lecture #11 — wed oct 9, 2002

- news
  - homework #3 will be posted later today
- go to recitations!
  - material in recitations will be tools that we are NOT covering in class
  - hand-outs will be given in recitation
  - some quiz questions will be on material covered in recitation
  - CRF has set it up so that you can use the machines in the CLIC lab during recitation, even if you don't have a CS account
- today
  - unix processes and threads and sockets
  - sources:
    * lecture slides by Henning Schulzrinne, cs3995, spring 2002
    * http://www.cs.rpi.edu/courses/sysprog/sockets/sock.html

what is a process? (1)

- fundamental to almost all operating systems
- = program in execution
- address space, usually separate
- program counter, stack pointer, hardware registers
- simple computer: one program, never stops

what is a process? (2)

- timesharing system: alternate between processes, interrupted by OS:
  - run on CPU
  - clock interrupt happens
  - save process state
    * registers (PC, SP, numeric)
    * memory map
    * memory (core image) → possibly swapped to disk
    * → process table
  - continue some other process

process relationships

- process tree structure: child processes
- inherit properties from parent
- processes can:
  - terminate
  - request more (virtual) memory
  - wait for a child process to terminate
  - overlay program with different one
  - send messages to other processes

processes

- in reality, each CPU can only run one program at a time
- but it appears to the user that many people are getting short (10-100 ms) time slices
  - pseudo-parallelism → *multiprogramming*
  - modeled as sequential processes
  - *context switch*

process creation

- processes are created:
  - system initialization
  - by another process
  - user request (from shell)
  - batch job (timed, Unix at or cron)
- foreground processes interact with user
- background processes don't (also called *daemons*)

unix processes — example (1)

- the `ps` command gives you information on the processes that are currently running (in unix)

```
unix$ ps -ef
     UID   PID  PPID  C    STIME TTY      TIME CMD
    root     0     0  0   Mar 31 ?        0:17 sched
    root     1     0  0   Mar 31 ?        0:09 /etc/init -
    root     2     0  0   Mar 31 ?        0:00 pageout
    root     3     0  0   Mar 31 ?       54:35 fsflush
    root   334     1  0   Mar 31 ?        0:00 /usr/lib/saf/sac -t
    root 24695     1  0 19:38:45 console  0:00 /usr/lib/saf/ttymon
    root   132     1  0   Mar 31 ?        1:57 /usr/local/sbin/ssh
    root   178     1  0   Mar 31 ?        0:01 /usr/sbin/inetd -s
  daemon    99     1  0   Mar 31 ?        0:00 /sbin/lpd
    root   139     1  0   Mar 31 ?        0:37 /usr/sbin/rpcbind
    root   119     1  0   Mar 31 ?        0:06 /usr/sbin/in.rdisc
    root   142     1  0   Mar 31 ?        0:00 /usr/sbin/keyserv
```


unix processes — example (2)

- process `0` — process scheduler ("swapper") system process
- process `1` — init process, invoked after bootstrap (`/sbin/init`)
- (note: unix `ps` is like the windows task manager)

unix process creation: forking

```
#include <sys/types.h>
#include <unistd.h>
pid_t  fork( void );
int v = 42;
if (( pid = fork() ) < 0 ) {
  perror( "fork" );
  exit( 1 );
}
else if ( pid == 0 ) {
  printf( "child %d of parent %d\n", getpid(), getppid() );
  v++;
}
else {
  sleep(10);
}
```


`fork()`

- called once, returns twice
- child: returns 0
- parent: process ID of child process
- both parent and child continue executing after fork
- child is clone of parent (copy!)
- copy-on-write: only copy page if child writes
- all file descriptors are duplicated in child
  - including file offset
  - network servers: often child and parent close unneeded file descriptors

user identities

- who we really are: real user and group ID
  - taken from `/etc/passwd` file:
    `eis2003:asvy735:95548:316:ELIZABETH I SKLAR,,,:/u/3/e/eis2003:/b`
- check file access permissions: effective user and group ID, supplementary group ID
  - supplementary IDs via group membership: `/etc/group`
  - special bits for file: "when this file is executed, set the effective IDs to be the owner of the file"
    → set-user-ID bit, set-group-ID bit
    - `/usr/bin/passwd` needs to access password files

aside: file permissions

S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

process identifiers

`pid_t getpid( void );`	process identifier
`pid_t getpgid( pid_t pid );`	process group
`pid_t getppid( void );`	parent PID
`uid_t getuid( void );`	real user ID
`uid_t geteuid( void );`	effective user ID
`gid_t getgid( void );`	real group ID
`gid_t getegid( void );`	effective group ID

process properties inherited

- user and group ids
- process group id
- controlling terminal
- setuid flag
- current working directory
- root directory (chroot)
- file creation mask
- signal masks
- close-on-exec flag
- environment
- shared memory
- resource limits

differences parent-child

- return value of `fork()`
- process IDs and parent process IDs
- accounting information
- file locks
- pending alarms

waiting for a child to terminate (1)

- asynchronous event
- SIGCHLD signal
- process can block waiting for child termination

```
pid = fork();
...
if ( wait( &status ) != pid ) {
  // something's wrong
}
```

waiting for a child to terminate (2)

```
pid_t waitpid( pid_t pid, int *statloc, int options );
```

`pid` $= -1$	any child process
`pid` > 0	specific process
`pid` $= 0$	any child with some process group id
`pid` < 0	any child with `PID = abs( pid )`

race conditions

- race = shared data and outcome depends on the order in which processes run
- e.g., parent or child runs first?
- waiting for *parent* to terminate
- generally, need some signaling mechanism
  - signals
  - stream pipes

exec: running another program

- replace current process by new program
  - text, