#), [pthread_setschedprio\(3\)](#), [ulimit\(3\)](#), [core\(5\)](#), [limits.conf\(5\)](#), [lxc.container.conf\(5\)](#), [proc\(5\)](#), [systemd.exec\(5\)](#), [systemd-system.conf\(5\)](#), [capabilities\(7\)](#), [cgroups\(7\)](#),



[credentials\(7\)](#), [fanotify\(7\)](#), [mq_overview\(7\)](#), [pthreads\(7\)](#), [sched\(7\)](#), [signal\(7\)](#), [time\(7\)](#), [unix\(7\)](#), [systemd-coredump\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



Materials for Topic 9: CPU Scheduling

Full C Programs

- [scheduling_examples.c](#) - a C program that demonstrates the uses of functions mentioned in the Topic 9 lecture notes: `sched_yield()`, `nice()`, `getpriority()`, `setpriority()`, `CPU_ZERO()`, `CPU_ISSET()`, `CPU_SET()`, `CPU_CLR()`, `sched_getaffinity()`, `sched_setaffinity()`, `sched_getscheduler()`, `sched_setscheduler()`, `memset()`, `sched_getparam()`, `sched_get_priority_min()`, `sched_get_priority_max()`, `sched_rr_get_interval()`, `setrlimit()`, and `getrlimit()`.

Runnable Linux Commands

Quick Links:

- [gcc](#)
 - [sudo ./scheduling_examples](#)
 - [./scheduling_examples](#)
 - [./short_prompt](#)
 - [./long_prompt](#)
-

- The command:

```
gcc -Wall -Wextra -O2 -g -o scheduling_examples
```

```
scheduling_examples.c
```

compiles the C source code located inside the file `scheduling_examples.c`. See more details [here](#).

- The command:

```
sudo ./scheduling_examples
```

executes the program `scheduling_examples` as the root user. Not all users of Linux are allowed to elevate their privileges to those of the root, so calling `sudo` might not be permitted for your user. However, if you are the owner of a virtual machine, you can call `sudo`, type the password, and view the full execution of this program with no permission limitations.

When I run this command on my Linux virtual machine (I am allowed to use `sudo`,) I get the following output:

```
The current nice value is: 0.
```

```
We increased the nice value to: 5.
```

```
The current nice value is: 5.
```

```
The new nice value is: 6.
```

```
cpu = 0 is set
```

```
cpu = 1 is set
```

```
cpu = 2 is unset
```

```
cpu = 3 is unset
```

```
cpu = 4 is unset
```

```
cpu = 5 is unset
```

```
cpu = 6 is unset
```

```
cpu = 7 is unset
```



cpu = 8 is unset

cpu = 9 is unset

cpu = 10 is unset

cpu = 11 is unset

cpu = 12 is unset

cpu = 13 is unset

cpu = 14 is unset

cpu = 15 is unset

This device allows checking up to 1024 CPUs.

cpu = 0 is set

cpu = 1 is unset

cpu = 2 is unset

cpu = 3 is unset

cpu = 4 is unset

cpu = 5 is unset

cpu = 6 is unset

cpu = 7 is unset

cpu = 8 is unset

cpu = 9 is unset

cpu = 10 is unset

cpu = 11 is unset

cpu = 12 is unset

cpu = 13 is unset

cpu = 14 is unset

cpu = 15 is unset

```
Policy is normal.
```

```
Policy was changed to round-robin.
```

```
Our priority is 1.
```

```
SCHED_RR priority range is: 1 - 99.
```

```
The time slice is 0 seconds and 100000000 nanosecs.
```

```
setrlimit: Operation not permitted
```

```
RLIMIT_NOFILE limits: soft = 1024 hard = 1048576.
```

The reason for the `setrlimit: Operation not permitted` error is due to a bug in earlier versions of `sudo`. The version of `sudo` that is installed on my virtual machine is 1.8.21p2, whereas the bug is fixed only in version 1.8.31p1 and above. I checked the version of the `sudo` command by typing: `sudo -V` into the terminal.

- The command:

```
./scheduling_examples
```

executes the program `scheduling_examples` as your real user. Some of the function calls in this program might return errors because they require elevated permissions, which can only be achieved by calling `sudo ./scheduling_examples`.

Note: If you are a regular/unprivileged Linux user (for example, if you are using your Brooklyn College Linux Account,) you might have no permission to use the `sudo` command. No homework assignment or exam will ask you

to use the `sudo` command, so don't worry about this current inability to use `sudo`. If you have access to a Linux virtual machine in which you are an administrator user, you should be able to use `sudo`.

- The command:

```
./short_prompt
```

executes code inside a file named `.short_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).

- The command:

```
./long_prompt
```

executes code inside a file named `.long_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).



This website by [Miriam Briskman](#) is licensed under [CC BY-NC 4.0](#).

```

1  /* A C program that demonstrates the uses of
2  *   functions mentioned in the Topic 9 lecture
3  *   notes: sched_yield(), nice(), getpriority(),
4  *   setpriority(), CPU_ZERO(), CPU_ISSET(),
5  *   CPU_SET(), CPU_CLR(), sched_getaffinity(),
6  *   sched_setaffinity(), sched_getscheduler(),
7  *   sched_setscheduler(), memset(),
8  *   sched_getparam(), sched_get_priority_min(),
9  *   sched_get_priority_max(),
10 *   sched_rr_get_interval(), setrlimit(), and
11 *   getrlimit().
12 *
13 *   Miriam Briskman, 3/27/2023
14 *   CISC 3350, Brooklyn College
15 *   Licensed under CC BY-NC 4.0
16 */
17
18 #define _GNU_SOURCE
19
20 #include <sys/types.h>    // pid_t, gid_t, etc.
21 #include <unistd.h>      // Defines some system calls.
22 #include <stdio.h>       // perror(), printf(), fprintf().
23 #include <stdlib.h>      // EXIT_SUCCESS, EXIT_FAILURE.
24 #include <sched.h>       // scheduling functions and constants.
25 #include <sys/time.h>    // Time-related functions and constants.
26 #include <sys/resource.h> // Resource functions and constants.
27 #include <errno.h>       // 'errno' variable.
28 #include <string.h>      // memset().
29
30 int main ()
31 {
32     // Yield the process from the CPU. This will place this
33     // process at the tail of the ready queue. It is
34     // similar to a player giving up their turn in a
35     // certain game's round.
36     if (sched_yield() == -1)
37     {
38         perror("sched_yield");
39     }
40
41     /*****\
42     |               Nice Values               |
43     \*****/
44
45     // Get the nice value of the process:
46     errno = 0;
47     int nice_value = nice(0);

```



```
48 // To check for errors, we check the value of
49 //     errno, since nice() can return -1, which is
50 //     a valid nice value, so we can't use the
51 //     returned value of nice() to check for errors.
52 if (errno != 0)
53 {
54     perror("nice");
55 }
56
57 printf("The current nice value is: %d.\n", nice_value);
58
59 // Let's increase the nice value by 5:
60 if (nice_value + 5 <= 19)
61 {
62     errno = 0;
63     int res = nice(5);
64     if (errno != 0)
65     {
66         perror("nice");
67     }
68     printf("We increased the nice value to: %d.\n", res);
69 }
70
71 // Get the nice value of the process again:
72 errno = 0;
73 int ret = getpriority (PRIO_PROCESS, 0);
74 if (errno != 0)
75 {
76     perror("getpriority");
77 }
78
79 printf ("The current nice value is: %d.\n", ret);
80
81 // Decrease the priority by 1:
82 ret = setpriority (PRIO_PROCESS, 0, ret + 1);
83 if (ret == -1)
84 {
85     perror ("setpriority");
86 }
87
88 // Get the nice value of the process again:
89 errno = 0;
90 ret = getpriority (PRIO_PROCESS, 0);
91 if (errno != 0)
92 {
93     perror("getpriority");
94 }
95
```




```

96     printf ("The new nice value is: %d.\n", ret);
97
98     /*****\
99     |                Affinities                |
100    \*****/
101
102    // The following code for getting CPU affinity is
103    //     taken from slide 17 of the Chapter 6
104    //     lecture notes.
105    cpu_set_t set;
106    int i;
107
108    CPU_ZERO (&set); // Nullify the set: remove all the CPUs.
109
110    // Find the affinity for the current process, and store
111    //     the results in 'set':
112    if (sched_getaffinity (0, sizeof (cpu_set_t), &set) == -1)
113    {
114        perror ("sched_getaffinity");
115    }
116
117    // Iterates through the first 16 virtual CPUs, and
118    //     check if each of them is bound to the process:
119    for (i = 0; i < 16; i++)
120    {
121        int cpu = CPU_ISSET (i, &set);
122        printf ("cpu = %i is %s\n", i, cpu ? "set" : "unset");
123    }
124
125    // Print the value of the CPU_SETSIZE constant:
126    printf("This device allows checking up to %d CPUs.\n", CPU_SETSIZE);
127
128    // Now, we'll set the affinity of the current process only
129    //     to CPU-0:
130    // The following code for setting CPU affinity is
131    //     taken from slide 17 of the Chapter 6
132    //     lecture notes.
133    CPU_ZERO (&set); /* clear all CPUs */
134    CPU_SET (0, &set); /* allow CPU #0 */
135    CPU_CLR (1, &set); /* disallow CPU #1 */
136
137    if (sched_setaffinity (0, sizeof (cpu_set_t), &set) == -1)
138    {
139        perror ("sched_setaffinity");
140    }
141
142    // Iterates through the first 16 virtual CPUs, and
143    //     check if each of them is bound to the process:

```



```
144     for (i = 0; i < 16; i++)
145     {
146         int cpu = CPU_ISSET (i, &set);
147         printf ("cpu = %i is %s\n", i, cpu ? "set" : "unset");
148     }
149
150     /*****\
151     |           Policies           |
152     \*****/
153
154     // The following code for getting the scheduling
155     //     policy is taken from slide 17 of the
156     //     Chapter 6 lecture notes.
157
158     /* get our scheduling policy */
159     int policy = sched_getscheduler (0);
160
161     switch (policy)
162     {
163         case SCHED_OTHER: printf ("Policy is normal.\n");
164                         break;
165         case SCHED_RR:   printf ("Policy is round-robin.\n");
166                         break;
167         case SCHED_FIFO: printf ("Policy is FIFO.\n");
168                         break;
169         case SCHED_BATCH: printf ("Policy is idle.\n");
170                         break;
171         case -1: perror ("sched_getscheduler");
172                 break;
173         default: fprintf (stderr, "Unknown policy!\n");
174     }
175
176     // Setting the policy to round-robin with priority of 1:
177     struct sched_param sp = { .sched_priority = 1 };
178     if (sched_setscheduler (0, SCHED_RR, &sp) == -1)
179     {
180         perror ("sched_setscheduler");
181     }
182
183     // Checking if the policy was indeed set:
184     policy = sched_getscheduler (0);
185
186     switch (policy)
187     {
188         case SCHED_OTHER: printf ("Policy was changed to normal.\n");
189                         break;
190         case SCHED_RR:   printf ("Policy was changed to round-robin.\n");
191                         break;
```



```
192     case SCHED_FIFO: printf ("Policy was changed to FIFO.\n");
193                     break;
194     case SCHED_BATCH: printf ("Policy was changed to idle.\n");
195                       break;
196     case -1: perror ("sched_getscheduler");
197              break;
198     default: fprintf (stderr, "Unknown policy!\n");
199 }
200
201 // Nullify sp (since we need to use it again:)
202 memset (&sp, 0, sizeof (struct sched_param));
203
204 // Finding the priority of the process:
205 if (sched_getparam (0, &sp)== -1)
206 {
207     perror ("sched_getparam");
208 }
209
210 printf ("Our priority is %d.\n", sp.sched_priority);
211
212 // Nullify sp (since we need to use it again:)
213 memset (&sp, 0, sizeof (struct sched_param));
214
215 // Setting the priority of the process to 1:
216 sp.sched_priority = 1;
217 if (sched_setparam (0, &sp)== -1)
218 {
219     perror ("sched_setparam");
220 }
221
222 // Finding the min and max priorities for round-robin:
223 int min = sched_get_priority_min (SCHED_RR);
224 if (min == -1)
225 {
226     perror ("sched_get_priority_min");
227 }
228
229 int max = sched_get_priority_max (SCHED_RR);
230 if (max == -1)
231 {
232     perror ("sched_get_priority_max");
233 }
234
235 printf ("SCHED_RR priority range is: %d - %d.\n", min, max);
236
237 // Getting the current process's time slice length:
238 struct timespec tp;
239
```



```
240     if (sched_rr_get_interval (0, &tp) == -1)
241     {
242         perror ("sched_rr_get_interval");
243     }
244
245     printf ("The time slice is %lld seconds and %lld nanosecs.\n",
246           (long long int) tp.tv_sec, (long long int) tp.tv_nsec);
247
248     /*****\
249     |                               |
250     \*****/
251
252     // Setting a soft limit of RLIMIT_NOFILE to 50:
253     struct rlimit rlim;
254     rlim.rlim_cur = 50; /* Changing the soft limit. */
255     rlim.rlim_max = RLIM_INFINITY; /* Hard limit: leaving it alone. */
256
257     if (setrlimit (RLIMIT_NOFILE, &rlim) == -1)
258     {
259         perror ("setrlimit");
260     }
261
262     // Printing the hard and soft limits of RLIMIT_NOFILE:
263     if (getrlimit (RLIMIT_NOFILE, &rlim) == -1)
264     {
265         perror ("getrlimit");
266     }
267
268     printf ("RLIMIT_NOFILE limits: soft = %ld hard = %ld.\n",
269           rlim.rlim_cur, rlim.rlim_max);
270
271     return EXIT_SUCCESS;
272 }
273
```



Topic 10: Signals

Note: A copy of each one of the sources listed below is included in this packet. To jump to the location of a source copy, click the page link under “Page”.

In case a source isn’t included (e.g., a YouTube video,) click the [↗](#) (external link) symbol under “Page” to open the external source in a browser window.

| # | Citation & Source Link | Page |
|----|--|------|
| 1 | “signal(7) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man7/signal.7.html | 1127 |
| 2 | “signal(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/signal.2.html | 1142 |
| 3 | “pause(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/pause.2.html | 1147 |
| 4 | “strsignal(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/strsignal.3.html | 1149 |
| 5 | “psignal(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/psignal.3.html | 1153 |
| 6 | “kill(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/kill.2.html | 1156 |
| 7 | “raise(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/raise.3.html | 1160 |
| 8 | “killpg(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/killpg.3.html | 1163 |
| 9 | “signal-safety(7) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man7/signal-safety.7.html | 1166 |
| 10 | “sigsetops(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/sigsetops.3.html | 1174 |
| 11 | “sigprocmask(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/sigprocmask.2.html | 1178 |
| 12 | “sigpending(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/sigpending.2.html | 1183 |
| 13 | “sigsuspend(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/sigsuspend.2.html | 1186 |
| 14 | “sigaction(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/sigaction.2.html | 1189 |
| 15 | “sigqueue(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/sigqueue.3.html | 1206 |
| 16 | Valente, Salvatore and Karel Zak. “kill(1) - Linux manual page”, <i>man7.org</i> , 19 Jul. 2023. URL: https://man7.org/linux/man-pages/man1/kill.1.html | 1210 |

| # | Citation & Source Link | Page |
|----|--|------|
| 17 | Briskman, Miriam. “Materials for Topic 10: Signals.” <i>Topic 10: Signals — CISC 3350 Materials</i> , 2023. URL: www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_10/ | 1216 |
| 18 | Briskman, Miriam. “division_by_0.c .” (C source code) 26 Apr. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_10/division_by_0.c | 1219 |
| 19 | Love, Robert. “handling_sigint.c .” (C source code), <i>Linux System Programming: Talking Directly to the Kernel and C Library</i> , O’Reilly Media, Inc., Sebastopol, CA, 2013, p. 342. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_10/handling_sigint.c | 1221 |
| 20 | Love, Robert. “more_signals.c .” (C source code), <i>Linux System Programming: Talking Directly to the Kernel and C Library</i> , O’Reilly Media, Inc., Sebastopol, CA, 2013, pp. 343-344. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_10/more_signals.c | 1223 |
| 21 | Briskman, Miriam. “sigaction_example.c .” (C source code) 30 Apr. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_10/sigaction_example.c | 1225 |
| 22 | Briskman, Miriam. “issuing_signals.c .” (C source code) 30 Apr. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_10/issuing_signals.c | 1227 |
| 23 | Briskman, Miriam. “payload_sending.c .” (C source code) 30 Apr. 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_10/payload_sending.c | 1232 |

signal(7) — Linux manual page

[NAME](#) | [DESCRIPTION](#) | [STANDARDS](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

signal(7)

Miscellaneous Information Manual

signal(7)

NAME [top](#)

signal - overview of signals

DESCRIPTION [top](#)

Linux supports both POSIX reliable signals (hereinafter "standard signals") and POSIX real-time signals.

Signal dispositions

Each signal has a current *disposition*, which determines how the process behaves when it is delivered the signal.

The entries in the "Action" column of the table below specify the default disposition for each signal, as follows:

| | |
|------|--|
| Term | Default action is to terminate the process. |
| Ign | Default action is to ignore the signal. |
| Core | Default action is to terminate the process and dump core (see core(5)). |
| Stop | Default action is to stop the process. |
| Cont | Default action is to continue the process if it is currently stopped. |

A process can change the disposition of a signal using [sigaction\(2\)](#) or [signal\(2\)](#). (The latter is less portable when establishing a signal handler; see [signal\(2\)](#) for details.) Using these system calls, a process can elect one of the following behaviors to occur on delivery of the signal: perform the default action; ignore the signal; or catch the signal with a *signal*

handler, a programmer-defined function that is automatically invoked when the signal is delivered.

By default, a signal handler is invoked on the normal process stack. It is possible to arrange that the signal handler uses an alternate stack; see [sigaltstack\(2\)](#) for a discussion of how to do this and when it might be useful.

The signal disposition is a per-process attribute: in a multithreaded application, the disposition of a particular signal is the same for all threads.

A child created via [fork\(2\)](#) inherits a copy of its parent's signal dispositions. During an [execve\(2\)](#), the dispositions of handled signals are reset to the default; the dispositions of ignored signals are left unchanged.

Sending a signal

The following system calls and library functions allow the caller to send a signal:

[raise\(3\)](#)

Sends a signal to the calling thread.

[kill\(2\)](#)

Sends a signal to a specified process, to all members of a specified process group, or to all processes on the system.

[pidfd_send_signal\(2\)](#)

Sends a signal to a process identified by a PID file descriptor.

[killpg\(3\)](#)

Sends a signal to all of the members of a specified process group.

[pthread_kill\(3\)](#)

Sends a signal to a specified POSIX thread in the same process as the caller.

[tgkill\(2\)](#)

Sends a signal to a specified thread within a specific process. (This is the system call used to implement [pthread_kill\(3\)](#).)

[sigqueue\(3\)](#)

Sends a real-time signal with accompanying data to a

specified process.

Waiting for a signal to be caught

The following system calls suspend execution of the calling thread until a signal is caught (or an unhandled signal terminates the process):

`pause(2)`

Suspends execution until any signal is caught.

`sigsuspend(2)`

Temporarily changes the signal mask (see below) and suspends execution until one of the unmasked signals is caught.

Synchronously accepting a signal

Rather than asynchronously catching a signal via a signal handler, it is possible to synchronously accept the signal, that is, to block execution until the signal is delivered, at which point the kernel returns information about the signal to the caller. There are two general ways to do this:

- `sigwaitinfo(2)`, `sigtimedwait(2)`, and `sigwait(3)` suspend execution until one of the signals in a specified set is delivered. Each of these calls returns information about the delivered signal.
- `signalfd(2)` returns a file descriptor that can be used to read information about signals that are delivered to the caller. Each `read(2)` from this file descriptor blocks until one of the signals in the set specified in the `signalfd(2)` call is delivered to the caller. The buffer returned by `read(2)` contains a structure describing the signal.

Signal mask and pending signals

A signal may be *blocked*, which means that it will not be delivered until it is later unblocked. Between the time when it is generated and when it is delivered a signal is said to be *pending*.

Each thread in a process has an independent *signal mask*, which indicates the set of signals that the thread is currently blocking. A thread can manipulate its signal mask using `pthread_sigmask(3)`. In a traditional single-threaded application, `sigprocmask(2)` can be used to manipulate the signal mask.

A child created via `fork(2)` inherits a copy of its parent's

signal mask; the signal mask is preserved across `execve(2)`.

A signal may be process-directed or thread-directed. A process-directed signal is one that is targeted at (and thus pending for) the process as a whole. A signal may be process-directed because it was generated by the kernel for reasons other than a hardware exception, or because it was sent using `kill(2)` or `sigqueue(3)`. A thread-directed signal is one that is targeted at a specific thread. A signal may be thread-directed because it was generated as a consequence of executing a specific machine-language instruction that triggered a hardware exception (e.g., **SIGSEGV** for an invalid memory access, or **SIGFPE** for a math error), or because it was targeted at a specific thread using interfaces such as `tgkill(2)` or `pthread_kill(3)`.

A process-directed signal may be delivered to any one of the threads that does not currently have the signal blocked. If more than one of the threads has the signal unblocked, then the kernel chooses an arbitrary thread to which to deliver the signal.

A thread can obtain the set of signals that it currently has pending using `sigpending(2)`. This set will consist of the union of the set of pending process-directed signals and the set of signals pending for the calling thread.

A child created via `fork(2)` initially has an empty pending signal set; the pending signal set is preserved across an `execve(2)`.

Execution of signal handlers

Whenever there is a transition from kernel-mode to user-mode execution (e.g., on return from a system call or scheduling of a thread onto the CPU), the kernel checks whether there is a pending unblocked signal for which the process has established a signal handler. If there is such a pending signal, the following steps occur:

- (1) The kernel performs the necessary preparatory steps for execution of the signal handler:
 - (1.1) The signal is removed from the set of pending signals.
 - (1.2) If the signal handler was installed by a call to `sigaction(2)` that specified the **SA_ONSTACK** flag and the thread has defined an alternate signal stack (using `sigaltstack(2)`), then that stack is installed.
 - (1.3) Various pieces of signal-related context are saved



into a special frame that is created on the stack. The saved information includes:

- the program counter register (i.e., the address of the next instruction in the main program that should be executed when the signal handler returns);
- architecture-specific register state required for resuming the interrupted program;
- the thread's current signal mask;
- the thread's alternate signal stack settings.

(If the signal handler was installed using the [sigaction\(2\)](#) **SA_SIGINFO** flag, then the above information is accessible via the `ucontext_t` object that is pointed to by the third argument of the signal handler.)

- (1.4) Any signals specified in `act->sa_mask` when registering the handler with [sigprocmask\(2\)](#) are added to the thread's signal mask. The signal being delivered is also added to the signal mask, unless **SA_NODEFER** was specified when registering the handler. These signals are thus blocked while the handler executes.
- (2) The kernel constructs a frame for the signal handler on the stack. The kernel sets the program counter for the thread to point to the first instruction of the signal handler function, and configures the return address for that function to point to a piece of user-space code known as the signal trampoline (described in [sigreturn\(2\)](#)).
- (3) The kernel passes control back to user-space, where execution commences at the start of the signal handler function.
- (4) When the signal handler returns, control passes to the signal trampoline code.
- (5) The signal trampoline calls [sigreturn\(2\)](#), a system call that uses the information in the stack frame created in step 1 to restore the thread to its state before the signal handler was called. The thread's signal mask and alternate signal stack settings are restored as part of this procedure. Upon



completion of the call to `sigreturn(2)`, the kernel transfers control back to user space, and the thread recommences execution at the point where it was interrupted by the signal handler.

Note that if the signal handler does not return (e.g., control is transferred out of the handler using `siglongjmp(3)`, or the handler executes a new program with `execve(2)`), then the final step is not performed. In particular, in such scenarios it is the programmer's responsibility to restore the state of the signal mask (using `sigprocmask(2)`), if it is desired to unblock the signals that were blocked on entry to the signal handler. (Note that `siglongjmp(3)` may or may not restore the signal mask, depending on the `savesigs` value that was specified in the corresponding call to `sigsetjmp(3)`.)

From the kernel's point of view, execution of the signal handler code is exactly the same as the execution of any other user-space code. That is to say, the kernel does not record any special state information indicating that the thread is currently executing inside a signal handler. All necessary state information is maintained in user-space registers and the user-space stack. The depth to which nested signal handlers may be invoked is thus limited only by the user-space stack (and sensible software design!).

Standard signals

Linux supports the standard signals listed below. The second column of the table indicates which standard (if any) specified the signal: "P1990" indicates that the signal is described in the original POSIX.1-1990 standard; "P2001" indicates that the signal was added in SUSv2 and POSIX.1-2001.

| Signal | Standard | Action | Comment |
|----------------|----------|--------|---|
| SIGABRT | P1990 | Core | Abort signal from <code>abort(3)</code> |
| SIGALRM | P1990 | Term | Timer signal from <code>alarm(2)</code> |
| SIGBUS | P2001 | Core | Bus error (bad memory access) |
| SIGCHLD | P1990 | Ign | Child stopped or terminated |
| SIGCLD | - | Ign | A synonym for SIGCHLD |
| SIGCONT | P1990 | Cont | Continue if stopped |
| SIGEMT | - | Term | Emulator trap |
| SIGFPE | P1990 | Core | Floating-point exception |
| SIGHUP | P1990 | Term | Hangup detected on controlling terminal or death of controlling process |
| SIGILL | P1990 | Core | Illegal Instruction |
| SIGINFO | - | | A synonym for SIGPWR |
| SIGINT | P1990 | Term | Interrupt from keyboard |
| SIGIO | - | Term | I/O now possible (4.2BSD) |



| | | | |
|------------------|-------|------|---|
| SIGIOT | - | Core | IOT trap. A synonym for SIGABRT |
| SIGKILL | P1990 | Term | Kill signal |
| SIGLOST | - | Term | File lock lost (unused) |
| SIGPIPE | P1990 | Term | Broken pipe: write to pipe with no readers; see pipe(7) |
| SIGPOLL | P2001 | Term | Pollable event (Sys V); synonym for SIGIO |
| SIGPROF | P2001 | Term | Profiling timer expired |
| SIGPWR | - | Term | Power failure (System V) |
| SIGQUIT | P1990 | Core | Quit from keyboard |
| SIGSEGV | P1990 | Core | Invalid memory reference |
| SIGSTKFLT | - | Term | Stack fault on coprocessor (unused) |
| SIGSTOP | P1990 | Stop | Stop process |
| SIGTSTP | P1990 | Stop | Stop typed at terminal |
| SIGSYS | P2001 | Core | Bad system call (SVr4); see also seccomp(2) |
| SIGTERM | P1990 | Term | Termination signal |
| SIGTRAP | P2001 | Core | Trace/breakpoint trap |
| SIGTTIN | P1990 | Stop | Terminal input for background process |
| SIGTTOU | P1990 | Stop | Terminal output for background process |
| SIGUNUSED | - | Core | Synonymous with SIGSYS |
| SIGURG | P2001 | Ign | Urgent condition on socket (4.2BSD) |
| SIGUSR1 | P1990 | Term | User-defined signal 1 |
| SIGUSR2 | P1990 | Term | User-defined signal 2 |
| SIGVTALRM | P2001 | Term | Virtual alarm clock (4.2BSD) |
| SIGXCPU | P2001 | Core | CPU time limit exceeded (4.2BSD); see setrlimit(2) |
| SIGXFSZ | P2001 | Core | File size limit exceeded (4.2BSD); see setrlimit(2) |
| SIGWINCH | - | Ign | Window resize signal (4.3BSD, Sun) |

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

Up to and including Linux 2.2, the default behavior for **SIGSYS**, **SIGXCPU**, **SIGXFSZ**, and (on architectures other than SPARC and MIPS) **SIGBUS** was to terminate the process (without a core dump). (On some other UNIX systems the default action for **SIGXCPU** and **SIGXFSZ** is to terminate the process without a core dump.) Linux 2.4 conforms to the POSIX.1-2001 requirements for these signals, terminating the process with a core dump.

SIGEMT is not specified in POSIX.1-2001, but nevertheless appears on most other UNIX systems, where its default action is typically to terminate the process with a core dump.

SIGPWR (which is not specified in POSIX.1-2001) is typically ignored by default on those other UNIX systems where it appears.



SIGIO (which is not specified in POSIX.1-2001) is ignored by default on several other UNIX systems.

Queueing and delivery semantics for standard signals

If multiple standard signals are pending for a process, the order in which the signals are delivered is unspecified.

Standard signals do not queue. If multiple instances of a standard signal are generated while that signal is blocked, then only one instance of the signal is marked as pending (and the signal will be delivered just once when it is unblocked). In the case where a standard signal is already pending, the *siginfo_t* structure (see [sigaction\(2\)](#)) associated with that signal is not overwritten on arrival of subsequent instances of the same signal. Thus, the process will receive the information associated with the first instance of the signal.

Signal numbering for standard signals

The numeric value for each signal is given in the table below. As shown in the table, many signals have different numeric values on different architectures. The first numeric value in each table row shows the signal number on x86, ARM, and most other architectures; the second value is for Alpha and SPARC; the third is for MIPS; and the last is for PARISC. A dash (-) denotes that a signal is absent on the corresponding architecture.

| Signal | x86/ARM most others | Alpha/ SPARC | MIPS | PARISC | Notes |
|--------|------------------------|-----------------|------|--------|-------|
|--------|------------------------|-----------------|------|--------|-------|

| | | | | | |
|------------------|----|----|----|----|--|
| SIGHUP | 1 | 1 | 1 | 1 | |
| SIGINT | 2 | 2 | 2 | 2 | |
| SIGQUIT | 3 | 3 | 3 | 3 | |
| SIGILL | 4 | 4 | 4 | 4 | |
| SIGTRAP | 5 | 5 | 5 | 5 | |
| SIGABRT | 6 | 6 | 6 | 6 | |
| SIGIOT | 6 | 6 | 6 | 6 | |
| SIGBUS | 7 | 10 | 10 | 10 | |
| SIGEMT | - | 7 | 7 | - | |
| SIGFPE | 8 | 8 | 8 | 8 | |
| SIGKILL | 9 | 9 | 9 | 9 | |
| SIGUSR1 | 10 | 30 | 16 | 16 | |
| SIGSEGV | 11 | 11 | 11 | 11 | |
| SIGUSR2 | 12 | 31 | 17 | 17 | |
| SIGPIPE | 13 | 13 | 13 | 13 | |
| SIGALRM | 14 | 14 | 14 | 14 | |
| SIGTERM | 15 | 15 | 15 | 15 | |
| SIGSTKFLT | 16 | - | - | 7 | |
| SIGCHLD | 17 | 20 | 18 | 18 | |



| | | | | | |
|------------------|----|------|----|----|---------------|
| SIGCLD | - | - | 18 | - | |
| SIGCONT | 18 | 19 | 25 | 26 | |
| SIGSTOP | 19 | 17 | 23 | 24 | |
| SIGTSTP | 20 | 18 | 24 | 25 | |
| SIGTTIN | 21 | 21 | 26 | 27 | |
| SIGTTOU | 22 | 22 | 27 | 28 | |
| SIGURG | 23 | 16 | 21 | 29 | |
| SIGXCPU | 24 | 24 | 30 | 12 | |
| SIGXFSZ | 25 | 25 | 31 | 30 | |
| SIGVTALRM | 26 | 26 | 28 | 20 | |
| SIGPROF | 27 | 27 | 29 | 21 | |
| SIGWINCH | 28 | 28 | 20 | 23 | |
| SIGIO | 29 | 23 | 22 | 22 | |
| SIGPOLL | | | | | Same as SIGIO |
| SIGPWR | 30 | 29/- | 19 | 19 | |
| SIGINFO | - | 29/- | - | - | |
| SIGLOST | - | -/29 | - | - | |
| SIGSYS | 31 | 12 | 12 | 31 | |
| SIGUNUSED | 31 | - | - | 31 | |

Note the following:

- Where defined, **SIGUNUSED** is synonymous with **SIGSYS**. Since glibc 2.26, **SIGUNUSED** is no longer defined on any architecture.
- Signal 29 is **SIGINFO/SIGPWR** (synonyms for the same value) on Alpha but **SIGLOST** on SPARC.

Real-time signals

Starting with Linux 2.2, Linux supports real-time signals as originally defined in the POSIX.1b real-time extensions (and now included in POSIX.1-2001). The range of supported real-time signals is defined by the macros **SIGRTMIN** and **SIGRTMAX**. POSIX.1-2001 requires that an implementation support at least **_POSIX_RTSIG_MAX** (8) real-time signals.

The Linux kernel supports a range of 33 different real-time signals, numbered 32 to 64. However, the glibc POSIX threads implementation internally uses two (for NPTL) or three (for LinuxThreads) real-time signals (see [pthreads\(7\)](#)), and adjusts the value of **SIGRTMIN** suitably (to 34 or 35). Because the range of available real-time signals varies according to the glibc threading implementation (and this variation can occur at run time according to the available kernel and glibc), and indeed the range of real-time signals varies across UNIX systems, programs should *never refer to real-time signals using hard-coded numbers*, but instead should always refer to real-time signals using the



notation **SIGRTMIN+n**, and include suitable (run-time) checks that **SIGRTMIN+n** does not exceed **SIGRTMAX**.

Unlike standard signals, real-time signals have no predefined meanings: the entire set of real-time signals can be used for application-defined purposes.

The default action for an unhandled real-time signal is to terminate the receiving process.

Real-time signals are distinguished by the following:

- Multiple instances of real-time signals can be queued. By contrast, if multiple instances of a standard signal are delivered while that signal is currently blocked, then only one instance is queued.
- If the signal is sent using `sigqueue(3)`, an accompanying value (either an integer or a pointer) can be sent with the signal. If the receiving process establishes a handler for this signal using the **SA_SIGINFO** flag to `sigaction(2)`, then it can obtain this data via the `si_value` field of the `siginfo_t` structure passed as the second argument to the handler. Furthermore, the `si_pid` and `si_uid` fields of this structure can be used to obtain the PID and real user ID of the process sending the signal.
- Real-time signals are delivered in a guaranteed order. Multiple real-time signals of the same type are delivered in the order they were sent. If different real-time signals are sent to a process, they are delivered starting with the lowest-numbered signal. (I.e., low-numbered signals have highest priority.) By contrast, if multiple standard signals are pending for a process, the order in which they are delivered is unspecified.

If both standard and real-time signals are pending for a process, POSIX leaves it unspecified which is delivered first. Linux, like many other implementations, gives priority to standard signals in this case.

According to POSIX, an implementation should permit at least **_POSIX_SIGQUEUE_MAX** (32) real-time signals to be queued to a process. However, Linux does things differently. Up to and including Linux 2.6.7, Linux imposes a system-wide limit on the number of queued real-time signals for all processes. This limit can be viewed and (with privilege) changed via the `/proc/sys/kernel/rtsig-max` file. A related file,



`/proc/sys/kernel/rtsig-nr`, can be used to find out how many real-time signals are currently queued. In Linux 2.6.8, these `/proc` interfaces were replaced by the **RLIMIT_SIGPENDING** resource limit, which specifies a per-user limit for queued signals; see [setrlimit\(2\)](#) for further details.

The addition of real-time signals required the widening of the signal set structure (`sigset_t`) from 32 to 64 bits. Consequently, various system calls were superseded by new system calls that supported the larger signal sets. The old and new system calls are as follows:

| Linux 2.0 and earlier | Linux 2.2 and later |
|---------------------------------|------------------------------------|
| sigaction(2) | rt_sigaction(2) |
| sigpending(2) | rt_sigpending(2) |
| sigprocmask(2) | rt_sigprocmask(2) |
| sigreturn(2) | rt_sigreturn(2) |
| sigsuspend(2) | rt_sigsuspend(2) |
| sigtimedwait(2) | rt_sigtimedwait(2) |

Interruption of system calls and library functions by signal handlers

If a signal handler is invoked while a system call or library function call is blocked, then either:

- the call is automatically restarted after the signal handler returns; or
- the call fails with the error **EINTR**.

Which of these two behaviors occurs depends on the interface and whether or not the signal handler was established using the **SA_RESTART** flag (see [sigaction\(2\)](#)). The details vary across UNIX systems; below, the details for Linux.

If a blocked call to one of the following interfaces is interrupted by a signal handler, then the call is automatically restarted after the signal handler returns if the **SA_RESTART** flag was used; otherwise the call fails with the error **EINTR**:

- [read\(2\)](#), [readv\(2\)](#), [write\(2\)](#), [writev\(2\)](#), and [ioctl\(2\)](#) calls on "slow" devices. A "slow" device is one where the I/O call may block for an indefinite time, for example, a terminal, pipe, or socket. If an I/O call on a slow device has already transferred some data by the time it is interrupted by a signal handler, then the call will return a success status (normally, the number of bytes transferred). Note that a (local) disk is not a slow device according to this definition; I/O operations on disk devices are not interrupted



by signals.

- `open(2)`, if it can block (e.g., when opening a FIFO; see `fifo(7)`).
- `wait(2)`, `wait3(2)`, `wait4(2)`, `waitid(2)`, and `waitpid(2)`.
- Socket interfaces: `accept(2)`, `connect(2)`, `recv(2)`, `recvfrom(2)`, `recvmsg(2)`, `recvfrom(2)`, `send(2)`, `sendto(2)`, and `sendmsg(2)`, unless a timeout has been set on the socket (see below).
- File locking interfaces: `flock(2)` and the `F_SETLKW` and `F_OFD_SETLKW` operations of `fcntl(2)`
- POSIX message queue interfaces: `mq_receive(3)`, `mq_timedreceive(3)`, `mq_send(3)`, and `mq_timedsend(3)`.
- `futex(2)` `FUTEX_WAIT` (since Linux 2.6.22; beforehand, always failed with `EINTR`).
- `getrandom(2)`.
- `pthread_mutex_lock(3)`, `pthread_cond_wait(3)`, and related APIs.
- `futex(2)` `FUTEX_WAIT_BITSET`.
- POSIX semaphore interfaces: `sem_wait(3)` and `sem_timedwait(3)` (since Linux 2.6.22; beforehand, always failed with `EINTR`).
- `read(2)` from an `inotify(7)` file descriptor (since Linux 3.8; beforehand, always failed with `EINTR`).

The following interfaces are never restarted after being interrupted by a signal handler, regardless of the use of `SA_RESTART`; they always fail with the error `EINTR` when interrupted by a signal handler:

- "Input" socket interfaces, when a timeout (`SO_RCVTIMEO`) has been set on the socket using `setsockopt(2)`: `accept(2)`, `recv(2)`, `recvfrom(2)`, `recvmsg(2)` (also with a non-NULL `timeout` argument), and `recvmsg(2)`.
- "Output" socket interfaces, when a timeout (`SO_RCVTIMEO`) has been set on the socket using `setsockopt(2)`: `connect(2)`, `send(2)`, `sendto(2)`, and `sendmsg(2)`.
- Interfaces used to wait for signals: `pause(2)`, `sigsuspend(2)`,



[sigtimedwait\(2\)](#), and [sigwaitinfo\(2\)](#).

- File descriptor multiplexing interfaces: [epoll_wait\(2\)](#), [epoll_pwait\(2\)](#), [poll\(2\)](#), [ppoll\(2\)](#), [select\(2\)](#), and [pselect\(2\)](#).
- System V IPC interfaces: [msgrcv\(2\)](#), [msgsnd\(2\)](#), [semop\(2\)](#), and [semtimedop\(2\)](#).
- Sleep interfaces: [clock_nanosleep\(2\)](#), [nanosleep\(2\)](#), and [usleep\(3\)](#).
- [io_getevents\(2\)](#).

The [sleep\(3\)](#) function is also never restarted if interrupted by a handler, but gives a success return: the number of seconds remaining to sleep.

In certain circumstances, the [seccomp\(2\)](#) user-space notification feature can lead to restarting of system calls that would otherwise never be restarted by **SA_RESTART**; for details, see [seccomp_unotify\(2\)](#).

Interruption of system calls and library functions by stop signals

On Linux, even in the absence of signal handlers, certain blocking interfaces can fail with the error **EINTR** after the process is stopped by one of the stop signals and then resumed via **SIGCONT**. This behavior is not sanctioned by POSIX.1, and doesn't occur on other systems.

The Linux interfaces that display this behavior are:

- "Input" socket interfaces, when a timeout (**SO_RCVTIMEO**) has been set on the socket using [setsockopt\(2\)](#): [accept\(2\)](#), [recv\(2\)](#), [recvfrom\(2\)](#), [recvmsg\(2\)](#) (also with a non-NULL *timeout* argument), and [recvmsg\(2\)](#).
- "Output" socket interfaces, when a timeout (**SO_RCVTIMEO**) has been set on the socket using [setsockopt\(2\)](#): [connect\(2\)](#), [send\(2\)](#), [sendto\(2\)](#), and [sendmsg\(2\)](#), if a send timeout (**SO_SNDTIMEO**) has been set.
- [epoll_wait\(2\)](#), [epoll_pwait\(2\)](#).
- [semop\(2\)](#), [semtimedop\(2\)](#).
- [sigtimedwait\(2\)](#), [sigwaitinfo\(2\)](#).
- Linux 3.7 and earlier: [read\(2\)](#) from an [inotify\(7\)](#) file



descriptor

- Linux 2.6.21 and earlier: [futex\(2\)](#) **FUTEX_WAIT**, [sem_timedwait\(3\)](#), [sem_wait\(3\)](#).
- Linux 2.6.8 and earlier: [msgrcv\(2\)](#), [msgsnd\(2\)](#).
- Linux 2.4 and earlier: [nanosleep\(2\)](#).

STANDARDS [top](#)

POSIX.1, except as noted.

NOTES [top](#)

For a discussion of async-signal-safe functions, see [signal-safety\(7\)](#).

The `/proc/pid/task/tid/status` file contains various fields that show the signals that a thread is blocking (*SigBlk*), catching (*SigCgt*), or ignoring (*SigIgn*). (The set of signals that are caught or ignored will be the same across all threads in a process.) Other fields show the set of pending signals that are directed to the thread (*SigPnd*) as well as the set of pending signals that are directed to the process as a whole (*ShdPnd*). The corresponding fields in `/proc/pid/status` show the information for the main thread. See [proc\(5\)](#) for further details.

BUGS [top](#)

There are six signals that can be delivered as a consequence of a hardware exception: **SIGBUS**, **SIGEMT**, **SIGFPE**, **SIGILL**, **SIGSEGV**, and **SIGTRAP**. Which of these signals is delivered, for any given hardware exception, is not documented and does not always make sense.

For example, an invalid memory access that causes delivery of **SIGSEGV** on one CPU architecture may cause delivery of **SIGBUS** on another architecture, or vice versa.

For another example, using the x86 *int* instruction with a forbidden argument (any number other than 3 or 128) causes delivery of **SIGSEGV**, even though **SIGILL** would make more sense, because of how the CPU reports the forbidden operation to the kernel.

SEE ALSO [top](#)

[kill\(1\)](#), [clone\(2\)](#), [getrlimit\(2\)](#), [kill\(2\)](#), [pidfd_send_signal\(2\)](#), [restart_syscall\(2\)](#), [rt_sigqueueinfo\(2\)](#), [setitimer\(2\)](#), [setrlimit\(2\)](#), [sgetmask\(2\)](#), [sigaction\(2\)](#), [sigaltstack\(2\)](#), [signal\(2\)](#), [signalfd\(2\)](#), [sigpending\(2\)](#), [sigprocmask\(2\)](#), [sigreturn\(2\)](#), [sigsuspend\(2\)](#), [sigwaitinfo\(2\)](#), [abort\(3\)](#), [bsd_signal\(3\)](#), [killpg\(3\)](#), [longjmp\(3\)](#), [pthread_sigqueue\(3\)](#), [raise\(3\)](#), [sigqueue\(3\)](#), [sigset\(3\)](#), [sigsetops\(3\)](#), [sigvec\(3\)](#), [sigwait\(3\)](#), [strsignal\(3\)](#), [swapcontext\(3\)](#), [sysv_signal\(3\)](#), [core\(5\)](#), [proc\(5\)](#), [nptl\(7\)](#), [pthreads\(7\)](#), [sigevent\(7\)](#)

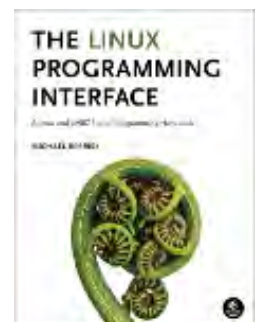
Linux man-pages (unreleased) (date) [signal\(7\)](#)

Pages that refer to this page: [env\(1\)](#), [kill\(1\)](#), [kill\(1@@procps-ng\)](#), [pgrep\(1\)](#), [ps\(1\)](#), [skill\(1\)](#), [strace\(1\)](#), [systemd-nspawn\(1\)](#), [xargs\(1\)](#), [accept\(2\)](#), [clock_nanosleep\(2\)](#), [close\(2\)](#), [connect\(2\)](#), [dup\(2\)](#), [epoll_wait\(2\)](#), [execve\(2\)](#), [fallocate\(2\)](#), [fcntl\(2\)](#), [flock\(2\)](#), [fsync\(2\)](#), [futex\(2\)](#), [getrandom\(2\)](#), [getrlimit\(2\)](#), [intro\(2\)](#), [io_getevents\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [kcmp\(2\)](#), [kill\(2\)](#), [msgop\(2\)](#), [nanosleep\(2\)](#), [open\(2\)](#), [pidfd_send_signal\(2\)](#), [poll\(2\)](#), [prctl\(2\)](#), [ptrace\(2\)](#), [read\(2\)](#), [recv\(2\)](#), [request_key\(2\)](#), [restart_syscall\(2\)](#), [rt_sigqueueinfo\(2\)](#), [s390_runtime_instr\(2\)](#), [sched_setattr\(2\)](#), [seccomp\(2\)](#), [seccomp_unotify\(2\)](#), [select\(2\)](#), [semop\(2\)](#), [send\(2\)](#), [sgetmask\(2\)](#), [sigaction\(2\)](#), [sigaltstack\(2\)](#), [signal\(2\)](#), [signalfd\(2\)](#), [sigpending\(2\)](#), [sigprocmask\(2\)](#), [sigreturn\(2\)](#), [sigsuspend\(2\)](#), [sigwaitinfo\(2\)](#), [spu_run\(2\)](#), [statfs\(2\)](#), [syscalls\(2\)](#), [timer_create\(2\)](#), [timer_getoverrun\(2\)](#), [truncate\(2\)](#), [wait\(2\)](#), [wait4\(2\)](#), [write\(2\)](#), [aio_suspend\(3\)](#), [bsd_signal\(3\)](#), [errno\(3\)](#), [getcontext\(3\)](#), [getgrent\(3\)](#), [getgrnam\(3\)](#), [getpwent\(3\)](#), [getpwnam\(3\)](#), [intro\(3\)](#), [killpg\(3\)](#), [lio_listio\(3\)](#), [lockf\(3\)](#), [mq_receive\(3\)](#), [mq_send\(3\)](#), [psignal\(3\)](#), [pthread_attr_setsigmask_np\(3\)](#), [pthread_kill\(3\)](#), [pthread_sigmask\(3\)](#), [pthread_sigqueue\(3\)](#), [raise\(3\)](#), [scanf\(3\)](#), [sd_event_add_signal\(3\)](#), [sd_journal_print\(3\)](#), [sem_wait\(3\)](#), [setjmp\(3\)](#), [sigqueue\(3\)](#), [sigset\(3\)](#), [sigvec\(3\)](#), [sigwait\(3\)](#), [sleep\(3\)](#), [statvfs\(3\)](#), [system\(3\)](#), [sysv_signal\(3\)](#), [tmpfile\(3\)](#), [ualarm\(3\)](#), [usleep\(3\)](#), [core\(5\)](#), [proc\(5\)](#), [systemd.kill\(5\)](#), [systemd.nspawn\(5\)](#), [systemd.service\(5\)](#), [credentials\(7\)](#), [fanotify\(7\)](#), [inotify\(7\)](#), [nptl\(7\)](#), [pthreads\(7\)](#), [random\(7\)](#), [signal-safety\(7\)](#), [system_data_types\(7\)](#), [cmirror\(8\)](#), [systemd-journald.service\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



signal(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 signal(2)

System Calls Manual

signal(2)**NAME** [top](#)

signal - ANSI C signal handling

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

DESCRIPTION [top](#)

WARNING: the behavior of `signal()` varies across UNIX versions, and has also varied historically across different versions of Linux. **Avoid its use:** use [sigaction\(2\)](#) instead. See [Portability](#) below.

`signal()` sets the disposition of the signal *signum* to *handler*, which is either `SIG_IGN`, `SIG_DFL`, or the address of a programmer-defined function (a "signal handler").

If the signal *signum* is delivered to the process, then one of the following happens:

- * If the disposition is set to **SIG_IGN**, then the signal is ignored.
- * If the disposition is set to **SIG_DFL**, then the default action associated with the signal (see [signal\(7\)](#)) occurs.
- * If the disposition is set to a function, then first either the disposition is reset to **SIG_DFL**, or the signal is blocked (see *Portability* below), and then *handler* is called with argument *signum*. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler.

The signals **SIGKILL** and **SIGSTOP** cannot be caught or ignored.

RETURN VALUE [top](#)

signal() returns the previous value of the signal handler. On failure, it returns **SIG_ERR**, and *errno* is set to indicate the error.

ERRORS [top](#)

EINVAL *signum* is invalid.

VERSIONS [top](#)

The use of *sighandler_t* is a GNU extension, exposed if **_GNU_SOURCE** is defined; glibc also defines (the BSD-derived) *sig_t* if **_BSD_SOURCE** (glibc 2.19 and earlier) or **_DEFAULT_SOURCE** (glibc 2.19 and later) is defined. Without use of such a type, the declaration of **signal()** is the somewhat harder to read:

```
void ( *signal(int signum, void (*handler)(int)) ) (int);
```

Portability

The only portable use of **signal()** is to set a signal's

disposition to **SIG_DFL** or **SIG_IGN**. The semantics when using **signal()** to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation); **do not use it for this purpose**.

POSIX.1 solved the portability mess by specifying [sigaction\(2\)](#), which provides explicit control of the semantics when a signal handler is invoked; use that interface instead of **signal()**.

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

C89, POSIX.1-2001.

In the original UNIX systems, when a handler that was established using **signal()** was invoked by the delivery of a signal, the disposition of the signal would be reset to **SIG_DFL**, and the system did not block delivery of further instances of the signal. This is equivalent to calling [sigaction\(2\)](#) with the following flags:

```
sa.sa_flags = SA_RESETHAND | SA_NODEFER;
```

System V also provides these semantics for **signal()**. This was bad because the signal might be delivered again before the handler had a chance to reestablish itself. Furthermore, rapid deliveries of the same signal could result in recursive invocations of the handler.

BSD improved on this situation, but unfortunately also changed the semantics of the existing **signal()** interface while doing so. On BSD, when a signal handler is invoked, the signal disposition is not reset, and further instances of the signal are blocked from being delivered while the handler is executing. Furthermore, certain blocking system calls are automatically restarted if interrupted by a signal handler (see [signal\(7\)](#)). The BSD semantics are equivalent to calling [sigaction\(2\)](#) with the following flags:


```
sa.sa_flags = SA_RESTART;
```

The situation on Linux is as follows:

- The kernel's **signal()** system call provides System V semantics.
- By default, in glibc 2 and later, the **signal()** wrapper function does not invoke the kernel system call. Instead, it calls [sigaction\(2\)](#) using flags that supply BSD semantics. This default behavior is provided as long as a suitable feature test macro is defined: **_BSD_SOURCE** on glibc 2.19 and earlier or **_DEFAULT_SOURCE** in glibc 2.19 and later. (By default, these macros are defined; see [feature_test_macros\(7\)](#) for details.) If such a feature test macro is not defined, then **signal()** provides System V semantics.

NOTES

[top](#)

The effects of **signal()** in a multithreaded process are unspecified.

According to POSIX, the behavior of a process is undefined after it ignores a **SIGFPE**, **SIGILL**, or **SIGSEGV** signal that was not generated by [kill\(2\)](#) or [raise\(3\)](#). Integer division by zero has undefined result. On some architectures it will generate a **SIGFPE** signal. (Also dividing the most negative integer by -1 may generate **SIGFPE**.) Ignoring this signal might lead to an endless loop.

See [sigaction\(2\)](#) for details on what happens when the disposition **SIGCHLD** is set to **SIG_IGN**.

See [signal-safety\(7\)](#) for a list of the async-signal-safe functions that can be safely called from inside a signal handler.

SEE ALSO

[top](#)

[kill\(1\)](#), [alarm\(2\)](#), [kill\(2\)](#), [pause\(2\)](#), [sigaction\(2\)](#), [signalfd\(2\)](#), [sigpending\(2\)](#), [sigprocmask\(2\)](#), [sigsuspend\(2\)](#), [bsd_signal\(3\)](#), [killpg\(3\)](#), [raise\(3\)](#), [siginterrupt\(3\)](#), [sigqueue\(3\)](#), [sigsetops\(3\)](#), [sigvec\(3\)](#), [sysv_signal\(3\)](#), [signal\(7\)](#)



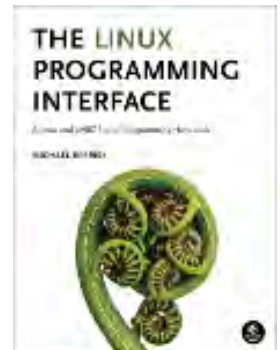
Linux man-pages (unreleased) (date) *signal(2)*

Pages that refer to this page: [alarm\(2\)](#), [getitimer\(2\)](#), [kill\(2\)](#), [pause\(2\)](#), [prctl\(2\)](#), [sigaction\(2\)](#), [sigpending\(2\)](#), [sigprocmask\(2\)](#), [sigreturn\(2\)](#), [sigsuspend\(2\)](#), [sigwaitinfo\(2\)](#), [syscalls\(2\)](#), [wait\(2\)](#), [wait4\(2\)](#), [bsd_signal\(3\)](#), [gsignal\(3\)](#), [killpg\(3\)](#), [profil\(3\)](#), [raise\(3\)](#), [siginterrupt\(3\)](#), [sigqueue\(3\)](#), [sigset\(3\)](#), [sigvec\(3\)](#), [sleep\(3\)](#), [sysv_signal\(3\)](#), [systemd.exec\(5\)](#), [fifo\(7\)](#), [signal\(7\)](#), [signal-safety\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



pause(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

pause(2)

System Calls Manual

pause(2)

NAME [top](#)

pause - wait for signal

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int pause(void);
```

DESCRIPTION [top](#)

`pause()` causes the calling process (or thread) to sleep until a signal is delivered that either terminates the process or causes the invocation of a signal-catching function.

RETURN VALUE [top](#)

`pause()` returns only when a signal was caught and the signal-catching function returned. In this case, `pause()` returns `-1`, and `errno` is set to `EINTR`.

ERRORS [top](#)

EINTR a signal was caught and the `signal-catching` function returned.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.3BSD.

SEE ALSO [top](#)

[kill\(2\)](#), [select\(2\)](#), [signal\(2\)](#), [sigsuspend\(2\)](#)

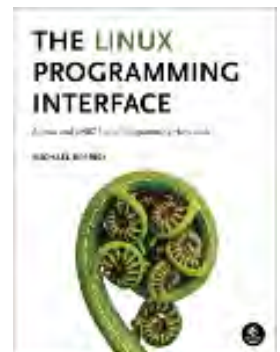
Linux man-pages (unreleased) [\(date\)](#) [pause\(2\)](#)

Pages that refer to this page: [pmsleep\(1\)](#), [alarm\(2\)](#), [ptrace\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [sigprocmask\(2\)](#), [sigsuspend\(2\)](#), [syscalls\(2\)](#), [wait\(2\)](#), [sigset\(3\)](#), [sigvec\(3\)](#), [signal\(7\)](#), [signal-safety\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



strsignal(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 strsignal(3)

Library Functions Manual

strsignal(3)**NAME** [top](#)

strsignal, sigabbrev_np, sigdescr_np, sys_siglist - return string describing signal

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <string.h>
```

```
char *strsignal(int sig);  
const char *sigdescr_np(int sig);  
const char *sigabbrev_np(int sig);
```

```
[[deprecated]] extern const char *const sys_siglist[];
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
sigabbrev_np(), sigdescr_np():  
    _GNU_SOURCE
```

```
strsignal():  
    From glibc 2.10 to glibc 2.31:
```

```
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE
```

sys_siglist:

```
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE
```

DESCRIPTION [top](#)

The **strsignal()** function returns a string describing the signal number passed in the argument *sig*. The string can be used only until the next call to **strsignal()**. The string returned by **strsignal()** is localized according to the **LC_MESSAGES** category in the current locale.

The **sigdescr_np()** function returns a string describing the signal number passed in the argument *sig*. Unlike **strsignal()** this string is not influenced by the current locale.

The **sigabbrev_np()** function returns the abbreviated name of the signal, *sig*. For example, given the value **SIGINT**, it returns the string "INT".

The (deprecated) array *sys_siglist* holds the signal description strings indexed by signal number. The **strsignal()** or the **sigdescr_np()** function should be used instead of this array; see also VERSIONS.

RETURN VALUE [top](#)

The **strsignal()** function returns the appropriate description string, or an unknown signal message if the signal number is invalid. On some systems (but not on Linux), NULL may instead be returned for an invalid signal number.

The **sigdescr_np()** and **sigabbrev_np()** functions return the appropriate description string. The returned string is statically allocated and valid for the lifetime of the program. These functions return NULL for an invalid signal number.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|---|---------------|------------------------------------|
| <code>strsignal()</code> | Thread safety | MT-Unsafe race:strsignal locale |
| <code>sigdescr_np()</code> , <code>sigabbrev_np()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

`strsignal()`
POSIX.1-2008.

`sigdescr_np()`
`sigabbrev_np()`
GNU.

`sys_siglist`
None.

HISTORY [top](#)

`strsignal()`
POSIX.1-2008. Solaris, BSD.

`sigdescr_np()`
`sigabbrev_np()`
glibc 2.32.

`sys_siglist`
Removed in glibc 2.32.

NOTES [top](#)

sigdescr_np() and **sigabbrev_np()** are thread-safe and async-signal-safe.

SEE ALSO [top](#)

[psignal\(3\)](#), [strerror\(3\)](#)

Linux man-pages (unreleased) (date)

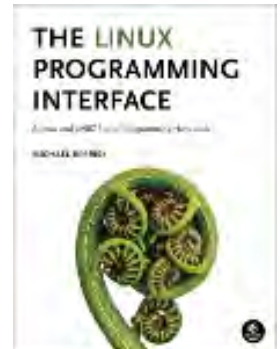
[strsignal\(3\)](#)

Pages that refer to this page: [psignal\(3\)](#), [strerror\(3\)](#), [signal\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



psignal(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [BUGS](#) | [SEE ALSO](#)

psignal(3)

Library Functions Manual

psignal(3)

NAME [top](#)

psignal, psiginfo - print signal description

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <signal.h>
```

```
void psignal(int sig, const char *s);  
void psiginfo(const siginfo_t *pinfo, const char *s);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
psignal():  
    Since glibc 2.19:  
        _DEFAULT_SOURCE  
    glibc 2.19 and earlier:  
        _BSD_SOURCE || _SVID_SOURCE
```

```
psiginfo():  
    _POSIX_C_SOURCE >= 200809L
```

DESCRIPTION [top](#)

The `psignal()` function displays a message on `stderr` consisting of the string `s`, a colon, a space, a string describing the signal number `sig`, and a trailing newline. If the string `s` is NULL or empty, the colon and space are omitted. If `sig` is invalid, the message displayed will indicate an unknown signal.

The `psiginfo()` function is like `psignal()`, except that it displays information about the signal described by `pinfo`, which should point to a valid `siginfo_t` structure. As well as the signal description, `psiginfo()` displays information about the origin of the signal, and other information relevant to the signal (e.g., the relevant memory address for hardware-generated signals, the child process ID for `SIGCHLD`, and the user ID and process ID of the sender, for signals set using `kill(2)` or `sigqueue(3)`).

RETURN VALUE [top](#)

The `psignal()` and `psiginfo()` functions return no value.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|--|---------------|----------------|
| <code>psignal()</code> , <code>psiginfo()</code> | Thread safety | MT-Safe locale |

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

glibc 2.10. POSIX.1-2008, 4.3BSD.

BUGS [top](#)

Up to glibc 2.12, `psiginfo()` had the following bugs:

- In some circumstances, a trailing newline is not printed.
- Additional details are not displayed for real-time signals.

SEE ALSO [top](#)

[sigaction\(2\)](#), [perror\(3\)](#), [strsignal\(3\)](#), [signal\(7\)](#)

Linux man-pages (unreleased) (date)

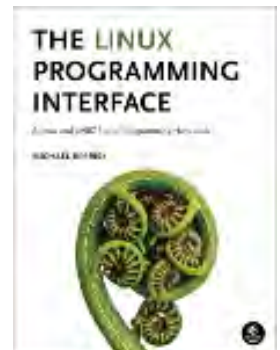
psignal(3)

Pages that refer to this page: [strsignal\(3\)](#), [system_data_types\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



kill(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

 kill(2)

System Calls Manual

kill(2)

NAME [top](#)

kill - send signal to a process

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
kill():  
    _POSIX_C_SOURCE
```

DESCRIPTION [top](#)

The `kill()` system call can be used to send any signal to any process group or process.

If `pid` is positive, then signal `sig` is sent to the process with

the ID specified by *pid*.

If *pid* equals 0, then *sig* is sent to every process in the process group of the calling process.

If *pid* equals -1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (*init*), but see below.

If *pid* is less than -1, then *sig* is sent to every process in the process group whose ID is *-pid*.

If *sig* is 0, then no signal is sent, but existence and permission checks are still performed; this can be used to check for the existence of a process ID or process group ID that the caller is permitted to signal.

For a process to have permission to send a signal, it must either be privileged (under Linux: have the **CAP_KILL** capability in the user namespace of the target process), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of **SIGCONT**, it suffices when the sending and receiving processes belong to the same session. (Historically, the rules were different; see NOTES.)

RETURN VALUE [top](#)

On success (at least one signal was sent), zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EINVAL An invalid signal was specified.

EPERM The calling process does not have permission to send the signal to any of the target processes.

ESRCH The target process or process group does not exist. Note that an existing process might be a zombie, a process that has terminated execution, but has not yet been `wait(2)`ed for.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.3BSD.

Linux notes

Across different kernel versions, Linux has enforced different rules for the permissions required for an unprivileged process to send a signal to another process. In Linux 1.0 to 1.2.2, a signal could be sent if the effective user ID of the sender matched effective user ID of the target, or the real user ID of the sender matched the real user ID of the target. From Linux 1.2.3 until 1.3.77, a signal could be sent if the effective user ID of the sender matched either the real or effective user ID of the target. The current rules, which conform to POSIX.1, were adopted in Linux 1.3.78.

NOTES [top](#)

The only signals that can be sent to process ID 1, the *init* process, are those for which *init* has explicitly installed signal handlers. This is done to assure the system is not brought down accidentally.

POSIX.1 requires that *kill(-1,sig)* send *sig* to all processes that the calling process may send signals to, except possibly for some implementation-defined system processes. Linux allows a process to signal itself, but on Linux the call *kill(-1,sig)* does not signal the calling process.

POSIX.1 requires that if a process sends a signal to itself, and the sending thread does not have the signal blocked, and no other thread has it unblocked or is waiting for it in *sigwait(3)*, at least one unblocked signal must be delivered to the sending thread before the *kill()* returns.

BUGS [top](#)

In Linux 2.6 up to and including Linux 2.6.7, there was a bug that meant that when sending signals to a process group, `kill()` failed with the error `EPERM` if the caller did not have permission to send the signal to *any* (rather than *all*) of the members of the process group. Notwithstanding this error return, the signal was still delivered to all of the processes for which the caller had permission to signal.

SEE ALSO [top](#)

[kill\(1\)](#), [_exit\(2\)](#), [pidfd_send_signal\(2\)](#), [signal\(2\)](#), [tkill\(2\)](#), [exit\(3\)](#), [killpg\(3\)](#), [sigqueue\(3\)](#), [capabilities\(7\)](#), [credentials\(7\)](#), [signal\(7\)](#)

Linux man-pages (unreleased) (date) [kill\(2\)](#)

Pages that refer to this page: [capsh\(1\)](#), [fuser\(1\)](#), [kill\(1@@coreutils\)](#), [kill\(1\)](#), [kill\(1@@procps-ng\)](#), [killall\(1\)](#), [pgrep\(1\)](#), [skill\(1\)](#), [strace\(1\)](#), [clone\(2\)](#), [_exit\(2\)](#), [fcntl\(2\)](#), [getpid\(2\)](#), [getrlimit\(2\)](#), [pause\(2\)](#), [pidfd_open\(2\)](#), [pidfd_send_signal\(2\)](#), [ptrace\(2\)](#), [rt_sigqueueinfo\(2\)](#), [setfsgid\(2\)](#), [setfsuid\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [sigpending\(2\)](#), [sigprocmask\(2\)](#), [sigreturn\(2\)](#), [sigsuspend\(2\)](#), [sigwaitinfo\(2\)](#), [syscalls\(2\)](#), [tkill\(2\)](#), [wait\(2\)](#), [gsignal\(3\)](#), [id_t\(3type\)](#), [killpg\(3\)](#), [psignal\(3\)](#), [pthread_kill\(3\)](#), [raise\(3\)](#), [sd_event_add_child\(3\)](#), [sigpause\(3\)](#), [sigqueue\(3\)](#), [sigset\(3\)](#), [sigvec\(3\)](#), [systemd.exec\(5\)](#), [systemd.kill\(5\)](#), [capabilities\(7\)](#), [cpuset\(7\)](#), [credentials\(7\)](#), [pid_namespaces\(7\)](#), [pthreads\(7\)](#), [signal\(7\)](#), [signal-safety\(7\)](#), [systemd-coredump\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



raise(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

raise(3)

Library Functions Manual

raise(3)

NAME [top](#)

raise - send a signal to the caller

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <signal.h>
```

```
int raise(int sig);
```

DESCRIPTION [top](#)

The `raise()` function sends a signal to the calling process or thread. In a single-threaded program it is equivalent to

```
kill(getpid(), sig);
```

In a multithreaded program it is equivalent to

```
pthread_kill(pthread_self(), sig);
```


If the signal causes a handler to be called, **raise()** will return only after the signal handler has returned.

RETURN VALUE [top](#)

raise() returns 0 on success, and nonzero for failure.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|----------------|---------------|---------|
| raise() | Thread safety | MT-Safe |

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, C89.

Since glibc 2.3.3, **raise()** is implemented by calling [tgkill\(2\)](#), if the kernel supports that system call. Older glibc versions implemented **raise()** using [kill\(2\)](#).

SEE ALSO [top](#)

[getpid\(2\)](#), [kill\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [pthread_kill\(3\)](#), [signal\(7\)](#)

Linux man-pages (unreleased) (date) [raise\(3\)](#)

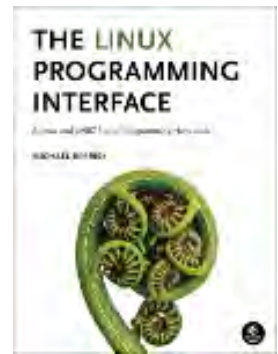
Pages that refer to this page: [sigaction\(2\)](#), [signal\(2\)](#), [sigprocmask\(2\)](#), [abort\(3\)](#), [gsignal\(3\)](#), [pthread_kill\(3\)](#), [sigset\(3\)](#), [sigvec\(3\)](#), [signal\(7\)](#), [signal-safety\(7\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



killpg(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 killpg(3)

Library Functions Manual

killpg(3)**NAME** [top](#)

killpg - send signal to a process group

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <signal.h>
```

```
int killpg(int pgrp, int sig);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
killpg():
    _XOPEN_SOURCE >= 500
    || /* Since glibc 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _BSD_SOURCE
```

DESCRIPTION [top](#)

killpg() sends the signal *sig* to the process group *pgrp*. See [signal\(7\)](#) for a list of signals.

If *pgrp* is 0, **killpg()** sends the signal to the calling process's process group. (POSIX says: if *pgrp* is less than or equal to 1, the behavior is undefined.)

For the permissions required to send a signal to another process, see [kill\(2\)](#).

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EINVAL *sig* is not a valid signal number.

EPERM The process does not have permission to send the signal to any of the target processes. For the required permissions, see [kill\(2\)](#).

ESRCH No process can be found in the process group specified by *pgrp*.

ESRCH The process group was given as 0 but the sending process does not have a process group.

VERSIONS [top](#)

There are various differences between the permission checking in BSD-type systems and System V-type systems. See the POSIX rationale for [kill\(3p\)](#). A difference not mentioned by POSIX concerns the return value **EPERM**: BSD documents that no signal is sent and **EPERM** returned when the permission check failed for at least one target process, while POSIX documents **EPERM** only when the permission check failed for all target processes.

C library/kernel differences

On Linux, **killpg()** is implemented as a library function that makes the call *kill(-pgrp, sig)*.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.4BSD (first appeared in 4BSD).

SEE ALSO [top](#)

[getpgrp\(2\)](#), [kill\(2\)](#), [signal\(2\)](#), [capabilities\(7\)](#), [credentials\(7\)](#)

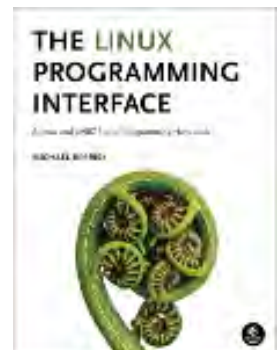
Linux man-pages (unreleased) (date) [killpg\(3\)](#)

Pages that refer to this page: [kill\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [credentials\(7\)](#), [signal\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



signal-safety(7) — Linux manual page

[NAME](#) | [DESCRIPTION](#) | [SEE ALSO](#)

Search online pages

signal-safety(7) Miscellaneous Information Manual *signal-safety(7)*

NAME [top](#)

signal-safety - async-signal-safe functions

DESCRIPTION [top](#)

An *async-signal-safe* function is one that can be safely called from within a signal handler. Many functions are *not* async-signal-safe. In particular, nonreentrant functions are generally unsafe to call from a signal handler.

The kinds of issues that render a function unsafe can be quickly understood when one considers the implementation of the *stdio* library, all of whose functions are not async-signal-safe.

When performing buffered I/O on a file, the *stdio* functions must maintain a statically allocated data buffer along with associated counters and indexes (or pointers) that record the amount of data and the current position in the buffer. Suppose that the main program is in the middle of a call to a *stdio* function such as [printf\(3\)](#) where the buffer and associated variables have been partially updated. If, at that moment, the program is interrupted by a signal handler that also calls [printf\(3\)](#), then the second call to [printf\(3\)](#) will operate on inconsistent data, with unpredictable results.

To avoid problems with unsafe functions, there are two possible choices:

- (a) Ensure that (1) the signal handler calls only `async-signal-safe` functions, and (2) the signal handler itself is reentrant with respect to global variables in the main program.
- (b) Block signal delivery in the main program when calling functions that are unsafe or operating on global data that is also accessed by the signal handler.

Generally, the second choice is difficult in programs of any complexity, so the first choice is taken.

POSIX.1 specifies a set of functions that an implementation must make `async-signal-safe`. (An implementation may provide safe implementations of additional functions, but this is not required by the standard and other implementations may not provide the same guarantees.)

In general, a function is `async-signal-safe` either because it is reentrant or because it is atomic with respect to signals (i.e., its execution can't be interrupted by a signal handler).

The set of functions required to be `async-signal-safe` by POSIX.1 is shown in the following table. The functions not otherwise noted were required to be `async-signal-safe` in POSIX.1-2001; the table details changes in the subsequent standards.

| Function | Notes |
|-------------------------------|---------------------------|
| <code>abort(3)</code> | Added in POSIX.1-2001 TC1 |
| <code>accept(2)</code> | |
| <code>access(2)</code> | |
| <code>aio_error(3)</code> | |
| <code>aio_return(3)</code> | |
| <code>aio_suspend(3)</code> | See notes below |
| <code>alarm(2)</code> | |
| <code>bind(2)</code> | |
| <code>cfgetispeed(3)</code> | |
| <code>cfgetospeed(3)</code> | |
| <code>cfsetispeed(3)</code> | |
| <code>cfsetospeed(3)</code> | |
| <code>chdir(2)</code> | |
| <code>chmod(2)</code> | |
| <code>chown(2)</code> | |
| <code>clock_gettime(2)</code> | |



close(2)
connect(2)
creat(2)
dup(2)
dup2(2)
execl(3) Added in POSIX.1-2008;
see notes below
execle(3) See notes below
execv(3) Added in POSIX.1-2008
execve(2)
_exit(2)
_Exit(2)
faccessat(2) Added in POSIX.1-2008
fchdir(2) Added in POSIX.1-2008 TC1
fchmod(2)
fchmodat(2) Added in POSIX.1-2008
fchown(2)
fchownat(2) Added in POSIX.1-2008
fcntl(2)
fdatasync(2)
fexecve(3) Added in POSIX.1-2008
ffs(3) Added in POSIX.1-2008 TC2
fork(2) See notes below
fstat(2)
fstatat(2) Added in POSIX.1-2008
fsync(2)
ftruncate(2)
futimens(3) Added in POSIX.1-2008
getegid(2)
geteuid(2)
getgid(2)
getgroups(2)
getpeername(2)
getpgrp(2)
getpid(2)
getppid(2)
getsockname(2)
getsockopt(2)
getuid(2)
htonl(3) Added in POSIX.1-2008 TC2
htons(3) Added in POSIX.1-2008 TC2
kill(2)
link(2)
linkat(2) Added in POSIX.1-2008



| | |
|--|---|
| <code>listen(2)</code> | |
| <code>longjmp(3)</code> | Added in POSIX.1-2008 TC2; see notes below |
| <code>lseek(2)</code> | |
| <code>lstat(2)</code> | |
| <code>memccpy(3)</code> | Added in POSIX.1-2008 TC2 |
| <code>memchr(3)</code> | Added in POSIX.1-2008 TC2 |
| <code>memcmp(3)</code> | Added in POSIX.1-2008 TC2 |
| <code>memcpy(3)</code> | Added in POSIX.1-2008 TC2 |
| <code>memmove(3)</code> | Added in POSIX.1-2008 TC2 |
| <code>memset(3)</code> | Added in POSIX.1-2008 TC2 |
| <code>mkdir(2)</code> | |
| <code>mkdirat(2)</code> | Added in POSIX.1-2008 |
| <code>mkfifo(3)</code> | |
| <code>mkfifoat(3)</code> | Added in POSIX.1-2008 |
| <code>mknod(2)</code> | Added in POSIX.1-2008 |
| <code>mknodat(2)</code> | Added in POSIX.1-2008 |
| <code>ntohl(3)</code> | Added in POSIX.1-2008 TC2 |
| <code>ntohs(3)</code> | Added in POSIX.1-2008 TC2 |
| <code>open(2)</code> | |
| <code>openat(2)</code> | Added in POSIX.1-2008 |
| <code>pause(2)</code> | |
| <code>pipe(2)</code> | |
| <code>poll(2)</code> | |
| <code>posix_trace_event(3)</code> | |
| <code>pselect(2)</code> | |
| <code>pthread_kill(3)</code> | Added in POSIX.1-2008 TC1 |
| <code>pthread_self(3)</code> | Added in POSIX.1-2008 TC1 |
| <code>pthread_sigmask(3)</code> | Added in POSIX.1-2008 TC1 |
| <code>raise(3)</code> | |
| <code>read(2)</code> | |
| <code>readlink(2)</code> | |
| <code>readlinkat(2)</code> | Added in POSIX.1-2008 |
| <code>recv(2)</code> | |
| <code>recvfrom(2)</code> | |
| <code>recvmsg(2)</code> | |
| <code>rename(2)</code> | |
| <code>renameat(2)</code> | Added in POSIX.1-2008 |
| <code>rmdir(2)</code> | |
| <code>select(2)</code> | |
| <code>sem_post(3)</code> | |
| <code>send(2)</code> | |
| <code>sendmsg(2)</code> | |
| <code>sendto(2)</code> | |



| | |
|--------------------|---|
| setgid(2) | |
| setpgid(2) | |
| setsid(2) | |
| setsockopt(2) | |
| setuid(2) | |
| shutdown(2) | |
| sigaction(2) | |
| sigaddset(3) | |
| sigdelset(3) | |
| sigemptyset(3) | |
| sigfillset(3) | |
| sigismember(3) | |
| siglongjmp(3) | Added in POSIX.1-2008 TC2; see notes below |
| signal(2) | |
| sigpause(3) | |
| sigpending(2) | |
| sigprocmask(2) | |
| sigqueue(2) | |
| sigset(3) | |
| sigsuspend(2) | |
| sleep(3) | |
| socketatmark(3) | Added in POSIX.1-2001 TC2 |
| socket(2) | |
| socketpair(2) | |
| stat(2) | |
| stpcpy(3) | Added in POSIX.1-2008 TC2 |
| stpncpy(3) | Added in POSIX.1-2008 TC2 |
| strcat(3) | Added in POSIX.1-2008 TC2 |
| strchr(3) | Added in POSIX.1-2008 TC2 |
| strcmp(3) | Added in POSIX.1-2008 TC2 |
| strcpy(3) | Added in POSIX.1-2008 TC2 |
| strcspn(3) | Added in POSIX.1-2008 TC2 |
| strlen(3) | Added in POSIX.1-2008 TC2 |
| strncat(3) | Added in POSIX.1-2008 TC2 |
| strncmp(3) | Added in POSIX.1-2008 TC2 |
| strncpy(3) | Added in POSIX.1-2008 TC2 |
| strnlen(3) | Added in POSIX.1-2008 TC2 |
| strpbrk(3) | Added in POSIX.1-2008 TC2 |
| strrchr(3) | Added in POSIX.1-2008 TC2 |
| strspn(3) | Added in POSIX.1-2008 TC2 |
| strstr(3) | Added in POSIX.1-2008 TC2 |
| strtok_r(3) | Added in POSIX.1-2008 TC2 |
| symlink(2) | |



| | | |
|---------------------|-----------------------|-----|
| symlinkat(2) | Added in POSIX.1-2008 | |
| tcdrain(3) | | |
| tcflow(3) | | |
| tcflush(3) | | |
| tcgetattr(3) | | |
| tcgetpgrp(3) | | |
| tcsendbreak(3) | | |
| tcsetattr(3) | | |
| tcsetpgrp(3) | | |
| time(2) | | |
| timer_getoverrun(2) | | |
| timer_gettime(2) | | |
| timer_settime(2) | | |
| times(2) | | |
| umask(2) | | |
| uname(2) | | |
| unlink(2) | | |
| unlinkat(2) | Added in POSIX.1-2008 | |
| utime(2) | | |
| utimensat(2) | Added in POSIX.1-2008 | |
| utimes(2) | Added in POSIX.1-2008 | |
| wait(2) | | |
| waitpid(2) | | |
| wcpcpy(3) | Added in POSIX.1-2008 | TC2 |
| wcpncpy(3) | Added in POSIX.1-2008 | TC2 |
| wcscat(3) | Added in POSIX.1-2008 | TC2 |
| wcschr(3) | Added in POSIX.1-2008 | TC2 |
| wcscmp(3) | Added in POSIX.1-2008 | TC2 |
| wcscpy(3) | Added in POSIX.1-2008 | TC2 |
| wcscspn(3) | Added in POSIX.1-2008 | TC2 |
| wcslen(3) | Added in POSIX.1-2008 | TC2 |
| wcsncat(3) | Added in POSIX.1-2008 | TC2 |
| wcsncmp(3) | Added in POSIX.1-2008 | TC2 |
| wcsncpy(3) | Added in POSIX.1-2008 | TC2 |
| wcsnlen(3) | Added in POSIX.1-2008 | TC2 |
| wcspbrk(3) | Added in POSIX.1-2008 | TC2 |
| wcsrchr(3) | Added in POSIX.1-2008 | TC2 |
| wcsspn(3) | Added in POSIX.1-2008 | TC2 |
| wcsstr(3) | Added in POSIX.1-2008 | TC2 |
| wcstok(3) | Added in POSIX.1-2008 | TC2 |
| wmemchr(3) | Added in POSIX.1-2008 | TC2 |
| wmemcmp(3) | Added in POSIX.1-2008 | TC2 |
| wmemcpy(3) | Added in POSIX.1-2008 | TC2 |
| wmemmove(3) | Added in POSIX.1-2008 | TC2 |



`wmemset(3)`
`write(2)`

Added in POSIX.1-2008 TC2

Notes:

- POSIX.1-2001 and POSIX.1-2001 TC2 required the functions `fpathconf(3)`, `pathconf(3)`, and `sysconf(3)` to be `async-signal-safe`, but this requirement was removed in POSIX.1-2008.
- If a signal handler interrupts the execution of an unsafe function, and the handler terminates via a call to `longjmp(3)` or `siglongjmp(3)` and the program subsequently calls an unsafe function, then the behavior of the program is undefined.
- POSIX.1-2001 TC1 clarified that if an application calls `fork(2)` from a signal handler and any of the fork handlers registered by `pthread_atfork(3)` calls a function that is not `async-signal-safe`, the behavior is undefined. A future revision of the standard is likely to remove `fork(2)` from the list of `async-signal-safe` functions.
- Asynchronous signal handlers that call functions which are cancellation points and nest over regions of deferred cancellation may trigger cancellation whose behavior is as if asynchronous cancellation had occurred and may cause application state to become inconsistent.

errno

Fetching and setting the value of `errno` is `async-signal-safe` provided that the signal handler saves `errno` on entry and restores its value before returning.

Deviations in the GNU C library

The following known deviations from the standard occur in the GNU C library:

- Before glibc 2.24, `execl(3)` and `execle(3)` employed `realloc(3)` internally and were consequently not `async-signal-safe`. This was fixed in glibc 2.24.
- The glibc implementation of `aio_suspend(3)` is not `async-signal-safe` because it uses `pthread_mutex_lock(3)` internally.



SEE ALSO [top](#)

[sigaction\(2\)](#), [signal\(7\)](#), [standards\(7\)](#)

Linux man-pages (unreleased) **(date)**

signal-safety(7)

Pages that refer to this page: [fork\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [pthread_atfork\(3\)](#), [sd_journal_get_fd\(3\)](#), [sd_journal_print\(3\)](#), [sd_journal_stream_fd\(3\)](#), [seccomp_load\(3\)](#), [seccomp_precompute\(3\)](#), [sem_post\(3\)](#), [setjmp\(3\)](#), [attributes\(7\)](#), [signal\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sigsetops(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 SIGSETOPS(3)

Library Functions Manual

SIGSETOPS(3)**NAME** [top](#)

sigemptyset, sigfillset, sigaddset, sigdelset, sigismember -
POSIX signal set operations

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);

int sigismember(const sigset_t *set, int signum);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
sigemptyset(), sigfillset(), sigaddset(), sigdelset(),
sigismember():
    _POSIX_C_SOURCE
```

DESCRIPTION [top](#)

These functions allow the manipulation of POSIX signal sets.

sigemptyset() initializes the signal set given by *set* to empty, with all signals excluded from the set.

sigfillset() initializes *set* to full, including all signals.

sigaddset() and **sigdelset()** add and delete respectively signal *signal* from *set*.

sigismember() tests whether *signal* is a member of *set*.

Objects of type *sigset_t* must be initialized by a call to either **sigemptyset()** or **sigfillset()** before being passed to the functions **sigaddset()**, **sigdelset()**, and **sigismember()** or the additional glibc functions described below (**sigisemptyset()**, **sigandset()**, and **sigorset()**). The results are undefined if this is not done.

RETURN VALUE [top](#)

sigemptyset(), **sigfillset()**, **sigaddset()**, and **sigdelset()** return 0 on success and -1 on error.

sigismember() returns 1 if *signal* is a member of *set*, 0 if *signal* is not a member, and -1 on error.

On error, these functions set *errno* to indicate the error.

ERRORS [top](#)

EINVAL *signal* is not a valid signal.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------|-----------|-------|
|-----------|-----------|-------|



| | | |
|--|---------------|---------|
| <code>sigemptyset()</code> , <code>sigfillset()</code> , <code>sigaddset()</code> , <code>sigdelset()</code> , <code>sigismember()</code> , <code>sigisemptyset()</code> , <code>sigorset()</code> , <code>sigandset()</code> | Thread safety | MT-Safe |
|--|---------------|---------|

VERSIONS [top](#)

GNU

If the `_GNU_SOURCE` feature test macro is defined, then `<signal.h>` exposes three other functions for manipulating signal sets:

```
int sigisemptyset(const sigset_t *set);
int sigorset(sigset_t *dest, const sigset_t *left,
             const sigset_t *right);
int sigandset(sigset_t *dest, const sigset_t *left,
              const sigset_t *right);
```

`sigisemptyset()` returns 1 if `set` contains no signals, and 0 otherwise.

`sigorset()` places the union of the sets `left` and `right` in `dest`. `sigandset()` places the intersection of the sets `left` and `right` in `dest`. Both functions return 0 on success, and -1 on failure.

These functions are nonstandard (a few other systems provide similar functions) and their use should be avoided in portable applications.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

NOTES [top](#)

When creating a filled signal set, the glibc `sigfillset()` function does not include the two real-time signals used internally by the NPTL threading implementation. See [nptl\(7\)](#) for details.

SEE ALSO [top](#)

[sigaction\(2\)](#), [sigpending\(2\)](#), [sigprocmask\(2\)](#), [sigsuspend\(2\)](#)

Linux man-pages (unreleased) (date)

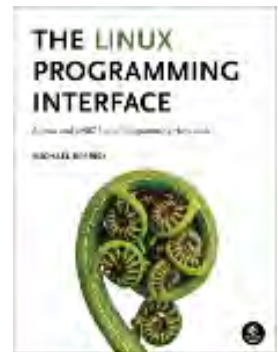
SIGSETOPS(3)

Pages that refer to this page: [sigaction\(2\)](#), [signal\(2\)](#), [signalfd\(2\)](#), [sigpending\(2\)](#), [sigprocmask\(2\)](#), [sigsuspend\(2\)](#), [sigwaitinfo\(2\)](#), [pthread_attr_setsigmask_np\(3\)](#), [pthread_sigmask\(3\)](#), [sigwait\(3\)](#), [nptl\(7\)](#), [signal\(7\)](#), [signal-safety\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sigprocmask(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

sigprocmask(2)

System Calls Manual

sigprocmask(2)

NAME [top](#)

sigprocmask, rt_sigprocmask - examine and change blocked signals

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <signal.h>
```

```
/* Prototype for the glibc wrapper function */  
int sigprocmask(int how, const sigset_t *_Nullable restrict set,  
               sigset_t *_Nullable restrict oldset);
```

```
#include <signal.h>           /* Definition of SIG_* constants */  
#include <sys/syscall.h>     /* Definition of SYS_* constants */  
#include <unistd.h>
```

```
/* Prototype for the underlying system call */  
int syscall(SYS_rt_sigprocmask, int how,  
           const kernel_sigset_t *_Nullable set,  
           kernel_sigset_t *_Nullable oldset,  
           size_t sigsetsize);
```

```
/* Prototype for the legacy system call */  
[[deprecated]] int syscall(SYS_sigprocmask, int how,  
                          const old_kernel_sigset_t *_Nullable set,
```

```
old_kernel_sigset_t *_Nullable oldset);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
sigprocmask():
    _POSIX_C_SOURCE
```

DESCRIPTION [top](#)

sigprocmask() is used to fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller (see also [signal\(7\)](#) for more details).

The behavior of the call is dependent on the value of *how*, as follows.

SIG_BLOCK

The set of blocked signals is the union of the current set and the *set* argument.

SIG_UNBLOCK

The signals in *set* are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.

SIG_SETMASK

The set of blocked signals is set to the argument *set*.

If *oldset* is non-NULL, the previous value of the signal mask is stored in *oldset*.

If *set* is NULL, then the signal mask is unchanged (i.e., *how* is ignored), but the current value of the signal mask is nevertheless returned in *oldset* (if it is not NULL).

A set of functions for modifying and inspecting variables of type *sigset_t* ("signal sets") is described in [sigsetops\(3\)](#).

The use of **sigprocmask()** is unspecified in a multithreaded process; see [pthread_sigmask\(3\)](#).

RETURN VALUE [top](#)

sigprocmask() returns 0 on success. On failure, -1 is returned and *errno* is set to indicate the error.

ERRORS [top](#)

EFAULT The *set* or *oldset* argument points outside the process's allocated address space.

EINVAL Either the value specified in *how* was invalid or the kernel does not support the size passed in *sigsetsize*.

VERSIONS [top](#)

C library/kernel differences

The kernel's definition of *sigset_t* differs in size from that used by the C library. In this manual page, the former is referred to as *kernel_sigset_t* (it is nevertheless named *sigset_t* in the kernel sources).

The glibc wrapper function for **sigprocmask()** silently ignores attempts to block the two real-time signals that are used internally by the NPTL threading implementation. See [nptl\(7\)](#) for details.

The original Linux system call was named **sigprocmask()**. However, with the addition of real-time signals in Linux 2.2, the fixed-size, 32-bit *sigset_t* (referred to as *old_kernel_sigset_t* in this manual page) type supported by that system call was no longer fit for purpose. Consequently, a new system call, **rt_sigprocmask()**, was added to support an enlarged *sigset_t* type (referred to as *kernel_sigset_t* in this manual page). The new system call takes a fourth argument, *size_t sigsetsize*, which specifies the size in bytes of the signal sets in *set* and *oldset*. This argument is currently required to have a fixed architecture specific value (equal to *sizeof(kernel_sigset_t)*).

The glibc **sigprocmask()** wrapper function hides these details from us, transparently calling **rt_sigprocmask()** when the kernel provides it.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

NOTES [top](#)

It is not possible to block **SIGKILL** or **SIGSTOP**. Attempts to do so are silently ignored.

Each of the threads in a process has its own signal mask.

A child created via `fork(2)` inherits a copy of its parent's signal mask; the signal mask is preserved across `execve(2)`.

If **SIGBUS**, **SIGFPE**, **SIGILL**, or **SIGSEGV** are generated while they are blocked, the result is undefined, unless the signal was generated by `kill(2)`, `sigqueue(3)`, or `raise(3)`.

See `sigsetops(3)` for details on manipulating signal sets.

Note that it is permissible (although not very useful) to specify both `set` and `oldset` as NULL.

SEE ALSO [top](#)

`kill(2)`, `pause(2)`, `sigaction(2)`, `signal(2)`, `sigpending(2)`, `sigsuspend(2)`, `pthread_sigmask(3)`, `sigqueue(3)`, `sigsetops(3)`, `signal(7)`

Linux man-pages (unreleased) (date) [sigprocmask\(2\)](#)

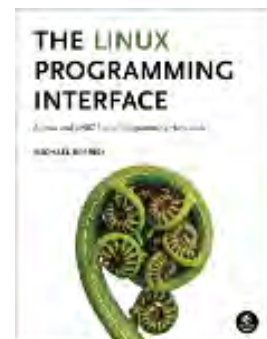
Pages that refer to this page: [env\(1\)](#), [clone\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [poll\(2\)](#), [ptrace\(2\)](#), [rt_sigqueueinfo\(2\)](#), [seccomp\(2\)](#), [select\(2\)](#), [select_tut\(2\)](#), [sgetmask\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [signalfd\(2\)](#), [sigpending\(2\)](#), [sigsuspend\(2\)](#), [sigwaitinfo\(2\)](#), [syscalls\(2\)](#), [getcontext\(3\)](#), [makecontext\(3\)](#), [posix_spawn\(3\)](#), [pthread_attr_setsigmask_np\(3\)](#), [pthread_sigmask\(3\)](#), [sd_event_add_child\(3\)](#), [sd_event_add_signal\(3\)](#), [sigpause\(3\)](#), [sigset\(3\)](#), [sigsetops\(3\)](#), [sigvec\(3\)](#), [system\(3\)](#), [systemd.exec\(5\)](#), [nptl\(7\)](#), [signal\(7\)](#), [signal-safety\(7\)](#), [system_data_types\(7\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sigpending(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

 sigpending(2)

System Calls Manual

sigpending(2)**NAME** [top](#)

sigpending, rt_sigpending - examine pending signals

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <signal.h>
```

```
int sigpending(sigset_t *set);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
sigpending():  
    _POSIX_C_SOURCE
```

DESCRIPTION [top](#)

sigpending() returns the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). The mask of pending signals is returned in *set*.

RETURN VALUE [top](#)

`sigpending()` returns 0 on success. On failure, -1 is returned and `errno` is set to indicate the error.

ERRORS [top](#)

EFAULT `set` points to memory which is not a valid part of the process address space.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

C library/kernel differences

The original Linux system call was named `sigpending()`. However, with the addition of real-time signals in Linux 2.2, the fixed-size, 32-bit `sigset_t` argument supported by that system call was no longer fit for purpose. Consequently, a new system call, `rt_sigpending()`, was added to support an enlarged `sigset_t` type. The new system call takes a second argument, `size_t sigsetsize`, which specifies the size in bytes of the signal set in `set`. The glibc `sigpending()` wrapper function hides these details from us, transparently calling `rt_sigpending()` when the kernel provides it.

NOTES [top](#)

See [sigsetops\(3\)](#) for details on manipulating signal sets.

If a signal is both blocked and has a disposition of "ignored", it is *not* added to the mask of pending signals when generated.

The set of signals that is pending for a thread is the union of the set of signals that is pending for that thread and the set of signals that is pending for the process as a whole; see

[signal\(7\)](#).

A child created via [fork\(2\)](#) initially has an empty pending signal set; the pending signal set is preserved across an [execve\(2\)](#).

BUGS [top](#)

Up to and including glibc 2.2.1, there is a bug in the wrapper function for [sigpending\(\)](#) which means that information about pending real-time signals is not correctly returned.

SEE ALSO [top](#)

[kill\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [sigprocmask\(2\)](#), [sigsuspend\(2\)](#), [sigsetops\(3\)](#), [signal\(7\)](#)

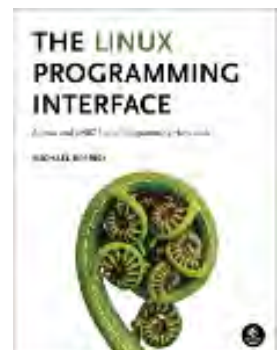
Linux man-pages (unreleased) (date) [sigpending\(2\)](#)

Pages that refer to this page: [clone\(2\)](#), [fork\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [sigprocmask\(2\)](#), [sigwaitinfo\(2\)](#), [syscalls\(2\)](#), [pthread_create\(3\)](#), [pthread_kill\(3\)](#), [pthread_sigmask\(3\)](#), [sigsetops\(3\)](#), [sigwait\(3\)](#), [signal\(7\)](#), [signal-safety\(7\)](#), [system_data_types\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sigsuspend(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 sigsuspend(2)

System Calls Manual

sigsuspend(2)

NAME [top](#)

sigsuspend, rt_sigsuspend - wait for a signal

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <signal.h>
```

```
int sigsuspend(const sigset_t *mask);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
sigsuspend():  
    _POSIX_C_SOURCE
```

DESCRIPTION [top](#)

sigsuspend() temporarily replaces the signal mask of the calling thread with the mask given by *mask* and then suspends the thread until delivery of a signal whose action is to invoke a signal handler or to terminate a process.

If the signal terminates the process, then **sigsuspend()** does not return. If the signal is caught, then **sigsuspend()** returns after the signal handler returns, and the signal mask is restored to the state before the call to **sigsuspend()**.

It is not possible to block **SIGKILL** or **SIGSTOP**; specifying these signals in *mask*, has no effect on the thread's signal mask.

RETURN VALUE [top](#)

sigsuspend() always returns -1, with *errno* set to indicate the error (normally, **EINTR**).

ERRORS [top](#)

EFAULT *mask* points to memory which is not a valid part of the process address space.

EINTR The call was interrupted by a signal; [signal\(7\)](#).

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

C library/kernel differences

The original Linux system call was named **sigsuspend()**. However, with the addition of real-time signals in Linux 2.2, the fixed-size, 32-bit *sigset_t* type supported by that system call was no longer fit for purpose. Consequently, a new system call, **rt_sigsuspend()**, was added to support an enlarged *sigset_t* type. The new system call takes a second argument, *size_t sigsetsize*, which specifies the size in bytes of the signal set in *mask*. This argument is currently required to have the value *sizeof(sigset_t)* (or the error **EINVAL** results). The glibc **sigsuspend()** wrapper function hides these details from us,

transparently calling `rt_sigsuspend()` when the kernel provides it.

NOTES [top](#)

Normally, `sigsuspend()` is used in conjunction with `sigprocmask(2)` in order to prevent delivery of a signal during the execution of a critical code section. The caller first blocks the signals with `sigprocmask(2)`. When the critical code has completed, the caller then waits for the signals by calling `sigsuspend()` with the signal mask that was returned by `sigprocmask(2)` (in the *oldset* argument).

See [sigsetops\(3\)](#) for details on manipulating signal sets.

SEE ALSO [top](#)

[kill\(2\)](#), [pause\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [sigprocmask\(2\)](#), [sigwaitinfo\(2\)](#), [sigsetops\(3\)](#), [sigwait\(3\)](#), [signal\(7\)](#)

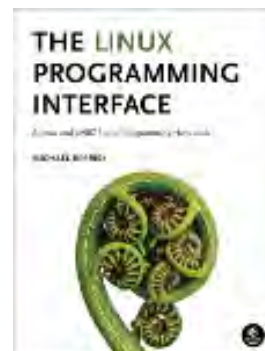
Linux man-pages (unreleased) (date) [sigsuspend\(2\)](#)

Pages that refer to this page: [pause\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [sigpending\(2\)](#), [sigprocmask\(2\)](#), [syscalls\(2\)](#), [sigpause\(3\)](#), [sigset\(3\)](#), [sigsetops\(3\)](#), [sigwait\(3\)](#), [signal\(7\)](#), [signal-safety\(7\)](#), [system_data_types\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sigaction(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [EXAMPLES](#) | [SEE ALSO](#)

 sigaction(2)

System Calls Manual

sigaction(2)**NAME** [top](#)

sigaction, rt_sigaction - examine and change a signal action

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <signal.h>
```

```
int sigaction(int signum,
              const struct sigaction *_Nullable restrict act,
              struct sigaction *_Nullable restrict oldact);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
sigaction():
    _POSIX_C_SOURCE
```

```
siginfo_t:
    _POSIX_C_SOURCE >= 199309L
```

DESCRIPTION [top](#)

The **sigaction()** system call is used to change the action taken by a process on receipt of a specific signal. (See [signal\(7\)](#) for an overview of signals.)

signum specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

If *act* is non-NULL, the new action for signal *signum* is installed from *act*. If *oldact* is non-NULL, the previous action is saved in *oldact*.

The *sigaction* structure is defined as something like:

```
struct sigaction {
    void      (*sa_handler)(int);
    void      (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t  sa_mask;
    int       sa_flags;
    void      (*sa_restorer)(void);
};
```

On some architectures a union is involved: do not assign to both *sa_handler* and *sa_sigaction*.

The *sa_restorer* field is not intended for application use. (POSIX does not specify a *sa_restorer* field.) Some further details of the purpose of this field can be found in [sigreturn\(2\)](#).

sa_handler specifies the action to be associated with *signum* and can be one of the following:

- **SIG_DFL** for the default action.
- **SIG_IGN** to ignore this signal.
- A pointer to a signal handling function. This function receives the signal number as its only argument.

If **SA_SIGINFO** is specified in *sa_flags*, then *sa_sigaction* (instead of *sa_handler*) specifies the signal-handling function for *signum*. This function receives three arguments, as described below.

sa_mask specifies a mask of signals which should be blocked (i.e., added to the signal mask of the thread in which the signal handler is invoked) during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the **SA_NODEFER** flag is used.

sa_flags specifies a set of flags which modify the behavior of

the signal. It is formed by the bitwise OR of zero or more of the following:

SA_NOCLDSTOP

If *signum* is **SIGCHLD**, do not receive notification when child processes stop (i.e., when they receive one of **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, or **SIGTTOU**) or resume (i.e., they receive **SIGCONT**) (see [wait\(2\)](#)). This flag is meaningful only when establishing a handler for **SIGCHLD**.

SA_NOCLDWAIT (since Linux 2.6)

If *signum* is **SIGCHLD**, do not transform children into zombies when they terminate. See also [waitpid\(2\)](#). This flag is meaningful only when establishing a handler for **SIGCHLD**, or when setting that signal's disposition to **SIG_DFL**.

If the **SA_NOCLDWAIT** flag is set when establishing a handler for **SIGCHLD**, POSIX.1 leaves it unspecified whether a **SIGCHLD** signal is generated when a child process terminates. On Linux, a **SIGCHLD** signal is generated in this case; on some other implementations, it is not.

SA_NODEFER

Do not add the signal to the thread's signal mask while the handler is executing, unless the signal is specified in *act.sa_mask*. Consequently, a further instance of the signal may be delivered to the thread while it is executing the handler. This flag is meaningful only when establishing a signal handler.

SA_NOMASK is an obsolete, nonstandard synonym for this flag.

SA_ONSTACK

Call the signal handler on an alternate signal stack provided by [sigaltstack\(2\)](#). If an alternate stack is not available, the default stack will be used. This flag is meaningful only when establishing a signal handler.

SA_RESETHAND

Restore the signal action to the default upon entry to the signal handler. This flag is meaningful only when establishing a signal handler.

SA_ONESHOT is an obsolete, nonstandard synonym for this flag.

SA_RESTART

Provide behavior compatible with BSD signal semantics by making certain system calls restartable across signals. This flag is meaningful only when establishing a signal handler. See [signal\(7\)](#) for a discussion of system call restarting.

SA_RESTORER

Not intended for application use. This flag is used by C libraries to indicate that the *sa_restorer* field contains the address of a "signal trampoline". See [sigreturn\(2\)](#) for more details.

SA_SIGINFO (since Linux 2.2)

The signal handler takes three arguments, not one. In this case, *sa_sigaction* should be set instead of *sa_handler*. This flag is meaningful only when establishing a signal handler.

SA_UNSUPPORTED (since Linux 5.11)

Used to dynamically probe for flag bit support.

If an attempt to register a handler succeeds with this flag set in *act->sa_flags* alongside other flags that are potentially unsupported by the kernel, and an immediately subsequent **sigaction()** call specifying the same signal number and with a non-NULL *oldact* argument yields **SA_UNSUPPORTED clear** in *oldact->sa_flags*, then *oldact->sa_flags* may be used as a bitmask describing which of the potentially unsupported flags are, in fact, supported. See the section "Dynamically probing for flag bit support" below for more details.

SA_EXPOSE_TAGBITS (since Linux 5.11)

Normally, when delivering a signal, an architecture-specific set of tag bits are cleared from the *si_addr* field of *siginfo_t*. If this flag is set, an architecture-specific subset of the tag bits will be preserved in *si_addr*.

Programs that need to be compatible with Linux versions older than 5.11 must use **SA_UNSUPPORTED** to probe for support.

The *siginfo_t* argument to a **SA_SIGINFO** handler

When the **SA_SIGINFO** flag is specified in *act.sa_flags*, the signal handler address is passed via the *act.sa_sigaction* field. This handler takes three arguments, as follows:


```

void
handler(int sig, siginfo_t *info, void *ucontext)
{
    ...
}

```

These three arguments are as follows

- sig* The number of the signal that caused invocation of the handler.
- info* A pointer to a *siginfo_t*, which is a structure containing further information about the signal, as described below.
- ucontext*
This is a pointer to a *ucontext_t* structure, cast to *void **. The structure pointed to by this field contains signal context information that was saved on the user-space stack by the kernel; for details, see [sigreturn\(2\)](#). Further information about the *ucontext_t* structure can be found in [getcontext\(3\)](#) and [signal\(7\)](#). Commonly, the handler function doesn't make any use of the third argument.

The *siginfo_t* data type is a structure with the following fields:

```

siginfo_t {
    int      si_signo;      /* Signal number */
    int      si_errno;     /* An errno value */
    int      si_code;      /* Signal code */
    int      si_trapno;     /* Trap number that caused
                           hardware-generated signal
                           (unused on most architectures) */
    pid_t    si_pid;       /* Sending process ID */
    uid_t    si_uid;       /* Real user ID of sending process */
    int      si_status;     /* Exit value or signal */
    clock_t  si_utime;     /* User time consumed */
    clock_t  si_stime;     /* System time consumed */
    union sigval si_value; /* Signal value */
    int      si_int;       /* POSIX.1b signal */
    void     *si_ptr;      /* POSIX.1b signal */
    int      si_overrun;   /* Timer overrun count;
                           POSIX.1b timers */
    int      si_timerid;   /* Timer ID; POSIX.1b timers */
    void     *si_addr;     /* Memory location which caused fault */
    long     si_band;      /* Band event (was int in
                           glibc 2.3.2 and earlier) */
}

```



```

int      si_fd;          /* File descriptor */
short    si_addr_lsb;   /* Least significant bit of address
                        (since Linux 2.6.32) */
void     *si_lower;     /* Lower bound when address violation
                        occurred (since Linux 3.19) */
void     *si_upper;     /* Upper bound when address violation
                        occurred (since Linux 3.19) */
int      si_pkey;       /* Protection key on PTE that caused
                        fault (since Linux 4.6) */
void     *si_call_addr; /* Address of system call instruction
                        (since Linux 3.5) */
int      si_syscall;    /* Number of attempted system call
                        (since Linux 3.5) */
unsigned int si_arch;   /* Architecture of attempted system call
                        (since Linux 3.5) */
}

```

si_signo, *si_errno* and *si_code* are defined for all signals. (*si_errno* is generally unused on Linux.) The rest of the struct may be a union, so that one should read only the fields that are meaningful for the given signal:

- Signals sent with `kill(2)` and `sigqueue(3)` fill in *si_pid* and *si_uid*. In addition, signals sent with `sigqueue(3)` fill in *si_int* and *si_ptr* with the values specified by the sender of the signal; see `sigqueue(3)` for more details.
- Signals sent by POSIX.1b timers (since Linux 2.6) fill in *si_overrun* and *si_timerid*. The *si_timerid* field is an internal ID used by the kernel to identify the timer; it is not the same as the timer ID returned by `timer_create(2)`. The *si_overrun* field is the timer overrun count; this is the same information as is obtained by a call to `timer_getoverrun(2)`. These fields are nonstandard Linux extensions.
- Signals sent for message queue notification (see the description of **SIGEV_SIGNAL** in `mq_notify(3)`) fill in *si_int/si_ptr*, with the *sigev_value* supplied to `mq_notify(3)`; *si_pid*, with the process ID of the message sender; and *si_uid*, with the real user ID of the message sender.
- **SIGCHLD** fills in *si_pid*, *si_uid*, *si_status*, *si_utime*, and *si_stime*, providing information about the child. The *si_pid* field is the process ID of the child; *si_uid* is the child's real user ID. The *si_status* field contains the exit status of the child (if *si_code* is **CLD_EXITED**), or the signal number that caused the process to change state. The *si_utime* and *si_stime* contain the user and system CPU time used by the



child process; these fields do not include the times used by waited-for children (unlike `getrusage(2)` and `times(2)`). Up to Linux 2.6, and since Linux 2.6.27, these fields report CPU time in units of `sysconf(_SC_CLK_TCK)`. In Linux 2.6 kernels before Linux 2.6.27, a bug meant that these fields reported time in units of the (configurable) system jiffy (see `time(7)`).

- **SIGILL**, **SIGFPE**, **SIGSEGV**, **SIGBUS**, and **SIGTRAP** fill in `si_addr` with the address of the fault. On some architectures, these signals also fill in the `si_trapno` field.

Some suberrors of **SIGBUS**, in particular **BUS_MCEERR_AO** and **BUS_MCEERR_AR**, also fill in `si_addr_lsb`. This field indicates the least significant bit of the reported address and therefore the extent of the corruption. For example, if a full page was corrupted, `si_addr_lsb` contains `Log2(sysconf(_SC_PAGESIZE))`. When **SIGTRAP** is delivered in response to a `ptrace(2)` event (`PTRACE_EVENT_foo`), `si_addr` is not populated, but `si_pid` and `si_uid` are populated with the respective process ID and user ID responsible for delivering the trap. In the case of `seccomp(2)`, the tracee will be shown as delivering the event. **BUS_MCEERR_*** and `si_addr_lsb` are Linux-specific extensions.

The **SEGV_BNDERR** suberror of **SIGSEGV** populates `si_lower` and `si_upper`.

The **SEGV_PKUERR** suberror of **SIGSEGV** populates `si_pkey`.

- **SIGIO/SIGPOLL** (the two names are synonyms on Linux) fills in `si_band` and `si_fd`. The `si_band` event is a bit mask containing the same values as are filled in the `revents` field by `poll(2)`. The `si_fd` field indicates the file descriptor for which the I/O event occurred; for further details, see the description of **F_SETSIG** in `fcntl(2)`.
- **SIGSYS**, generated (since Linux 3.5) when a seccomp filter returns **SECCOMP_RET_TRAP**, fills in `si_call_addr`, `si_syscall`, `si_arch`, `si_errno`, and other fields as described in `seccomp(2)`.

The `si_code` field

The `si_code` field inside the `siginfo_t` argument that is passed to a **SA_SIGINFO** signal handler is a value (not a bit mask) indicating why this signal was sent. For a `ptrace(2)` event, `si_code` will contain **SIGTRAP** and have the ptrace event in the high byte:

(SIGTRAP | PTRACE_EVENT_foo << 8).

For a non-`ptrace(2)` event, the values that can appear in `si_code` are described in the remainder of this section. Since glibc 2.20, the definitions of most of these symbols are obtained from `<signal.h>` by defining feature test macros (before including *any* header file) as follows:

- `_XOPEN_SOURCE` with the value 500 or greater;
- `_XOPEN_SOURCE` and `_XOPEN_SOURCE_EXTENDED`; or
- `_POSIX_C_SOURCE` with the value 200809L or greater.

For the `TRAP_*` constants, the symbol definitions are provided only in the first two cases. Before glibc 2.20, no feature test macros were required to obtain these symbols.

For a regular signal, the following list shows the values which can be placed in `si_code` for any signal, along with the reason that the signal was generated.

SI_USER

`kill(2)`.

SI_KERNEL

Sent by the kernel.

SI_QUEUE

`sigqueue(3)`.

SI_TIMER

POSIX timer expired.

SI_MESGQ (since Linux 2.6.6)

POSIX message queue state changed; see `mq_notify(3)`.

SI_ASYNCIO

AIO completed.

SI_SIGIO

Queued **SIGIO** (only up to Linux 2.2; from Linux 2.4 onward **SIGIO/SIGPOLL** fills in `si_code` as described below).

SI_TKILL (since Linux 2.4.19)

`tkill(2)` or `tgkill(2)`.



The following values can be placed in *si_code* for a **SIGILL** signal:

- ILL_ILLOPC**
Illegal opcode.
- ILL_ILLOPN**
Illegal operand.
- ILL_ILLADR**
Illegal addressing mode.
- ILL_ILLTRP**
Illegal trap.
- ILL_PRVOPC**
Privileged opcode.
- ILL_PRVREG**
Privileged register.
- ILL_COPROC**
Coprocessor error.
- ILL_BADSTK**
Internal stack error.

The following values can be placed in *si_code* for a **SIGFPE** signal:

- FPE_INTDIV**
Integer divide by zero.
- FPE_INTOVF**
Integer overflow.
- FPE_FLTDIV**
Floating-point divide by zero.
- FPE_FLTOVF**
Floating-point overflow.
- FPE_FLTUND**
Floating-point underflow.
- FPE_FLTRES**
Floating-point inexact result.



FPE_FLTINV

Floating-point invalid operation.

FPE_FLTSUB

Subscript out of range.

The following values can be placed in *si_code* for a **SIGSEGV** signal:

SEGV_MAPERR

Address not mapped to object.

SEGV_ACCERR

Invalid permissions for mapped object.

SEGV_BNDERR (since Linux 3.19)

Failed address bound checks.

SEGV_PKUERR (since Linux 4.6)

Access was denied by memory protection keys. See [pkeys\(7\)](#). The protection key which applied to this access is available via *si_pkey*.

The following values can be placed in *si_code* for a **SIGBUS** signal:

BUS_ADRALN

Invalid address alignment.

BUS_ADRERR

Nonexistent physical address.

BUS_OBJERR

Object-specific hardware error.

BUS_MCEERR_AR (since Linux 2.6.32)

Hardware memory error consumed on a machine check; action required.

BUS_MCEERR_AO (since Linux 2.6.32)

Hardware memory error detected in process but not consumed; action optional.

The following values can be placed in *si_code* for a **SIGTRAP** signal:

TRAP_BRKPT

Process breakpoint.

TRAP_TRACE

Process trace trap.

TRAP_BRANCH (since Linux 2.4, IA64 only)

Process taken branch trap.

TRAP_HWBKPT (since Linux 2.4, IA64 only)

Hardware breakpoint/watchpoint.

The following values can be placed in *si_code* for a **SIGCHLD** signal:

CLD_EXITED

Child has exited.

CLD_KILLED

Child was killed.

CLD_DUMPED

Child terminated abnormally.

CLD_TRAPPED

Traced child has trapped.

CLD_STOPPED

Child has stopped.

CLD_CONTINUED (since Linux 2.6.9)

Stopped child has continued.

The following values can be placed in *si_code* for a **SIGIO/SIGPOLL** signal:

POLL_IN

Data input available.

POLL_OUT

Output buffers available.

POLL_MSG

Input message available.

POLL_ERR

I/O error.

POLL_PRI

High priority input available.

POLL_HUP

Device disconnected.

The following value can be placed in *si_code* for a **SIGSYS** signal:

SYS_SECCOMP (since Linux 3.5)

Triggered by a [seccomp\(2\)](#) filter rule.

Dynamically probing for flag bit support

The **sigaction()** call on Linux accepts unknown bits set in *act->sa_flags* without error. The behavior of the kernel starting with Linux 5.11 is that a second **sigaction()** will clear unknown bits from *oldact->sa_flags*. However, historically, a second **sigaction()** call would typically leave those bits set in *oldact->sa_flags*.

This means that support for new flags cannot be detected simply by testing for a flag in *sa_flags*, and a program must test that **SA_UNSUPPORTED** has been cleared before relying on the contents of *sa_flags*.

Since the behavior of the signal handler cannot be guaranteed unless the check passes, it is wise to either block the affected signal while registering the handler and performing the check in this case, or where this is not possible, for example if the signal is synchronous, to issue the second **sigaction()** in the signal handler itself.

In kernels that do not support a specific flag, the kernel's behavior is as if the flag was not set, even if the flag was set in *act->sa_flags*.

The flags **SA_NOCLDSTOP**, **SA_NOCLDWAIT**, **SA_SIGINFO**, **SA_ONSTACK**, **SA_RESTART**, **SA_NODEFER**, **SA_RESETHAND**, and, if defined by the architecture, **SA_RESTORER** may not be reliably probed for using this mechanism, because they were introduced before Linux 5.11. However, in general, programs may assume that these flags are supported, since they have all been supported since Linux 2.6, which was released in the year 2003.

See EXAMPLES below for a demonstration of the use of **SA_UNSUPPORTED**.

RETURN VALUE [top](#)

sigaction() returns 0 on success; on error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EFAULT *act* or *oldact* points to memory which is not a valid part of the process address space.

EINVAL An invalid signal was specified. This will also be generated if an attempt is made to change the action for **SIGKILL** or **SIGSTOP**, which cannot be caught or ignored.

VERSIONS [top](#)

C library/kernel differences

The glibc wrapper function for **sigaction()** gives an error (**EINVAL**) on attempts to change the disposition of the two real-time signals used internally by the NPTL threading implementation. See [nptl\(7\)](#) for details.

On architectures where the signal trampoline resides in the C library, the glibc wrapper function for **sigaction()** places the address of the trampoline code in the *act.sa_restorer* field and sets the **SA_RESTORER** flag in the *act.sa_flags* field. See [sigreturn\(2\)](#).

The original Linux system call was named **sigaction()**. However, with the addition of real-time signals in Linux 2.2, the fixed-size, 32-bit *sigset_t* type supported by that system call was no longer fit for purpose. Consequently, a new system call, **rt_sigaction()**, was added to support an enlarged *sigset_t* type. The new system call takes a fourth argument, *size_t sigsetsize*, which specifies the size in bytes of the signal sets in *act.sa_mask* and *oldact.sa_mask*. This argument is currently required to have the value *sizeof(sigset_t)* (or the error **EINVAL** results). The glibc **sigaction()** wrapper function hides these details from us, transparently calling **rt_sigaction()** when the kernel provides it.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4.

POSIX.1-1990 disallowed setting the action for **SIGCHLD** to **SIG_IGN**. POSIX.1-2001 and later allow this possibility, so that ignoring **SIGCHLD** can be used to prevent the creation of zombies (see `wait(2)`). Nevertheless, the historical BSD and System V behaviors for ignoring **SIGCHLD** differ, so that the only completely portable method of ensuring that terminated children do not become zombies is to catch the **SIGCHLD** signal and perform a `wait(2)` or similar.

POSIX.1-1990 specified only **SA_NOCLDSTOP**. POSIX.1-2001 added **SA_NOCLDSTOP**, **SA_NOCLDWAIT**, **SA_NODEFER**, **SA_ONSTACK**, **SA_RESETHAND**, **SA_RESTART**, and **SA_SIGINFO**. Use of these latter values in *sa_flags* may be less portable in applications intended for older UNIX implementations.

The **SA_RESETHAND** flag is compatible with the SVr4 flag of the same name.

The **SA_NODEFER** flag is compatible with the SVr4 flag of the same name under kernels 1.3.9 and later. On older kernels the Linux implementation allowed the receipt of any signal, not just the one we are installing (effectively overriding any *sa_mask* settings).

NOTES [top](#)

A child created via `fork(2)` inherits a copy of its parent's signal dispositions. During an `execve(2)`, the dispositions of handled signals are reset to the default; the dispositions of ignored signals are left unchanged.

According to POSIX, the behavior of a process is undefined after it ignores a **SIGFPE**, **SIGILL**, or **SIGSEGV** signal that was not generated by `kill(2)` or `raise(3)`. Integer division by zero has undefined result. On some architectures it will generate a **SIGFPE** signal. (Also dividing the most negative integer by -1 may generate **SIGFPE**.) Ignoring this signal might lead to an endless loop.

`sigaction()` can be called with a NULL second argument to query the current signal handler. It can also be used to check whether a given signal is valid for the current machine by calling it with NULL second and third arguments.

It is not possible to block **SIGKILL** or **SIGSTOP** (by specifying



them in `sa_mask`). Attempts to do so are silently ignored.

See [sigsetops\(3\)](#) for details on manipulating signal sets.

See [signal-safety\(7\)](#) for a list of the async-signal-safe functions that can be safely called inside from inside a signal handler.

Undocumented

Before the introduction of **SA_SIGINFO**, it was also possible to get some additional information about the signal. This was done by providing an `sa_handler` signal handler with a second argument of type `struct sigcontext`, which is the same structure as the one that is passed in the `uc_mcontext` field of the `ucontext` structure that is passed (via a pointer) in the third argument of the `sa_sigaction` handler. See the relevant Linux kernel sources for details. This use is obsolete now.

BUGS [top](#)

When delivering a signal with a **SA_SIGINFO** handler, the kernel does not always provide meaningful values for all of the fields of the `siginfo_t` that are relevant for that signal.

Up to and including Linux 2.6.13, specifying **SA_NODEFER** in `sa_flags` prevents not only the delivered signal from being masked during execution of the handler, but also the signals specified in `sa_mask`. This bug was fixed in Linux 2.6.14.

EXAMPLES [top](#)

See [mprotect\(2\)](#).

Probing for flag support

The following example program exits with status **EXIT_SUCCESS** if **SA_EXPOSE_TAGBITS** is determined to be supported, and **EXIT_FAILURE** otherwise.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
handler(int signo, siginfo_t *info, void *context)
{
```



```

struct sigaction oldact;

if (sigaction(SIGSEGV, NULL, &oldact) == -1
    || (oldact.sa_flags & SA_UNSUPPORTED)
    || !(oldact.sa_flags & SA_EXPOSE_TAGBITS))
{
    _exit(EXIT_FAILURE);
}
_exit(EXIT_SUCCESS);
}

int
main(void)
{
    struct sigaction act = { 0 };

    act.sa_flags = SA_SIGINFO | SA_UNSUPPORTED | SA_EXPOSE_TAGBITS;
    act.sa_sigaction = &handler;
    if (sigaction(SIGSEGV, &act, NULL) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    raise(SIGSEGV);
}

```

SEE ALSO [top](#)

[kill\(1\)](#), [kill\(2\)](#), [pause\(2\)](#), [pidfd_send_signal\(2\)](#),
[restart_syscall\(2\)](#), [seccomp\(2\)](#), [sigaltstack\(2\)](#), [signal\(2\)](#),
[signalfd\(2\)](#), [sigpending\(2\)](#), [sigprocmask\(2\)](#), [sigreturn\(2\)](#),
[sigsuspend\(2\)](#), [wait\(2\)](#), [killpg\(3\)](#), [raise\(3\)](#), [siginterrupt\(3\)](#),
[sigqueue\(3\)](#), [sigsetops\(3\)](#), [sigvec\(3\)](#), [core\(5\)](#), [signal\(7\)](#)

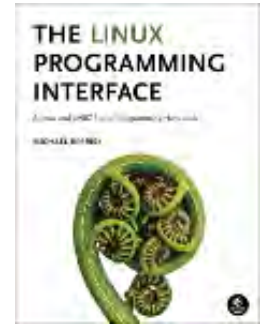
Linux man-pages (unreleased) **(date)** ***sigaction(2)***

Pages that refer to this page: [env\(1\)](#), [kill\(1\)](#), [kill\(1@@procps-ng\)](#), [pgrep\(1\)](#), [alarm\(2\)](#), [clock_nanosleep\(2\)](#), [clone\(2\)](#), [fcntl\(2\)](#), [getitimer\(2\)](#), [pidfd_open\(2\)](#), [pidfd_send_signal\(2\)](#), [prctl\(2\)](#), [ptrace\(2\)](#), [restart_syscall\(2\)](#), [rt_sigqueueinfo\(2\)](#), [seccomp\(2\)](#), [seccomp_unotify\(2\)](#), [semop\(2\)](#), [send\(2\)](#), [sigaltstack\(2\)](#), [signal\(2\)](#), [signalfd\(2\)](#), [sigpending\(2\)](#), [sigprocmask\(2\)](#), [sigreturn\(2\)](#), [sigsuspend\(2\)](#), [sigwaitinfo\(2\)](#), [syscalls\(2\)](#), [timer_getoverrun\(2\)](#), [wait\(2\)](#), [wait4\(2\)](#), [abort\(3\)](#), [bsd_signal\(3\)](#), [getcontext\(3\)](#), [makecontext\(3\)](#), [posix_spawn\(3\)](#), [profil\(3\)](#), [psignal\(3\)](#), [pthread_kill\(3\)](#), [pthread_sigmask\(3\)](#), [pthread_sigqueue\(3\)](#), [raise\(3\)](#), [seccomp_init\(3\)](#), [siginterrupt\(3\)](#), [sigpause\(3\)](#), [sigqueue\(3\)](#), [sigset\(3\)](#), [sigsetops\(3\)](#), [sigvec\(3\)](#), [sigwait\(3\)](#), [system\(3\)](#), [sysv_signal\(3\)](#), [core\(5\)](#), [proc\(5\)](#), [fifo\(7\)](#), [inotify\(7\)](#), [nptl\(7\)](#), [pid_namespaces\(7\)](#), [pkeys\(7\)](#), [sigevent\(7\)](#), [signal\(7\)](#), [signal-safety\(7\)](#), [socket\(7\)](#), [system_data_types\(7\)](#), [user_namespaces\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



sigqueue(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 sigqueue(3)

Library Functions Manual

sigqueue(3)

NAME [top](#)

sigqueue - queue a signal and data to a process

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <signal.h>
```

```
int sigqueue(pid_t pid, int sig, const union sigval value);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
sigqueue():  
    _POSIX_C_SOURCE >= 199309L
```

DESCRIPTION [top](#)

sigqueue() sends the signal specified in *sig* to the process whose PID is given in *pid*. The permissions required to send a signal are the same as for [kill\(2\)](#). As with [kill\(2\)](#), the null signal (0) can be used to check if a process with a given PID exists.

The *value* argument is used to specify an accompanying item of

data (either an integer or a pointer value) to be sent with the signal, and has the following type:

```
union sigval {
    int    sival_int;
    void *sival_ptr;
};
```

If the receiving process has installed a handler for this signal using the **SA_SIGINFO** flag to `sigaction(2)`, then it can obtain this data via the *si_value* field of the *siginfo_t* structure passed as the second argument to the handler. Furthermore, the *si_code* field of that structure will be set to **SI_QUEUE**.

RETURN VALUE [top](#)

On success, `sigqueue()` returns 0, indicating that the signal was successfully queued to the receiving process. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS [top](#)

EAGAIN The limit of signals which may be queued has been reached. (See `signal(7)` for further information.)

EINVAL *sig* was invalid.

EPERM The process does not have permission to send the signal to the receiving process. For the required permissions, see `kill(2)`.

ESRCH No process has a PID matching *pid*.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see `attributes(7)`.

| Interface | Attribute | Value |
|-------------------------|---------------|---------|
| <code>sigqueue()</code> | Thread safety | MT-Safe |

VERSIONS [top](#)

C library/kernel differences

On Linux, `sigqueue()` is implemented using the `rt_sigqueueinfo(2)` system call. The system call differs in its third argument, which is the `siginfo_t` structure that will be supplied to the receiving process's signal handler or returned by the receiving process's `sigtimedwait(2)` call. Inside the glibc `sigqueue()` wrapper, this argument, `uinfo`, is initialized as follows:

```
uinfo.si_signo = sig;      /* Argument supplied to sigqueue() */
uinfo.si_code = SI_QUEUE;
uinfo.si_pid = getpid();  /* Process ID of sender */
uinfo.si_uid = getuid();  /* Real UID of sender */
uinfo.si_value = val;     /* Argument supplied to sigqueue() */
```

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

Linux 2.2. POSIX.1-2001.

NOTES [top](#)

If this function results in the sending of a signal to the process that invoked it, and that signal was not blocked by the calling thread, and no other threads were willing to handle this signal (either by having it unblocked, or by waiting for it using `sigwait(3)`), then at least some signal must be delivered to this thread before this function returns.

SEE ALSO [top](#)

`kill(2)`, `rt_sigqueueinfo(2)`, `sigaction(2)`, `signal(2)`,
`pthread_sigqueue(3)`, `sigwait(3)`, `signal(7)`

Linux man-pages (unreleased) (date)

`sigqueue(3)`

Pages that refer to this page: [kill\(1\)](#), [kill\(1@@procps-ng\)](#), [pgrep\(1\)](#), [systemctl\(1\)](#), [clone\(2\)](#), [getrlimit\(2\)](#), [kill\(2\)](#), [ptrace\(2\)](#), [rt_sigqueueinfo\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [signalfd\(2\)](#), [sigprocmask\(2\)](#), [sigwaitinfo\(2\)](#), [id_t\(3type\)](#), [psignal\(3\)](#), [pthread_sigqueue\(3\)](#), [org.freedesktop.systemd1\(5\)](#), [credentials\(7\)](#), [signal\(7\)](#), [system_data_types\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



Other versions of this page are provided by these projects: [coreutils](#) [procps-ng](#)

kill(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [ARGUMENTS](#) | [OPTIONS](#) | [EXIT STATUS](#) | [NOTES](#) | [AUTHORS](#) | [SEE ALSO](#) | [REPORTING BUGS](#) | [AVAILABILITY](#)

 KILL(1)

User Commands

KILL(1)**NAME** [top](#)

`kill` - terminate a process

SYNOPSIS [top](#)

```
kill [-signal|-s signal|-p] [-q value] [-a] [--timeout  
milliseconds signal] [--] pid|name...
```

```
kill -l [number] | -L
```

DESCRIPTION [top](#)

The command **kill** sends the specified *signal* to the specified processes or process groups.

If no signal is specified, the **TERM** signal is sent. The default action for this signal is to terminate the process. This signal should be used in preference to the **KILL** signal (number 9), since a process may install a handler for the TERM signal in order to perform clean-up steps before terminating in an orderly fashion. If a process does not terminate after a **TERM** signal has been sent, then the **KILL** signal may be used; be aware that the latter signal cannot be caught, and so does not give the target process the opportunity to perform any clean-up before terminating.



Most modern shells have a builtin **kill** command, with a usage rather similar to that of the command described here. The **--all**, **--pid**, and **--queue** options, and the possibility to specify processes by command name, are local extensions.

If *signal* is 0, then no actual signal is sent, but error checking is still performed.

ARGUMENTS [top](#)

The list of processes to be signaled can be a mixture of names and PIDs.

pid

Each *pid* can be expressed in one of the following ways:

n

where *n* is larger than 0. The process with PID *n* is signaled.

0

All processes in the current process group are signaled.

-1

All processes with a PID larger than 1 are signaled.

-*n*

where *n* is larger than 1. All processes in process group *n* are signaled. When an argument of the form '-*n*' is given, and it is meant to denote a process group, either a signal must be specified first, or the argument must be preceded by a '--' option, otherwise it will be taken as the signal to send.

name

All processes invoked using this *name* will be signaled.

OPTIONS [top](#)

-s, --signal *signal*

The signal to send. It may be given as a name or a number.

- l, --list** *[number]*
Print a list of signal names, or convert the given signal number to a name. The signals can be found in */usr/include/linux/signal.h*.
- L, --table**
Similar to **-l**, but it will print signal names and their corresponding numbers.
- a, --all**
Do not restrict the command-name-to-PID conversion to processes with the same UID as the present process.
- p, --pid**
Only print the process ID (PID) of the named processes, do not send any signals.
- r, --require-handler**
Do not send the signal if it is not caught in userspace by the signalled process.
- verbose**
Print PID(s) that will be signaled with **kill** along with the signal.
- q, --queue** *value*
Send the signal using **sigqueue(3)** rather than **kill(2)**. The *value* argument is an integer that is sent along with the signal. If the receiving process has installed a handler for this signal using the **SA_SIGINFO** flag to **sigaction(2)**, then it can obtain this data via the *si_sigval* field of the *siginfo_t* structure.
- timeout** *milliseconds signal*
Send a signal defined in the usual way to a process, followed by an additional signal after a specified delay. The **--timeout** option causes **kill** to wait for a period defined in *milliseconds* before sending a follow-up *signal* to the process. This feature is implemented using the Linux kernel PID file descriptor feature in order to guarantee that the follow-up signal is sent to the same process or not sent if the process no longer exists.



Note that the operating system may re-use PIDs and implementing an equivalent feature in a shell using **kill** and **sleep** would be subject to races whereby the follow-up signal might be sent to a different process that used a recycled PID.

The **--timeout** option can be specified multiple times: the signals are sent sequentially with the specified timeouts. The **--timeout** option can be combined with the **--queue** option.

As an example, the following command sends the signals **QUIT**, **TERM** and **KILL** in sequence and waits for 1000 milliseconds between sending the signals:

```
kill --verbose --timeout 1000 TERM --timeout 1000 KILL \  
    --signal QUIT 12345
```

EXIT STATUS [top](#)

kill has the following exit status values:

- 0**
success
- 1**
failure
- 64**
partial success (when more than one process specified)

NOTES [top](#)

Although it is possible to specify the TID (thread ID, see [gettid\(2\)](#)) of one of the threads in a multithreaded process as the argument of **kill**, the signal is nevertheless directed to the process (i.e., the entire thread group). In other words, it is not possible to send a signal to an explicitly selected thread in a multithreaded process. The signal will be delivered to an arbitrarily selected thread in the target process that is not blocking the signal. For more details, see [signal\(7\)](#) and the description of **CLONE_THREAD** in [clone\(2\)](#).

Various shells provide a builtin **kill** command that is preferred in relation to the `kill(1)` executable described by this manual. The easiest way to ensure one is executing the command described in this page is to use the full path when calling the command, for example: **`/bin/kill --version`**

AUTHORS [top](#)

Salvatore Valente <svalente@mit.edu>, Karel Zak <kzak@redhat.com>

The original version was taken from BSD 4.4.

SEE ALSO [top](#)

[bash\(1\)](#), [tcsh\(1\)](#), [sigaction\(2\)](#), [kill\(2\)](#), [sigqueue\(3\)](#), [signal\(7\)](#)

REPORTING BUGS [top](#)

For bug reports, use the issue tracker at <https://github.com/util-linux/util-linux/issues>.

AVAILABILITY [top](#)

The **kill** command is part of the util-linux package which can be downloaded from Linux Kernel Archive <<https://www.kernel.org/pub/linux/utils/util-linux/>>. This page is part of the *util-linux* (a random collection of Linux utilities) project. Information about the project can be found at <<https://www.kernel.org/pub/linux/utils/util-linux/>>. If you have a bug report for this manual page, send it to util-linux@vger.kernel.org. This page was obtained from the project's upstream Git repository (<git://git.kernel.org/pub/scm/utils/util-linux/util-linux.git>) on 2023-12-22. (At that time, the date of the most recent commit that was found in the repository was 2023-12-14.) If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

util-linux 2.39.594-1e0ad**2023-07-19****KILL(1)**

Pages that refer to this page: [fuser\(1\)](#), [kill\(1\)](#), [killall\(1\)](#), [pgrep\(1\)](#), [pmsignal\(1\)](#), [skill\(1\)](#), [tcpdump\(1\)](#), [timeout\(1\)](#), [xargs\(1\)](#), [kill\(2\)](#), [sigaction\(2\)](#), [signal\(2\)](#), [posix_spawn\(3\)](#), [signal\(7\)](#), [ldattach\(8\)](#), [lsof\(8\)](#), [systemd-coredump\(8\)](#), [tcpdump\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



Materials for Topic 10: Signals

Full C Programs

- [division_by_0.c](#) - a C program that catches the `SIGFPE` signal during a division by 0.
- [handling_sigint.c](#) - a C program that catches the `SIGINT` signal whenever you press `ctrl+c` during its execution.
- [more_signals.c](#) - a C program that catches the `SIGINT` signal whenever you press `ctrl+c`, catches the `SIGTERM` signal whenever another app or terminal calls the `kill` command of the ID of the process, ignores the `SIGHUP` signal, and resets the behavior for `SIGPROF` to the default.
- [sigaction_example.c](#) - a C program that ignores the `SIGINT` signal using the `sigaction()` function.
- [issuing_signals.c](#) - a C program that creates a child process and sends it the `SIGINT` signal via `kill()`. The child handles the `SIGINT` signal by calling the `write()` system call, which is reentrant. Finally, the parent sends a signal to itself using `raise()`.
- [payload_sending.c](#) - a C program that creates a child process and sends it the `SIGUSR2` signal with an integer payload that the user types via the terminal using the `sigqueue()` function.

Runnable Linux Commands

Top

Quick Links:

- [gcc](#)
 - [./short_prompt](#)
 - [./long_prompt](#)
 - [kill -l](#)
 - [kill -NAME PID](#)
 - [kill PID](#)
-

- The command:

```
gcc -Wall -Wextra -O2 -g -o program program.c
```

compiles the C source code located inside the file `scheduling_examples.c`. See more details [here](#).

- The command:

```
./short_prompt
```

executes code inside a file named `short_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).

- The command:

```
./long_prompt
```

executes code inside a file named `long_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).

- The command:

```
kill -l
```



(- l is "dash L") display a list of the names of signals supported by the OS.

- The command:

```
kill -NAME PID
```

sends the signal NAME (e.g., HUP) to the process with id of PID.

- The command:

```
kill PID
```

sends the signal TERM (i.e., SIGTERM) to the process with id of PID.



This website by [Miriam Briskman](#) is licensed under [CC BY-NC 4.0](#).

```
1  /* A program that raises SIGFPE when we divide a
2     *   number by 0.
3     *
4     *   Miriam Briskman, 4/26/2023
5     *   CISC 3350, Brooklyn College
6     *   Licensed under CC BY-NC 4.0
7     */
8
9  #include <stdlib.h>
10 #include <stdio.h>
11 #include <unistd.h>
12 #include <signal.h>
13
14 // Handler for the SIGFPE signal::
15 static void sigint_handler (int signo)
16 {
17     // We shouldn't use the function printf()
18     //   in a signal handler; we use it here
19     //   only for demonstration of how a
20     //   handler works. We will see the reason
21     //   that printf() shouldn't be used when
22     //   we discuss the idea of reentrancy.
23     printf ("Caught %d, which is SIGFPE!\n", signo);
24     exit (EXIT_SUCCESS);
25 }
26
27 int main (void)
28 {
29     // Register 'sigint_handler' as our signal
30     //   handler for SIGFPE:
31     if (signal (SIGFPE, sigint_handler) == SIG_ERR)
32     {
33         fprintf (stderr, "Cannot handle SIGFPE!\n");
34         exit (EXIT_FAILURE);
35     }
36
37     printf ("Our favorite number in Math "
38            "is: %d.\n", 1/0);
39
40     pause ();
41
42     return EXIT_SUCCESS;
```



43 | }
44

```
1 // A program that handles SIGINT by printing a
2 // message, and then terminating the program.
3
4 // This program is taken from Linux System Programming:
5 // Talking Directly to the Kernel and C Library,
6 // 2nd Edition, by Love. ISBN: 978-1-44933953-1,
7 // page 342.
8
9 #include <stdlib.h>
10 #include <stdio.h>
11 #include <unistd.h>
12 #include <signal.h>
13
14 // Handler function for SIGINT:
15 static void sigint_handler (int signo)
16 {
17     // We shouldn't use the function printf()
18     // in a signal handler; we use it here
19     // only for demonstration of how a
20     // handler works. We will see the reason
21     // that printf() shouldn't be used when
22     // we discuss the idea of reentrancy.
23     printf ("\nCaught %d, which is SIGINT!\n",
24             signo);
25     exit (EXIT_SUCCESS);
26 }
27
28 int main (void)
29 {
30     // Register 'sigint_handler' as our signal
31     // handler for SIGINT:
32     if (signal (SIGINT, sigint_handler) == SIG_ERR)
33     {
34         fprintf (stderr, "Cannot handle SIGINT!\n");
35         exit (EXIT_FAILURE);
36     }
37
38     printf ("To activate the SIGINT signal, "
39            "press Ctrl+C.\n");
40
41     for (;;)
42         pause ();
```



```
43 |  
44 | return EXIT_SUCCESS;  
45 | }  
46 |
```

```
1 // A program that registers the same handler for
2 //   SIGTERM and SIGINT. It also resets the
3 //   behavior for SIGPROF to the default (which
4 //   is to terminate the process) and ignores
5 //   SIGHUP (which would otherwise terminate
6 //   the process.)
7
8 // This program is taken from Linux System Programming:
9 //   Talking Directly to the Kernel and C Library,
10 //   2nd Edition, by Love. ISBN: 978-1-44933953-1,
11 //   pages 343-344.
12
13 #include <stdlib.h>
14 #include <stdio.h>
15 #include <unistd.h>
16 #include <signal.h>
17
18 // Handler function for SIGINT and SIGTERM:
19 static void signal_handler (int signo)
20 {
21     if (signo == SIGINT)
22     {
23         printf ("\nCaught SIGINT!\n");
24         exit (EXIT_FAILURE);
25     }
26     else if (signo == SIGTERM)
27         printf ("Caught SIGTERM!\n");
28     else
29     {
30         /* This shouldn't happen. */
31         fprintf (stderr,
32                 "\nUnexpected signal: number %d\n",
33                 signo);
34         exit (EXIT_FAILURE);
35     }
36 }
37
38 int main (void)
39 {
40     // Register 'sigint_handler' as our signal
41     //   handler for SIGINT:
42     if (signal (SIGINT, signal_handler) == SIG_ERR)
```



```
43     {
44         fprintf (stderr, "Cannot handle SIGINT!\n");
45         exit (EXIT_FAILURE);
46     }
47
48     // Register 'sigint_handler' as our signal
49     // handler for SIGTERM:
50     if (signal (SIGTERM, signal_handler) == SIG_ERR)
51     {
52         fprintf (stderr, "Cannot handle SIGTERM!\n");
53         exit (EXIT_FAILURE);
54     }
55
56     // Reset SIGPROF's behavior to the default:
57     if (signal (SIGPROF, SIG_DFL) == SIG_ERR)
58     {
59         fprintf (stderr, "Cannot reset SIGPROF!\n");
60         exit (EXIT_FAILURE);
61     }
62
63     // Ignore SIGHUP:
64     if (signal (SIGHUP, SIG_IGN) == SIG_ERR)
65     {
66         fprintf (stderr, "Cannot ignore SIGHUP!\n");
67         exit (EXIT_FAILURE);
68     }
69
70     for (;;)
71         pause ();
72
73     return EXIT_SUCCESS;
74 }
75
```




```
1  /* A C program that ignores the SIGINT signal using
2     *   the sigaction() function.
3     *
4     *   Miriam Briskman, 4/30/2023
5     *   CISC 3350, Brooklyn College
6     *   Licensed under CC BY-NC 4.0
7     */
8
9  #include <signal.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <unistd.h>
13
14 int main()
15 {
16     struct sigaction act;
17
18     // Find the current action of SIGINT:
19     if (sigaction (SIGINT, NULL, &act) == -1)
20     {
21         perror ("Could not obtain old action "
22             "of SIGINT");
23         exit (EXIT_FAILURE);
24     }
25
26     // If the action is the default (termination),
27     //   change it to 'ignore':
28     if (act.sa_handler == SIG_DFL)
29     {
30         act.sa_handler = SIG_IGN;
31         if (sigaction (SIGINT, &act, NULL) == -1)
32         {
33             perror ("Could not ignore SIGINT");
34             exit (EXIT_FAILURE);
35         }
36     }
37
38     // Invoke sleep to try and kill the process
39     //   with Ctrl+C. You'll see that Ctrl+C
40     //   won't kill the program, because we set
41     //   the program to ignore a SIGINT signal!
42     sleep (10);
```



```
43  
44     printf ("\nThe result is %d.\n", 10/5);  
45  
46     return EXIT_SUCCESS;  
47 }  
48
```



```
1  /* A program that creates a child process and
2     * sends it the SIGINT signal via kill().
3     * The child handles the SIGINT signal by
4     * calling the write() system call, which is
5     * reentrant. Finally, the parent sends a
6     * signal to itself using raise().
7     *
8     * Miriam Briskman, 4/30/2023
9     * CISC 3350, Brooklyn College
10    * Licensed under CC BY-NC 4.0
11   */
12
13  #include <unistd.h> // write().
14  #include <sys/wait.h> // waitpid().
15  #include <stdio.h> // perror(), printf(), fprintf().
16  #include <stdlib.h> // EXIT_SUCCESS, EXIT_FAILURE.
17  #include <string.h> // strlen().
18
19  static void signal_handler_child (int signo)
20  {
21      // write() is a reentrant function, so we can
22      // use it safely instead of using printf():
23      if (write (STDOUT_FILENO,
24              "Child: Received the signal: ",
25              strlen("Child: Received the signal: "))
26          == -1 ||
27          write (STDOUT_FILENO,
28              sys_siglist[signo],
29              strlen(sys_siglist[signo]))
30          == -1 ||
31          write (STDOUT_FILENO,
32              ".\n",
33              strlen(".\n"))
34          == -1)
35      {
36          perror ("write");
37          exit (EXIT_FAILURE);
38      }
39  }
40
41  static void signal_handler_parent (int signo)
42  {
```



```
43     if (write (STDOUT_FILENO,
44             "Parent: Received the signal: ",
45             strlen("Parent: Received the signal: "))
46         == -1 ||
47         write (STDOUT_FILENO,
48             sys_siglist[signo],
49             strlen(sys_siglist[signo]))
50         == -1 ||
51         write (STDOUT_FILENO,
52             ".\n",
53             strlen(".\n"))
54         == -1)
55     {
56         perror ("write");
57         exit (EXIT_FAILURE);
58     }
59 }
60
61 int main ()
62 {
63     printf ("Parent: Started executing now!\n");
64
65     // Register signal_handler_parent as for
66     // SIGUSR1:
67     if (signal (SIGUSR1, signal_handler_parent)
68         == SIG_ERR)
69     {
70         fprintf (stderr, "Cannot handle SIGUSR1!\n");
71         exit (EXIT_FAILURE);
72     }
73
74     // Fork (create) a child process:
75     pid_t childpid = fork();
76     if (childpid == -1)
77     {
78         perror ("fork");
79         exit (EXIT_FAILURE);
80     }
81     // We enter this 'if' only inside the parent process:
82     else if (childpid)
83     {
84         printf ("Parent: A child with "
85             "pid = %d was forked!\n",
```



```
86         childpid);
87     // Check if we have the permission to send
88     // signals to the child:
89     int ret = kill (childpid, 0);
90     if (ret)
91     {
92         fprintf (stderr,
93                 "We can't send signals "
94                 "to the child.\n");
95     }
96     else
97     {
98         // Wait several seconds for the child to
99         // launch:
100        sleep(5);
101        // Send SIGINT to the child:
102        printf ("Parent: I'll send SIGINT to "
103               "the child now!\n");
104        ret = kill (childpid, SIGINT);
105        if (ret)
106        {
107            perror ("kill");
108            exit (EXIT_FAILURE);
109        }
110    }
111
112    // Wait for the child to return:
113    int status;
114
115    pid_t temp = waitpid (childpid, &status, 0);
116    if (temp == -1)
117    {
118        perror ("waitpid");
119        exit (EXIT_FAILURE);
120    }
121
122    printf ("Parent: The child that just "
123           "returned has the pid = %d.\n",
124           temp);
125
126    // Let's check what happened with the child:
127    if (WIFEXITED (status))
128        printf ("Parent: It terminated normally "
```



```
129         "with the exit status = %d.\n",
130         WEXITSTATUS (status));
131     if (WIFSIGNALED (status))
132         printf ("Parent: It was killed by the "
133             "signal = %d%s.\n",
134             WTERMSIG (status),
135             WCOREDUMP (status) ? " (dumped core)" : "");
136     if (WIFSTOPPED (status))
137         printf ("Parent: It was stopped by the "
138             "signal = %d.\n",
139             WSTOPSIG (status));
140     if (WIFCONTINUED (status))
141         printf ("Parent: Its execution "
142             "continued.\n");
143
144     printf ("Parent: I'll send SIGUSR1 "
145         "to myself now!\n");
146     // Send SIGUSR1 to itself (the parent):
147     ret = raise (SIGUSR1);
148     if (ret)
149     {
150         fprintf (stderr, "raise() failed.\n");
151         exit (EXIT_FAILURE);
152     }
153 }
154 else // We are inside the child process:
155 {
156     printf ("Child: Started executing now! "
157         "My pid is %d.\n",
158         getpid());
159
160     // Register signal_handler_child for SIGINT:
161     if (signal (SIGINT, signal_handler_child)
162         == SIG_ERR)
163     {
164         fprintf (stderr, "Cannot handle SIGINT!\n");
165         exit (EXIT_FAILURE);
166     }
167     printf ("Child: I just changed "
168         "the action of SIGINT.\n");
169     // Wait for the signal to be sent
170     // by the parent:
171     pause();
```



```
172     printf ("Child: Returning...\n");
173 }
174
175 return EXIT_SUCCESS;
176 }
177
```

```
1  /* A program that creates a child process and
2  *   sends it the SIGUSR2 signal with a payload
3  *   that the user types via the terminal.
4  *
5  *   Miriam Briskman, 4/30/2023
6  *   CISC 3350, Brooklyn College
7  *   Licensed under CC BY-NC 4.0
8  */
9
10 #include <unistd.h> // write().
11 #include <sys/wait.h> // waitpid().
12 #include <stdio.h> // perror(), printf(), fprintf().
13 #include <stdlib.h> // EXIT_SUCCESS, EXIT_FAILURE.
14 #include <string.h> // strlen().
15 #include <signal.h> // sigqueue(), signal struct.
16
17 void signal_handler_child (int signo,
18                             siginfo_t *si,
19                             void *ucontext)
20 {
21     // Let the compiler know ucontext is unused:
22     (void)(ucontext);
23
24     // write() is a reentrant function, so we can
25     // use it safely instead of using printf():
26     if (write (STDOUT_FILENO,
27               "Child: Received the signal: ",
28               strlen("Child: Received the signal: "))
29         == -1 ||
30         write (STDOUT_FILENO,
31               sys_siglist[signo],
32               strlen(sys_siglist[signo]))
33         == -1 ||
34         write (STDOUT_FILENO,
35               ".\n",
36               strlen(".\n"))
37         == -1)
38     {
39         perror ("write");
40         exit (EXIT_FAILURE);
41     }
42 }
```




```
43 char buffer[20];
44 sprintf (buffer, "%d", si->si_int);
45
46 // Print some siginfo information:
47 if (write (STDOUT_FILENO,
48           "Child: The received payload is: ",
49           strlen("Child: The received payload is: "))
50     == -1 ||
51     write (STDOUT_FILENO,
52           buffer,
53           strlen(buffer))
54     == -1 ||
55     write (STDOUT_FILENO,
56           ".\n",
57           strlen(".\n"))
58     == -1)
59 {
60     perror ("write");
61     exit (EXIT_FAILURE);
62 }
63 }
64
65 int main ()
66 {
67     printf ("Parent: Started executing now!\n");
68
69     union sigval value; // The sigval union struct.
70
71     // Ask the user to type an integer via
72     // the keyboard:
73     printf ("Parent: please enter an integer: >> ");
74     scanf ("%d", &(value.sival_int));
75
76     printf ("Parent: I'll send the number you "
77           "entered: %d to the child!\n",
78           value.sival_int);
79
80     // Fork (create) a child process:
81     pid_t childpid = fork();
82     if (childpid == -1)
83     {
84         perror ("fork");
85         exit (EXIT_FAILURE);
```



```
86     }
87     // We enter this 'if' only inside the parent
88     // process:
89     else if (childpid)
90     {
91         printf ("Parent: A child with "
92                "pid = %d was forked!\n",
93                childpid);
94         // Check if we have the permission to
95         // send signals to the child:
96         int ret = sigqueue (childpid, 0, value);
97         if (ret)
98         {
99             fprintf (stderr,
100                    "We can't send signals "
101                    "to the child.\n");
102         }
103     else
104     {
105         // Wait several seconds for the child
106         // to launch:
107         sleep(5);
108         // Send SIGINT to the child:
109         printf ("Parent: I'll send SIGUSR2 "
110                "to the child now!\n");
111         ret = sigqueue (childpid,
112                        SIGUSR2,
113                        value);
114         if (ret)
115         {
116             perror ("sigqueue");
117             exit (EXIT_FAILURE);
118         }
119     }
120
121     // Wait for the child to return:
122     int status;
123
124     pid_t temp = waitpid (childpid, &status, 0);
125     if (temp == -1)
126     {
127         perror ("waitpid");
128         exit (EXIT_FAILURE);
```



```
129     }
130
131     printf ("Parent: The child that just "
132            "returned has the pid = %d.\n",
133            temp);
134
135     // Let's check what happened with the child:
136     if (WIFEXITED (status))
137         printf ("Parent: It terminated normally "
138                "with the exit status = %d.\n",
139                WEXITSTATUS (status));
140     if (WIFSIGNALED (status))
141         printf ("Parent: It was killed by "
142                "the signal = %d%s.\n",
143                WTERMSIG (status),
144                WCOREDUMP (status) ? " (dumped core)" : "");
145     if (WIFSTOPPED (status))
146         printf ("Parent: It was stopped "
147                "by the signal = %d.\n",
148                WSTOPSIG (status));
149     if (WIFCONTINUED (status))
150         printf ("Parent: Its execution "
151                "continued.\n");
152 }
153 else // We are inside the child process:
154 {
155     printf ("Child: Started executing now! "
156            "My pid is %d.\n",
157            getpid());
158
159     struct sigaction act;
160     act.sa_flags = SA_SIGINFO;
161     act.sa_sigaction = signal_handler_child;
162
163     // Register signal_handler_child:
164     if (sigaction (SIGUSR2, &act, NULL) == -1)
165     {
166         perror ("Cannot handle SIGUSR2!\n");
167         exit (EXIT_FAILURE);
168     }
169     printf ("Child: I just changed the "
170            "action of SIGUSR2.\n");
171     // Wait for the signal to be sent
```



```
172     //    by the parent:
173     pause();
174     printf ("Child: Returning...\n");
175 }
176
177 return EXIT_SUCCESS;
178 }
179
```



Topic 11: Memory Management

Note: A copy of each one of the sources listed below is included in this packet. To jump to the location of a source copy, click the page link under “Page”.

In case a source isn’t included (e.g., a YouTube video,) click the [↗](#) (external link) symbol under “Page” to open the external source in a browser window.

| # | Citation & Source Link | Page |
|----|---|------|
| 1 | Tychonievich, Luther. “An Overview of Memory.” CS 2130, University of Virginia, 2023. URL: https://www.cs.virginia.edu/~jh2jf/courses/cs2130/spring2023/readings/memory.html | 133 |
| 2 | Rosenberg, Burt. <i>Stack Layouts: with a look forward to virtual addressing</i> , University of Miami, Sept. 2015. URL: https://www.cs.miami.edu/home/burt/learning/Csc421.171/workbook/stack-memory.html | 147 |
| 3 | “malloc(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/malloc.3.html | 1238 |
| 4 | “calloc(3p) - Linux manual page”, <i>man7.org</i> , 2017. URL: https://man7.org/linux/man-pages/man3/calloc.3p.html | 1245 |
| 5 | “realloc(3p) - Linux manual page”, <i>man7.org</i> , 2017. URL: https://man7.org/linux/man-pages/man3/realloc.3p.html | 1249 |
| 6 | “free(3p) - Linux manual page”, <i>man7.org</i> , 2017. URL: https://man7.org/linux/man-pages/man3/free.3p.html | 1254 |
| 7 | “memset(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/memset.3.html | 1257 |
| 8 | “memcmp(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/memcmp.3.html | 1259 |
| 9 | “memmove(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/memmove.3.html | 1262 |
| 10 | “memcpy(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/memcpy.3.html | 1264 |
| 11 | “memchr(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/memchr.3.html | 1267 |
| 12 | Briskman, Miriam. “Materials for Topic 11: Memory Management.” <i>Topic 11: Memory Management — CISC 3350 Materials</i> , 2023. URL: www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_11/ | 1270 |
| 13 | Briskman, Miriam. “memory_functions.c.” (C source code) 14 May 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_11/memory_functions.c | 1271 |

malloc(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [EXAMPLES](#) | [SEE ALSO](#)

 malLoc(3)

Library Functions Manual

malLoc(3)**NAME** [top](#)

malloc, free, calloc, realloc, reallocarray - allocate and free dynamic memory

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdlib.h>
```

```
void *malloc(size_t size);  
void free(void *_Nullable ptr);  
void *calloc(size_t nmemb, size_t size);  
void *realloc(void *_Nullable ptr, size_t size);  
void *reallocarray(void *_Nullable ptr, size_t nmemb, size_t size);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
reallocarray():  
    Since glibc 2.29:  
        _DEFAULT_SOURCE  
    glibc 2.28 and earlier:  
        _GNU_SOURCE
```

DESCRIPTION [top](#)

malloc()

The **malloc()** function allocates *size* bytes and returns a pointer to the allocated memory. *The memory is not initialized.* If *size* is 0, then **malloc()** returns a unique pointer value that can later be successfully passed to **free()**. (See "Nonportable behavior" for portability issues.)

free()

The **free()** function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc()** or related functions. Otherwise, or if *ptr* has already been freed, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

calloc()

The **calloc()** function allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero. If *nmemb* or *size* is 0, then **calloc()** returns a unique pointer value that can later be successfully passed to **free()**.

If the multiplication of *nmemb* and *size* would result in integer overflow, then **calloc()** returns an error. By contrast, an integer overflow would not be detected in the following call to **malloc()**, with the result that an incorrectly sized block of memory would be allocated:

```
malloc(nmemb * size);
```

realloc()

The **realloc()** function changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents of the memory will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will *not* be initialized.

If *ptr* is NULL, then the call is equivalent to **malloc(size)**, for all values of *size*.

If *size* is equal to zero, and *ptr* is not NULL, then the call is equivalent to **free(ptr)** (but see "Nonportable behavior" for portability issues).

Unless *ptr* is NULL, it must have been returned by an earlier call to **malloc** or related functions. If the area pointed to was moved, a **free(ptr)** is done.

reallocarray()



The **reallocarray()** function changes the size of (and possibly moves) the memory block pointed to by *ptr* to be large enough for an array of *nmemb* elements, each of which is *size* bytes. It is equivalent to the call

```
realloc(ptr, nmemb * size);
```

However, unlike that **realloc()** call, **reallocarray()** fails safely in the case where the multiplication would overflow. If such an overflow occurs, **reallocarray()** returns an error.

RETURN VALUE [top](#)

The **malloc()**, **calloc()**, **realloc()**, and **reallocarray()** functions return a pointer to the allocated memory, which is suitably aligned for any type that fits into the requested size or less. On error, these functions return NULL and set *errno*. Attempting to allocate more than **PTRDIFF_MAX** bytes is considered an error, as an object that large could cause later pointer subtraction to overflow.

The **free()** function returns no value, and preserves *errno*.

The **realloc()** and **reallocarray()** functions return NULL if *ptr* is not NULL and the requested size is zero; this is not considered an error. (See "Nonportable behavior" for portability issues.) Otherwise, the returned pointer may be the same as *ptr* if the allocation was not moved (e.g., there was room to expand the allocation in-place), or different from *ptr* if the allocation was moved to a new address. If these functions fail, the original block is left untouched; it is not freed or moved.

ERRORS [top](#)

calloc(), **malloc()**, **realloc()**, and **reallocarray()** can fail with the following error:

ENOMEM Out of memory. Possibly, the application hit the **RLIMIT_AS** or **RLIMIT_DATA** limit described in [getrlimit\(2\)](#). Another reason could be that the number of mappings created by the caller process exceeded the limit specified by [/proc/sys/vm/max_map_count](#).

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|---|---------------|---------|
| <code>malloc()</code> , <code>free()</code> , <code>calloc()</code> , <code>realloc()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

`malloc()`
`free()`
`calloc()`
`realloc()`
 C11, POSIX.1-2008.

`reallocarray()`
 None.

HISTORY [top](#)

`malloc()`
`free()`
`calloc()`
`realloc()`
 POSIX.1-2001, C89.

`reallocarray()`
 glibc 2.26. OpenBSD 5.6, FreeBSD 11.0.

`malloc()` and related functions rejected sizes greater than `PTRDIFF_MAX` starting in glibc 2.30.

`free()` preserved `errno` starting in glibc 2.33.

NOTES [top](#)

By default, Linux follows an optimistic memory allocation strategy. This means that when `malloc()` returns non-NULL there is no guarantee that the memory really is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer. For more information, see the description of `/proc/sys/vm/overcommit_memory` and `/proc/sys/vm/oom_adj` in [proc\(5\)](#), and the Linux kernel source file `Documentation/vm/overcommit-accounting.rst`.

Normally, `malloc()` allocates memory from the heap, and adjusts the size of the heap as required, using `sbrk(2)`. When allocating blocks of memory larger than `MMAP_THRESHOLD` bytes, the glibc `malloc()` implementation allocates the memory as a private anonymous mapping using `mmap(2)`. `MMAP_THRESHOLD` is 128 kB by default, but is adjustable using `mallopt(3)`. Prior to Linux 4.7 allocations performed using `mmap(2)` were unaffected by the `RLIMIT_DATA` resource limit; since Linux 4.7, this limit is also enforced for allocations performed using `mmap(2)`.

To avoid corruption in multithreaded applications, mutexes are used internally to protect the memory-management data structures employed by these functions. In a multithreaded application in which threads simultaneously allocate and free memory, there could be contention for these mutexes. To scalably handle memory allocation in multithreaded applications, glibc creates additional *memory allocation arenas* if mutex contention is detected. Each arena is a large region of memory that is internally allocated by the system (using `brk(2)` or `mmap(2)`), and managed with its own mutexes.

If your program uses a private memory allocator, it should do so by replacing `malloc()`, `free()`, `calloc()`, and `realloc()`. The replacement functions must implement the documented glibc behaviors, including *errno* handling, size-zero allocations, and overflow checking; otherwise, other library routines may crash or operate incorrectly. For example, if the replacement `free()` does not preserve *errno*, then seemingly unrelated library routines may fail without having a valid reason in *errno*. Private memory allocators may also need to replace other glibc functions; see "Replacing malloc" in the glibc manual for details.

Crashes in memory allocators are almost always related to heap corruption, such as overflowing an allocated chunk or freeing the same pointer twice.

The `malloc()` implementation is tunable via environment variables; see `mallopt(3)` for details.

Nonportable behavior

The behavior of these functions when the requested size is zero is glibc specific; other implementations may return NULL without setting *errno*, and portable POSIX programs should tolerate such behavior. See `realloc(3p)`.

POSIX requires memory allocators to set *errno* upon failure. However, the C standard does not require this, and applications

portable to non-POSIX platforms should not assume this.

Portable programs should not use private memory allocators, as POSIX and the C standard do not allow replacement of **malloc()**, **free()**, **calloc()**, and **realloc()**.

EXAMPLES [top](#)

```
#include <err.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MALLOCARRAY(n, type) ((type *) my_mallocarray(n, sizeof(type)))
#define MALLOC(type) MALLOCARRAY(1, type)

static inline void *my_mallocarray(size_t nmemb, size_t size);

int
main(void)
{
    char *p;

    p = MALLOCARRAY(32, char);
    if (p == NULL)
        err(EXIT_FAILURE, "malloc");

    strcpy(p, "foo", 32);
    puts(p);
}

static inline void *
my_mallocarray(size_t nmemb, size_t size)
{
    return reallocarray(NULL, nmemb, size);
}
```

SEE ALSO [top](#)

[valgrind\(1\)](#), [brk\(2\)](#), [mmap\(2\)](#), [alloca\(3\)](#), [malloc_get_state\(3\)](#), [malloc_info\(3\)](#), [malloc_trim\(3\)](#), [malloc_usable_size\(3\)](#), [mallopt\(3\)](#), [mcheck\(3\)](#), [mtrace\(3\)](#), [posix_memalign\(3\)](#)

For details of the GNU C library implementation, see <https://sourceware.org/glibc/wiki/MallocInternals>.

Linux man-pages (unreleased) (date)

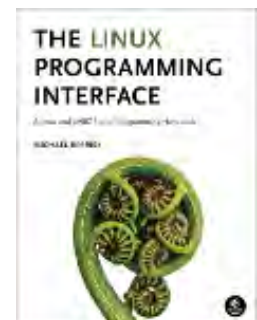
maLLoc(3)

Pages that refer to this page: [memusage\(1\)](#), [brk\(2\)](#), [clone\(2\)](#), [getrlimit\(2\)](#), [io_uring_register\(2\)](#), [mlock\(2\)](#), [mmap\(2\)](#), [mremap\(2\)](#), [alloca\(3\)](#), [argz_add\(3\)](#), [asprintf\(3\)](#), [ausearch_add_expression\(3\)](#), [avc_init\(3\)](#), [backtrace\(3\)](#), [basename\(3\)](#), [canonicalize_file_name\(3\)](#), [cfree\(3\)](#), [CPU_SET\(3\)](#), [dbopen\(3\)](#), [end\(3\)](#), [exec\(3\)](#), [fopen\(3\)](#), [fseek\(3\)](#), [fts\(3\)](#), [ftw\(3\)](#), [getcwd\(3\)](#), [getgrent\(3\)](#), [getgrnam\(3\)](#), [getifaddrs\(3\)](#), [getline\(3\)](#), [getpwent\(3\)](#), [getpwnam\(3\)](#), [glob\(3\)](#), [hsearch\(3\)](#), [if_nameindex\(3\)](#), [lber-memory\(3\)](#), [ldap_memory\(3\)](#), [mallinfo\(3\)](#), [malloc_get_state\(3\)](#), [malloc_hook\(3\)](#), [malloc_info\(3\)](#), [malloc_stats\(3\)](#), [malloc_trim\(3\)](#), [malloc_usable_size\(3\)](#), [mallopt\(3\)](#), [mcheck\(3\)](#), [mpool\(3\)](#), [mtrace\(3\)](#), [numa\(3\)](#), [open_memstream\(3\)](#), [pam_conv\(3\)](#), [pmaddderived\(3\)](#), [__pmaf\(3\)](#), [pmagetlog\(3\)](#), [pmapi\(3\)](#), [pmarewritedata\(3\)](#), [pmarewritemeta\(3\)](#), [pmdachildren\(3\)](#), [pmdafetch\(3\)](#), [pmdainstance\(3\)](#), [pmdalabel\(3\)](#), [pmdatext\(3\)](#), [pmdatrace\(3\)](#), [pmdiscoverservices\(3\)](#), [pmextractvalue\(3\)](#), [pmfault\(3\)](#), [pmfetch\(3\)](#), [pmfetcharchive\(3\)](#), [pmfetchgroup\(3\)](#), [pmfreelabelsets\(3\)](#), [pmfreeprofile\(3\)](#), [pmfreeresult\(3\)](#), [pmfstring\(3\)](#), [pmgetchildren\(3\)](#), [pmgetchildrenstatus\(3\)](#), [pmgetindom\(3\)](#), [pmgetindomarchive\(3\)](#), [pmlookupindomtext\(3\)](#), [pmlookuptext\(3\)](#), [pmnameall\(3\)](#), [pmnameid\(3\)](#), [pmnameindom\(3\)](#), [pmnameindomarchive\(3\)](#), [pmnewcontextzone\(3\)](#), [pmnewzone\(3\)](#), [pmnomem\(3\)](#), [__pmparsectime\(3\)](#), [pmparsehostattrsspec\(3\)](#), [pmparsehostspect\(3\)](#), [pmparseinterval\(3\)](#), [pmparsemetricspec\(3\)](#), [__pmparsetime\(3\)](#), [pmparsetimewindow\(3\)](#), [pmparseunitsstr\(3\)](#), [pmregisterderived\(3\)](#), [posix_memalign\(3\)](#), [pthread_setcancelstate\(3\)](#), [random_r\(3\)](#), [readdir\(3\)](#), [readline\(3\)](#), [realpath\(3\)](#), [scandir\(3\)](#), [sd_bus_creds_get_pid\(3\)](#), [sd_bus_error\(3\)](#), [sd_bus_path_encode\(3\)](#), [sd_get_seats\(3\)](#), [sd_journal_get_catalog\(3\)](#), [sd_journal_get_cursor\(3\)](#), [sd-login\(3\)](#), [sd_machine_get_class\(3\)](#), [sd_path_lookup\(3\)](#), [sd_pid_get_owner_uid\(3\)](#), [sd_seat_get_active\(3\)](#), [sd_session_is_active\(3\)](#), [sd_uid_get_state\(3\)](#), [seccomp_syscall_resolve_name\(3\)](#), [security_class_to_string\(3\)](#), [selabel_get_digests_all_partial_matches\(3\)](#), [selinux_boolean_sub\(3\)](#), [selinux_getpolicytype\(3\)](#), [selinux_raw_context_to_color\(3\)](#), [setbuf\(3\)](#), [sscanf\(3\)](#), [strdup\(3\)](#), [string\(3\)](#), [tempnam\(3\)](#), [tracefs_event_get_file\(3\)](#), [tracefs_instance_set_affinity\(3\)](#), [tracefs_tracers\(3\)](#), [void\(3type\)](#), [wcsdup\(3\)](#), [proc\(5\)](#), [environ\(7\)](#), [feature_test_macros\(7\)](#), [signal-safety\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



calloc(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 CALLOC(3P)

POSIX Programmer's Manual

CALLOC(3P)

PROLOG [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

`calloc` – a memory allocator

SYNOPSIS [top](#)

```
#include <stdlib.h>
```

```
void *calloc(size_t nelem, size_t elsize);
```

DESCRIPTION [top](#)

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The `calloc()` function shall allocate unused space for an array of *nelem* elements each of whose size in bytes is *elsize*. The space

shall be initialized to all bits 0.

The order and contiguity of storage allocated by successive calls to `calloc()` is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned shall point to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned. If the size of the space requested is 0, the behavior is implementation-defined: either a null pointer shall be returned, or the behavior shall be as if the size were some non-zero value, except that the behavior is undefined if the returned pointer is used to access an object.

RETURN VALUE [top](#)

Upon successful completion with both `nlem` and `elsize` non-zero, `calloc()` shall return a pointer to the allocated space. If either `nlem` or `elsize` is 0, then either:

- * A null pointer shall be returned and `errno` may be set to an implementation-defined value, or
- * A pointer to the allocated space shall be returned. The application shall ensure that the pointer is not used to access an object.

Otherwise, it shall return a null pointer and set `errno` to indicate the error.

ERRORS [top](#)

The `calloc()` function shall fail if:

ENOMEM Insufficient memory is available.

The following sections are informative.

EXAMPLES [top](#)

None.

APPLICATION USAGE [top](#)

There is now no requirement for the implementation to support the inclusion of `<malloc.h>`.

RATIONALE [top](#)

None.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

[free\(3p\)](#), [malloc\(3p\)](#), [realloc\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [stdlib.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see

https://www.kernel.org/doc/man-pages/reporting_bugs.html .

IEEE/The Open Group

2017

CALLOC(3P)

Pages that refer to this page: [stdlib.h\(0p\)](#), [free\(3p\)](#), [malloc\(3p\)](#), [realloc\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



realloc(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 REALLOC(3P)

POSIX Programmer's Manual

REALLOC(3P)**PROLOG** [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

realloc – memory reallocator

SYNOPSIS [top](#)

```
#include <stdlib.h>
```

```
void *realloc(void *ptr, size_t size);
```

DESCRIPTION [top](#)

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The `realloc()` function shall deallocate the old object pointed to by `ptr` and return a pointer to a new object that has the size

specified by *size*. The contents of the new object shall be the same as that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values. If the size of the space requested is zero, the behavior shall be implementation-defined: either a null pointer is returned, or the behavior shall be as if the size were some non-zero value, except that the behavior is undefined if the returned pointer is used to access an object. If the space cannot be allocated, the object shall remain unchanged.

If *ptr* is a null pointer, *realloc()* shall be equivalent to *malloc()* for the specified size.

If *ptr* does not match a pointer returned earlier by *calloc()*, *malloc()*, or *realloc()* or if the space has previously been deallocated by a call to *free()* or *realloc()*, the behavior is undefined.

The order and contiguity of storage allocated by successive calls to *realloc()* is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned shall point to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned.

RETURN VALUE [top](#)

Upon successful completion, *realloc()* shall return a pointer to the (possibly moved) allocated space. If *size* is 0, either:

- * A null pointer shall be returned and, if *ptr* is not a null pointer, *errno* shall be set to an implementation-defined value.
- * A pointer to the allocated space shall be returned, and the memory object pointed to by *ptr* shall be freed. The application shall ensure that the pointer is not used to access an object.

If there is not enough available memory, `realloc()` shall return a null pointer and set `errno` to `[ENOMEM]`. If `realloc()` returns a null pointer and `errno` has been set to `[ENOMEM]`, the memory referenced by `ptr` shall not be changed.

ERRORS [top](#)

The `realloc()` function shall fail if:

ENOMEM Insufficient memory is available.

The following sections are informative.

EXAMPLES [top](#)

None.

APPLICATION USAGE [top](#)

The description of `realloc()` has been modified from previous versions of this standard to align with the ISO/IEC 9899:1999 standard. Previous versions explicitly permitted a call to `realloc(p, 0)` to free the space pointed to by `p` and return a null pointer. While this behavior could be interpreted as permitted by this version of the standard, the C language committee have indicated that this interpretation is incorrect. Applications should assume that if `realloc()` returns a null pointer, the space pointed to by `p` has not been freed. Since this could lead to double-frees, implementations should also set `errno` if a null pointer actually indicates a failure, and applications should only free the space if `errno` was changed.

RATIONALE [top](#)

None.

FUTURE DIRECTIONS [top](#)

This standard defers to the ISO C standard. While that standard currently has language that might permit `realloc(p, 0)`, where `p` is not a null pointer, to free `p` while still returning a null pointer, the committee responsible for that standard is considering clarifying the language to explicitly prohibit that alternative.

SEE ALSO [top](#)

[calloc\(3p\)](#), [free\(3p\)](#), [malloc\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [stdlib.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

IEEE/The Open Group

2017

REALLOC(3P)

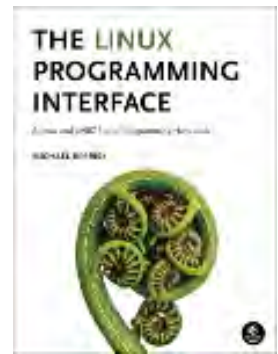
Pages that refer to this page: [stdlib.h\(0p\)](#), [calloc\(3p\)](#), [free\(3p\)](#), [getdelim\(3p\)](#), [malloc\(3\)](#), [malloc\(3p\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



free(3p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [EXAMPLES](#) | [APPLICATION USAGE](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 FREE(3P)

POSIX Programmer's Manual

FREE(3P)

PROLOG [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

free – free allocated memory

SYNOPSIS [top](#)

```
#include <stdlib.h>

void free(void *ptr);
```

DESCRIPTION [top](#)

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The `free()` function shall cause the space pointed to by `ptr` to be deallocated; that is, made available for further allocation. If

ptr is a null pointer, no action shall occur. Otherwise, if the argument does not match a pointer earlier returned by a function in POSIX.1-2008 that allocates memory as if by *malloc()*, or if the space has been deallocated by a call to *free()* or *realloc()*, the behavior is undefined.

Any use of a pointer that refers to freed space results in undefined behavior.

RETURN VALUE [top](#)

The *free()* function shall not return a value.

ERRORS [top](#)

No errors are defined.

The following sections are informative.

EXAMPLES [top](#)

None.

APPLICATION USAGE [top](#)

There is now no requirement for the implementation to support the inclusion of *<malloc.h>*.

RATIONALE [top](#)

None.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)



[calloc\(3p\)](#), [malloc\(3p\)](#), [posix_memalign\(3p\)](#), [realloc\(3p\)](#)

The Base Definitions volume of POSIX.1-2017, [stdlib.h\(0p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

IEEE/The Open Group

2017

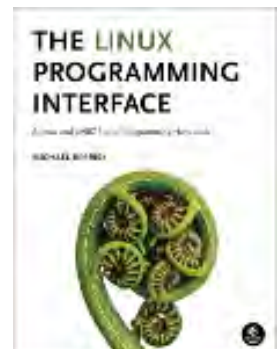
FREE(3P)

Pages that refer to this page: [stdlib.h\(0p\)](#), [calloc\(3p\)](#), [getdelim\(3p\)](#), [malloc\(3p\)](#), [open_memstream\(3p\)](#), [posix_memalign\(3p\)](#), [putenv\(3p\)](#), [realloc\(3p\)](#), [realpath\(3p\)](#), [strdup\(3p\)](#), [tempnam\(3p\)](#), [wcsdup\(3p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



memset(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 memset(3)

Library Functions Manual

memset(3)

NAME [top](#)

memset - fill memory with a constant byte

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <string.h>
```

```
void *memset(void s[.n], int c, size_t n);
```

DESCRIPTION [top](#)

The `memset()` function fills the first *n* bytes of the memory area pointed to by *s* with the constant byte *c*.

RETURN VALUE [top](#)

The `memset()` function returns a pointer to the memory area *s*.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------------------|---------------|---------|
| <code>memset()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, C89, SVr4, 4.3BSD.

SEE ALSO [top](#)

[bstring\(3\)](#), [bzero\(3\)](#), [swab\(3\)](#), [wmemset\(3\)](#)

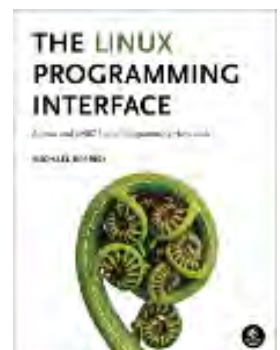
Linux man-pages (unreleased) [\(date\)](#) [memset\(3\)](#)

Pages that refer to this page: [mount_setattr\(2\)](#), [openat2\(2\)](#), [set_thread_area\(2\)](#), [bstring\(3\)](#), [bzero\(3\)](#), [NULL\(3const\)](#), [size_t\(3type\)](#), [void\(3type\)](#), [wmemset\(3\)](#), [aio\(7\)](#), [feature_test_macros\(7\)](#), [signal-safety\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



memcmp(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [CAVEATS](#) | [SEE ALSO](#)

 memcmp(3)

Library Functions Manual

memcmp(3)

NAME [top](#)

memcmp - compare memory areas

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <string.h>
```

```
int memcmp(const void s1[], const void s2[], size_t n);
```

DESCRIPTION [top](#)

The `memcmp()` function compares the first *n* bytes (each interpreted as *unsigned char*) of the memory areas *s1* and *s2*.

RETURN VALUE [top](#)

The `memcmp()` function returns an integer less than, equal to, or greater than zero if the first *n* bytes of *s1* is found, respectively, to be less than, to match, or be greater than the first *n* bytes of *s2*.

For a nonzero return value, the sign is determined by the sign of the difference between the first pair of bytes (interpreted as *unsigned char*) that differ in *s1* and *s2*.

If *n* is zero, the return value is zero.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------------------|---------------|---------|
| <code>memcmp()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, C89, SVr4, 4.3BSD.

CAVEATS [top](#)

Do not use `memcmp()` to compare confidential data, such as cryptographic secrets, because the CPU time required for the comparison depends on the contents of the addresses compared, this function is subject to timing-based side-channel attacks. In such cases, a function that performs comparisons in deterministic time, depending only on *n* (the quantity of bytes compared) is required. Some operating systems provide such a function (e.g., NetBSD's `consttime_memequal()`), but no such function is specified in POSIX. On Linux, you may need to implement such a function yourself.

SEE ALSO [top](#)



[bstring\(3\)](#), [strcasecmp\(3\)](#), [strcmp\(3\)](#), [strcoll\(3\)](#), [strncasecmp\(3\)](#),
[strncmp\(3\)](#), [wmemcmp\(3\)](#)

Linux man-pages (unreleased) (date)

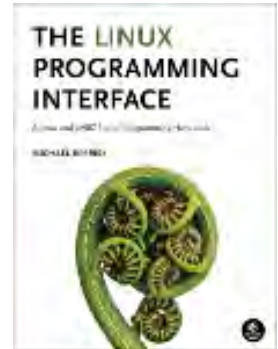
memcmp(3)

Pages that refer to this page: [bcmp\(3\)](#), [bstring\(3\)](#), [size_t\(3type\)](#), [strcasecmp\(3\)](#),
[strcmp\(3\)](#), [strcoll\(3\)](#), [strxfrm\(3\)](#), [void\(3type\)](#), [wmemcmp\(3\)](#), [signal-safety\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of
[The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



memmove(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 memmove(3)

Library Functions Manual

memmove(3)

NAME [top](#)

memmove - copy memory area

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <string.h>
```

```
void *memmove(void dest[.n], const void src[.n], size_t n);
```

DESCRIPTION [top](#)

The `memmove()` function copies *n* bytes from memory area *src* to memory area *dest*. The memory areas may overlap: copying takes place as though the bytes in *src* are first copied into a temporary array that does not overlap *src* or *dest*, and the bytes are then copied from the temporary array to *dest*.

RETURN VALUE [top](#)

The `memmove()` function returns a pointer to *dest*.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|------------------------|---------------|---------|
| <code>memmove()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, C89, SVr4, 4.3BSD.

SEE ALSO [top](#)

[bcopy\(3\)](#), [bstring\(3\)](#), [memccpy\(3\)](#), [memcpy\(3\)](#), [strcpy\(3\)](#), [strncpy\(3\)](#), [wmemmove\(3\)](#)

Linux man-pages (unreleased) (date) [memmove\(3\)](#)

Pages that refer to this page: [bcopy\(3\)](#), [bstring\(3\)](#), [memccpy\(3\)](#), [memcpy\(3\)](#), [mempcpy\(3\)](#), [wmemmove\(3\)](#), [signal-safety\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



memcpy(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [CAVEATS](#) | [SEE ALSO](#)

 memcpy(3)

Library Functions Manual

memcpy(3)

NAME [top](#)

memcpy - copy memory area

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <string.h>
```

```
void *memcpy(void dest[restrict .n], const void src[restrict .n],  
             size_t n);
```

DESCRIPTION [top](#)

The **memcpy()** function copies *n* bytes from memory area *src* to memory area *dest*. The memory areas must not overlap. Use [memmove\(3\)](#) if the memory areas do overlap.

RETURN VALUE [top](#)

The **memcpy()** function returns a pointer to *dest*.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------------------|---------------|---------|
| <code>memcpy()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, C89, SVr4, 4.3BSD.

CAVEATS [top](#)

Failure to observe the requirement that the memory areas do not overlap has been the source of significant bugs. (POSIX and the C standards are explicit that employing `memcpy()` with overlapping areas produces undefined behavior.) Most notably, in glibc 2.13 a performance optimization of `memcpy()` on some platforms (including x86-64) included changing the order in which bytes were copied from *src* to *dest*.

This change revealed breakages in a number of applications that performed copying with overlapping areas. Under the previous implementation, the order in which the bytes were copied had fortuitously hidden the bug, which was revealed when the copying order was reversed. In glibc 2.14, a versioned symbol was added so that old binaries (i.e., those linked against glibc versions earlier than 2.14) employed a `memcpy()` implementation that safely handles the overlapping buffers case (by providing an "older" `memcpy()` implementation that was aliased to `memmove(3)`).

SEE ALSO [top](#)



[bcopy\(3\)](#), [bstring\(3\)](#), [memccpy\(3\)](#), [memmove\(3\)](#), [mempcpy\(3\)](#),
[strcpy\(3\)](#), [strncpy\(3\)](#), [wmemcpy\(3\)](#)

Linux man-pages (unreleased) (date)

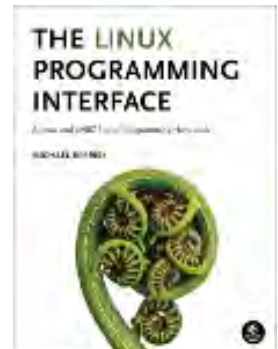
memcpy(3)

Pages that refer to this page: [bcopy\(3\)](#), [bstring\(3\)](#), [cmsg\(3\)](#), [CPU_SET\(3\)](#),
[memccpy\(3\)](#), [memmove\(3\)](#), [mempcpy\(3\)](#), [size_t\(3type\)](#), [void\(3type\)](#), [wmemcpy\(3\)](#),
[feature_test_macros\(7\)](#), [signal-safety\(7\)](#), [string_copying\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of
The Linux Programming Interface.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



memchr(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 memchr(3)

Library Functions Manual

memchr(3)

NAME [top](#)

memchr, memrchr, rawmemchr - scan memory for a character

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <string.h>
```

```
void *memchr(const void s[.n], int c, size_t n);  
void *memrchr(const void s[.n], int c, size_t n);
```

```
[[deprecated]] void *rawmemchr(const void *s, int c);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
memrchr(), rawmemchr():  
    _GNU_SOURCE
```

DESCRIPTION [top](#)

The `memchr()` function scans the initial `n` bytes of the memory area pointed to by `s` for the first instance of `c`. Both `c` and the bytes of the memory area pointed to by `s` are interpreted as *unsigned char*.

The `memrchr()` function is like the `memchr()` function, except that it searches backward from the end of the `n` bytes pointed to by `s` instead of forward from the beginning.

The `rawmemchr()` function is similar to `memchr()`, but it assumes (i.e., the programmer knows for certain) that an instance of `c` lies somewhere in the memory area starting at the location pointed to by `s`. If an instance of `c` is not found, the behavior is undefined. Use either `strlen(3)` or `memchr(3)` instead.

RETURN VALUE [top](#)

The `memchr()` and `memrchr()` functions return a pointer to the matching byte or NULL if the character does not occur in the given memory area.

The `rawmemchr()` function returns a pointer to the matching byte.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|---|---------------|---------|
| <code>memchr()</code> , <code>memrchr()</code> , <code>rawmemchr()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

`memchr()`
C11, POSIX.1-2008.

`memrchr()`
`rawmemchr()`
GNU.



HISTORY [top](#)

memchr()
POSIX.1-2001, C89, SVr4, 4.3BSD.

memrchr()
glibc 2.2.

rawmemchr()
glibc 2.1.

SEE ALSO [top](#)

[bstring\(3\)](#), [ffs\(3\)](#), [memmem\(3\)](#), [strchr\(3\)](#), [strpbrk\(3\)](#), [strrchr\(3\)](#),
[strsep\(3\)](#), [strspn\(3\)](#), [strstr\(3\)](#), [wmemchr\(3\)](#)

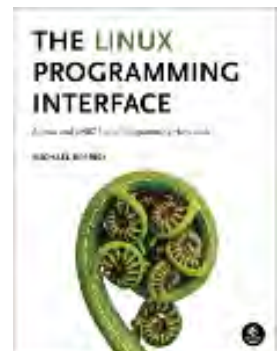
Linux man-pages (unreleased) (date) [memchr\(3\)](#)

Pages that refer to this page: [bstring\(3\)](#), [ffs\(3\)](#), [memchr\(3\)](#), [strchr\(3\)](#), [strpbrk\(3\)](#),
[strsep\(3\)](#), [strspn\(3\)](#), [strstr\(3\)](#), [strtok\(3\)](#), [wmemchr\(3\)](#), [signal-safety\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



Materials for Topic 11: Memory Management

Full C Programs

- [memory_functions.c](#) - a C program demonstrating uses of some C memory functions, including `malloc()`, `calloc()`, `realloc()`, `free()`, `memset()`, `memcpy()`, and `memcmp()`.

Runnable Linux Commands

- The command:

```
gcc -Wall -Wextra -O2 -g -o memory_functions  
memory_functions.c
```

compiles the C source code located inside the file `memory_functions.c`. See more details [here](#).

- The command:

```
./short_prompt
```

executes code inside a file named `short_prompt` and sources it (applies all the changes to the current session.)

See more details [here](#).

- The command:

```
./long_prompt
```

executes code inside a file named `long_prompt` and sources it (applies all the changes to the current session.)

See more details [here](#).



This website by [Miriam Briskman](#) is licensed under [CC BY-NC 4.0](#).

```
1  /* A program demonstrating uses of some C memory
2  *   functions: malloc(), calloc(), realloc(),
3  *   free(), memset(), memcpy(), and memcmp().
4  *
5  *   Miriam Briskman, 5/14/2023
6  *   CISC 3350, Brooklyn College
7  *   Licensed under CC BY-NC 4.0
8  */
9
10 // For perror() and printf():
11 #include <stdio.h>
12 // For malloc(), calloc(), realloc(), and free():
13 #include <stdlib.h>
14 // For memset(), memcpy(), memcmp(), strcmp(),
15 //   and strncmp():
16 #include <string.h>
17
18 #define INI_SIZE 10
19 #define NEW_SIZE 20
20
21 int main ()
22 {
23     // Allocate memory for a char array:
24     char * small_A = malloc (INI_SIZE * sizeof(char));
25
26     if (!small_A) // Same as (small_A == NULL)
27     {
28         perror ("malloc");
29         exit (EXIT_FAILURE);
30     }
31
32     // Set small_A to contain "AAAAAAAAA":
33     memset (small_A, 'A', 9);
34     // Note that none of the mem functions,
35     //   including memset(), returns errors.
36
37     // Set small_A[9] to '\0':
38     small_A[9] = '\0';
39
40     // At this point, small_A will contain:
41     //   AAAAAAAAA\0
42
```



```
43 // Let's resize small_A to 20 spots:
44 char * A = realloc (small_A, NEW_SIZE);
45 if (!A)
46 {
47     perror ("realloc");
48     exit (EXIT_FAILURE);
49 }
50
51 // From this point on, we'll use 'A' rather
52 //   than 'small_A' since the OS might
53 //   have allocated memory at a location
54 //   different from the location of small_A.
55
56 // Set A[10] to A[19] to 'A':
57 memset (A + 10, 'A', 10);
58
59 // At this point, A should contain:
60 //   AAAAAAAAAA\0AAAAAAAAAA
61 //   without any \0 at the end.
62
63 // calloc() an array B:
64 char * B = calloc (NEW_SIZE, sizeof(char));
65 if (!B)
66 {
67     perror ("calloc");
68     return EXIT_FAILURE;
69 }
70
71 // calloc() not only allocates memory, but
72 //   also sets every spot in it to \0.
73
74 // Copy 9 'A' characters from A to B:
75 memcpy (B, A, 9);
76
77 // B should contain the following now:
78 //   AAAAAAAAAA\0\0\0\0\0\0\0\0\0\0
79
80 // Set the last 10 spots of B to "BBBBBBBBBB":
81 memset (B + 10, 'B', 10);
82
83 // B should contain the following now:
84 //   AAAAAAAAAA\0BBBBBBBBBB
85
```




```
86 // Printing the two strings using loops:
87 int i;
88
89 printf ("A contains: ");
90
91 for (i = 0; i < NEW_SIZE; i++)
92 {
93     if (A[i] == '\0')
94         printf ("\0");
95     else
96         printf ("%c", A[i]);
97 }
98
99 printf ("\nB contains: ");
100
101 for (i = 0; i < NEW_SIZE; i++)
102 {
103     if (B[i] == '\0')
104         printf ("\0");
105     else
106         printf ("%c", B[i]);
107 }
108
109 // Compare A and B using strcmp():
110 int result = strcmp (A, B);
111 // strcmp() will stop the character
112 // comparison when it sees '\0'.
113
114 printf ("\nAccording to strcmp(), ");
115
116 if (result < 0)
117     printf ("A appears before B "
118             "in the dictionary.\n");
119 else if (result > 0)
120     printf ("A appears after B "
121             "in the dictionary.\n");
122 else
123     printf ("Both A and B contain "
124             "the same characters.\n");
125
126 // Follow-up question #1:
127 // Which one of the 3 statements above
128 // will be printed to the screen?
```



```
129 // Hint:
130 //   A contains: AAAAAAAAA\0AAAAAAAAA
131 //   B contains: AAAAAAAAA\0BBBBBBBBB
132 //   and strcmp() stops the comparison
133 //   at '\0'.
134
135 // Compare A and B using strncmp():
136 result = strncmp (A, B, NEW_SIZE);
137 // strncmp() is asked to compare
138 //   the first 20 characters, but
139 //   it stops the comparison at '\0'.
140
141 printf ("According to strcmp(), ");
142
143 if (result < 0)
144     printf ("A appears before B "
145            "in the dictionary.\n");
146 else if (result > 0)
147     printf ("A appears after B "
148            "in the dictionary.\n");
149 else
150     printf ("Both A and B contain "
151            "the same characters.\n");
152
153 // Follow-up question #2:
154 // Which one of the 3 statements above
155 //   will be printed to the screen?
156 // Hint:
157 //   A contains: AAAAAAAAA\0AAAAAAAAA
158 //   B contains: AAAAAAAAA\0BBBBBBBBB
159 //   and strncmp(), too, stops at '\0'.
160
161 // Compare A and B using memcmp():
162 result = memcmp (A, B, NEW_SIZE);
163 // memcmp() will compare exactly
164 //   the first 20 characters (it
165 //   won't stop at '\0'.)
166
167 printf ("According to memcmp(), ");
168
169 if (result < 0)
170     printf ("A appears before B "
171            "in the dictionary.\n");
```




```
172     else if (result > 0)
173         printf ("A appears after B "
174                "in the dictionary.\n");
175     else
176         printf ("both A and B contain "
177                "the same characters.\n");
178
179     // Follow-up question #3:
180     // Which one of the 3 statements above
181     // will be printed to the screen?
182     // Hint:
183     //   A contains: AAAAAAAAAA\0AAAAAAAAAA
184     //   B contains: AAAAAAAAAA\0BBBBBBBBBB
185     //   and memcmp() WON'T stop at '\0'.
186
187     printf ("The reason for the difference "
188            "is that strcmp() and strcmp() stop "
189            "comparing \nwhen they encounter the "
190            "null character, \\0, but memcmp() "
191            "keeps comparing \n"
192            "until all NEW_SIZE characters "
193            "are compared.\n");
194
195     // Freeing memory:
196     free (A);
197     free (B);
198
199     return EXIT_SUCCESS;
200 }
201
```



Topic 12: File and Directory Management

Note: A copy of each one of the sources listed below is included in this packet. To jump to the location of a source copy, click the page link under “Page”.

In case a source isn’t included (e.g., a YouTube video,) click the  (external link) symbol under “Page” to open the external source in a browser window.

| # | Citation & Source Link | Page |
|----|--|------|
| 1 | “inode(7) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man7/inode.7.html | 1279 |
| 2 | Stallman, Richard M. and MacKenzie, David. “ls(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/ls.1.html | 171 |
| 3 | “stat(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/stat.2.html | 1287 |
| 4 | “chmod(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/chmod.2.html | 1299 |
| 5 | “chown(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/chown.2.html | 1305 |
| 6 | “getxattr(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/getxattr.2.html | 1313 |
| 7 | “setxattr(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/setxattr.2.html | 1317 |
| 8 | “listxattr(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/listxattr.2.html | 1321 |
| 9 | “removexattr(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/removexattr.2.html | 1328 |
| 10 | “getcwd(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/getcwd.3.html | 1331 |
| 11 | “chdir(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/chdir.2.html | 1337 |
| 12 | “passwd(5) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man5/passwd.5.html | 1340 |
| 13 | “getent(1) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man1/getent.1.html | 1344 |
| 14 | “mkdir(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/mkdir.2.html | 1349 |
| 15 | “umask(1p) - Linux manual page”, <i>man7.org</i> , 2017. URL: https://man7.org/linux/man-pages/man1/umask.1p.html | 1354 |

| # | Citation & Source Link | Page |
|----|--|------|
| 16 | “rmdir(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/rmdir.2.html | 1362 |
| 17 | MacKenzie, David. “rmdir(1) - Linux manual page”, <i>man7.org</i> , Aug. 2023. URL: https://man7.org/linux/man-pages/man1/rmdir.1.html | 195 |
| 18 | “opendir(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/opendir.3.html | 1365 |
| 19 | “dirfd(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/dirfd.3.html | 1369 |
| 20 | “readdir(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/readdir.3.html | 1372 |
| 21 | “closedir(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/closedir.3.html | 1378 |
| 22 | “link(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/link.2.html | 1381 |
| 23 | “symlink(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/symlink.2.html | 1388 |
| 24 | “unlink(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/unlink.2.html | 1393 |
| 25 | “remove(3) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man3/remove.3.html | 1399 |
| 26 | “rename(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/rename.2.html | 1402 |
| 27 | “ioctl(2) - Linux manual page”, <i>man7.org</i> . URL: https://man7.org/linux/man-pages/man2/ioctl.2.html | 1410 |
| 28 | Nudelman, Mark. “less(1) - Linux manual page”, <i>man7.org</i> , 20 Jul. 2023. URL: https://man7.org/linux/man-pages/man1/less.1.html | 1414 |
| 29 | Briskman, Miriam. “Materials for Topic 12: File and Directory Management.” <i>Topic 12: File and Directory Management — CISC 3350 Materials</i> , 2023. URL: www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_12/ | 1417 |
| 30 | Love, Robert. “get_file_size.c .” (C source code), <i>Linux System Programming: Talking Directly to the Kernel and C Library</i> , O’Reilly Media, Inc., Sebastopol, CA, 2013, p. 244. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_12/get_file_size.c | 1422 |
| 31 | Love, Robert. “find_file_type.c .” (C source code), <i>Linux System Programming: Talking Directly to the Kernel and C Library</i> , O’Reilly Media, Inc., Sebastopol, CA, 2013, pp. 244-245. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_12/find_file_type.c | 1424 |

| # | Citation & Source Link | Page |
|----|---|------|
| 32 | Briskman, Miriam. “permissions_changing.c .” (C source code) 8 May 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_12/permissions_changing.c | 1426 |
| 33 | Briskman, Miriam. “extended_attributes.c .” (C source code) 8 May 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_12/extended_attributes.c | 1431 |
| 34 | Love, Robert. “switch_directories.c .” (C source code), <i>Linux System Programming: Talking Directly to the Kernel and C Library</i> , O’Reilly Media, Inc., Sebastopol, CA, 2013, p. 265. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_12/switch_directories.c | 1434 |
| 35 | Love, Robert. “my_ls.c .” (C source code), <i>Linux System Programming: Talking Directly to the Kernel and C Library</i> , O’Reilly Media, Inc., Sebastopol, CA, 2013, p. 270. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_12/my_ls.c | 1437 |
| 36 | Briskman, Miriam. “adding_links.c .” (C source code) 10 May 2023. Created for CISC 3350, Brooklyn College. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_12/adding_links.c | 1439 |
| 37 | Love, Robert. “eject_cd_rom.c .” (C source code), <i>Linux System Programming: Talking Directly to the Kernel and C Library</i> , O’Reilly Media, Inc., Sebastopol, CA, 2013, pp. 282-283. URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/lecture_notes/topic_12/eject_cd_rom.c | 1443 |

inode(7) — Linux manual page

[NAME](#) | [DESCRIPTION](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

inode(7)

Miscellaneous Information Manual

inode(7)

NAME [top](#)

inode - file inode information

DESCRIPTION [top](#)

Each file has an inode containing metadata about the file. An application can retrieve this metadata using [stat\(2\)](#) (or related calls), which returns a *stat* structure, or [statx\(2\)](#), which returns a *statx* structure.

The following is a list of the information typically found in, or associated with, the file inode, with the names of the corresponding structure fields returned by [stat\(2\)](#) and [statx\(2\)](#):

Device where inode resides

stat.st_dev; *statx.stx_dev_minor* and *statx.stx_dev_major*

Each inode (as well as the associated file) resides in a filesystem that is hosted on a device. That device is identified by the combination of its major ID (which identifies the general class of device) and minor ID (which identifies a specific instance in the general class).

Inode number

stat.st_ino; *statx.stx_ino*

Each file in a filesystem has a unique inode number.

Inode numbers are guaranteed to be unique only within a filesystem (i.e., the same inode numbers may be used by different filesystems, which is the reason that hard links may not cross filesystem boundaries). This field contains the file's inode number.

File type and mode

stat.st_mode; statx.stx_mode

See the discussion of file type and mode, below.

Link count

stat.st_nlink; statx.stx_nlink

This field contains the number of hard links to the file. Additional links to an existing file are created using [link\(2\)](#).

User ID

st_uid stat.st_uid; statx.stx_uid

This field records the user ID of the owner of the file. For newly created files, the file user ID is the effective user ID of the creating process. The user ID of a file can be changed using [chown\(2\)](#).

Group ID

stat.st_gid; statx.stx_gid

The inode records the ID of the group owner of the file. For newly created files, the file group ID is either the group ID of the parent directory or the effective group ID of the creating process, depending on whether or not the set-group-ID bit is set on the parent directory (see below). The group ID of a file can be changed using [chown\(2\)](#).

Device represented by this inode

stat.st_rdev; statx.stx_rdev_minor and
statx.stx_rdev_major

If this file (inode) represents a device, then the inode records the major and minor ID of that device.



File size

stat.st_size; *statx.stx_size*

This field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symbolic link is the length of the pathname it contains, without a terminating null byte.

Preferred block size for I/O

stat.st_blksize; *statx.stx_blksize*

This field gives the "preferred" blocksize for efficient filesystem I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

Number of blocks allocated to the file

stat.st_blocks; *statx.stx_size*

This field indicates the number of blocks allocated to the file, 512-byte units, (This may be smaller than *st_size*/512 when the file has holes.)

The POSIX.1 standard notes that the unit for the *st_blocks* member of the *stat* structure is not defined by the standard. On many implementations it is 512 bytes; on a few systems, a different unit is used, such as 1024. Furthermore, the unit may differ on a per-filesystem basis.

Last access timestamp (atime)

stat.st_atime; *statx.stx_atime*

This is the file's last access timestamp. It is changed by file accesses, for example, by `execve(2)`, `mknod(2)`, `pipe(2)`, `utime(2)`, and `read(2)` (of more than zero bytes). Other interfaces, such as `mmap(2)`, may or may not update the atime timestamp

Some filesystem types allow mounting in such a way that file and/or directory accesses do not cause an update of the atime timestamp. (See *noatime*, *nodiratime*, and *relatime* in `mount(8)`, and related information in `mount(2)`.) In addition, the atime timestamp is not updated if a file is opened with the **O_NOATIME** flag; see

`open(2)`.

File creation (birth) timestamp (`btime`)

(not returned in the `stat` structure); `statx.stx_btime`

The file's creation timestamp. This is set on file creation and not changed subsequently.

The `btime` timestamp was not historically present on UNIX systems and is not currently supported by most Linux filesystems.

Last modification timestamp (`mtime`)

`stat.st_mtime`; `statx.stx_mtime`

This is the file's last modification timestamp. It is changed by file modifications, for example, by `mknod(2)`, `truncate(2)`, `utime(2)`, and `write(2)` (of more than zero bytes). Moreover, the `mtime` timestamp of a directory is changed by the creation or deletion of files in that directory. The `mtime` timestamp is *not* changed for changes in owner, group, hard link count, or mode.

Last status change timestamp (`ctime`)

`stat.st_ctime`; `statx.stx_ctime`

This is the file's last status change timestamp. It is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

The timestamp fields report time measured with a zero point at the *Epoch*, 1970-01-01 00:00:00 +0000, UTC (see `time(7)`).

Nanosecond timestamps are supported on XFS, JFS, Btrfs, and ext4 (since Linux 2.6.23). Nanosecond timestamps are not supported in ext2, ext3, and Reiserfs. In order to return timestamps with nanosecond precision, the timestamp fields in the `stat` and `statx` structures are defined as structures that include a nanosecond component. See `stat(2)` and `statx(2)` for details. On filesystems that do not support subsecond timestamps, the nanosecond fields in the `stat` and `statx` structures are returned with the value 0.

The file type and mode

The `stat.st_mode` field (for `statx(2)`, the `statx.stx_mode` field)

contains the file type and mode.

POSIX refers to the *stat.st_mode* bits corresponding to the mask **S_IFMT** (see below) as the *file type*, the 12 bits corresponding to the mask 07777 as the *file mode bits* and the least significant 9 bits (0777) as the *file permission bits*.

The following mask values are defined for the file type:

| | | |
|-----------------|---------|--------------------------------------|
| S_IFMT | 0170000 | bit mask for the file type bit field |
| S_IFSOCK | 0140000 | socket |
| S_IFLNK | 0120000 | symbolic link |
| S_IFREG | 0100000 | regular file |
| S_IFBLK | 0060000 | block device |
| S_IFDIR | 0040000 | directory |
| S_IFCHR | 0020000 | character device |
| S_IFIFO | 0010000 | FIFO |

Thus, to test for a regular file (for example), one could write:

```
stat(pathname, &sb);
if ((sb.st_mode & S_IFMT) == S_IFREG) {
    /* Handle regular file */
}
```

Because tests of the above form are common, additional macros are defined by POSIX to allow the test of the file type in *st_mode* to be written more concisely:

S_ISREG(m)
is it a regular file?

S_ISDIR(m)
directory?

S_ISCHR(m)
character device?

S_ISBLK(m)
block device?

S_ISFIFO(m)
FIFO (named pipe)?



S_ISLNK(m)
symbolic link? (Not in POSIX.1-1996.)

S_ISSOCK(m)
socket? (Not in POSIX.1-1996.)

The preceding code snippet could thus be rewritten as:

```
stat(pathname, &sb);
if (S_ISREG(sb.st_mode)) {
    /* Handle regular file */
}
```

The definitions of most of the above file type test macros are provided if any of the following feature test macros is defined: **_BSD_SOURCE** (in glibc 2.19 and earlier), **_SVID_SOURCE** (in glibc 2.19 and earlier), or **_DEFAULT_SOURCE** (in glibc 2.20 and later). In addition, definitions of all of the above macros except **S_IFSOCK** and **S_ISSOCK()** are provided if **_XOPEN_SOURCE** is defined.

The definition of **S_IFSOCK** can also be exposed either by defining **_XOPEN_SOURCE** with a value of 500 or greater or (since glibc 2.24) by defining both **_XOPEN_SOURCE** and **_XOPEN_SOURCE_EXTENDED**.

The definition of **S_ISSOCK()** is exposed if any of the following feature test macros is defined: **_BSD_SOURCE** (in glibc 2.19 and earlier), **_DEFAULT_SOURCE** (in glibc 2.20 and later), **_XOPEN_SOURCE** with a value of 500 or greater, **_POSIX_C_SOURCE** with a value of 200112L or greater, or (since glibc 2.24) by defining both **_XOPEN_SOURCE** and **_XOPEN_SOURCE_EXTENDED**.

The following mask values are defined for the file mode component of the *st_mode* field:

| | | |
|----------------|-------|--|
| S_ISUID | 04000 | set-user-ID bit (see execve(2)) |
| S_ISGID | 02000 | set-group-ID bit (see below) |
| S_ISVTX | 01000 | sticky bit (see below) |
| S_IRWXU | 00700 | owner has read, write, and execute permission |
| S_IRUSR | 00400 | owner has read permission |
| S_IWUSR | 00200 | owner has write permission |
| S_IXUSR | 00100 | owner has execute permission |
| S_IRWXG | 00070 | group has read, write, and execute |



| | | |
|----------------|-------|--|
| | | permission |
| S_IRGRP | 00040 | group has read permission |
| S_IWGRP | 00020 | group has write permission |
| S_IXGRP | 00010 | group has execute permission |
| S_IRWXO | 00007 | others (not in group) have read, write, and execute permission |
| S_IROTH | 00004 | others have read permission |
| S_IWOTH | 00002 | others have write permission |
| S_IXOTH | 00001 | others have execute permission |

The set-group-ID bit (**S_ISGID**) has several special uses. For a directory, it indicates that BSD semantics are to be used for that directory: files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the **S_ISGID** bit set. For an executable file, the set-group-ID bit causes the effective group ID of a process that executes the file to change as described in [execve\(2\)](#). For a file that does not have the group execution bit (**S_IXGRP**) set, the set-group-ID bit indicates mandatory file/record locking.

The sticky bit (**S_ISVTX**) on a directory means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, and by a privileged process.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

POSIX.1-1990 did not describe the **S_IFMT**, **S_IFSOCK**, **S_IFLNK**, **S_IFREG**, **S_IFBLK**, **S_IFDIR**, **S_IFCHR**, **S_IFIFO**, and **S_ISVTX** constants, but instead specified the use of the macros **S_ISDIR()** and so on.

The **S_ISLNK()** and **S_ISSOCK()** macros were not in POSIX.1-1996; the former is from SVID 4, the latter from SUSv2.



UNIX V7 (and later systems) had **S_IREAD**, **S_IWRITE**, **S_IEXEC**, and where POSIX prescribes the synonyms **S_IRUSR**, **S_IWUSR**, and **S_IXUSR**.

NOTES [top](#)

For pseudofiles that are autogenerated by the kernel, the file size (`stat.st_size`; `statx.stx_size`) reported by the kernel is not accurate. For example, the value 0 is returned for many files under the `/proc` directory, while various files under `/sys` report a size of 4096 bytes, even though the file content is smaller. For such files, one should simply try to read as many bytes as possible (and append `'\0'` to the returned buffer if it is to be interpreted as a string).

SEE ALSO [top](#)

[stat\(1\)](#), [stat\(2\)](#), [statx\(2\)](#), [symlink\(7\)](#)

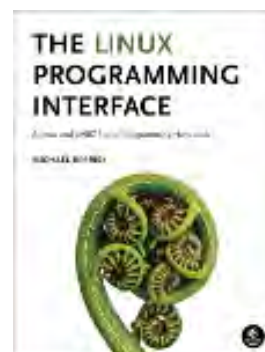
Linux man-pages (unreleased) (date) [inode\(7\)](#)

Pages that refer to this page: [chmod\(2\)](#), [fsync\(2\)](#), [getdents\(2\)](#), [mkdir\(2\)](#), [mknod\(2\)](#), [open\(2\)](#), [stat\(2\)](#), [statx\(2\)](#), [truncate\(2\)](#), [umask\(2\)](#), [utime\(2\)](#), [utimensat\(2\)](#), [stat\(3type\)](#), [systemd.exec\(5\)](#), [symlink\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



stat(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [EXAMPLES](#) | [SEE ALSO](#)

 stat(2)

System Calls Manual

stat(2)**NAME** [top](#)

stat, fstat, lstat, fstatat - get file status

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/stat.h>
```

```
int stat(const char *restrict pathname,
         struct stat *restrict statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *restrict pathname,
         struct stat *restrict statbuf);
```

```
#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/stat.h>
```

```
int fstatat(int dirfd, const char *restrict pathname,
            struct stat *restrict statbuf, int flags);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
lstat():
    /* Since glibc 2.20 */ _DEFAULT_SOURCE
    || _XOPEN_SOURCE >= 500
    || /* Since glibc 2.10: */ _POSIX_C_SOURCE >= 200112L
```

```
|| /* glibc 2.19 and earlier */ _BSD_SOURCE
```

fstatat():

Since glibc 2.10:

```
_POSIX_C_SOURCE >= 200809L
```

Before glibc 2.10:

```
_ATFILE_SOURCE
```

DESCRIPTION [top](#)

These functions return information about a file, in the buffer pointed to by *statbuf*. No permissions are required on the file itself, but—in the case of **stat()**, **fstatat()**, and **lstat()**—execute (search) permission is required on all of the directories in *pathname* that lead to the file.

stat() and **fstatat()** retrieve information about the file pointed to by *pathname*; the differences for **fstatat()** are described below.

lstat() is identical to **stat()**, except that if *pathname* is a symbolic link, then it returns information about the link itself, not the file that the link refers to.

fstat() is identical to **stat()**, except that the file about which information is to be retrieved is specified by the file descriptor *fd*.

The stat structure

All of these system calls return a *stat* structure (see [stat\(3type\)](#)).

Note: for performance and simplicity reasons, different fields in the *stat* structure may contain state information from different moments during the execution of the system call. For example, if *st_mode* or *st_uid* is changed by another process by calling [chmod\(2\)](#) or [chown\(2\)](#), **stat()** might return the old *st_mode* together with the new *st_uid*, or the old *st_uid* together with the new *st_mode*.

fstatat()

The **fstatat()** system call is a more general interface for accessing file information which can still provide exactly the behavior of each of **stat()**, **lstat()**, and **fstat()**.

If the *pathname* given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file



descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **stat()** and **lstat()** for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like **stat()** and **lstat()**).

If *pathname* is absolute, then *dirfd* is ignored.

flags can either be 0, or include one or more of the following flags ORed:

AT_EMPTY_PATH (since Linux 2.6.39)

If *pathname* is an empty string, operate on the file referred to by *dirfd* (which may have been obtained using the **open(2)** **O_PATH** flag). In this case, *dirfd* can refer to any type of file, not just a directory, and the behavior of **fstatat()** is similar to that of **fstat()**. If *dirfd* is **AT_FDCWD**, the call operates on the current working directory. This flag is Linux-specific; define **_GNU_SOURCE** to obtain its definition.

AT_NO_AUTOMOUNT (since Linux 2.6.38)

Don't automount the terminal ("basename") component of *pathname*. Since Linux 3.1 this flag is ignored. Since Linux 4.11 this flag is implied.

AT_SYMLINK_NOFOLLOW

If *pathname* is a symbolic link, do not dereference it: instead return information about the link itself, like **lstat()**. (By default, **fstatat()** dereferences symbolic links, like **stat()**.)

See **openat(2)** for an explanation of the need for **fstatat()**.

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EACCES Search permission is denied for one of the directories in the path prefix of *pathname*. (See also **path_resolution(7)**.)



EBADF *fd* is not a valid open file descriptor.

EBADF (`fstatat()`) *pathname* is relative but *dirfd* is neither `AT_FDCWD` nor a valid file descriptor.

EFAULT Bad address.

EINVAL (`fstatat()`) Invalid flag specified in *flags*.

ELOOP Too many symbolic links encountered while traversing the path.

ENAMETOOLONG
pathname is too long.

ENOENT A component of *pathname* does not exist or is a dangling symbolic link.

ENOENT *pathname* is an empty string and `AT_EMPTY_PATH` was not specified in *flags*.

ENOMEM Out of memory (i.e., kernel memory).

ENOTDIR
A component of the path prefix of *pathname* is not a directory.

ENOTDIR
(`fstatat()`) *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

EOVERFLOW
pathname or *fd* refers to a file whose size, inode number, or number of blocks cannot be represented in, respectively, the types *off_t*, *ino_t*, or *blkcnt_t*. This error can occur when, for example, an application compiled on a 32-bit platform without `-D_FILE_OFFSET_BITS=64` calls `stat()` on a file whose size exceeds $(1 \ll 31) - 1$ bytes.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

stat()
fstat()
lstat()

SVr4, 4.3BSD, POSIX.1-2001.

fstatat()

POSIX.1-2008. Linux 2.6.16, glibc 2.4.

According to POSIX.1-2001, **lstat()** on a symbolic link need return valid information only in the *st_size* field and the file type of the *st_mode* field of the *stat* structure. POSIX.1-2008 tightens the specification, requiring **lstat()** to return valid information in all fields except the mode bits in *st_mode*.

Use of the *st_blocks* and *st_blksize* fields may be less portable. (They were introduced in BSD. The interpretation differs between systems, and possibly on a single system when NFS mounts are involved.)

C library/kernel differences

Over time, increases in the size of the *stat* structure have led to three successive versions of **stat()**: *sys_stat()* (slot *__NR_oldstat*), *sys_newstat()* (slot *__NR_stat*), and *sys_stat64()* (slot *__NR_stat64*) on 32-bit platforms such as i386. The first two versions were already present in Linux 1.0 (albeit with different names); the last was added in Linux 2.4. Similar remarks apply for **fstat()** and **lstat()**.

The kernel-internal versions of the *stat* structure dealt with by the different versions are, respectively:

__old_kernel_stat

The original structure, with rather narrow fields, and no padding.

stat Larger *st_ino* field and padding added to various parts of the structure to allow for future expansion.

stat64 Even larger *st_ino* field, larger *st_uid* and *st_gid* fields to accommodate the Linux-2.4 expansion of UIDs and GIDs to 32 bits, and various other enlarged fields and further padding in the structure. (Various padding bytes were eventually consumed in Linux 2.6, with the advent of 32-bit device IDs and nanosecond components for the timestamp fields.)

The glibc **stat()** wrapper function hides these details from applications, invoking the most recent version of the system call



provided by the kernel, and repacking the returned information if required for old binaries.

On modern 64-bit systems, life is simpler: there is a single **stat()** system call and the kernel deals with a *stat* structure that contains fields of a sufficient size.

The underlying system call employed by the glibc **fstatat()** wrapper function is actually called **fstatat64()** or, on some architectures, **newfstatat()**.

EXAMPLES [top](#)

The following program calls **lstat()** and displays selected fields in the returned *stat* structure.

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/sysmacros.h>
#include <time.h>

int
main(int argc, char *argv[])
{
    struct stat sb;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (lstat(argv[1], &sb) == -1) {
        perror("lstat");
        exit(EXIT_FAILURE);
    }

    printf("ID of containing device:  [%x,%x]\n",
           major(sb.st_dev),
           minor(sb.st_dev));

    printf("File type:                ");

    switch (sb.st_mode & S_IFMT) {
        case S_IFBLK:  printf("block device\n");           break;
        case S_IFCHR:  printf("character device\n");        break;
    }
}
```



```

case S_IFDIR:  printf("directory\n");          break;
case S_IFIFO:  printf("FIFO/pipe\n");          break;
case S_IFLNK:  printf("symlink\n");           break;
case S_IFREG:  printf("regular file\n");       break;
case S_IFSOCK: printf("socket\n");             break;
default:       printf("unknown?\n");           break;
}

printf("I-node number:          %ju\n", (uintmax_t) sb.st_ino);

printf("Mode:                   %jo (octal)\n",
       (uintmax_t) sb.st_mode);

printf("Link count:             %ju\n", (uintmax_t) sb.st_nlink);
printf("Ownership:              UID=%ju   GID=%ju\n",
       (uintmax_t) sb.st_uid, (uintmax_t) sb.st_gid);

printf("Preferred I/O block size: %jd bytes\n",
       (intmax_t) sb.st_blksize);
printf("File size:               %jd bytes\n",
       (intmax_t) sb.st_size);
printf("Blocks allocated:        %jd\n",
       (intmax_t) sb.st_blocks);

printf("Last status change:      %s", ctime(&sb.st_ctime));
printf("Last file access:        %s", ctime(&sb.st_atime));
printf("Last file modification:  %s", ctime(&sb.st_mtime));

exit(EXIT_SUCCESS);
}

```

SEE ALSO [top](#)

[ls\(1\)](#), [stat\(1\)](#), [access\(2\)](#), [chmod\(2\)](#), [chown\(2\)](#), [readlink\(2\)](#), [statx\(2\)](#), [utime\(2\)](#), [stat\(3type\)](#), [capabilities\(7\)](#), [inode\(7\)](#), [symlink\(7\)](#)

Linux man-pages (unreleased) **(date)** **[stat\(2\)](#)**

Pages that refer to this page: [bash\(1\)](#), [find\(1\)](#), [fuser\(1\)](#), [git-update-index\(1\)](#), [lsfd\(1\)](#), [pv\(1\)](#), [rsync\(1\)](#), [stat\(1\)](#), [strace\(1\)](#), [systemd-analyze\(1\)](#), [access\(2\)](#), [chmod\(2\)](#), [fallocate\(2\)](#), [fanotify_init\(2\)](#), [futimesat\(2\)](#), [getxattr\(2\)](#), [ioctl_ns\(2\)](#), [link\(2\)](#), [listxattr\(2\)](#), [mkdir\(2\)](#), [mknod\(2\)](#), [mount\(2\)](#), [open\(2\)](#), [pivot_root\(2\)](#), [readlink\(2\)](#), [removexattr\(2\)](#), [setxattr\(2\)](#), [spu_create\(2\)](#), [statfs\(2\)](#), [statx\(2\)](#), [symlink\(2\)](#), [syscalls\(2\)](#), [truncate\(2\)](#), [umask\(2\)](#), [ustat\(2\)](#), [utime\(2\)](#), [utimensat\(2\)](#), [dirfd\(3\)](#), [eidaccess\(3\)](#), [fseek\(3\)](#), [ftok\(3\)](#), [fts\(3\)](#), [ftw\(3\)](#), [getfilecon\(3\)](#), [getseuserbyname\(3\)](#), [glob\(3\)](#), [isatty\(3\)](#), [isfdtype\(3\)](#), [makedev\(3\)](#), [mkfifo\(3\)](#), [readdir\(3\)](#), [readline\(3\)](#), [sd_is_fifo\(3\)](#), [selabel_lookup_best_match\(3\)](#), [setfilecon\(3\)](#), [shm_open\(3\)](#),

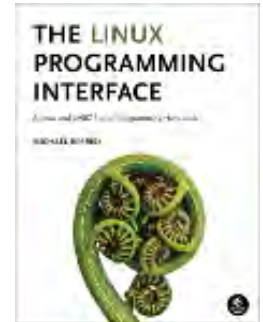


[stat\(3type\)](#), [ttyname\(3\)](#), [fuse\(4\)](#), [nfs\(5\)](#), [proc\(5\)](#), [selabel_file\(5\)](#), [sysfs\(5\)](#), [inode\(7\)](#), [inotify\(7\)](#), [landlock\(7\)](#), [namespaces\(7\)](#), [path_resolution\(7\)](#), [pipe\(7\)](#), [shm_overview\(7\)](#), [signal-safety\(7\)](#), [spufs\(7\)](#), [symlink\(7\)](#), [time\(7\)](#), [user_namespaces\(7\)](#), [xattr\(7\)](#), [lsof\(8\)](#), [umount\(8\)](#), [xfs_db\(8\)](#), [xfs_io\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



stat(3type) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#)
| [SEE ALSO](#)

stat(3type)

stat(3type)

NAME [top](#)

stat - file status

LIBRARY [top](#)

Standard C library (*libc*)

SYNOPSIS [top](#)

```
#include <sys/stat.h>
```

```
struct stat {
    dev_t      st_dev;      /* ID of device containing file */
    ino_t      st_ino;     /* Inode number */
    mode_t     st_mode;    /* File type and mode */
    nlink_t    st_nlink;   /* Number of hard links */
    uid_t      st_uid;     /* User ID of owner */
    gid_t      st_gid;     /* Group ID of owner */
    dev_t      st_rdev;    /* Device ID (if special file) */
    off_t      st_size;    /* Total size, in bytes */
    blksize_t  st_blksize; /* Block size for filesystem I/O */
    blkcnt_t   st_blocks;  /* Number of 512 B blocks allocated */
};
```

```
/* Since POSIX.1-2008, this structure supports nanosecond
   precision for the following timestamp fields.
   For the details before POSIX.1-2008, see VERSIONS. */
```

```

struct timespec  st_atim;  /* Time of last access */
struct timespec  st_mtim;  /* Time of last modification */
struct timespec  st_ctim;  /* Time of last status change */

#define st_atime  st_atim.tv_sec  /* Backward compatibility */
#define st_mtime  st_mtim.tv_sec
#define st_ctime  st_ctim.tv_sec
};

```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```

st_atim, st_mtim, st_ctim:
  Since glibc 2.12:
    _POSIX_C_SOURCE >= 200809L || _XOPEN_SOURCE >= 700
  glibc 2.19 and earlier:
    _BSD_SOURCE || _SVID_SOURCE

```

DESCRIPTION [top](#)

Describes information about a file.

The fields are as follows:

st_dev This field describes the device on which this file resides. (The [major\(3\)](#) and [minor\(3\)](#) macros may be useful to decompose the device ID in this field.)

st_ino This field contains the file's inode number.

st_mode
This field contains the file type and mode. See [inode\(7\)](#) for further information.

st_nlink
This field contains the number of hard links to the file.

st_uid This field contains the user ID of the owner of the file.

st_gid This field contains the ID of the group owner of the file.

st_rdev
This field describes the device that this file (inode) represents.

st_size

This field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symbolic link is the length of the pathname it contains, without a terminating null byte.

st_blksize

This field gives the "preferred" block size for efficient filesystem I/O.

st_blocks

This field indicates the number of blocks allocated to the file, in 512-byte units. (This may be smaller than *st_size*/512 when the file has holes.)

st_atime

This is the time of the last access of file data.

st_mtime

This is the time of last modification of file data.

st_ctime

This is the file's last status change timestamp (time of last change to the inode).

For further information on the above fields, see [inode\(7\)](#).

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001.

Old kernels and old standards did not support nanosecond timestamp fields. Instead, there were three timestamp fields—*st_atime*, *st_mtime*, and *st_ctime*—typed as *time_t* that recorded timestamps with one-second precision.

Since Linux 2.5.48, the *stat* structure supports nanosecond resolution for the three file timestamp fields. The nanosecond

components of each timestamp are available via names of the form `st_atim.tv_nsec`, if suitable test macros are defined. Nanosecond timestamps were standardized in POSIX.1-2008, and, starting with glibc 2.12, glibc exposes the nanosecond component names if `_POSIX_C_SOURCE` is defined with the value 200809L or greater, or `_XOPEN_SOURCE` is defined with the value 700 or greater. Up to and including glibc 2.19, the definitions of the nanoseconds components are also defined if `_BSD_SOURCE` or `_SVID_SOURCE` is defined. If none of the aforementioned macros are defined, then the nanosecond values are exposed with names of the form `st_atimensec`.

NOTES [top](#)

The following header also provides this type: `<ftw.h>`.

SEE ALSO [top](#)

[stat\(2\)](#), [inode\(7\)](#)

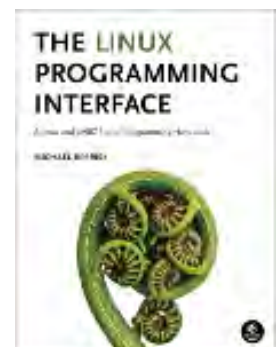
Linux man-pages (unreleased) (date) [stat\(3type\)](#)

Pages that refer to this page: [stat\(2\)](#), [blkcnt_t\(3type\)](#), [blksize_t\(3type\)](#), [dev_t\(3type\)](#), [mode_t\(3type\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



chmod(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 [chmod\(2\)](#)

System Calls Manual

[chmod\(2\)](#)

NAME [top](#)

chmod, fchmod, fchmodat - change permissions of a file

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/stat.h>
```

```
int chmod(const char *pathname, mode_t mode);  
int fchmod(int fd, mode_t mode);
```

```
#include <fcntl.h> /* Definition of AT_* constants */  
#include <sys/stat.h>
```

```
int fchmodat(int dirfd, const char *pathname, mode_t mode, int flags);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
fchmod():  
    Since glibc 2.24:  
        _POSIX_C_SOURCE >= 199309L  
glibc 2.19 to glibc 2.23  
    _POSIX_C_SOURCE  
glibc 2.16 to glibc 2.19:  
    _BSD_SOURCE || _POSIX_C_SOURCE
```

```
glibc 2.12 to glibc 2.16:
    _BSD_SOURCE || _XOPEN_SOURCE >= 500
    || _POSIX_C_SOURCE >= 200809L
glibc 2.11 and earlier:
    _BSD_SOURCE || _XOPEN_SOURCE >= 500
```

fchmodat():

```
Since glibc 2.10:
    _POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
    _ATFILE_SOURCE
```

DESCRIPTION [top](#)

The **chmod()** and **fchmod()** system calls change a file's mode bits. (The file mode consists of the file permission bits plus the set-user-ID, set-group-ID, and sticky bits.) These system calls differ only in how the file is specified:

- **chmod()** changes the mode of the file specified whose pathname is given in *pathname*, which is dereferenced if it is a symbolic link.
- **fchmod()** changes the mode of the file referred to by the open file descriptor *fd*.

The new file mode is specified in *mode*, which is a bit mask created by ORing together zero or more of the following:

S_ISUID (04000)
set-user-ID (set process effective user ID on [execve\(2\)](#))

S_ISGID (02000)
set-group-ID (set process effective group ID on [execve\(2\)](#)); mandatory locking, as described in [fcntl\(2\)](#); take a new file's group from parent directory, as described in [chown\(2\)](#) and [mkdir\(2\)](#))

S_ISVTX (01000)
sticky bit (restricted deletion flag, as described in [unlink\(2\)](#))

S_IRUSR (00400)
read by owner

S_IWUSR (00200)



write by owner

S_IXUSR (00100)

execute/search by owner ("search" applies for directories, and means that entries within the directory can be accessed)

S_IRGRP (00040)

read by group

S_IWGRP (00020)

write by group

S_IXGRP (00010)

execute/search by group

S_IROTH (00004)

read by others

S_IWOTH (00002)

write by others

S_IXOTH (00001)

execute/search by others

The effective UID of the calling process must match the owner of the file, or the process must be privileged (Linux: it must have the **CAP_FOWNER** capability).

If the calling process is not privileged (Linux: does not have the **CAP_FSETID** capability), and the group of the file does not match the effective group ID of the process or one of its supplementary group IDs, the **S_ISGID** bit will be turned off, but this will not cause an error to be returned.

As a security measure, depending on the filesystem, the set-user-ID and set-group-ID execution bits may be turned off if a file is written. (On Linux, this occurs if the writing process does not have the **CAP_FSETID** capability.) On some filesystems, only the superuser can set the sticky bit, which may have a special meaning. For the sticky bit, and for set-user-ID and set-group-ID bits on directories, see [inode\(7\)](#).

On NFS filesystems, restricting the permissions will immediately influence already open files, because the access control is done on the server, but open files are maintained by the client. Widening the permissions may be delayed for other clients if

attribute caching is enabled on them.

fchmodat()

The **fchmodat()** system call operates in exactly the same way as **chmod()**, except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **chmod()** for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like **chmod()**).

If *pathname* is absolute, then *dirfd* is ignored.

flags can either be 0, or include the following flag:

AT_SYMLINK_NOFOLLOW

If *pathname* is a symbolic link, do not dereference it: instead operate on the link itself. This flag is not currently implemented.

See [openat\(2\)](#) for an explanation of the need for **fchmodat()**.

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

Depending on the filesystem, errors other than those listed below can be returned.

The more general errors for **chmod()** are listed below:

EACCES Search permission is denied on a component of the path prefix. (See also [path_resolution\(7\)](#).)

EBADF (**fchmod()**) The file descriptor *fd* is not valid.

EBADF (**fchmodat()**) *pathname* is relative but *dirfd* is neither



AT_FDCWD nor a valid file descriptor.

EFAULT *pathname* points outside your accessible address space.

EINVAL (**fchmodat()**) Invalid flag specified in *flags*.

EIO An I/O error occurred.

ELOOP Too many symbolic links were encountered in resolving *pathname*.

ENAMETOOLONG
pathname is too long.

ENOENT The file does not exist.

ENOMEM Insufficient kernel memory was available.

ENOTDIR
A component of the path prefix is not a directory.

ENOTDIR
(**fchmodat()**) *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

ENOTSUP
(**fchmodat()**) *flags* specified **AT_SYMLINK_NOFOLLOW**, which is not supported.

EPERM The effective UID does not match the owner of the file, and the process is not privileged (Linux: it does not have the **CAP_FOWNER** capability).

EPERM The file is marked immutable or append-only. (See [ioctl_iflags\(2\)](#).)

EROFS The named file resides on a read-only filesystem.

VERSIONS [top](#)

C library/kernel differences

The GNU C library **fchmodat()** wrapper function implements the POSIX-specified interface described in this page. This interface differs from the underlying Linux system call, which does *not* have a *flags* argument.

glibc notes

On older kernels where `fchmodat()` is unavailable, the glibc wrapper function falls back to the use of `chmod()`. When *pathname* is a relative pathname, glibc constructs a pathname based on the symbolic link in `/proc/self/fd` that corresponds to the *dirfd* argument.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

`chmod()`
`fchmod()`
4.4BSD, SVr4, POSIX.1-2001.

`fchmodat()`
POSIX.1-2008. Linux 2.6.16, glibc 2.4.

SEE ALSO [top](#)

[chmod\(1\)](#), [chown\(2\)](#), [execve\(2\)](#), [open\(2\)](#), [stat\(2\)](#), [inode\(7\)](#),
[path_resolution\(7\)](#), [symlink\(7\)](#)

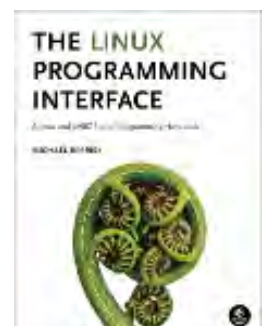
Linux man-pages (unreleased) (date) [chmod\(2\)](#)

Pages that refer to this page: [chmod\(1\)](#), [access\(2\)](#), [chown\(2\)](#), [execve\(2\)](#), [fcntl\(2\)](#), [mkdir\(2\)](#), [mknod\(2\)](#), [open\(2\)](#), [rename\(2\)](#), [rmdir\(2\)](#), [stat\(2\)](#), [statx\(2\)](#), [syscalls\(2\)](#), [umask\(2\)](#), [unlink\(2\)](#), [euidaccess\(3\)](#), [mode_t\(3type\)](#), [shm_open\(3\)](#), [capabilities\(7\)](#), [inotify\(7\)](#), [landlock\(7\)](#), [shm_overview\(7\)](#), [signal-safety\(7\)](#), [spufs\(7\)](#), [symlink\(7\)](#), [unix\(7\)](#), [logrotate\(8\)](#), [xfs_db\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



chown(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [EXAMPLES](#) | [SEE ALSO](#)

 chown(2)

System Calls Manual

chown(2)

NAME [top](#)

chown, fchown, lchown, fchownat - change ownership of a file

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *pathname, uid_t owner, gid_t group);
```

```
#include <fcntl.h>          /* Definition of AT_* constants */
#include <unistd.h>
```

```
int fchownat(int dirfd, const char *pathname,
             uid_t owner, gid_t group, int flags);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
fchown(), lchown():
    /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
    || _XOPEN_SOURCE >= 500
    || /* glibc <= 2.19: */ _BSD_SOURCE
```

fchownat():

Since glibc 2.10:

```
_POSIX_C_SOURCE >= 200809L
```

Before glibc 2.10:

```
_ATFILE_SOURCE
```

DESCRIPTION [top](#)

These system calls change the owner and group of a file. The **chown()**, **fchown()**, and **lchown()** system calls differ only in how the file is specified:

- **chown()** changes the ownership of the file specified by *pathname*, which is dereferenced if it is a symbolic link.
- **fchown()** changes the ownership of the file referred to by the open file descriptor *fd*.
- **lchown()** is like **chown()**, but does not dereference symbolic links.

Only a privileged process (Linux: one with the **CAP_CHOWN** capability) may change the owner of a file. The owner of a file may change the group of the file to any group of which that owner is a member. A privileged process (Linux: with **CAP_CHOWN**) may change the group arbitrarily.

If the *owner* or *group* is specified as -1, then that ID is not changed.

When the owner or group of an executable file is changed by an unprivileged user, the **S_ISUID** and **S_ISGID** mode bits are cleared. POSIX does not specify whether this also should happen when root does the **chown()**; the Linux behavior depends on the kernel version, and since Linux 2.2.13, root is treated like other users. In case of a non-group-executable file (i.e., one for which the **S_IXGRP** bit is not set) the **S_ISGID** bit indicates mandatory locking, and is not cleared by a **chown()**.

When the owner or group of an executable file is changed (by any user), all capability sets for the file are cleared.

fchownat()

The **fchownat()** system call operates in exactly the same way as **chown()**, except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **chown()** for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like **chown()**).

If *pathname* is absolute, then *dirfd* is ignored.

The *flags* argument is a bit mask created by ORing together 0 or more of the following values;

AT_EMPTY_PATH (since Linux 2.6.39)

If *pathname* is an empty string, operate on the file referred to by *dirfd* (which may have been obtained using the **open(2)** **O_PATH** flag). In this case, *dirfd* can refer to any type of file, not just a directory. If *dirfd* is **AT_FDCWD**, the call operates on the current working directory. This flag is Linux-specific; define **_GNU_SOURCE** to obtain its definition.

AT_SYMLINK_NOFOLLOW

If *pathname* is a symbolic link, do not dereference it: instead operate on the link itself, like **lchown()**. (By default, **fchownat()** dereferences symbolic links, like **chown()**.)

See **openat(2)** for an explanation of the need for **fchownat()**.

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

Depending on the filesystem, errors other than those listed below can be returned.

The more general errors for **chown()** are listed below.

EACCES Search permission is denied on a component of the path



prefix. (See also [path_resolution\(7\)](#).)

EBADF (`fchown()`) *fd* is not a valid open file descriptor.

EBADF (`fchownat()`) *pathname* is relative but *dirfd* is neither `AT_FDCWD` nor a valid file descriptor.

EFAULT *pathname* points outside your accessible address space.

EINVAL (`fchownat()`) Invalid flag specified in *flags*.

EIO (`fchown()`) A low-level I/O error occurred while modifying the inode.

ELOOP Too many symbolic links were encountered in resolving *pathname*.

ENAMETOOLONG
pathname is too long.

ENOENT The file does not exist.

ENOMEM Insufficient kernel memory was available.

ENOTDIR
A component of the path prefix is not a directory.

ENOTDIR
(`fchownat()`) *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

EPERM The calling process did not have the required permissions (see above) to change owner and/or group.

EPERM The file is marked immutable or append-only. (See [ioctl_iflags\(2\)](#).)

EROFS The named file resides on a read-only filesystem.

VERSIONS [top](#)

The 4.4BSD version can be used only by the superuser (that is, ordinary users cannot give away files).

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

chown()

fchown()

lchown()

4.4BSD, SVr4, POSIX.1-2001.

fchownat()

POSIX.1-2008. Linux 2.6.16, glibc 2.4.

NOTES [top](#)

Ownership of new files

When a new file is created (by, for example, [open\(2\)](#) or [mkdir\(2\)](#)), its owner is made the same as the filesystem user ID of the creating process. The group of the file depends on a range of factors, including the type of filesystem, the options used to mount the filesystem, and whether or not the set-group-ID mode bit is enabled on the parent directory. If the filesystem supports the **-o grpuid** (or, synonymously **-o bsdgroups**) and **-o nogrpuid** (or, synonymously **-o sysvgroups**) [mount\(8\)](#) options, then the rules are as follows:

- If the filesystem is mounted with **-o grpuid**, then the group of a new file is made the same as that of the parent directory.
- If the filesystem is mounted with **-o nogrpuid** and the set-group-ID bit is disabled on the parent directory, then the group of a new file is made the same as the process's filesystem GID.
- If the filesystem is mounted with **-o nogrpuid** and the set-group-ID bit is enabled on the parent directory, then the group of a new file is made the same as that of the parent directory.

As at Linux 4.12, the **-o grpuid** and **-o nogrpuid** mount options are supported by ext2, ext3, ext4, and XFS. Filesystems that don't support these mount options follow the **-o nogrpuid** rules.

glibc notes

On older kernels where **fchownat()** is unavailable, the glibc wrapper function falls back to the use of **chown()** and **lchown()**.

When *pathname* is a relative pathname, glibc constructs a pathname based on the symbolic link in */proc/self/fd* that corresponds to the *dirfd* argument.

NFS

The **chown()** semantics are deliberately violated on NFS filesystems which have UID mapping enabled. Additionally, the semantics of all system calls which access the file contents are violated, because **chown()** may cause immediate access revocation on already open files. Client side caching may lead to a delay between the time where ownership have been changed to allow access for a user and the time where the file can actually be accessed by the user on other clients.

Historical details

The original Linux **chown()**, **fchown()**, and **lchown()** system calls supported only 16-bit user and group IDs. Subsequently, Linux 2.4 added **chown32()**, **fchown32()**, and **lchown32()**, supporting 32-bit IDs. The glibc **chown()**, **fchown()**, and **lchown()** wrapper functions transparently deal with the variations across kernel versions.

Before Linux 2.1.81 (except 2.1.46), **chown()** did not follow symbolic links. Since Linux 2.1.81, **chown()** does follow symbolic links, and there is a new system call **lchown()** that does not follow symbolic links. Since Linux 2.1.86, this new call (that has the same semantics as the old **chown()**) has got the same syscall number, and **chown()** got the newly introduced number.

EXAMPLES [top](#)

The following program changes the ownership of the file named in its second command-line argument to the value specified in its first command-line argument. The new owner can be specified either as a numeric user ID, or as a username (which is converted to a user ID by using [getpwnam\(3\)](#) to perform a lookup in the system password file).

Program source

```
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
```



```
{
    char          *endptr;
    uid_t         uid;
    struct passwd *pwd;

    if (argc != 3 || argv[1][0] == '\0') {
        fprintf(stderr, "%s <owner> <file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    uid = strtol(argv[1], &endptr, 10); /* Allow a numeric string */

    if (*endptr != '\0') {             /* Was not pure numeric string */
        pwd = getpwnam(argv[1]);       /* Try getting UID for username */
        if (pwd == NULL) {
            perror("getpwnam");
            exit(EXIT_FAILURE);
        }

        uid = pwd->pw_uid;
    }

    if (chown(argv[2], uid, -1) == -1) {
        perror("chown");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

SEE ALSO [top](#)

[chgrp\(1\)](#), [chown\(1\)](#), [chmod\(2\)](#), [flock\(2\)](#), [path_resolution\(7\)](#),
[symlink\(7\)](#)

Linux man-pages (unreleased) (date) [chown\(2\)](#)

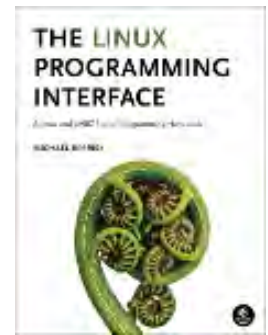
Pages that refer to this page: [chgrp\(1\)](#), [chown\(1\)](#), [access\(2\)](#), [chmod\(2\)](#), [fcntl\(2\)](#), [mkdir\(2\)](#),
[mknod\(2\)](#), [mount_setattr\(2\)](#), [open\(2\)](#), [open_by_handle_at\(2\)](#), [stat\(2\)](#), [statx\(2\)](#),
[symlink\(2\)](#), [syscalls\(2\)](#), [euidaccess\(3\)](#), [fpathconf\(3\)](#), [id_t\(3type\)](#), [shm_open\(3\)](#),
[systemd.exec\(5\)](#), [capabilities\(7\)](#), [inode\(7\)](#), [inotify\(7\)](#), [landlock\(7\)](#), [shm_overview\(7\)](#),
[signal-safety\(7\)](#), [spufs\(7\)](#), [symlink\(7\)](#), [unix\(7\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



getxattr(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [EXAMPLES](#) | [SEE ALSO](#)

 getxattr(2)

System Calls Manual

getxattr(2)

NAME [top](#)

getxattr, lgetxattr, fgetxattr - retrieve an extended attribute value

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/xattr.h>

ssize_t getxattr(const char *path, const char *name,
                void value[.size], size_t size);
ssize_t lgetxattr(const char *path, const char *name,
                  void value[.size], size_t size);
ssize_t fgetxattr(int fd, const char *name,
                  void value[.size], size_t size);
```

DESCRIPTION [top](#)

Extended attributes are *name:value* pairs associated with inodes (files, directories, symbolic links, etc.). They are extensions to the normal attributes which are associated with all inodes in the system (i.e., the [stat\(2\)](#) data). A complete overview of

extended attributes concepts can be found in [xattr\(7\)](#).

getxattr() retrieves the value of the extended attribute identified by *name* and associated with the given *path* in the filesystem. The attribute value is placed in the buffer pointed to by *value*; *size* specifies the size of that buffer. The return value of the call is the number of bytes placed in *value*.

lgetxattr() is identical to **getxattr()**, except in the case of a symbolic link, where the link itself is interrogated, not the file that it refers to.

fgetxattr() is identical to **getxattr()**, only the open file referred to by *fd* (as returned by [open\(2\)](#)) is interrogated in place of *path*.

An extended attribute *name* is a null-terminated string. The name includes a namespace prefix; there may be several, disjoint namespaces associated with an individual inode. The value of an extended attribute is a chunk of arbitrary textual or binary data that was assigned using [setxattr\(2\)](#).

If *size* is specified as zero, these calls return the current size of the named extended attribute (and leave *value* unchanged). This can be used to determine the size of the buffer that should be supplied in a subsequent call. (But, bear in mind that there is a possibility that the attribute value may change between the two calls, so that it is still necessary to check the return status from the second call.)

RETURN VALUE [top](#)

On success, these calls return a nonnegative value which is the size (in bytes) of the extended attribute value. On failure, -1 is returned and *errno* is set to indicate the error.

ERRORS [top](#)

E2BIG The size of the attribute value is larger than the maximum size allowed; the attribute cannot be retrieved. This can happen on filesystems that support very large attribute values such as NFSv4, for example.

ENODATA

The named attribute does not exist, or the process has no access to this attribute.

ENOTSUP

Extended attributes are not supported by the filesystem, or are disabled.

ERANGE The *size* of the *value* buffer is too small to hold the result.

In addition, the errors documented in [stat\(2\)](#) can also occur.

STANDARDS [top](#)

Linux.

HISTORY [top](#)

Linux 2.4, glibc 2.3.

EXAMPLES [top](#)

See [listxattr\(2\)](#).

SEE ALSO [top](#)

[getfattr\(1\)](#), [setfattr\(1\)](#), [listxattr\(2\)](#), [open\(2\)](#), [removexattr\(2\)](#), [setxattr\(2\)](#), [stat\(2\)](#), [symlink\(7\)](#), [xattr\(7\)](#)

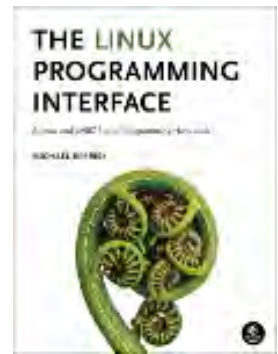
Linux man-pages (unreleased) (date) [getxattr\(2\)](#)

Pages that refer to this page: [listxattr\(2\)](#), [open\(2\)](#), [removexattr\(2\)](#), [setxattr\(2\)](#), [syscalls\(2\)](#), [io_uring_prep_fgetxattr\(3\)](#), [io_uring_prep_getxattr\(3\)](#), [capabilities\(7\)](#), [symlink\(7\)](#), [xattr\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



setxattr(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 setxattr(2)

System Calls Manual

setxattr(2)

NAME [top](#)

setxattr, lsetxattr, fsetxattr - set an extended attribute value

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/xattr.h>
```

```
int setxattr(const char *path, const char *name,  
            const void value[.size], size_t size, int flags);  
int lsetxattr(const char *path, const char *name,  
            const void value[.size], size_t size, int flags);  
int fsetxattr(int fd, const char *name,  
            const void value[.size], size_t size, int flags);
```

DESCRIPTION [top](#)

Extended attributes are *name:value* pairs associated with inodes (files, directories, symbolic links, etc.). They are extensions to the normal attributes which are associated with all inodes in the system (i.e., the [stat\(2\)](#) data). A complete overview of extended attributes concepts can be found in [xattr\(7\)](#).

setxattr() sets the *value* of the extended attribute identified by *name* and associated with the given *path* in the filesystem. The *size* argument specifies the size (in bytes) of *value*; a zero-length value is permitted.

lsetxattr() is identical to **setxattr()**, except in the case of a symbolic link, where the extended attribute is set on the link itself, not the file that it refers to.

fsetxattr() is identical to **setxattr()**, only the extended attribute is set on the open file referred to by *fd* (as returned by **open(2)**) in place of *path*.

An extended attribute name is a null-terminated string. The *name* includes a namespace prefix; there may be several, disjoint namespaces associated with an individual inode. The *value* of an extended attribute is a chunk of arbitrary textual or binary data of specified length.

By default (i.e., *flags* is zero), the extended attribute will be created if it does not exist, or the value will be replaced if the attribute already exists. To modify these semantics, one of the following values can be specified in *flags*:

XATTR_CREATE

Perform a pure create, which fails if the named attribute exists already.

XATTR_REPLACE

Perform a pure replace operation, which fails if the named attribute does not already exist.

RETURN VALUE [top](#)

On success, zero is returned. On failure, -1 is returned and *errno* is set to indicate the error.

ERRORS [top](#)

EDQUOT Disk quota limits meant that there is insufficient space remaining to store the extended attribute.



EEXIST **XATTR_CREATE** was specified, and the attribute exists already.

ENODATA

XATTR_REPLACE was specified, and the attribute does not exist.

ENOSPC There is insufficient space remaining to store the extended attribute.

ENOTSUP

The namespace prefix of *name* is not valid.

ENOTSUP

Extended attributes are not supported by the filesystem, or are disabled,

EPERM The file is marked immutable or append-only. (See [ioctl_iflags\(2\)](#).)

In addition, the errors documented in [stat\(2\)](#) can also occur.

ERANGE The size of *name* or *value* exceeds a filesystem-specific limit.

STANDARDS [top](#)

Linux.

HISTORY [top](#)

Linux 2.4, glibc 2.3.

SEE ALSO [top](#)

[getfattr\(1\)](#), [setfattr\(1\)](#), [getxattr\(2\)](#), [listxattr\(2\)](#), [open\(2\)](#), [removexattr\(2\)](#), [stat\(2\)](#), [symlink\(7\)](#), [xattr\(7\)](#)

Linux man-pages (unreleased) (date)

[setxattr\(2\)](#)

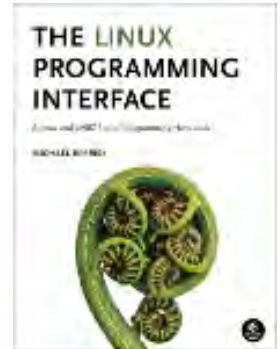


Pages that refer to this page: [getxattr\(2\)](#), [listxattr\(2\)](#), [open\(2\)](#), [removexattr\(2\)](#), [syscalls\(2\)](#), [io_uring_prep_fsetxattr\(3\)](#), [io_uring_prep_setxattr\(3\)](#), [capabilities\(7\)](#), [inotify\(7\)](#), [landlock\(7\)](#), [symlink\(7\)](#), [xattr\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



listxattr(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [BUGS](#) | [EXAMPLES](#) | [SEE ALSO](#)

 [listxattr\(2\)](#)

System Calls Manual

[listxattr\(2\)](#)

NAME [top](#)

listxattr, llistxattr, flistxattr - list extended attribute names

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/xattr.h>
```

```
ssize_t listxattr(const char *path, char *_Nullable list, size_t size);  
ssize_t llistxattr(const char *path, char *_Nullable list, size_t size);  
ssize_t flistxattr(int fd, char *_Nullable list, size_t size);
```

DESCRIPTION [top](#)

Extended attributes are *name:value* pairs associated with inodes (files, directories, symbolic links, etc.). They are extensions to the normal attributes which are associated with all inodes in the system (i.e., the [stat\(2\)](#) data). A complete overview of extended attributes concepts can be found in [xattr\(7\)](#).

listxattr() retrieves the list of extended attribute names associated with the given *path* in the filesystem. The retrieved list is placed in *list*, a caller-allocated buffer whose size (in bytes) is specified in the argument *size*. The list is the set of (null-terminated) names, one after the other. Names of extended attributes to which the calling process does not have access may be omitted from the list. The length of the attribute name *list*

is returned.

llistxattr() is identical to **listxattr**(), except in the case of a symbolic link, where the list of names of extended attributes associated with the link itself is retrieved, not the file that it refers to.

flistxattr() is identical to **listxattr**(), only the open file referred to by *fd* (as returned by **open(2)**) is interrogated in place of *path*.

A single extended attribute *name* is a null-terminated string. The name includes a namespace prefix; there may be several, disjoint namespaces associated with an individual inode.

If *size* is specified as zero, these calls return the current size of the list of extended attribute names (and leave *list* unchanged). This can be used to determine the size of the buffer that should be supplied in a subsequent call. (But, bear in mind that there is a possibility that the set of extended attributes may change between the two calls, so that it is still necessary to check the return status from the second call.)

Example

The *list* of names is returned as an unordered array of null-terminated character strings (attribute names are separated by null bytes ('\0')), like this:

```
user.name1\0system.name1\0user.name2\0
```

Filesystems that implement POSIX ACLs using extended attributes might return a *list* like this:

```
system.posix_acl_access\0system.posix_acl_default\0
```

RETURN VALUE [top](#)

On success, a nonnegative number is returned indicating the size of the extended attribute name list. On failure, -1 is returned and *errno* is set to indicate the error.

ERRORS [top](#)

E2BIG The size of the list of extended attribute names is larger than the maximum size allowed; the list cannot be retrieved. This can happen on filesystems that support an unlimited number of extended attributes per file such as

XFS, for example. See [BUGS](#).

ENOTSUP

Extended attributes are not supported by the filesystem, or are disabled.

ERANGE The *size* of the *list* buffer is too small to hold the result.

In addition, the errors documented in [stat\(2\)](#) can also occur.

STANDARDS [top](#)

Linux.

HISTORY [top](#)

Linux 2.4, glibc 2.3.

BUGS [top](#)

As noted in [xattr\(7\)](#), the VFS imposes a limit of 64 kB on the size of the extended attribute name list returned by **listxattr()**. If the total size of attribute names attached to a file exceeds this limit, it is no longer possible to retrieve the list of attribute names.

EXAMPLES [top](#)

The following program demonstrates the usage of **listxattr()** and [getxattr\(2\)](#). For the file whose pathname is provided as a command-line argument, it lists all extended file attributes and their values.

To keep the code simple, the program assumes that attribute keys and values are constant during the execution of the program. A production program should expect and handle changes during execution of the program. For example, the number of bytes required for attribute keys might increase between the two calls to **listxattr()**. An application could handle this possibility using a loop that retries the call (perhaps up to a predetermined maximum number of attempts) with a larger buffer each time it fails with the error **ERANGE**. Calls to [getxattr\(2\)](#) could be handled similarly.

The following output was recorded by first creating a file, setting some extended file attributes, and then listing the attributes with the example program.

Example output

```
$ touch /tmp/foo
$ setfattr -n user.fred -v chocolate /tmp/foo
$ setfattr -n user.frieda -v bar /tmp/foo
$ setfattr -n user.empty /tmp/foo
$ ./listxattr /tmp/foo
user.fred: chocolate
user.frieda: bar
user.empty: <no value>
```

Program source (listxattr.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/xattr.h>

int
main(int argc, char *argv[])
{
    char    *buf, *key, *val;
    ssize_t  buflen, keylen, vallen;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s path\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /*
     * Determine the length of the buffer needed.
     */
    buflen = listxattr(argv[1], NULL, 0);
    if (buflen == -1) {
        perror("listxattr");
        exit(EXIT_FAILURE);
    }
    if (buflen == 0) {
        printf("%s has no attributes.\n", argv[1]);
        exit(EXIT_SUCCESS);
    }

    /*
     * Allocate the buffer.
     */
    buf = malloc(buflen);
```



```
if (buf == NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
}

/*
 * Copy the list of attribute keys to the buffer.
 */
buflen = listxattr(argv[1], buf, buflen);
if (buflen == -1) {
    perror("listxattr");
    exit(EXIT_FAILURE);
}

/*
 * Loop over the list of zero terminated strings with the
 * attribute keys. Use the remaining buffer length to determine
 * the end of the list.
 */
key = buf;
while (buflen > 0) {

    /*
     * Output attribute key.
     */
    printf("%s: ", key);

    /*
     * Determine length of the value.
     */
    vallen = getxattr(argv[1], key, NULL, 0);
    if (vallen == -1)
        perror("getxattr");

    if (vallen > 0) {

        /*
         * Allocate value buffer.
         * One extra byte is needed to append 0x00.
         */
        val = malloc(vallen + 1);
        if (val == NULL) {
            perror("malloc");
            exit(EXIT_FAILURE);
        }

        /*
         * Copy value to buffer.

```



```
    */
    vallen = getxattr(argv[1], key, val, vallen);
    if (vallen == -1) {
        perror("getxattr");
    } else {
        /*
         * Output attribute value.
         */
        val[vallen] = 0;
        printf("%s", val);
    }

    free(val);
} else if (vallen == 0) {
    printf("<no value>");
}

printf("\n");

/*
 * Forward to next attribute key.
 */
keylen = strlen(key) + 1;
buflen -= keylen;
key += keylen;
}

free(buf);
exit(EXIT_SUCCESS);
}
```

SEE ALSO [top](#)

[getfattr\(1\)](#), [setfattr\(1\)](#), [getxattr\(2\)](#), [open\(2\)](#), [removexattr\(2\)](#),
[setxattr\(2\)](#), [stat\(2\)](#), [symlink\(7\)](#), [xattr\(7\)](#)

Linux man-pages (unreleased)**(date)*****Listxattr(2)***

Pages that refer to this page: [getxattr\(2\)](#), [removexattr\(2\)](#), [setxattr\(2\)](#), [syscalls\(2\)](#), [symlink\(7\)](#),
[xattr\(7\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



removexattr(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 [removexattr\(2\)](#)

System Calls Manual

[removexattr\(2\)](#)

NAME [top](#)

removexattr, lremovexattr, fremovexattr - remove an extended attribute

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/xattr.h>
```

```
int removexattr(const char *path, const char *name);  
int lremovexattr(const char *path, const char *name);  
int fremovexattr(int fd, const char *name);
```

DESCRIPTION [top](#)

Extended attributes are *name:value* pairs associated with inodes (files, directories, symbolic links, etc.). They are extensions to the normal attributes which are associated with all inodes in the system (i.e., the [stat\(2\)](#) data). A complete overview of extended attributes concepts can be found in [xattr\(7\)](#).

removexattr() removes the extended attribute identified by *name*

and associated with the given *path* in the filesystem.

lremovexattr() is identical to **removexattr()**, except in the case of a symbolic link, where the extended attribute is removed from the link itself, not the file that it refers to.

fremovexattr() is identical to **removexattr()**, only the extended attribute is removed from the open file referred to by *fd* (as returned by **open(2)**) in place of *path*.

An extended attribute name is a null-terminated string. The *name* includes a namespace prefix; there may be several, disjoint namespaces associated with an individual inode.

RETURN VALUE [top](#)

On success, zero is returned. On failure, -1 is returned and *errno* is set to indicate the error.

ERRORS [top](#)

ENODATA

The named attribute does not exist.

ENOTSUP

Extended attributes are not supported by the filesystem, or are disabled.

In addition, the errors documented in **stat(2)** can also occur.

STANDARDS [top](#)

Linux.

HISTORY [top](#)

Linux 2.4, glibc 2.3.

SEE ALSO [top](#)

[getfattr\(1\)](#), [setfattr\(1\)](#), [getxattr\(2\)](#), [listxattr\(2\)](#), [open\(2\)](#),
[setxattr\(2\)](#), [stat\(2\)](#), [symlink\(7\)](#), [xattr\(7\)](#)

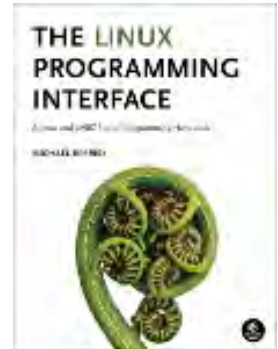
Linux man-pages (unreleased) **(date)** ***removexattr(2)***

Pages that refer to this page: [getxattr\(2\)](#), [listxattr\(2\)](#), [setxattr\(2\)](#), [syscalls\(2\)](#),
[symlink\(7\)](#), [xattr\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of
[The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



getcwd(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [VERSIONS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

 [getcwd\(3\)](#)

Library Functions Manual

[getcwd\(3\)](#)

NAME [top](#)

getcwd, getwd, get_current_dir_name - get current working directory

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
char *getcwd(char buf[.size], size_t size);  
char *get_current_dir_name(void);
```

```
[[deprecated]] char *getwd(char buf[PATH_MAX]);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
get_current_dir_name():  
    _GNU_SOURCE
```

```
getwd():  
    Since glibc 2.12:
```

```

(_XOPEN_SOURCE >= 500) && ! (_POSIX_C_SOURCE >= 200809L)
    || /* glibc >= 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _BSD_SOURCE
Before glibc 2.12:
    _BSD_SOURCE || _XOPEN_SOURCE >= 500

```

DESCRIPTION [top](#)

These functions return a null-terminated string containing an absolute pathname that is the current working directory of the calling process. The pathname is returned as the function result and via the argument *buf*, if present.

The **getcwd()** function copies an absolute pathname of the current working directory to the array pointed to by *buf*, which is of length *size*.

If the length of the absolute pathname of the current working directory, including the terminating null byte, exceeds *size* bytes, NULL is returned, and *errno* is set to **ERANGE**; an application should check for this error, and allocate a larger buffer if necessary.

As an extension to the POSIX.1-2001 standard, glibc's **getcwd()** allocates the buffer dynamically using **malloc(3)** if *buf* is NULL. In this case, the allocated buffer has the length *size* unless *size* is zero, when *buf* is allocated as big as necessary. The caller should **free(3)** the returned buffer.

get_current_dir_name() will **malloc(3)** an array big enough to hold the absolute pathname of the current working directory. If the environment variable **PWD** is set, and its value is correct, then that value will be returned. The caller should **free(3)** the returned buffer.

getwd() does not **malloc(3)** any memory. The *buf* argument should be a pointer to an array at least **PATH_MAX** bytes long. If the length of the absolute pathname of the current working directory, including the terminating null byte, exceeds **PATH_MAX** bytes, NULL is returned, and *errno* is set to **ENAMETOOLONG**. (Note that on some systems, **PATH_MAX** may not be a compile-time constant; furthermore, its value may depend on the filesystem, see

[pathconf\(3\)](#).) For portability and security reasons, use of `getwd()` is deprecated.

RETURN VALUE [top](#)

On success, these functions return a pointer to a string containing the pathname of the current working directory. In the case of `getcwd()` and `getwd()` this is the same value as *buf*.

On failure, these functions return NULL, and *errno* is set to indicate the error. The contents of the array pointed to by *buf* are undefined on error.

ERRORS [top](#)

EACCES Permission to read or search a component of the filename was denied.

EFAULT *buf* points to a bad address.

EINVAL The *size* argument is zero and *buf* is not a null pointer.

EINVAL `getwd()`: *buf* is NULL.

ENAMETOOLONG

`getwd()`: The size of the null-terminated absolute pathname string exceeds **PATH_MAX** bytes.

ENOENT The current working directory has been unlinked.

ENOMEM Out of memory.

ERANGE The *size* argument is less than the length of the absolute pathname of the working directory, including the terminating null byte. You need to allocate a bigger array and try again.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).



| Interface | Attribute | Value |
|--|---------------|-------------|
| <code>getcwd()</code> , <code>getwd()</code> | Thread safety | MT-Safe |
| <code>get_current_dir_name()</code> | Thread safety | MT-Safe env |

VERSIONS [top](#)

POSIX.1-2001 leaves the behavior of `getcwd()` unspecified if `buf` is NULL.

POSIX.1-2001 does not define any errors for `getwd()`.

VERSIONS [top](#)

C library/kernel differences

On Linux, the kernel provides a `getcwd()` system call, which the functions described in this page will use if possible. The system call takes the same arguments as the library function of the same name, but is limited to returning at most **PATH_MAX** bytes. (Before Linux 3.12, the limit on the size of the returned pathname was the system page size. On many architectures, **PATH_MAX** and the system page size are both 4096 bytes, but a few architectures have a larger page size.) If the length of the pathname of the current working directory exceeds this limit, then the system call fails with the error **ENAMETOOLONG**. In this case, the library functions fall back to a (slower) alternative implementation that returns the full pathname.

Following a change in Linux 2.6.36, the pathname returned by the `getcwd()` system call will be prefixed with the string "(unreachable)" if the current directory is not below the root directory of the current process (e.g., because the process set a new filesystem root using `chroot(2)` without changing its current directory into the new root). Such behavior can also be caused by an unprivileged user by changing the current directory into another mount namespace. When dealing with pathname from untrusted sources, callers of the functions described in this page should consider checking whether the returned pathname

starts with '/' or '(' to avoid misinterpreting an unreachable path as a relative pathname.

STANDARDS [top](#)

getcwd()
POSIX.1-2008.

get_current_dir_name()
GNU.

getwd()
None.

HISTORY [top](#)

getcwd()
POSIX.1-2001.

getwd()
POSIX.1-2001, but marked LEGACY. Removed in POSIX.1-2008. Use **getcwd()** instead.

Under Linux, these functions make use of the **getcwd()** system call (available since Linux 2.1.92). On older systems they would query */proc/self/cwd*. If both system call and proc filesystem are missing, a generic implementation is called. Only in that case can these calls fail under Linux with **EACCES**.

NOTES [top](#)

These functions are often used to save the location of the current working directory for the purpose of returning to it later. Opening the current directory (".") and calling **fchdir(2)** to return is usually a faster and more reliable alternative when sufficiently many file descriptors are available, especially on platforms other than Linux.

BUGS [top](#)

Since the Linux 2.6.36 change that added "(unreachable)" in the circumstances described above, the glibc implementation of `getcwd()` has failed to conform to POSIX and returned a relative pathname when the API contract requires an absolute pathname. With glibc 2.27 onwards this is corrected; calling `getcwd()` from such a pathname will now result in failure with `ENOENT`.

SEE ALSO [top](#)

[pwd\(1\)](#), [chdir\(2\)](#), [fchdir\(2\)](#), [open\(2\)](#), [unlink\(2\)](#), [free\(3\)](#), [malloc\(3\)](#)

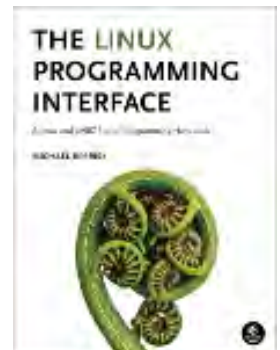
Linux man-pages (unreleased) (date) [getcwd\(3\)](#)

Pages that refer to this page: [pwd\(1\)](#), [chdir\(2\)](#), [syscalls\(2\)](#), [realpath\(3\)](#), [core\(5\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



chdir(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 chdir(2)

System Calls Manual

chdir(2)**NAME** [top](#)

chdir, fchdir - change working directory

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int chdir(const char *path);  
int fchdir(int fd);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
fchdir():  
    _XOPEN_SOURCE >= 500  
    || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L  
    || /* glibc up to and including 2.19: */ _BSD_SOURCE
```

DESCRIPTION [top](#)

chdir() changes the current working directory of the calling process to the directory specified in *path*.

fchdir() is identical to **chdir()**; the only difference is that the directory is given as an open file descriptor.

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

Depending on the filesystem, other errors can be returned. The more general errors for **chdir()** are listed below:

EACCES Search permission is denied for one of the components of *path*. (See also [path_resolution\(7\)](#).)

EFAULT *path* points outside your accessible address space.

EIO An I/O error occurred.

ELOOP Too many symbolic links were encountered in resolving *path*.

ENAMETOOLONG
path is too long.

ENOENT The directory specified in *path* does not exist.

ENOMEM Insufficient kernel memory was available.

ENOTDIR
A component of *path* is not a directory.

The general errors for **fchdir()** are listed below:

EACCES Search permission was denied on the directory open on *fd*.

EBADF *fd* is not a valid file descriptor.

ENOTDIR

`fd` does not refer to a directory.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.4BSD.

NOTES [top](#)

The current working directory is the starting point for interpreting relative pathnames (those not starting with '/').

A child process created via `fork(2)` inherits its parent's current working directory. The current working directory is left unchanged by `execve(2)`.

SEE ALSO [top](#)

[chroot\(2\)](#), [getcwd\(3\)](#), [path_resolution\(7\)](#)

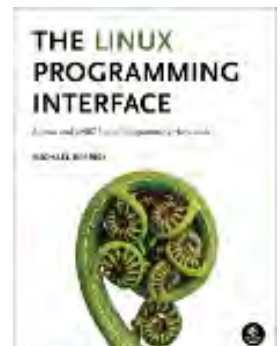
Linux man-pages (unreleased) (date) [chdir\(2\)](#)

Pages that refer to this page: [chroot\(2\)](#), [clone\(2\)](#), [open\(2\)](#), [pivot_root\(2\)](#), [rmdir\(2\)](#), [syscalls\(2\)](#), [unshare\(2\)](#), [dirfd\(3\)](#), [fts\(3\)](#), [ftw\(3\)](#), [getcwd\(3\)](#), [cpuset\(7\)](#), [landlock\(7\)](#), [path_resolution\(7\)](#), [pthreads\(7\)](#), [signal-safety\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



Another version of this page is provided by the [shadow-utils](#) project

passwd(5) — Linux manual page

[NAME](#) | [DESCRIPTION](#) | [FILES](#) | [NOTES](#) | [SEE ALSO](#)

Search online pages

passwd(5)

File Formats Manual

passwd(5)

NAME [top](#)

passwd - password file

DESCRIPTION [top](#)

The */etc/passwd* file is a text file that describes user login accounts for the system. It should have read permission allowed for all users (many utilities, like [ls\(1\)](#) use it to map user IDs to usernames), but write access only for the superuser.

In the good old days there was no great problem with this general read permission. Everybody could read the encrypted passwords, but the hardware was too slow to crack a well-chosen password, and moreover the basic assumption used to be that of a friendly user-community. These days many people run some version of the shadow password suite, where */etc/passwd* has an 'x' character in the password field, and the encrypted passwords are in */etc/shadow*, which is readable by the superuser only.

If the encrypted password, whether in */etc/passwd* or in */etc/shadow*, is an empty string, login is allowed without even asking for a password. Note that this functionality may be intentionally disabled in applications, or configurable (for example using the "**nullok**" or "**nonull**" arguments to [pam_unix\(8\)](#)).

If the encrypted password in */etc/passwd* is "***NP***" (without the



quotes), the shadow record should be obtained from an NIS+ server.

Regardless of whether shadow passwords are used, many system administrators use an asterisk (*) in the encrypted password field to make sure that this user can not authenticate themselves using a password. (But see NOTES below.)

If you create a new login, first put an asterisk (*) in the password field, then use `passwd(1)` to set it.

Each line of the file describes a single user, and contains seven colon-separated fields:

```
name:password:UID:GID:GECOS:directory:shell
```

The field are as follows:

name This is the user's login name. It should not contain capital letters.

password

This is either the encrypted user password, an asterisk (*), or the letter 'x'. (See `pwconv(8)` for an explanation of 'x'.)

UID The privileged *root* login account (superuser) has the user ID 0.

GID This is the numeric primary group ID for this user. (Additional groups for the user are defined in the system group file; see `group(5)`).

GECOS This field (sometimes called the "comment field") is optional and used only for informational purposes. Usually, it contains the full username. Some programs (for example, `finger(1)`) display information from this field.

GECOS stands for "General Electric Comprehensive Operating System", which was renamed to GCOS when GE's large systems division was sold to Honeywell. Dennis Ritchie has reported: "Sometimes we sent printer output or batch jobs to the GCOS machine. The gcos field in the password file



was a place to stash the information for the \$IDENTcard. Not elegant."

directory

This is the user's home directory: the initial directory where the user is placed after logging in. The value in this field is used to set the **HOME** environment variable.

shell

This is the program to run at login (if empty, use */bin/sh*). If set to a nonexistent executable, the user will be unable to login through [login\(1\)](#). The value in this field is used to set the **SHELL** environment variable.

FILES

[top](#)

/etc/passwd

NOTES

[top](#)

If you want to create user groups, there must be an entry in */etc/group*, or no group will exist.

If the encrypted password is set to an asterisk (*), the user will be unable to login using [login\(1\)](#), but may still login using **rlogin(1)**, run existing processes and initiate new ones through **rsh(1)**, [cron\(8\)](#), **at(1)**, or mail filters, etc. Trying to lock an account by simply changing the shell field yields the same result and additionally allows the use of [su\(1\)](#).

SEE ALSO

[top](#)

[chfn\(1\)](#), [chsh\(1\)](#), [login\(1\)](#), [passwd\(1\)](#), [su\(1\)](#), [crypt\(3\)](#), [getpwent\(3\)](#), [getpwnam\(3\)](#), [group\(5\)](#), [shadow\(5\)](#), [vipw\(8\)](#)

Linux man-pages (unreleased)

(date)

passwd(5)

Pages that refer to this page: [chage\(1\)](#), [chfn\(1\)](#), [chfn\(1@@shadow-utils\)](#), [chsh\(1\)](#), [chsh\(1@@shadow-utils\)](#), [expiry\(1\)](#), [login\(1\)](#), [login\(1@@shadow-utils\)](#), [lslogins\(1\)](#), [passwd\(1\)](#), [systemd-firstboot\(1\)](#), [crypt\(3\)](#), [fgetpwent\(3\)](#), [getgrouplist\(3\)](#), [getpw\(3\)](#), [getpwent\(3\)](#), [getpwent_r\(3\)](#), [getpwnam\(3\)](#), [sysexits.h\(3head\)](#), [ftusers\(5\)](#), [group\(5\)](#), [login.defs\(5\)](#), [shadow\(5\)](#), [slapd.backends\(5\)](#), [slapd-passwd\(5\)](#), [systemd.exec\(5\)](#),



[credentials\(7\)](#), [environ\(7\)](#), [grpck\(8\)](#), [newusers\(8\)](#), [nologin\(8\)](#), [nscd\(8\)](#), [pwck\(8\)](#), [vipw\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *[The Linux Programming Interface](#)*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



getent(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [EXIT STATUS](#) | [SEE ALSO](#)

getent(1)

General Commands Manual

getent(1)

NAME [top](#)

`getent` - get entries from Name Service Switch libraries

SYNOPSIS [top](#)

`getent` [*option*]... *database key*...

DESCRIPTION [top](#)

The `getent` command displays entries from databases supported by the Name Service Switch libraries, which are configured in `/etc/nsswitch.conf`. If one or more *key* arguments are provided, then only the entries that match the supplied keys will be displayed. Otherwise, if no *key* is provided, all entries will be displayed (unless the database does not support enumeration).

The *database* may be any of those supported by the GNU C Library, listed below:

ahosts When no *key* is provided, use [sethostent\(3\)](#), [gethostent\(3\)](#), and [endhostent\(3\)](#) to enumerate the hosts database. This is identical to using **hosts**. When one or more *key* arguments are provided, pass each *key* in succession to [getaddrinfo\(3\)](#) with the address family **AF_UNSPEC**, enumerating each socket address structure returned.

ahostsv4

Same as **ahosts**, but use the address family **AF_INET**.

ahostsv6

Same as **ahosts**, but use the address family **AF_INET6**. The call to `getaddrinfo(3)` in this case includes the **AI_V4MAPPED** flag.

aliases

When no *key* is provided, use `setaliasent(3)`, `getaliasent(3)`, and `endaliasent(3)` to enumerate the aliases database. When one or more *key* arguments are provided, pass each *key* in succession to `getaliasbyname(3)` and display the result.

ethers When one or more *key* arguments are provided, pass each *key* in succession to `ether_aton(3)` and `ether_hostton(3)` until a result is obtained, and display the result. Enumeration is not supported on **ethers**, so a *key* must be provided.

group When no *key* is provided, use `setgrent(3)`, `getgrent(3)`, and `endgrent(3)` to enumerate the group database. When one or more *key* arguments are provided, pass each numeric *key* to `getgrgid(3)` and each nonnumeric *key* to `getgrnam(3)` and display the result.

gshadow

When no *key* is provided, use `setsgent(3)`, `getsgent(3)`, and `endsgent(3)` to enumerate the gshadow database. When one or more *key* arguments are provided, pass each *key* in succession to `getsgnam(3)` and display the result.

hosts When no *key* is provided, use `sethostent(3)`, `gethostent(3)`, and `endhostent(3)` to enumerate the hosts database. When one or more *key* arguments are provided, pass each *key* to `gethostbyaddr(3)` or `gethostbyname2(3)`, depending on whether a call to `inet_pton(3)` indicates that the *key* is an IPv6 or IPv4 address or not, and display the result.

initgroups

When one or more *key* arguments are provided, pass each

key in succession to `getgrouplist(3)` and display the result. Enumeration is not supported on `initgroups`, so a *key* must be provided.

netgroup

When one *key* is provided, pass the *key* to `setnetgrent(3)` and, using `getnetgrent(3)` display the resulting string triple (*hostname*, *username*, *domainname*). Alternatively, three *keys* may be provided, which are interpreted as the *hostname*, *username*, and *domainname* to match to a netgroup name via `innetgr(3)`. Enumeration is not supported on `netgroup`, so either one or three *keys* must be provided.

networks

When no *key* is provided, use `setnetent(3)`, `getnetent(3)`, and `endnetent(3)` to enumerate the networks database. When one or more *key* arguments are provided, pass each numeric *key* to `getnetbyaddr(3)` and each nonnumeric *key* to `getnetbyname(3)` and display the result.

passwd When no *key* is provided, use `setpwent(3)`, `getpwent(3)`, and `endpwent(3)` to enumerate the passwd database. When one or more *key* arguments are provided, pass each numeric *key* to `getpwuid(3)` and each nonnumeric *key* to `getpwnam(3)` and display the result.

protocols

When no *key* is provided, use `setprotoent(3)`, `getprotoent(3)`, and `endprotoent(3)` to enumerate the protocols database. When one or more *key* arguments are provided, pass each numeric *key* to `getprotobynumber(3)` and each nonnumeric *key* to `getprotobyname(3)` and display the result.

rpc When no *key* is provided, use `setrpcent(3)`, `getrpcent(3)`, and `endrpcent(3)` to enumerate the rpc database. When one or more *key* arguments are provided, pass each numeric *key* to `getrpcbynumber(3)` and each nonnumeric *key* to `getrpcbyname(3)` and display the result.

services



When no *key* is provided, use `setservernt(3)`, `getservernt(3)`, and `endservernt(3)` to enumerate the services database. When one or more *key* arguments are provided, pass each numeric *key* to `getservbynumber(3)` and each nonnumeric *key* to `getservbyname(3)` and display the result.

shadow When no *key* is provided, use `setspent(3)`, `getspent(3)`, and `endspent(3)` to enumerate the shadow database. When one or more *key* arguments are provided, pass each *key* in succession to `getspnam(3)` and display the result.

OPTIONS [top](#)

- s *service*, --service *service***
Override all databases with the specified service. (Since glibc 2.2.5.)
- s *database:service*, --service *database:service***
Override only specified databases with the specified service. The option may be used multiple times, but only the last service for each database will be used. (Since glibc 2.4.)
- i, --no-idn**
Disables IDN encoding in lookups for **ahosts**/`getaddrinfo(3)` (Since glibc-2.13.)
- ?, --help**
Print a usage summary and exit.
- usage**
Print a short usage summary and exit.
- V, --version**
Print the version number, license, and disclaimer of warranty for **getent**.

EXIT STATUS [top](#)

One of the following exit values can be returned by **getent**:

- 0 Command completed successfully.
- 1 Missing arguments, or *database* unknown.
- 2 One or more supplied *key* could not be found in the *database*.
- 3 Enumeration not supported on this *database*.

SEE ALSO [top](#)

[nsswitch.conf\(5\)](#)

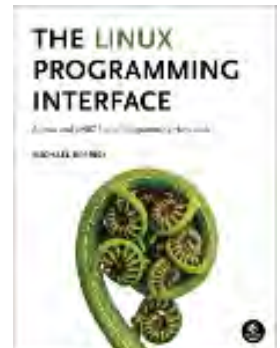
Linux man-pages (unreleased) (date) [getent\(1\)](#)

Pages that refer to this page: [groups\(1\)](#), [homectl\(1\)](#), [userdbctl\(1\)](#), [users\(1\)](#), [nsswitch.conf\(5\)](#), [passwd\(5@@shadow-utils\)](#), [nss-myhostname\(8\)](#), [nss-mymachines\(8\)](#), [nss-systemd\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



mkdir(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 mkdir(2)

System Calls Manual

mkdir(2)

NAME [top](#)

mkdir, mkdirat - create a directory

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/stat.h>
```

```
int mkdir(const char *pathname, mode_t mode);
```

```
#include <fcntl.h> /* Definition of AT_* constants */  
#include <sys/stat.h>
```

```
int mkdirat(int dirfd, const char *pathname, mode_t mode);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
mkdirat():  
    Since glibc 2.10:  
        _POSIX_C_SOURCE >= 200809L  
    Before glibc 2.10:  
        _ATFILE_SOURCE
```

DESCRIPTION [top](#)

mkdir() attempts to create a directory named *pathname*.

The argument *mode* specifies the mode for the new directory (see [inode\(7\)](#)). It is modified by the process's *umask* in the usual way: in the absence of a default ACL, the mode of the created directory is (*mode* & ~*umask* & 0777). Whether other *mode* bits are honored for the created directory depends on the operating system. For Linux, see NOTES below.

The newly created directory will be owned by the effective user ID of the process. If the directory containing the file has the set-group-ID bit set, or if the filesystem is mounted with BSD group semantics (*mount -o bsdgroups* or, synonymously *mount -o grpuid*), the new directory will inherit the group ownership from its parent; otherwise it will be owned by the effective group ID of the process.

If the parent directory has the set-group-ID bit set, then so will the newly created directory.

mkdirat()

The **mkdirat()** system call operates in exactly the same way as **mkdir()**, except for the differences described here.

If the *pathname* given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **mkdir()** for a relative *pathname*).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like **mkdir()**).

If *pathname* is absolute, then *dirfd* is ignored.

See [openat\(2\)](#) for an explanation of the need for **mkdirat()**.

RETURN VALUE [top](#)



mkdir() and **mkdirat()** return zero on success. On error, -1 is returned and *errno* is set to indicate the error.

ERRORS [top](#)

- EACCES** The parent directory does not allow write permission to the process, or one of the directories in *pathname* did not allow search permission. (See also [path_resolution\(7\)](#).)
- EBADF** (**mkdirat()**) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.
- EDQUOT** The user's quota of disk blocks or inodes on the filesystem has been exhausted.
- EEXIST** *pathname* already exists (not necessarily as a directory). This includes the case where *pathname* is a symbolic link, dangling or not.
- EFAULT** *pathname* points outside your accessible address space.
- EINVAL** The final component ("basename") of the new directory's *pathname* is invalid (e.g., it contains characters not permitted by the underlying filesystem).
- ELOOP** Too many symbolic links were encountered in resolving *pathname*.
- EMLINK** The number of links to the parent directory would exceed **LINK_MAX**.
- ENAMETOOLONG**
pathname was too long.
- ENOENT** A directory component in *pathname* does not exist or is a dangling symbolic link.
- ENOMEM** Insufficient kernel memory was available.
- ENOSPC** The device containing *pathname* has no room for the new directory.
- ENOSPC** The new directory cannot be created because the user's

disk quota is exhausted.

ENOTDIR

A component used as a directory in *pathname* is not, in fact, a directory.

ENOTDIR

(*mkdirat*()) *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

EPERM The filesystem containing *pathname* does not support the creation of directories.

EROFS *pathname* refers to a file on a read-only filesystem.

VERSIONS [top](#)

Under Linux, apart from the permission bits, the **S_ISVTX** *mode* bit is also honored.

glibc notes

On older kernels where *mkdirat*() is unavailable, the glibc wrapper function falls back to the use of *mkdir*(). When *pathname* is a relative pathname, glibc constructs a pathname based on the symbolic link in */proc/self/fd* that corresponds to the *dirfd* argument.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

mkdir()
SVr4, BSD, POSIX.1-2001.

mkdirat()
Linux 2.6.16, glibc 2.4.

NOTES [top](#)

There are many infelicities in the protocol underlying NFS. Some of these affect `mkdir()`.

SEE ALSO [top](#)

[mkdir\(1\)](#), [chmod\(2\)](#), [chown\(2\)](#), [mknod\(2\)](#), [mount\(2\)](#), [rmdir\(2\)](#), [stat\(2\)](#), [umask\(2\)](#), [unlink\(2\)](#), [acl\(5\)](#), [path_resolution\(7\)](#)

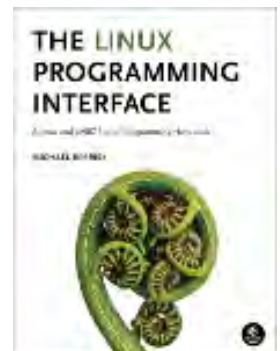
Linux man-pages (unreleased) (date) [mkdir\(2\)](#)

Pages that refer to this page: [mkdir\(1\)](#), [chmod\(2\)](#), [chown\(2\)](#), [fanotify_mark\(2\)](#), [fcntl\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [mknod\(2\)](#), [open\(2\)](#), [rmdir\(2\)](#), [seccomp_unotify\(2\)](#), [syscalls\(2\)](#), [umask\(2\)](#), [io_uring_prep_mkdir\(3\)](#), [io_uring_prep_mkdirat\(3\)](#), [mkdtemp\(3\)](#), [mode_t\(3type\)](#), [proc\(5\)](#), [cpuset\(7\)](#), [inotify\(7\)](#), [signal-safety\(7\)](#), [mount\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



umask(1p) — Linux manual page

[PROLOG](#) | [NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [OPTIONS](#) | [OPERANDS](#) | [STDIN](#) | [INPUT FILES](#) | [ENVIRONMENT VARIABLES](#) | [ASYNCHRONOUS EVENTS](#) | [STDOUT](#) | [STDERR](#) | [OUTPUT FILES](#) | [EXTENDED DESCRIPTION](#) | [EXIT STATUS](#) | [CONSEQUENCES OF ERRORS](#) | [APPLICATION USAGE](#) | [EXAMPLES](#) | [RATIONALE](#) | [FUTURE DIRECTIONS](#) | [SEE ALSO](#) | [COPYRIGHT](#)

 UMASK(1P)

POSIX Programmer's Manual

UMASK(1P)

PROLOG [top](#)

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME [top](#)

`umask` – get or set the file mode creation mask

SYNOPSIS [top](#)

```
umask [-S] [mask]
```

DESCRIPTION [top](#)

The *umask* utility shall set the file mode creation mask of the current shell execution environment (see [Section 2.12, Shell Execution Environment](#)) to the value specified by the *mask* operand. This mask shall affect the initial value of the file permission bits of subsequently created files. If *umask* is called in a subshell or separate utility execution environment, such as one of the following:



```
(umask 002)
nohup umask ...
find . -exec umask ... \;
```

it shall not affect the file mode creation mask of the caller's environment.

If the *mask* operand is not specified, the *umask* utility shall write to standard output the value of the file mode creation mask of the invoking process.

OPTIONS [top](#)

The *umask* utility shall conform to the Base Definitions volume of POSIX.1-2017, *Section 12.2, Utility Syntax Guidelines*.

The following option shall be supported:

-S Produce symbolic output.

The default output style is unspecified, but shall be recognized on a subsequent invocation of *umask* on the same system as a *mask* operand to restore the previous file mode creation mask.

OPERANDS [top](#)

The following operand shall be supported:

mask A string specifying the new file mode creation mask. The string is treated in the same way as the *mode* operand described in the EXTENDED DESCRIPTION section for *chmod*.

For a *symbolic_mode* value, the new value of the file mode creation mask shall be the logical complement of the file permission bits portion of the file mode specified by the *symbolic_mode* string.

In a *symbolic_mode* value, the permissions *op* characters '+' and '-' shall be interpreted relative to the current file mode creation mask; '+' shall cause the

bits for the indicated permissions to be cleared in the mask; '-' shall cause the bits for the indicated permissions to be set in the mask.

The interpretation of *mode* values that specify file mode bits other than the file permission bits is unspecified.

In the octal integer form of *mode*, the specified bits are set in the file mode creation mask.

The file mode creation mask shall be set to the resulting numeric value.

The default output of a prior invocation of *umask* on the same system with no operand also shall be recognized as a *mask* operand.

STDIN [top](#)

Not used.

INPUT FILES [top](#)

None.

ENVIRONMENT VARIABLES [top](#)

The following environment variables shall affect the execution of *umask*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of POSIX.1-2017, *Section 8.2, Internationalization Variables* for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.



LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH Determine the location of message catalogs for the processing of **LC_MESSAGES**.

ASYNCHRONOUS EVENTS [top](#)

Default.

STDOUT [top](#)

When the **mask** operand is not specified, the **umask** utility shall write a message to standard output that can later be used as a **umask mask** operand.

If **-S** is specified, the message shall be in the following format:

```
"u=%s,g=%s,o=%s\n", <owner permissions>, <group permissions>,
<other permissions>
```

where the three values shall be combinations of letters from the set {**r**, **w**, **x**}; the presence of a letter shall indicate that the corresponding bit is clear in the file mode creation mask.

If a **mask** operand is specified, there shall be no output written to standard output.

STDERR [top](#)

The standard error shall be used only for diagnostic messages.

OUTPUT FILES [top](#)

None.

EXTENDED DESCRIPTION [top](#)

None.

EXIT STATUS [top](#)

The following exit values shall be returned:

- 0 The file mode creation mask was successfully changed, or no *mask* operand was supplied.
- >0 An error occurred.

CONSEQUENCES OF ERRORS [top](#)

Default.

The following sections are informative.

APPLICATION USAGE [top](#)

Since *umask* affects the current shell execution environment, it is generally provided as a shell regular built-in.

In contrast to the negative permission logic provided by the file mode creation mask and the octal number form of the *mask* argument, the symbolic form of the *mask* argument specifies those permissions that are left alone.

EXAMPLES [top](#)

Either of the commands:

```
umask a=rx,ug+w
```

```
umask 002
```

sets the mode mask so that subsequently created files have their



S_IWOTH bit cleared.

After setting the mode mask with either of the above commands, the *umask* command can be used to write out the current value of the mode mask:

```
$ umask
0002
```

(The output format is unspecified, but historical implementations use the octal integer mode format.)

```
$ umask -S
u=rwx,g=rwx,o=rX
```

Either of these outputs can be used as the mask operand to a subsequent invocation of the *umask* utility.

Assuming the mode mask is set as above, the command:

```
umask g-w
```

sets the mode mask so that subsequently created files have their S_IWGRP and S_IWOTH bits cleared.

The command:

```
umask -- -w
```

sets the mode mask so that subsequently created files have all their write bits cleared. Note that *mask* operands *-r*, *-w*, *-x* or anything beginning with a <hyphen-minus>, must be preceded by "--" to keep it from being interpreted as an option.

RATIONALE [top](#)

Since *umask* affects the current shell execution environment, it is generally provided as a shell regular built-in. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(umask 002)
nohup umask ...
```



```
find . -exec umask ... \;
```

it does not affect the file mode creation mask of the environment of the caller.

The description of the historical utility was modified to allow it to use the symbolic modes of *chmod*. The **-s** option used in early proposals was changed to **-S** because **-s** could be confused with a *symbolic_mode* form of mask referring to the S_ISUID and S_ISGID bits.

The default output style is unspecified to permit implementors to provide migration to the new symbolic style at the time most appropriate to their users. A **-o** flag to force octal mode output was omitted because the octal mode may not be sufficient to specify all of the information that may be present in the file mode creation mask when more secure file access permission checks are implemented.

It has been suggested that trusted systems developers might appreciate ameliorating the requirement that the mode mask ``affects'' the file access permissions, since it seems access control lists might replace the mode mask to some degree. The wording has been changed to say that it affects the file permission bits, and it leaves the details of the behavior of how they affect the file access permissions to the description in the System Interfaces volume of POSIX.1-2017.

FUTURE DIRECTIONS [top](#)

None.

SEE ALSO [top](#)

Chapter 2, Shell Command Language, [chmod\(1p\)](#)

The Base Definitions volume of POSIX.1-2017, *Chapter 8, Environment Variables*, *Section 12.2, Utility Syntax Guidelines*

The System Interfaces volume of POSIX.1-2017, [umask\(3p\)](#)

COPYRIGHT [top](#)

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html> .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .

IEEE/The Open Group

2017

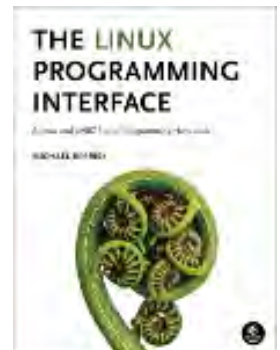
UMASK(1P)

Pages that refer to this page: [c99\(1p\)](#), [chmod\(1p\)](#), [fort77\(1p\)](#), [mkdir\(1p\)](#), [mkfifo\(1p\)](#), [sh\(1p\)](#), [uudecode\(1p\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



rmdir(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [BUGS](#) | [SEE ALSO](#)

rmdir(2)

System Calls Manual

rmdir(2)**NAME** [top](#)

rmdir - delete a directory

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int rmdir(const char *pathname);
```

DESCRIPTION [top](#)

rmdir() deletes a directory, which must be empty.

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

- EACCES** Write access to the directory containing *pathname* was not allowed, or one of the directories in the path prefix of *pathname* did not allow search permission. (See also [path_resolution\(7\)](#).)
- EBUSY** *pathname* is currently in use by the system or some process that prevents its removal. On Linux, this means *pathname* is currently used as a mount point or is the root directory of the calling process.
- EFAULT** *pathname* points outside your accessible address space.
- EINVAL** *pathname* has `.` as last component.
- ELOOP** Too many symbolic links were encountered in resolving *pathname*.
- ENAMETOOLONG**
pathname was too long.
- ENOENT** A directory component in *pathname* does not exist or is a dangling symbolic link.
- ENOMEM** Insufficient kernel memory was available.
- ENOTDIR**
pathname, or a component used as a directory in *pathname*, is not, in fact, a directory.
- ENOTEMPTY**
pathname contains entries other than `.` and `..` ; or, *pathname* has `..` as its final component. POSIX.1 also allows **EEXIST** for this condition.
- EPERM** The directory containing *pathname* has the sticky bit (**S_ISVTX**) set and the process's effective user ID is neither the user ID of the file to be deleted nor that of the directory containing it, and the process is not privileged (Linux: does not have the **CAP_FOWNER** capability).
- EPERM** The filesystem containing *pathname* does not support the removal of directories.



EROFS *pathname* refers to a directory on a read-only filesystem.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.3BSD.

BUGS [top](#)

Infelicities in the protocol underlying NFS can cause the unexpected disappearance of directories which are still being used.

SEE ALSO [top](#)

[rm\(1\)](#), [rmdir\(1\)](#), [chdir\(2\)](#), [chmod\(2\)](#), [mkdir\(2\)](#), [rename\(2\)](#), [unlink\(2\)](#), [unlinkat\(2\)](#)

Linux man-pages (unreleased) [\(date\)](#) *rmdir(2)*

Pages that refer to this page: [rmdir\(1\)](#), [fanotify_mark\(2\)](#), [fcntl\(2\)](#), [mkdir\(2\)](#), [syscalls\(2\)](#), [unlink\(2\)](#), [remove\(3\)](#), [cpuset\(7\)](#), [mount_namespaces\(7\)](#), [signal-safety\(7\)](#), [symlink\(7\)](#), [mount\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



opendir(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [STANDARDS](#) | [NOTES](#) | [SEE ALSO](#)

 [opendir\(3\)](#)

Library Functions Manual

[opendir\(3\)](#)

NAME [top](#)

opendir, fdopendir - open a directory

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
DIR *fdopendir(int fd);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
fdopendir():
    Since glibc 2.10:
        _POSIX_C_SOURCE >= 200809L
    Before glibc 2.10:
        _GNU_SOURCE
```

DESCRIPTION [top](#)

The **opendir()** function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

The **fdopendir()** function is like **opendir()**, but returns a directory stream for the directory referred to by the open file descriptor *fd*. After a successful call to **fdopendir()**, *fd* is used internally by the implementation, and should not otherwise be used by the application.

RETURN VALUE [top](#)

The **opendir()** and **fdopendir()** functions return a pointer to the directory stream. On error, NULL is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EACCES Permission denied.

EBADF *fd* is not a valid file descriptor opened for reading.

EMFILE The per-process limit on the number of open file descriptors has been reached.

ENFILE The system-wide limit on the total number of open files has been reached.

ENOENT Directory does not exist, or *name* is an empty string.

ENOMEM Insufficient memory to complete the operation.

ENOTDIR
name is not a directory.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).



| Interface | Attribute | Value |
|---|---------------|---------|
| <code>opendir()</code> , <code>fdopendir()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

POSIX.1-2008.

STANDARDS [top](#)

`opendir()`
SVr4, 4.3BSD, POSIX.1-2001.

`fdopendir()`
POSIX.1-2008. glibc 2.4.

NOTES [top](#)

Filename entries can be read from a directory stream using [readdir\(3\)](#).

The underlying file descriptor of the directory stream can be obtained using [dirfd\(3\)](#).

The `opendir()` function sets the close-on-exec flag for the file descriptor underlying the *DIR* *. The `fdopendir()` function leaves the setting of the close-on-exec flag unchanged for the file descriptor, *fd*. POSIX.1-200x leaves it unspecified whether a successful call to `fdopendir()` will set the close-on-exec flag for the file descriptor, *fd*.

SEE ALSO [top](#)

[open\(2\)](#), [closedir\(3\)](#), [dirfd\(3\)](#), [readdir\(3\)](#), [rewinddir\(3\)](#), [scandir\(3\)](#), [seekdir\(3\)](#), [telldir\(3\)](#)

Linux man-pages (unreleased) (date)

[opendir\(3\)](#)

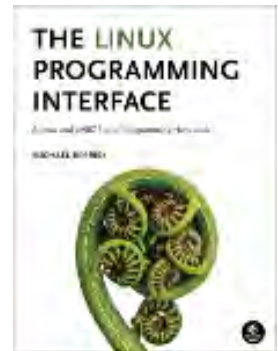


Pages that refer to this page: [close_range\(2\)](#), [execve\(2\)](#), [fanotify_mark\(2\)](#), [fork\(2\)](#), [open\(2\)](#), [closedir\(3\)](#), [dirfd\(3\)](#), [fts\(3\)](#), [getdirentries\(3\)](#), [glob\(3\)](#), [readdir\(3\)](#), [rewinddir\(3\)](#), [scandir\(3\)](#), [seekdir\(3\)](#), [telldir\(3\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



dirfd(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 dirfd(3)

Library Functions Manual

dirfd(3)

NAME [top](#)

dirfd - get directory stream file descriptor

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/types.h>
#include <dirent.h>

int dirfd(DIR *dirp);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
dirfd():
    /* Since glibc 2.10: */ _POSIX_C_SOURCE >= 200809L
    || /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

DESCRIPTION [top](#)

The function `dirfd()` returns the file descriptor associated with the directory stream *dirp*.

This file descriptor is the one used internally by the directory stream. As a result, it is useful only for functions which do not depend on or alter the file position, such as [fstat\(2\)](#) and [fchdir\(2\)](#). It will be automatically closed when [closedir\(3\)](#) is called.

RETURN VALUE [top](#)

On success, `dirfd()` returns a file descriptor (a nonnegative integer). On error, -1 is returned, and `errno` is set to indicate the error.

ERRORS [top](#)

POSIX.1-2008 specifies two errors, neither of which is returned by the current implementation.

EINVAL *dirp* does not refer to a valid directory stream.

ENOTSUP

The implementation does not support the association of a file descriptor with a directory.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|----------------------|---------------|---------|
| <code>dirfd()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

4.3BSD-Reno (not in 4.2BSD).

SEE ALSO [top](#)

[open\(2\)](#), [openat\(2\)](#), [closedir\(3\)](#), [opendir\(3\)](#), [readdir\(3\)](#),
[rewinddir\(3\)](#), [scandir\(3\)](#), [seekdir\(3\)](#), [telldir\(3\)](#)

Linux man-pages (unreleased) (date)

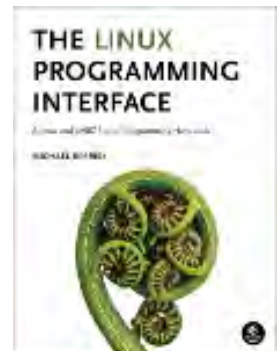
[dirfd\(3\)](#)

Pages that refer to this page: [open\(2\)](#), [opendir\(3\)](#), [readdir\(3\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



readdir(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 [readdir\(3\)](#)

Library Functions Manual

[readdir\(3\)](#)

NAME [top](#)

readdir - read a directory

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

DESCRIPTION [top](#)

The `readdir()` function returns a pointer to a *dirent* structure representing the next directory entry in the directory stream pointed to by *dirp*. It returns NULL on reaching the end of the directory stream or if an error occurred.

In the glibc implementation, the *dirent* structure is defined as follows:

```
struct dirent {
    ino_t      d_ino;      /* Inode number */
    off_t     d_off;      /* Not an offset; see below */
```



```
    unsigned short d_reclen;    /* Length of this record */
    unsigned char  d_type;      /* Type of file; not supported
                                by all filesystem types */
    char           d_name[256]; /* Null-terminated filename */
};
```

The only fields in the *dirent* structure that are mandated by POSIX.1 are *d_name* and *d_ino*. The other fields are unstandardized, and not present on all systems; see NOTES below for some further details.

The fields of the *dirent* structure are as follows:

d_ino This is the inode number of the file.

d_off The value returned in *d_off* is the same as would be returned by calling `telldir(3)` at the current position in the directory stream. Be aware that despite its type and name, the *d_off* field is seldom any kind of directory offset on modern filesystems. Applications should treat this field as an opaque value, making no assumptions about its contents; see also `telldir(3)`.

d_reclen

This is the size (in bytes) of the returned record. This may not match the size of the structure definition shown above; see NOTES.

d_type This field contains a value indicating the file type, making it possible to avoid the expense of calling `lstat(2)` if further actions depend on the type of the file.

When a suitable feature test macro is defined (`_DEFAULT_SOURCE` since glibc 2.19, or `_BSD_SOURCE` on glibc 2.19 and earlier), glibc defines the following macro constants for the value returned in *d_type*:

DT_BLK This is a block device.

DT_CHR This is a character device.

DT_DIR This is a directory.

DT_FIFO



This is a named pipe (FIFO).

DT_LNK This is a symbolic link.

DT_REG This is a regular file.

DT SOCK
This is a UNIX domain socket.

DT_UNKNOWN
The file type could not be determined.

Currently, only some filesystems (among them: Btrfs, ext2, ext3, and ext4) have full support for returning the file type in *d_type*. All applications must properly handle a return of **DT_UNKNOWN**.

d_name This field contains the null terminated filename. *See NOTES.*

The data returned by **readdir()** may be overwritten by subsequent calls to **readdir()** for the same directory stream.

RETURN VALUE [top](#)

On success, **readdir()** returns a pointer to a *dirent* structure. (This structure may be statically allocated; do not attempt to **free(3)** it.)

If the end of the directory stream is reached, NULL is returned and *errno* is not changed. If an error occurs, NULL is returned and *errno* is set to indicate the error. To distinguish end of stream from an error, set *errno* to zero before calling **readdir()** and then check the value of *errno* if NULL is returned.

ERRORS [top](#)

EBADF Invalid directory stream descriptor *dirp*.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|------------------------|---------------|--------------------------|
| <code>readdir()</code> | Thread safety | MT-Unsafe race:dirstream |

In the current POSIX.1 specification (POSIX.1-2008), `readdir()` is not required to be thread-safe. However, in modern implementations (including the glibc implementation), concurrent calls to `readdir()` that specify different directory streams are thread-safe. In cases where multiple threads must read from the same directory stream, using `readdir()` with external synchronization is still preferable to the use of the deprecated [readdir_r\(3\)](#) function. It is expected that a future version of POSIX.1 will require that `readdir()` be thread-safe when concurrently employed on different directory streams.

VERSIONS [top](#)

Only the fields `d_name` and (as an XSI extension) `d_ino` are specified in POSIX.1. Other than Linux, the `d_type` field is available mainly only on BSD systems. The remaining fields are available on many, but not all systems. Under glibc, programs can check for the availability of the fields not defined in POSIX.1 by testing whether the macros `_DIRENT_HAVE_D_NAMLEN`, `_DIRENT_HAVE_D_RECLEN`, `_DIRENT_HAVE_D_OFF`, or `_DIRENT_HAVE_D_TYPE` are defined.

The `d_name` field

The `dirent` structure definition shown above is taken from the glibc headers, and shows the `d_name` field with a fixed size.

Warning: applications should avoid any dependence on the size of the `d_name` field. POSIX defines it as `char d_name[]`, a character array of unspecified size, with at most `NAME_MAX` characters preceding the terminating null byte (`'\0'`).

POSIX.1 explicitly notes that this field should not be used as an lvalue. The standard also notes that the use of `sizeof(d_name)` is incorrect; use `strlen(d_name)` instead. (On some systems, this field is defined as `char d_name[1]!`) By implication, the use `sizeof(struct dirent)` to capture the size of the record including



the size of *d_name* is also incorrect.

Note that while the call

```
fpathconf(fd, _PC_NAME_MAX)
```

returns the value 255 for most filesystems, on some filesystems (e.g., CIFS, Windows SMB servers), the null-terminated filename that is (correctly) returned in *d_name* can actually exceed this size. In such cases, the *d_reclen* field will contain a value that exceeds the size of the glibc *dirent* structure shown above.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.3BSD.

NOTES [top](#)

A directory stream is opened using [opendir\(3\)](#).

The order in which filenames are read by successive calls to [readdir\(\)](#) depends on the filesystem implementation; it is unlikely that the names will be sorted in any fashion.

SEE ALSO [top](#)

[getdents\(2\)](#), [read\(2\)](#), [closedir\(3\)](#), [dirfd\(3\)](#), [ftw\(3\)](#), [offsetof\(3\)](#), [opendir\(3\)](#), [readdir_r\(3\)](#), [rewinddir\(3\)](#), [scandir\(3\)](#), [seekdir\(3\)](#), [telldir\(3\)](#)

Linux man-pages (unreleased) (date) [readdir\(3\)](#)

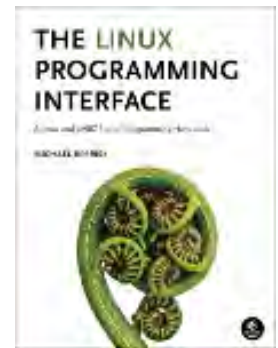
Pages that refer to this page: [sshfs\(1\)](#), [fanotify_mark\(2\)](#), [getdents\(2\)](#), [readdir\(2\)](#), [closedir\(3\)](#), [dirfd\(3\)](#), [fts\(3\)](#), [ftw\(3\)](#), [getdirentries\(3\)](#), [glob\(3\)](#), [opendir\(3\)](#), [readdir_r\(3\)](#), [rewinddir\(3\)](#), [scandir\(3\)](#), [seekdir\(3\)](#), [telldir\(3\)](#), [xfs_io\(8\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



closedir(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [SEE ALSO](#)

 closedir(3)

Library Functions Manual

closedir(3)**NAME** [top](#)

closedir - close a directory

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/types.h>
#include <dirent.h>

int closedir(DIR *dirp);
```

DESCRIPTION [top](#)

The **closedir()** function closes the directory stream associated with *dirp*. A successful call to **closedir()** also closes the underlying file descriptor associated with *dirp*. The directory stream descriptor *dirp* is not available after this call.

RETURN VALUE [top](#)

The **closedir()** function returns 0 on success. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EBADF Invalid directory stream descriptor *dirp*.

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-------------------------|---------------|---------|
| <code>closedir()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, SVr4, 4.3BSD.

SEE ALSO [top](#)

[close\(2\)](#), [opendir\(3\)](#), [readdir\(3\)](#), [rewinddir\(3\)](#), [scandir\(3\)](#), [seekdir\(3\)](#), [telldir\(3\)](#)

Linux man-pages (unreleased) [\(date\)](#) [closedir\(3\)](#)

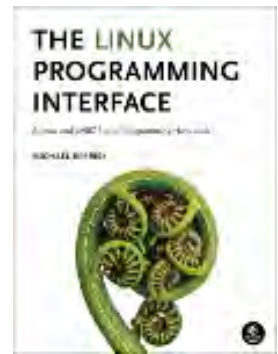
Pages that refer to this page: [fanotify_mark\(2\)](#), [dirfd\(3\)](#), [opendir\(3\)](#), [readdir\(3\)](#), [rewinddir\(3\)](#), [scandir\(3\)](#), [seekdir\(3\)](#), [telldir\(3\)](#)



HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



link(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [BUGS](#) | [SEE ALSO](#)

 [Link\(2\)](#)

System Calls Manual

[Link\(2\)](#)

NAME [top](#)

link, linkat - make a new name for a file

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int link(const char *oldpath, const char *newpath);
```

```
#include <fcntl.h> /* Definition of AT_* constants */  
#include <unistd.h>
```

```
int linkat(int olddirfd, const char *oldpath,  
           int newdirfd, const char *newpath, int flags);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
linkat():  
    Since glibc 2.10:  
        _POSIX_C_SOURCE >= 200809L
```

Before glibc 2.10:
_ATFILE_SOURCE

DESCRIPTION [top](#)

link() creates a new link (also known as a hard link) to an existing file.

If *newpath* exists, it will *not* be overwritten.

This new name may be used exactly as the old one for any operation; both names refer to the same file (and so have the same permissions and ownership) and it is impossible to tell which name was the "original".

linkat()

The **linkat()** system call operates in exactly the same way as **link()**, except for the differences described here.

If the pathname given in *oldpath* is relative, then it is interpreted relative to the directory referred to by the file descriptor *olddirfd* (rather than relative to the current working directory of the calling process, as is done by **link()** for a relative pathname).

If *oldpath* is relative and *olddirfd* is the special value **AT_FDCWD**, then *oldpath* is interpreted relative to the current working directory of the calling process (like **link()**).

If *oldpath* is absolute, then *olddirfd* is ignored.

The interpretation of *newpath* is as for *oldpath*, except that a relative pathname is interpreted relative to the directory referred to by the file descriptor *newdirfd*.

The following values can be bitwise ORed in *flags*:

AT_EMPTY_PATH (since Linux 2.6.39)

If *oldpath* is an empty string, create a link to the file referenced by *olddirfd* (which may have been obtained using the **open(2)** **O_PATH** flag). In this case, *olddirfd* can refer to any type of file except a directory. This will generally not work if the file has a link count of zero



(files created with **O_TMPFILE** and without **O_EXCL** are an exception). The caller must have the **CAP_DAC_READ_SEARCH** capability in order to use this flag. This flag is Linux-specific; define **_GNU_SOURCE** to obtain its definition.

AT_SYMLINK_FOLLOW (since Linux 2.6.18)

By default, **linkat()**, does not dereference *oldpath* if it is a symbolic link (like **link()**). The flag **AT_SYMLINK_FOLLOW** can be specified in *flags* to cause *oldpath* to be dereferenced if it is a symbolic link. If *procfs* is mounted, this can be used as an alternative to **AT_EMPTY_PATH**, like this:

```
linkat(AT_FDCWD, "/proc/self/fd/<fd>", newdirfd,  
      newname, AT_SYMLINK_FOLLOW);
```

Before Linux 2.6.18, the *flags* argument was unused, and had to be specified as 0.

See [openat\(2\)](#) for an explanation of the need for **linkat()**.

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EACCES Write access to the directory containing *newpath* is denied, or search permission is denied for one of the directories in the path prefix of *oldpath* or *newpath*. (See also [path_resolution\(7\)](#).)

EDQUOT The user's quota of disk blocks on the filesystem has been exhausted.

EEXIST *newpath* already exists.

EFAULT *oldpath* or *newpath* points outside your accessible address space.

EIO An I/O error occurred.



- ELOOP** Too many symbolic links were encountered in resolving *oldpath* or *newpath*.
- EMLINK** The file referred to by *oldpath* already has the maximum number of links to it. For example, on an `ext4(5)` filesystem that does not employ the *dir_index* feature, the limit on the number of hard links to a file is 65,000; on `btrfs(5)`, the limit is 65,535 links.
- ENAMETOOLONG**
oldpath or *newpath* was too long.
- ENOENT** A directory component in *oldpath* or *newpath* does not exist or is a dangling symbolic link.
- ENOMEM** Insufficient kernel memory was available.
- ENOSPC** The device containing the file has no room for the new directory entry.
- ENOTDIR**
A component used as a directory in *oldpath* or *newpath* is not, in fact, a directory.
- EPERM** *oldpath* is a directory.
- EPERM** The filesystem containing *oldpath* and *newpath* does not support the creation of hard links.
- EPERM** (since Linux 3.6)
The caller does not have permission to create a hard link to this file (see the description of */proc/sys/fs/protected_hardlinks* in `proc(5)`).
- EPERM** *oldpath* is marked immutable or append-only. (See `ioctl_iflags(2)`.)
- EROFS** The file is on a read-only filesystem.
- EXDEV** *oldpath* and *newpath* are not on the same mounted filesystem. (Linux permits a filesystem to be mounted at multiple points, but `link()` does not work across different mounts, even if the same filesystem is mounted on both.)



The following additional errors can occur for **linkat()**:

EBADF *oldpath* (*newpath*) is relative but *olddirfd* (*newdirfd*) is neither **AT_FDCWD** nor a valid file descriptor.

EINVAL An invalid flag value was specified in *flags*.

ENOENT **AT_EMPTY_PATH** was specified in *flags*, but the caller did not have the **CAP_DAC_READ_SEARCH** capability.

ENOENT An attempt was made to link to the */proc/self/fd/NN* file corresponding to a file descriptor created with

```
open(path, O_TMPFILE | O_EXCL, mode);
```

See [open\(2\)](#).

ENOENT An attempt was made to link to a */proc/self/fd/NN* file corresponding to a file that has been deleted.

ENOENT *oldpath* is a relative pathname and *olddirfd* refers to a directory that has been deleted, or *newpath* is a relative pathname and *newdirfd* refers to a directory that has been deleted.

ENOTDIR

oldpath is relative and *olddirfd* is a file descriptor referring to a file other than a directory; or similar for *newpath* and *newdirfd*

EPERM **AT_EMPTY_PATH** was specified in *flags*, *oldpath* is an empty string, and *olddirfd* refers to a directory.

VERSIONS

[top](#)

POSIX.1-2001 says that **link()** should dereference *oldpath* if it is a symbolic link. However, since Linux 2.0, Linux does not do so: if *oldpath* is a symbolic link, then *newpath* is created as a (hard) link to the same symbolic link file (i.e., *newpath* becomes a symbolic link to the same file that *oldpath* refers to). Some other implementations behave in the same manner as Linux. POSIX.1-2008 changes the specification of **link()**, making it



implementation-dependent whether or not *oldpath* is dereferenced if it is a symbolic link. For precise control over the treatment of symbolic links when creating a link, use **linkat()**.

glibc

On older kernels where **linkat()** is unavailable, the glibc wrapper function falls back to the use of **link()**, unless the **AT_SYMLINK_FOLLOW** is specified. When *oldpath* and *newpath* are relative pathnames, glibc constructs pathnames based on the symbolic links in */proc/self/fd* that correspond to the *olddirfd* and *newdirfd* arguments.

STANDARDS [top](#)

link() POSIX.1-2008.

HISTORY [top](#)

link() SVr4, 4.3BSD, POSIX.1-2001 (but see VERSIONS).

linkat()
POSIX.1-2008. Linux 2.6.16, glibc 2.4.

NOTES [top](#)

Hard links, as created by **link()**, cannot span filesystems. Use [symlink\(2\)](#) if this is required.

BUGS [top](#)

On NFS filesystems, the return code may be wrong in case the NFS server performs the link creation and dies before it can say so. Use [stat\(2\)](#) to find out if the link got created.

SEE ALSO [top](#)

[ln\(1\)](#), [open\(2\)](#), [rename\(2\)](#), [stat\(2\)](#), [symlink\(2\)](#), [unlink\(2\)](#), [path_resolution\(7\)](#), [symlink\(7\)](#)

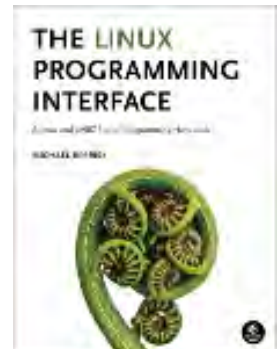


Pages that refer to this page: [link\(1\)](#), [ln\(1\)](#), [sshfs\(1\)](#), [fcntl\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [open\(2\)](#), [rename\(2\)](#), [symlink\(2\)](#), [syscalls\(2\)](#), [unlink\(2\)](#), [io_uring_prep_link\(3\)](#), [io_uring_prep_linkat\(3\)](#), [remove\(3\)](#), [capabilities\(7\)](#), [inode\(7\)](#), [inotify\(7\)](#), [signal-safety\(7\)](#), [symlink\(7\)](#), [mount\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



symlink(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 [symlink\(2\)](#)

System Calls Manual

[symlink\(2\)](#)

NAME [top](#)

symlink, symlinkat - make a new name for a file

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int symlink(const char *target, const char *linkpath);
```

```
#include <fcntl.h> /* Definition of AT_* constants */
```

```
#include <unistd.h>
```

```
int symlinkat(const char *target, int newdirfd, const char *linkpath);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

symlink():

```
_XOPEN_SOURCE >= 500 || _POSIX_C_SOURCE >= 200112L  
|| /* glibc <= 2.19: */ _BSD_SOURCE
```

symlinkat():

Since glibc 2.10:

```
_POSIX_C_SOURCE >= 200809L
```

Before glibc 2.10:
_ATFILE_SOURCE

DESCRIPTION [top](#)

symlink() creates a symbolic link named *linkpath* which contains the string *target*.

Symbolic links are interpreted at run time as if the contents of the link had been substituted into the path being followed to find a file or directory.

Symbolic links may contain `..` path components, which (if used at the start of the link) refer to the parent directories of that in which the link resides.

A symbolic link (also known as a soft link) may point to an existing file or to a nonexistent one; the latter case is known as a dangling link.

The permissions of a symbolic link are irrelevant; the ownership is ignored when following the link (except when the *protected_symlinks* feature is enabled, as explained in [proc\(5\)](#)), but is checked when removal or renaming of the link is requested and the link is in a directory with the sticky bit (**S_ISVTX**) set.

If *linkpath* exists, it will *not* be overwritten.

symlinkat()

The **symlinkat()** system call operates in exactly the same way as **symlink()**, except for the differences described here.

If the pathname given in *linkpath* is relative, then it is interpreted relative to the directory referred to by the file descriptor *newdirfd* (rather than relative to the current working directory of the calling process, as is done by **symlink()** for a relative pathname).

If *linkpath* is relative and *newdirfd* is the special value **AT_FDCWD**, then *linkpath* is interpreted relative to the current working directory of the calling process (like **symlink()**).

If *linkpath* is absolute, then *newdirfd* is ignored.

See [openat\(2\)](#) for an explanation of the need for **symlinkat()**.



RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EACCES Write access to the directory containing *linkpath* is denied, or one of the directories in the path prefix of *linkpath* did not allow search permission. (See also [path_resolution\(7\)](#).)

EBADF (`symlinkat()`) *linkpath* is relative but *newdirfd* is neither `AT_FDCWD` nor a valid file descriptor.

EDQUOT The user's quota of resources on the filesystem has been exhausted. The resources could be inodes or disk blocks, depending on the filesystem implementation.

EEXIST *linkpath* already exists.

EFAULT *target* or *linkpath* points outside your accessible address space.

EIO An I/O error occurred.

ELOOP Too many symbolic links were encountered in resolving *linkpath*.

ENAMETOOLONG *target* or *linkpath* was too long.

ENOENT A directory component in *linkpath* does not exist or is a dangling symbolic link, or *target* or *linkpath* is an empty string.

ENOENT (`symlinkat()`) *linkpath* is a relative pathname and *newdirfd* refers to a directory that has been deleted.

ENOMEM Insufficient kernel memory was available.

ENOSPC The device containing the file has no room for the new directory entry.

ENOTDIR A component used as a directory in *linkpath* is not, in

fact, a directory.

ENOTDIR

(`symlinkat()`) *linkpath* is relative and *newdirfd* is a file descriptor referring to a file other than a directory.

EPERM The filesystem containing *linkpath* does not support the creation of symbolic links.

EROFS *linkpath* is on a read-only filesystem.

STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

`symlink()`
SVr4, 4.3BSD, POSIX.1-2001.

`symlinkat()`
POSIX.1-2008. Linux 2.6.16, glibc 2.4.

glibc notes

On older kernels where `symlinkat()` is unavailable, the glibc wrapper function falls back to the use of `symlink()`. When *linkpath* is a relative pathname, glibc constructs a pathname based on the symbolic link in */proc/self/fd* that corresponds to the *newdirfd* argument.

NOTES [top](#)

No checking of *target* is done.

Deleting the name referred to by a symbolic link will actually delete the file (unless it also has other hard links). If this behavior is not desired, use `link(2)`.

SEE ALSO [top](#)

`ln(1)`, `namei(1)`, `lchown(2)`, `link(2)`, `lstat(2)`, `open(2)`, `readlink(2)`, `rename(2)`, `unlink(2)`, `path_resolution(7)`, `symlink(7)`

Linux man-pages (unreleased) (date)

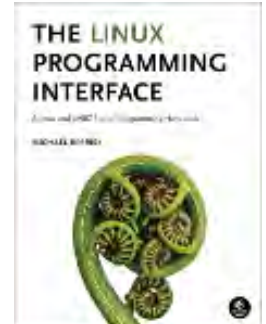
`symlink(2)`

Pages that refer to this page: [ln\(1\)](#), [fcntl\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [link\(2\)](#), [open\(2\)](#), [readlink\(2\)](#), [rename\(2\)](#), [syscalls\(2\)](#), [io_uring_prep_symlink\(3\)](#), [io_uring_prep_symlinkat\(3\)](#), [proc\(5\)](#), [inotify\(7\)](#), [signal-safety\(7\)](#), [symlink\(7\)](#), [mount\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



unlink(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [BUGS](#) | [SEE ALSO](#)

 [unlink\(2\)](#)

System Calls Manual

[unlink\(2\)](#)

NAME [top](#)

unlink, unlinkat - delete a name and possibly the file it refers to

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <unistd.h>
```

```
int unlink(const char *pathname);
```

```
#include <fcntl.h>           /* Definition of AT_* constants */  
#include <unistd.h>
```

```
int unlinkat(int dirfd, const char *pathname, int flags);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
unlinkat():  
    Since glibc 2.10:  
        _POSIX_C_SOURCE >= 200809L
```

Before glibc 2.10:
_ATFILE_SOURCE

DESCRIPTION [top](#)

unlink() deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.

If the name was the last link to a file but any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed.

If the name referred to a symbolic link, the link is removed.

If the name referred to a socket, FIFO, or device, the name for it is removed but processes which have the object open may continue to use it.

unlinkat()

The **unlinkat()** system call operates in exactly the same way as either **unlink()** or **rmdir(2)** (depending on whether or not *flags* includes the **AT_REMOVEDIR** flag) except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **unlink()** and **rmdir(2)** for a relative pathname).

If the pathname given in *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like **unlink()** and **rmdir(2)**).

If the pathname given in *pathname* is absolute, then *dirfd* is ignored.

flags is a bit mask that can either be specified as 0, or by ORing together flag values that control the operation of **unlinkat()**. Currently, only one such flag is defined:

AT_REMOVEDIR

By default, `unlinkat()` performs the equivalent of `unlink()` on *pathname*. If the `AT_REMOVEDIR` flag is specified, then performs the equivalent of `rmdir(2)` on *pathname*.

See [openat\(2\)](#) for an explanation of the need for `unlinkat()`.

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EACCES Write access to the directory containing *pathname* is not allowed for the process's effective UID, or one of the directories in *pathname* did not allow search permission. (See also [path_resolution\(7\)](#).)

EBUSY The file *pathname* cannot be unlinked because it is being used by the system or another process; for example, it is a mount point or the NFS client software created it to represent an active but otherwise nameless inode ("NFS silly renamed").

EFAULT *pathname* points outside your accessible address space.

EIO An I/O error occurred.

EISDIR *pathname* refers to a directory. (This is the non-POSIX value returned since Linux 2.1.132.)

ELOOP Too many symbolic links were encountered in translating *pathname*.

ENAMETOOLONG
pathname was too long.

ENOENT A component in *pathname* does not exist or is a dangling symbolic link, or *pathname* is empty.

ENOMEM Insufficient kernel memory was available.

ENOTDIR

A component used as a directory in *pathname* is not, in fact, a directory.

EPERM The system does not allow unlinking of directories, or unlinking of directories requires privileges that the calling process doesn't have. (This is the POSIX prescribed error return; as noted above, Linux returns **EISDIR** for this case.)

EPERM (Linux only)

The filesystem does not allow unlinking of files.

EPERM or **EACCES**

The directory containing *pathname* has the sticky bit (**S_ISVTX**) set and the process's effective UID is neither the UID of the file to be deleted nor that of the directory containing it, and the process is not privileged (Linux: does not have the **CAP_FOWNER** capability).

EPERM The file to be unlinked is marked immutable or append-only. (See [ioctl_iflags\(2\)](#).)

EROFS *pathname* refers to a file on a read-only filesystem.

The same errors that occur for **unlink()** and [rmdir\(2\)](#) can also occur for **unlinkat()**. The following additional errors can occur for **unlinkat()**:

EBADF *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EINVAL An invalid flag value was specified in *flags*.

EISDIR *pathname* refers to a directory, and **AT_REMOVEDIR** was not specified in *flags*.

ENOTDIR

pathname is relative and *dirfd* is a file descriptor referring to a file other than a directory.



STANDARDS [top](#)

POSIX.1-2008.

HISTORY [top](#)

unlink()
SVr4, 4.3BSD, POSIX.1-2001.

unlinkat()
POSIX.1-2008. Linux 2.6.16, glibc 2.4.

glibc

On older kernels where **unlinkat()** is unavailable, the glibc wrapper function falls back to the use of **unlink()** or **rmdir(2)**. When *pathname* is a relative pathname, glibc constructs a pathname based on the symbolic link in */proc/self/fd* that corresponds to the *dirfd* argument.

BUGS [top](#)

Infelicities in the protocol underlying NFS can cause the unexpected disappearance of files which are still being used.

SEE ALSO [top](#)

[rm\(1\)](#), [unlink\(1\)](#), [chmod\(2\)](#), [link\(2\)](#), [mknod\(2\)](#), [open\(2\)](#), [rename\(2\)](#), [rmdir\(2\)](#), [mkfifo\(3\)](#), [remove\(3\)](#), [path_resolution\(7\)](#), [symlink\(7\)](#)

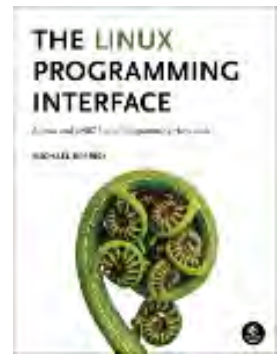
Linux man-pages (unreleased) (date) [unLink\(2\)](#)

Pages that refer to this page: [rm\(1\)](#), [unlink\(1\)](#), [chmod\(2\)](#), [close\(2\)](#), [fcntl\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [link\(2\)](#), [mkdir\(2\)](#), [mknod\(2\)](#), [open\(2\)](#), [rename\(2\)](#), [rmdir\(2\)](#), [symlink\(2\)](#), [syscalls\(2\)](#), [getcwd\(3\)](#), [io_uring_prep_unlink\(3\)](#), [io_uring_prep_unlinkat\(3\)](#), [remove\(3\)](#), [shm_open\(3\)](#), [inotify\(7\)](#), [signal-safety\(7\)](#), [symlink\(7\)](#), [unix\(7\)](#), [lsof\(8\)](#), [mount\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



remove(3) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) | [STANDARDS](#) | [HISTORY](#) | [BUGS](#) | [SEE ALSO](#)

 remove(3)

Library Functions Manual

remove(3)

NAME [top](#)

remove - remove a file or directory

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
int remove(const char *pathname);
```

DESCRIPTION [top](#)

`remove()` deletes a name from the filesystem. It calls `unlink(2)` for files, and `rmdir(2)` for directories.

If the removed name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.

If the name was the last link to a file, but any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed.

If the name referred to a symbolic link, the link is removed.

If the name referred to a socket, FIFO, or device, the name is removed, but processes which have the object open may continue to use it.

RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

The errors that occur are those for [unlink\(2\)](#) and [rmdir\(2\)](#).

ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

| Interface | Attribute | Value |
|-----------------------|---------------|---------|
| <code>remove()</code> | Thread safety | MT-Safe |

STANDARDS [top](#)

C11, POSIX.1-2008.

HISTORY [top](#)

POSIX.1-2001, C89, 4.3BSD.

BUGS [top](#)

Infelicities in the protocol underlying NFS can cause the unexpected disappearance of files which are still being used.



SEE ALSO [top](#)

[rm\(1\)](#), [unlink\(1\)](#), [link\(2\)](#), [mknod\(2\)](#), [open\(2\)](#), [rename\(2\)](#),
[rmdir\(2\)](#), [unlink\(2\)](#), [mkfifo\(3\)](#), [symlink\(7\)](#)

Linux man-pages (unreleased) **(date)**

remove(3)

Pages that refer to this page: [unlink\(2\)](#), [stdio\(3\)](#), [symlink\(7\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of [The Linux Programming Interface](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



rename(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [STANDARDS](#) | [HISTORY](#) | [BUGS](#) | [SEE ALSO](#)

 rename(2)

System Calls Manual

rename(2)**NAME** [top](#)

rename, renameat, renameat2 - change the name or location of a file

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdio.h>
```

```
int rename(const char *oldpath, const char *newpath);
```

```
#include <fcntl.h>          /* Definition of AT_* constants */  
#include <stdio.h>
```

```
int renameat(int olddirfd, const char *oldpath,  
             int newdirfd, const char *newpath);
```

```
int renameat2(int olddirfd, const char *oldpath,  
              int newdirfd, const char *newpath, unsigned int flags);
```

Feature Test Macro Requirements for glibc (see [feature_test_macros\(7\)](#)):

```
renameat():  
    Since glibc 2.10:  
        _POSIX_C_SOURCE >= 200809L
```

```
Before glibc 2.10:  
_ATFILE_SOURCE
```

```
renameat2():  
_GNU_SOURCE
```

DESCRIPTION [top](#)

rename() renames a file, moving it between directories if required. Any other hard links to the file (as created using [link\(2\)](#)) are unaffected. Open file descriptors for *oldpath* are also unaffected.

Various restrictions determine whether or not the rename operation succeeds: see ERRORS below.

If *newpath* already exists, it will be atomically replaced, so that there is no point at which another process attempting to access *newpath* will find it missing. However, there will probably be a window in which both *oldpath* and *newpath* refer to the file being renamed.

If *oldpath* and *newpath* are existing hard links referring to the same file, then **rename()** does nothing, and returns a success status.

If *newpath* exists but the operation fails for some reason, **rename()** guarantees to leave an instance of *newpath* in place.

oldpath can specify a directory. In this case, *newpath* must either not exist, or it must specify an empty directory.

If *oldpath* refers to a symbolic link, the link is renamed; if *newpath* refers to a symbolic link, the link will be overwritten.

renameat()

The **renameat()** system call operates in exactly the same way as **rename()**, except for the differences described here.

If the pathname given in *oldpath* is relative, then it is interpreted relative to the directory referred to by the file descriptor *olddirfd* (rather than relative to the current working directory of the calling process, as is done by **rename()** for a relative pathname).

If *oldpath* is relative and *olddirfd* is the special value **AT_FDCWD**, then *oldpath* is interpreted relative to the current working directory of the calling process (like **rename()**).

If *oldpath* is absolute, then *olddirfd* is ignored.

The interpretation of *newpath* is as for *oldpath*, except that a relative pathname is interpreted relative to the directory referred to by the file descriptor *newdirfd*.

See [openat\(2\)](#) for an explanation of the need for **renameat()**.

renameat2()

renameat2() has an additional *flags* argument. A **renameat2()** call with a zero *flags* argument is equivalent to **renameat()**.

The *flags* argument is a bit mask consisting of zero or more of the following flags:

RENAME_EXCHANGE

Atomically exchange *oldpath* and *newpath*. Both pathnames must exist but may be of different types (e.g., one could be a non-empty directory and the other a symbolic link).

RENAME_NOREPLACE

Don't overwrite *newpath* of the rename. Return an error if *newpath* already exists.

RENAME_NOREPLACE can't be employed together with **RENAME_EXCHANGE**.

RENAME_NOREPLACE requires support from the underlying filesystem. Support for various filesystems was added as follows:

- ext4 (Linux 3.15);
- btrfs, tmpfs, and cifs (Linux 3.17);
- xfs (Linux 4.0);
- Support for many other filesystems was added in Linux 4.9, including ext2, minix, reiserfs, jfs, vfat, and bpf.

RENAME_WHITEOUT (since Linux 3.18)

This operation makes sense only for overlay/union filesystem implementations.

Specifying **RENAME_WHITEOUT** creates a "whiteout" object at the source of the rename at the same time as performing the rename. The whole operation is atomic, so that if the rename succeeds then the whiteout will also have been created.

A "whiteout" is an object that has special meaning in union/overlay filesystem constructs. In these constructs, multiple layers exist and only the top one is ever modified. A whiteout on an upper layer will effectively hide a matching file in the lower layer, making it appear as if the file didn't exist.

When a file that exists on the lower layer is renamed, the file is first copied up (if not already on the upper layer) and then renamed on the upper, read-write layer. At the same time, the source file needs to be "whiteouted" (so that the version of the source file in the lower layer is rendered invisible). The whole operation needs to be done atomically.

When not part of a union/overlay, the whiteout appears as a character device with a {0,0} device number. (Note that other union/overlay implementations may employ different methods for storing whiteout entries; specifically, BSD union mount employs a separate inode type, **DT_WHT**, which, while supported by some filesystems available in Linux, such as CODA and XFS, is ignored by the kernel's whiteout support code, as of Linux 4.19, at least.)

RENAME_WHITEOUT requires the same privileges as creating a device node (i.e., the **CAP_MKNOD** capability).

RENAME_WHITEOUT can't be employed together with **RENAME_EXCHANGE**.

RENAME_WHITEOUT requires support from the underlying filesystem. Among the filesystems that support it are tmpfs (since Linux 3.18), ext4 (since Linux 3.18), XFS (since Linux 4.1), f2fs (since Linux 4.2), btrfs (since Linux 4.7), and ubifs (since Linux 4.9).



RETURN VALUE [top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

- EACCES** Write permission is denied for the directory containing *oldpath* or *newpath*, or, search permission is denied for one of the directories in the path prefix of *oldpath* or *newpath*, or *oldpath* is a directory and does not allow write permission (needed to update the *..* entry). (See also [path_resolution\(7\)](#).)
- EBUSY** The rename fails because *oldpath* or *newpath* is a directory that is in use by some process (perhaps as current working directory, or as root directory, or because it was open for reading) or is in use by the system (for example as a mount point), while the system considers this an error. (Note that there is no requirement to return **EBUSY** in such cases—there is nothing wrong with doing the rename anyway—but it is allowed to return **EBUSY** if the system cannot otherwise handle such situations.)
- EDQUOT** The user's quota of disk blocks on the filesystem has been exhausted.
- EFAULT** *oldpath* or *newpath* points outside your accessible address space.
- EINVAL** The new pathname contained a path prefix of the old, or, more generally, an attempt was made to make a directory a subdirectory of itself.
- EISDIR** *newpath* is an existing directory, but *oldpath* is not a directory.
- ELOOP** Too many symbolic links were encountered in resolving *oldpath* or *newpath*.
- EMLINK** *oldpath* already has the maximum number of links to it, or it was a directory and the directory containing *newpath* has the maximum number of links.

ENAMETOOLONG



oldpath or *newpath* was too long.

ENOENT The link named by *oldpath* does not exist; or, a directory component in *newpath* does not exist; or, *oldpath* or *newpath* is an empty string.

ENOMEM Insufficient kernel memory was available.

ENOSPC The device containing the file has no room for the new directory entry.

ENOTDIR

A component used as a directory in *oldpath* or *newpath* is not, in fact, a directory. Or, *oldpath* is a directory, and *newpath* exists but is not a directory.

ENOTEMPTY or **EEXIST**

newpath is a nonempty directory, that is, contains entries other than "." and "..".

EPERM or **EACCES**

The directory containing *oldpath* has the sticky bit (**S_ISVTX**) set and the process's effective user ID is neither the user ID of the file to be deleted nor that of the directory containing it, and the process is not privileged (Linux: does not have the **CAP_FOWNER** capability); or *newpath* is an existing file and the directory containing it has the sticky bit set and the process's effective user ID is neither the user ID of the file to be replaced nor that of the directory containing it, and the process is not privileged (Linux: does not have the **CAP_FOWNER** capability); or the filesystem containing *oldpath* does not support renaming of the type requested.

EROFS The file is on a read-only filesystem.

EXDEV *oldpath* and *newpath* are not on the same mounted filesystem. (Linux permits a filesystem to be mounted at multiple points, but **rename()** does not work across different mount points, even if the same filesystem is mounted on both.)

The following additional errors can occur for **renameat()** and **renameat2()**:



EBADF *oldpath* (*newpath*) is relative but *olddirfd* (*newdirfd*) is not a valid file descriptor.

ENOTDIR

oldpath is relative and *olddirfd* is a file descriptor referring to a file other than a directory; or similar for *newpath* and *newdirfd*

The following additional errors can occur for **renameat2()**:

EEXIST *flags* contains **RENAME_NOREPLACE** and *newpath* already exists.

EINVAL An invalid flag was specified in *flags*.

EINVAL Both **RENAME_NOREPLACE** and **RENAME_EXCHANGE** were specified in *flags*.

EINVAL Both **RENAME_WHITEOUT** and **RENAME_EXCHANGE** were specified in *flags*.

EINVAL The filesystem does not support one of the flags in *flags*.

ENOENT *flags* contains **RENAME_EXCHANGE** and *newpath* does not exist.

EPERM **RENAME_WHITEOUT** was specified in *flags*, but the caller does not have the **CAP_MKNOD** capability.

STANDARDS [top](#)

rename()
C11, POSIX.1-2008.

renameat()
POSIX.1-2008.

renameat2()
Linux.

HISTORY [top](#)

rename()
4.3BSD, C89, POSIX.1-2001.

renameat()

Linux 2.6.16, glibc 2.4.

renameat2()

Linux 3.15, glibc 2.28.

glibc notes

On older kernels where **renameat()** is unavailable, the glibc wrapper function falls back to the use of **rename()**. When *oldpath* and *newpath* are relative pathnames, glibc constructs pathnames based on the symbolic links in */proc/self/fd* that correspond to the *olddirfd* and *newdirfd* arguments.

BUGS[top](#)

On NFS filesystems, you can not assume that if the operation failed, the file was not renamed. If the server does the rename operation and then crashes, the retransmitted RPC which will be processed when the server is up again causes a failure. The application is expected to deal with this. See [link\(2\)](#) for a similar problem.

SEE ALSO[top](#)

[mv\(1\)](#), [rename\(1\)](#), [chmod\(2\)](#), [link\(2\)](#), [symlink\(2\)](#), [unlink\(2\)](#), [path_resolution\(7\)](#), [symlink\(7\)](#)

Linux man-pages (unreleased)

(date)

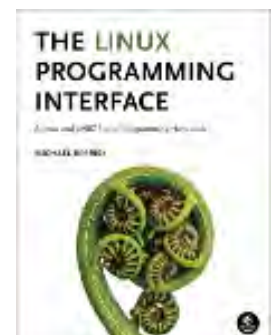
[rename\(2\)](#)

Pages that refer to this page: [exch\(1\)](#), [mv\(1\)](#), [fcntl\(2\)](#), [io_uring_enter2\(2\)](#), [io_uring_enter\(2\)](#), [link\(2\)](#), [open\(2\)](#), [rmdir\(2\)](#), [symlink\(2\)](#), [syscalls\(2\)](#), [unlink\(2\)](#), [io_uring_prep_rename\(3\)](#), [io_uring_prep_renameat\(3\)](#), [remove\(3\)](#), [cpuset\(7\)](#), [inotify\(7\)](#), [signal-safety\(7\)](#), [symlink\(7\)](#), [lsof\(8\)](#), [mount\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



ioctl(2) — Linux manual page

[NAME](#) | [LIBRARY](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [SEE ALSO](#)

 ioctl(2)

System Calls Manual

ioctl(2)**NAME** [top](#)

ioctl - control device

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <sys/ioctl.h>
```

```
int ioctl(int fd, unsigned long request, ...);
```

DESCRIPTION [top](#)

The **ioctl()** system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with **ioctl()** requests. The argument *fd* must be an open file descriptor.

The second argument is a device-dependent request code. The third argument is an untyped pointer to memory. It's traditionally **char *argp** (from the days before **void *** was valid C), and will be so named for this discussion.

An **ioctl()** *request* has encoded in it whether the argument is an *in* parameter or *out* parameter, and the size of the argument *argp* in bytes. Macros and defines used in specifying an **ioctl()** *request* are located in the file `<sys/ioctl.h>`. See NOTES.

RETURN VALUE [top](#)

Usually, on success zero is returned. A few **ioctl()** requests use the return value as an output parameter and return a nonnegative value on success. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS [top](#)

EBADF *fd* is not a valid file descriptor.

EFAULT *argp* references an inaccessible memory area.

EINVAL *request* or *argp* is not valid.

ENOTTY *fd* is not associated with a character special device.

ENOTTY The specified request does not apply to the kind of object that the file descriptor *fd* references.

VERSIONS [top](#)

Arguments, returns, and semantics of **ioctl()** vary according to the device driver in question (the call is used as a catch-all for operations that don't cleanly fit the UNIX stream I/O model).

STANDARDS [top](#)

None.

HISTORY [top](#)

Version 7 AT&T UNIX.

NOTES [top](#)

In order to use this call, one needs an open file descriptor. Often the `open(2)` call has unwanted side effects, that can be avoided under Linux by giving it the `O_NONBLOCK` flag.

ioctl structure

Ioctl command values are 32-bit constants. In principle these constants are completely arbitrary, but people have tried to build some structure into them.

The old Linux situation was that of mostly 16-bit constants, where the last byte is a serial number, and the preceding byte(s) give a type indicating the driver. Sometimes the major number was used: `0x03` for the `HDIO_*` ioctls, `0x06` for the `LP*` ioctls. And sometimes one or more ASCII letters were used. For example, `TCGETS` has value `0x00005401`, with `0x54 = 'T'` indicating the terminal driver, and `CYGETTIMEOUT` has value `0x00435906`, with `0x43 0x59 = 'C' 'Y'` indicating the cyclades driver.

Later (0.98p5) some more information was built into the number. One has 2 direction bits (`00`: none, `01`: write, `10`: read, `11`: read/write) followed by 14 size bits (giving the size of the argument), followed by an 8-bit type (collecting the ioctls in groups for a common purpose or a common driver), and an 8-bit serial number.

The macros describing this structure live in `<asm/ioctl.h>` and are `_IO(type,nr)` and `{_IOR,_IOW,_IOWR}(type,nr,size)`. They use `sizeof(size)` so that size is a misnomer here: this third argument is a data type.

Note that the size bits are very unreliable: in lots of cases they are wrong, either because of buggy macros using `sizeof(sizeof(struct))`, or because of legacy values.

Thus, it seems that the new structure only gave disadvantages: it does not help in checking, but it causes varying values for the various architectures.

SEE ALSO [top](#)

less(1) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [COMMANDS](#) | [OPTIONS](#) | [LINE EDITING](#) | [KEY BINDINGS](#) | [INPUT PREPROCESSOR](#) | [NATIONAL CHARACTER SETS](#) | [PROMPTS](#) | [SECURITY](#) | [COMPATIBILITY WITH MORE](#) | [ENVIRONMENT VARIABLES](#) | [SEE ALSO](#) | [COPYRIGHT](#) | [AUTHOR](#) | [COLOPHON](#)

 Search online pages**LESS(1)**

General Commands Manual

LESS(1)**NAME** [top](#)

less - opposite of more

SYNOPSIS [top](#)

```
less -?  
less --help  
less -V  
less --version  
less [-[+]aAbCcDeEfFgGiIjKlMmNnQqRrRsSuUVvWwX~]  
      [-b space] [-h lines] [-j line] [-k keyfile]  
      [-{oO} logfile] [-p pattern] [-P prompt] [-t tag]  
      [-T tagsfile] [-x tab,...] [-y lines] [-[z] lines]  
      [-# shift] [+ [+]cmd] [--] [filename]...
```

(See the OPTIONS section for alternate option syntax with long option names.)

DESCRIPTION [top](#)

Less is a program similar to [more\(1\)](#), but which allows backward movement in the file as well as forward movement. Also, **less** does not have to read the entire input file before starting, so with large input files it starts up faster than text editors like [vi\(1\)](#). **Less** uses termcap (or terminfo on some systems), so it can run on a variety of terminals. There is even limited support for hardcopy terminals. (On a hardcopy terminal, lines which should be printed at the top of the screen are prefixed with a caret.)

Commands are based on both [more](#) and [vi](#). Commands may be preceded by a decimal number, called N in the descriptions below. The number is used by some commands, as indicated.

COMMANDS [top](#)

In the following descriptions, ^X means control-X. ESC stands for the ESCAPE key; for example ESC-v means the two character sequence "ESCAPE", then "v".



Possible location of the history file; see the description of the LESSHISTFILE environment variable.

SEE ALSO [top](#)

[lesskey\(1\)](#), [lessecho\(1\)](#)

COPYRIGHT [top](#)

Copyright (C) 1984-2023 Mark Nudelman

less is part of the GNU project and is free software. You can redistribute it and/or modify it under the terms of either (1) the GNU General Public License as published by the Free Software Foundation; or (2) the Less License. See the file README in the less distribution for more details regarding redistribution. You should have received a copy of the GNU General Public License along with the source for less; see the file COPYING. If not, write to the Free Software Foundation, 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA. You should also have received a copy of the Less License; see the file LICENSE.

less is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

AUTHOR [top](#)

Mark Nudelman
Report bugs at <https://github.com/gsw/less/issues>.
For more information, see the less homepage at <https://greenwoodsoftware.com/less>

COLOPHON [top](#)

This page is part of the *less* (A file pager) project. Information about the project can be found at <http://www.greenwoodsoftware.com/less/>. If you have a bug report for this manual page, see <http://www.greenwoodsoftware.com/less/faq.html#bugs>. This page was obtained from the tarball less-643.tar.gz fetched from <http://www.greenwoodsoftware.com/less/download.html> on 2023-12-22. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is *not* part of the original manual page), send a mail to man-pages@man7.org

Version 643: 20 Jul 2023

LESS(1)

Pages that refer to this page: [homectl\(1\)](#), [journalctl\(1\)](#), [lessecho\(1\)](#), [lesskey\(1\)](#), [localectl\(1\)](#), [loginctl\(1\)](#), [machinectl\(1\)](#), [man\(1\)](#), [more\(1\)](#), [portablectl\(1\)](#), [quilt\(1\)](#), [systemctl\(1\)](#), [systemd\(1\)](#), [systemd-](#)



[analyze\(1\)](#), [systemd-inhibit\(1\)](#), [systemd-nspawn\(1\)](#), [systemd-vmspawn\(1\)](#), [timedatectl\(1\)](#), [userdbctl\(1\)](#), [environ\(7\)](#), [debugfs\(8\)](#), [systemd-tmpfiles\(8\)](#)

HTML rendering created 2023-12-22 by [Michael Kerrisk](#), author of *The Linux Programming Interface*.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



Materials for Topic 12: File and Directory Management

Full C Programs

- [get_file_size.c](#) - a C program outputting the size of a file whose name is passed as the 1st argument (`argv[1]`) to the program.
- [find_file_type.c](#) - a C program outputting the type of a file whose name is passed as the 1st argument (`argv[1]`) to the program.
- [permissions_changing.c](#) - a C program changing the permissions of a file and printing the update to the terminal.
- [extended_attributes.c](#) - a C program printing all the extended attributes of the file at `argv[1]` and their values.
- [switch_directories.c](#) - a C program switching the current directory `D` to `argv[1]` and then back to the previous directory `D`.
- [my_ls.c](#) - a C program listing the contents of a directory. If no additional arguments are passed, the contents of the current working directory (`.`) are printed. Otherwise, the contents of the directory whose path is stored in `argv[1]` are printed.
- [adding_links.c](#) - a C program that creates a hard link and a soft link for a file given at `argv[1]`. Both are created in

the same directory where the original file is stored. The program then calls `ls -il` to show that the 2 links were created. Finally, the program removes both links and calls `ls -il` again.

- [eject_cd_rom.c](#) - a C program using the `CDROMEJECT` request to eject the media tray from a CDROM device, which the user provides as the first argument on the program's command line.

Runnable Linux Commands

Quick Links:

- [gcc](#)
- [./short_prompt](#)
- [./long_prompt](#)
- [ls -i](#)
- [ls -i *](#)
- [cat /etc/passwd](#)
- [getent passwd "\\$\(_id -u \)"](#)
- [umask](#)
- [rmdir -r folder](#)
- [program | less](#)

-
- The command:

```
gcc -Wall -Wextra -O2 -g -o program program.c
```

compiles the C source code located inside the file `program.c`. See more details [here](#).

- The command:

```
./short_prompt
```

executes code inside a file named `.short_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).

- The command:

```
./long_prompt
```

executes code inside a file named `.long_prompt` and sources it (applies all the changes to the current session.) See more details [here](#).

- The command:

```
getfattr -d -m "*" myfile
```

shows the extended attributes of the file `myfile`.

- The command:

```
ls -i
```

shows the i-node number of every file and folder inside the current folder.

- The command:

```
ls -i *
```

shows the i-node number of every file inside the current folder, including those files within nested folders.

- The command:

```
cat /etc/passwd
```

prints the content of the `/etc/passwd` to the screen. Each

line in this file contains information about users in the Linux system, such as the username, the preferred shell interpreter, and the home directory.

- The command:

```
getent passwd "$( id -u )"
```

in case the file `/etc/passwd` doesn't include information about your user, use the command

```
getent passwd "$( id -u )"
```

to search the system for information about your user. It will print a line or a few lines including info only about your user. For example, your instructor sees the following info when calling this command:

```
briskman:x:4846:4846:Miriam Briskman (Miriam  
Briskman):/u/briskman:/bin/bash
```

The data are separated by colons (:), and there are 7 of them: (1) the username (`briskman`), (2), a placeholder for the password "`x`", (3) the user ID (`4846`), (4) the group ID (`4846`), (5) the name of the user (`Miriam Briskman (Miriam Briskman)`), (6) the home directory (`/u/briskman`), and (7) the default shell (`/bin/bash`).

- The command:

```
umask
```

prints to the terminal the permissions that the OS excludes by default from newly-created files. For example, if the output of this command is:

```
0002
```

it means that other users won't be allowed to write to or execute newly created files (but they can still read (view) the content of the file.)

- The command:

```
rmdir -r folder
```

deletes all the files and folders inside the `folder` directory and then deletes `folder` itself. Without the `-r` option, the `folder` directory won't be allowed to be deleted until you empty it from files and folders.

- The command:

```
program | less
```

displays the output from the command given by `program` in Linux's `less` content viewer. This is a great option especially when the output from `program` is very long and can't fit on the screen all at once.

To display the contents of a file (e.g., `myFile.txt`) instead of a program's output, type:

```
less myFile.txt
```

To scroll through the content that `less` displays, press the up or down arrow keys.

To exit the `less` content viewer, press the 'Q' key on the keyboard.



This website by [Miriam Briskman](#) is licensed under [CC BY-NC 4.0](#).



```
1 // A program outputing the size of a file whose name is
2 //   passed as the 1st argument (argv[1]) to the
3 //   program.
4
5 // This program is taken from Linux System Programming:
6 //   Talking Directly to the Kernel and C Library,
7 //   2nd Edition, by Love. ISBN: 978-1-44933953-1,
8 //   page 244.
9
10 #include <sys/types.h>
11 #include <sys/stat.h>
12 #include <unistd.h>
13 #include <stdlib.h>
14 #include <stdio.h>
15
16 int main (int argc, char *argv[])
17 {
18     struct stat sb; // The 'statistics' stucture.
19     int ret;
20
21     if (argc < 2)
22     {
23         fprintf (stderr,
24                 "usage: %s <file>\n",
25                 argv[0]);
26         exit (EXIT_FAILURE);
27     }
28
29     // Calling stat():
30     ret = stat (argv[1], &sb);
31     if (ret)
32     {
33         perror ("stat");
34         exit (EXIT_FAILURE);
35     }
36
37     // The st_size field carries the size of the
38     //   file in bytes.
39     printf ("File size of %s: %ld bytes.\n",
40            argv[1],
41            sb.st_size);
42
```




```
1 // A program outputing the type of a file whose name is
2 //   passed as the 1st argument (argv[1]) to the
3 //   program.
4
5 // This program is taken from Linux System Programming:
6 //   Talking Directly to the Kernel and C Library,
7 //   2nd Edition, by Love. ISBN: 978-1-44933953-1,
8 //   pages 244-245.
9
10 #include <sys/types.h>
11 #include <sys/stat.h>
12 #include <unistd.h>
13 #include <stdlib.h>
14 #include <stdio.h>
15
16 int main (int argc, char *argv[])
17 {
18     struct stat sb; // The 'statistics' structure.
19     int ret;
20
21     if (argc < 2)
22     {
23         fprintf (stderr,
24                 "usage: %s <file>\n",
25                 argv[0]);
26         exit (EXIT_FAILURE);
27     }
28
29     ret = stat (argv[1], &sb); // Calling stat().
30     if (ret)
31     {
32         perror ("stat");
33         exit (EXIT_FAILURE);
34     }
35
36     printf ("File type of %s: ", argv[1]);
37
38     // Below, we do a bitwise AND (using &) to
39     //   extract only the file type out of the
40     //   st_mode field (which also contains
41     //   information about file permission, but
42     //   we don't need them now so we ignore
```




```
43 // them.)
44 // See a more complete explanation at:
45 // https://stackoverflow.com/a/31449965/14167156
46 switch (sb.st_mode & S_IFMT)
47 {
48     case S_IFBLK: printf("block device node.\n");
49                 break;
50     case S_IFCHR: printf("character device node.\n");
51                 break;
52     case S_IFDIR: printf("directory.\n");
53                 break;
54     case S_IFIFO: printf("FIFO (a queue).\n");
55                 break;
56     case S_IFLNK: printf("symbolic link.\n");
57                 break;
58     case S_IFREG: printf("regular file.\n");
59                 break;
60     case S_IFSOCK: printf("socket.\n");
61                  break;
62     default:      printf("unknown.\n");
63 }
64
65 return EXIT_SUCCESS;
66 }
67
```



```
1  /* A program changing the permissions of a file
2     *   and printing the update to the terminal.
3     *
4     *   Miriam Briskman, 5/8/2023
5     *   CISC 3350, Brooklyn College
6     *   Licensed under CC BY-NC 4.0
7     */
8
9  #include <sys/types.h>
10 #include <sys/stat.h>
11 #include <unistd.h>
12 #include <stdlib.h>
13 #include <stdio.h>
14
15 // Function prototype:
16 void print_permissions (struct stat);
17 int int_to_permissions (int);
18
19 int main (int argc, char *argv[])
20 {
21     struct stat sb; // The 'statistics' structure.
22     int ret;
23
24     if (argc < 3)
25     {
26         fprintf (stderr,
27                 "usage: %s <file> <permissions "
28                 "as a number: 777>\n",
29                 argv[0]);
30         exit (EXIT_FAILURE);
31     }
32
33     // Calling stat():
34     ret = stat (argv[1], &sb);
35     if (ret)
36     {
37         perror ("stat");
38         exit (EXIT_FAILURE);
39     }
40
41     printf ("Current file permissions of %s:\n",
42            argv[1]);
```



```
43 print_permissions (sb);
44
45 ret = chmod (argv[1],
46             int_to_permissions (atoi (argv[2]]));
47 if (ret)
48 {
49     perror ("chmod");
50     exit (EXIT_FAILURE);
51 }
52
53 // Calling stat():
54 ret = stat (argv[1], &sb);
55 if (ret)
56 {
57     perror ("stat");
58     exit (EXIT_FAILURE);
59 }
60
61 // Print the updated permissions of the file:
62 printf ("Updated file permissions of %s:\n",
63        argv[1]);
64 print_permissions (sb);
65
66 return EXIT_SUCCESS;
67 }
68
69 // A function printing the current permissions
70 // of the file:
71 void print_permissions (struct stat sb)
72 {
73     switch (sb.st_mode & S_IFMT)
74     {
75         case S_IFBLK: printf("b");
76                     break;
77         case S_IFCHR: printf("c");
78                     break;
79         case S_IFDIR: printf("d");
80                     break;
81         case S_IFIFO: printf("f");
82                     break;
83         case S_IFLNK: printf("l");
84                     break;
85         case S_IFREG: printf("-");
```



```
86         break;
87     case S_IFSOCK: printf("s");
88         break;
89     default:      printf("u");
90 }
91
92 printf ( (sb.st_mode & S_IRUSR) ? "r" : "-");
93 printf ( (sb.st_mode & S_IWUSR) ? "w" : "-");
94 printf ( (sb.st_mode & S_IXUSR) ? "x" : "-");
95 printf ( (sb.st_mode & S_IRGRP) ? "r" : "-");
96 printf ( (sb.st_mode & S_IWGRP) ? "w" : "-");
97 printf ( (sb.st_mode & S_IXGRP) ? "x" : "-");
98 printf ( (sb.st_mode & S_IROTH) ? "r" : "-");
99 printf ( (sb.st_mode & S_IWOTH) ? "w" : "-");
100 printf ( (sb.st_mode & S_IXOTH) ? "x" : "-");
101
102 printf ("\n");
103 }
104
105 // A function that takes a numerical representation
106 //   of permissions, such as 644, and converts it
107 //   into a value that the chmod() function
108 //   understands.
109 // This value (integer) is returned by the function.
110 int int_to_permissions (int permissions)
111 {
112     int res = 0;
113
114     // First digit for the owner permissions:
115     switch ((permissions % 1000) / 100)
116     {
117         case 0: break;
118         case 1: res = res | S_IXUSR;
119                 break;
120         case 2: res = res | S_IWUSR;
121                 break;
122         case 3: res = res | S_IXUSR | S_IWUSR;
123                 break;
124         case 4: res = res | S_IRUSR;
125                 break;
126         case 5: res = res | S_IRUSR | S_IXUSR;
127                 break;
128         case 6: res = res | S_IRUSR | S_IWUSR;
```



```
129         break;
130     case 7: res = res | S_IRUSR | S_IWUSR | S_IXUSR;
131         break;
132 }
133
134 // 2nd digit for the group permissions:
135 switch ((permissions % 100) / 10)
136 {
137     case 0: break;
138     case 1: res = res | S_IXGRP;
139         break;
140     case 2: res = res | S_IWGRP;
141         break;
142     case 3: res = res | S_IXGRP | S_IWGRP;
143         break;
144     case 4: res = res | S_IRGRP;
145         break;
146     case 5: res = res | S_IRGRP | S_IXGRP;
147         break;
148     case 6: res = res | S_IRGRP | S_IWGRP;
149         break;
150     case 7: res = res | S_IRGRP | S_IWGRP | S_IXGRP;
151         break;
152 }
153
154 // 3rd digit for the others' permissions:
155 switch (permissions % 10)
156 {
157     case 0: break;
158     case 1: res = res | S_IXOTH;
159         break;
160     case 2: res = res | S_IWOTH;
161         break;
162     case 3: res = res | S_IXOTH | S_IWOTH;
163         break;
164     case 4: res = res | S_IROTH;
165         break;
166     case 5: res = res | S_IROTH | S_IXOTH;
167         break;
168     case 6: res = res | S_IROTH | S_IWOTH;
169         break;
170     case 7: res = res | S_IROTH | S_IWOTH | S_IXOTH;
171         break;
```



```
172     }  
173  
174     return res;  
175 }  
176
```

```
1  /* A program printing all the extended attributes
2  *    of the file at argv[1] and their values.
3  *
4  *    Miriam Briskman, 5/8/2023
5  *    CISC 3350, Brooklyn College
6  *    Licensed under CC BY-NC 4.0
7  */
8
9  #include <sys/types.h>
10 #include <sys/stat.h>
11 #include <sys/xattr.h>
12 #include <unistd.h>
13 #include <stdlib.h>
14 #include <stdio.h>
15
16 #define BUF_SIZE 32000
17
18 int main (int argc, char *argv[])
19 {
20     ssize_t ret;
21
22     if (argc < 2)
23     {
24         fprintf (stderr,
25                 "usage: %s <file>\n",
26                 argv[0]);
27         exit (EXIT_FAILURE);
28     }
29
30     // Setting an extended attribute:
31     int result = setxattr (argv[1],
32                           "user.slogan",
33                           "Hello World!",
34                           12,
35                           0);
36
37     if (result == -1)
38     {
39         perror ("setxattr");
40         exit (EXIT_FAILURE);
41     }
42
43     // Follow-up question:
```



```
43 // According to our lecture notes
44 //   about setattr():
45 //   (a) What does '12' above stand for?
46 //   (b) What does '0' above stand for?
47
48 // Creating a buffer of this size:
49 char * buffer = malloc (BUF_SIZE * sizeof(char));
50 if (buffer == NULL)
51 {
52     perror ("malloc");
53     exit (EXIT_FAILURE);
54 }
55
56 // Creating a buffer for the values of
57 //   attributes:
58 char * value = malloc (BUF_SIZE * sizeof(char));
59 if (value == NULL)
60 {
61     perror ("malloc");
62     exit (EXIT_FAILURE);
63 }
64
65 // Find all the extended attributes of argv[1]:
66 ret = listxattr (argv[1],
67                 buffer,
68                 BUF_SIZE);
69 if (ret == -1)
70 {
71     perror ("listxattr");
72     exit (EXIT_FAILURE);
73 }
74
75 printf ("The keys of the extended attributes "
76         "are:\n%s\n",
77         buffer);
78
79 int i = 0, j = 0, n = 1;
80
81 char temp [BUF_SIZE]; // A temporary buffer
82
83 // Print all the attributes' keys + values:
84 while (i < ret)
85 {
```




```
86     temp[j] = buffer[i];
87
88     if (buffer[i] == '\0')
89     {
90         // Get the key and value of the
91         // attribute:
92         ssize_t res = getxattr (argv[1],
93                                 temp,
94                                 (void *) value,
95                                 BUF_SIZE);
96
97         if (res == -1)
98         {
99             perror ("getxattr");
100            exit (EXIT_FAILURE);
101        }
102
103        // Print the attribute:
104        printf ("Attribute #%-d: %s ---> %s\n",
105              n,
106              temp,
107              value);
108
109        n++;
110        j = -1;
111    }
112
113    j++;
114    i++;
115
116    free (buffer);
117    free (value);
118
119    return EXIT_SUCCESS;
120 }
121
```



```
1 // A program switching the current directory D to
2 //   argv[1] and then back to the previous
3 //   directory D.
4
5 // This program is based on Linux System Programming:
6 //   Talking Directly to the Kernel and C Library,
7 //   2nd Edition, by Love. ISBN: 978-1-44933953-1,
8 //   page 265.
9
10 #include <sys/types.h>
11 #include <sys/stat.h>
12 #include <unistd.h>
13 #include <fcntl.h>
14 #include <stdlib.h>
15 #include <stdio.h>
16
17 #define BUF_SIZE 65000
18
19 int main (int argc, char * argv[])
20 {
21     int swd_fd, ret;
22     char cwd [BUF_SIZE];
23
24     if (argc < 2)
25     {
26         fprintf (stderr,
27                 "usage: %s <some directory's path>\n",
28                 argv[0]);
29         exit (EXIT_FAILURE);
30     }
31
32     // Get the current working directory:
33     if (!getcwd (cwd, BUF_SIZE))
34     {
35         perror ("getcwd");
36         exit (EXIT_FAILURE);
37     }
38
39     printf ("Current working directory: %s\n",
40            cwd);
41
42     // Open the current directory file:
```



```
43 swd_fd = open (".", O_RDONLY);
44 if (swd_fd == -1)
45 {
46     perror ("open");
47     exit (EXIT_FAILURE);
48 }
49
50 // Change to a different directory:
51 ret = chdir (argv[1]);
52 if (ret)
53 {
54     perror ("chdir");
55     exit (EXIT_FAILURE);
56 }
57
58 // Get the current working directory again:
59 if (!getcwd (cwd, BUF_SIZE))
60 {
61     perror ("getcwd");
62     exit (EXIT_FAILURE);
63 }
64
65 printf ("Updated current directory: %s\n",
66         cwd);
67
68 // Return to the saved directory:
69 ret = fchdir (swd_fd);
70 if (ret)
71 {
72     perror ("fchdir");
73     exit (EXIT_FAILURE);
74 }
75
76 // Get the current working directory
77 //   one more time:
78 if (!getcwd (cwd, BUF_SIZE))
79 {
80     perror ("getcwd");
81     exit (EXIT_FAILURE);
82 }
83
84 printf ("Finally, we are now back at: %s\n",
85         cwd);
```



```
86
87 // Close the directory's fd:
88 ret = close (swd_fd);
89 if (ret)
90 {
91     perror ("close");
92     exit (EXIT_FAILURE);
93 }
94
95 return EXIT_SUCCESS;
96 }
97
```



```
1 // A program listing the contents of a directory.
2
3 // This program is based on Linux System Programming:
4 //     Talking Directly to the Kernel and C Library,
5 //     2nd Edition, by Love. ISBN: 978-1-44933953-1,
6 //     page 270.
7
8 #include <sys/types.h>
9 #include <sys/stat.h>
10 #include <unistd.h>
11 #include <dirent.h>
12 #include <errno.h>
13 #include <stdlib.h>
14 #include <stdio.h>
15
16 int main (int argc, char *argv[])
17 {
18     struct dirent *entry;
19     DIR *dir;
20
21     if (argc == 1)
22         dir = opendir (".");
23     else
24         dir = opendir (argv[1]);
25
26     if (dir == NULL)
27     {
28         perror ("opendir");
29         exit (EXIT_FAILURE);
30     }
31
32     errno = 0;
33
34     while ((entry = readdir (dir)) != NULL)
35     {
36         printf ("%s\n", entry->d_name);
37     }
38
39     if (errno && !entry)
40     {
41         perror ("readdir");
42         exit (EXIT_FAILURE);
```



```
43     }
44
45     if (closedir (dir) == -1)
46     {
47         perror ("closedir");
48         exit (EXIT_FAILURE);
49     }
50
51     return EXIT_SUCCESS;
52 }
53
```

```
1  /* A program that creates a hard link and a soft
2  *   link for a file given at argv[1]. Both are
3  *   created in the same directory where the
4  *   original file is stored. The program then
5  *   calls "ls -il" to show that the 2 links were
6  *   created. Finally, the program removes both
7  *   links and calls "ls -il" again.
8  *
9  *   Miriam Briskman, 5/10/2023
10  *   CISC 3350, Brooklyn College
11  *   Licensed under CC BY-NC 4.0
12  */
13
14
15  #include <unistd.h> // chdir().
16  //#include <fcntl.h> //
17  #include <libgen.h> // dirname().
18  #include <string.h> // strlen(), strcpy().
19  #include <stdlib.h> // EXIT_SUCCESS, EXIT_FAILURE.
20  #include <stdio.h> // perror(), printf(), fprintf().
21
22  #define BUF_SIZE 65000
23
24  int main (int argc, char *argv[])
25  {
26      int ret;
27
28      if (argc < 2)
29      {
30          fprintf (stderr,
31                  "usage: %s <file>\n",
32                  argv[0]);
33          exit (EXIT_FAILURE);
34      }
35
36      // Creating an array to store a copy of argv[1].
37      // We do this because dirname(), which we call
38      // later, may modify argv[1], which we don't
39      // want, so we need to keep a copy!
40      char * pathcopy = malloc ((strlen (argv[1]) + 1)
41                               * sizeof(char));
42      if (pathcopy == NULL)
```



```
43     {
44         perror ("malloc");
45         exit (EXIT_FAILURE);
46     }
47
48     // Copying argv[1] to pathcopy:
49     pathcopy = strcpy (pathcopy, argv[1]);
50
51     // Change the directory to the file's
52     //     directory:
53     ret = chdir (dirname (pathcopy));
54     if (ret)
55     {
56         perror ("chdir");
57         exit (EXIT_FAILURE);
58     }
59
60     // Create a hard link:
61     ret = link (argv[1], "./hardlink");
62     if (ret)
63     {
64         perror ("link");
65         exit (EXIT_FAILURE);
66     }
67
68     // Create a soft link:
69     ret = symlink (argv[1], "./symlink");
70     if (ret)
71     {
72         perror ("symlink");
73         exit (EXIT_FAILURE);
74     }
75
76     printf ("Content of the directory "
77            "after adding the 2 links:\n");
78
79     // Calling "ls -il":
80     ret = system ("ls -il --color=auto");
81     if (ret == -1)
82     {
83         perror ("system");
84         exit (EXIT_FAILURE);
85     }
```




```
86
87 printf ("\nThe content of the file "
88         "referenced by the hard link is:\n");
89
90 // cat-ing the content of the hard link:
91 ret = system ("cat hardlink");
92 if (ret == -1)
93 {
94     perror ("system");
95     exit (EXIT_FAILURE);
96 }
97
98 printf ("\n");
99
100 printf ("The content of the file "
101         "referenced by the soft link is:\n");
102
103 ret = system ("cat symlink");
104 if (ret == -1)
105 {
106     perror ("system");
107     exit (EXIT_FAILURE);
108 }
109
110 // Removing the links:
111 ret = unlink ("./hardlink");
112 if (ret)
113 {
114     perror ("unlink");
115     exit (EXIT_FAILURE);
116 }
117
118 ret = unlink ("./symlink");
119 if (ret)
120 {
121     perror ("unlink");
122     exit (EXIT_FAILURE);
123 }
124
125 printf ("\nContent of the directory "
126         "after removing the links:\n");
127
128 // Calling "ls -il" to confirm the links were removed:
```



```
129 | ret = system ("ls -il --color=auto");
130 | if (ret == -1)
131 | {
132 |     perror ("system");
133 |     exit (EXIT_FAILURE);
134 | }
135 |
136 | return EXIT_SUCCESS;
137 | }
138 |
```



```
1 // The following program uses the CDROMEJECT request
2 // to eject the media tray from a CDROM device,
3 // which the user provides as the first argument
4 // on the program's command line.
5
6 // This program is taken from Linux System Programming:
7 // Talking Directly to the Kernel and C Library,
8 // 2nd Edition, by Love. ISBN: 978-1-44933953-1,
9 // pages 282-283.
10
11 #include <sys/types.h>
12 #include <sys/stat.h>
13 #include <fcntl.h>
14 #include <sys/ioctl.h>
15 #include <unistd.h>
16 #include <linux/cdrom.h>
17 #include <stdio.h>
18 #include <stdlib.h>
19
20 int main (int argc, char *argv[])
21 {
22     int fd, ret;
23
24     if (argc < 2)
25     {
26         fprintf (stderr,
27                 "usage: %s <device to eject>\n",
28                 argv[0]);
29         exit (EXIT_FAILURE);
30     }
31
32     // Open the CD-ROM device, read-only.
33     // O_NONBLOCK tells the kernel that we want to
34     // open the device even if there is no media
35     // present in the drive.
36     fd = open (argv[1], O_RDONLY | O_NONBLOCK);
37     if (fd < 0)
38     {
39         perror ("open");
40         exit (EXIT_FAILURE);
41     }
42
```



```
43 // Send the eject command to the CD-ROM device:
44 ret = ioctl (fd, CDROMEJECT, 0);
45 if (ret)
46 {
47     perror ("ioctl");
48     exit (EXIT_FAILURE);
49 }
50
51 ret = close (fd);
52 if (ret)
53 {
54     perror ("close");
55     exit (EXIT_FAILURE);
56 }
57
58 return EXIT_SUCCESS;
59 }
60
```

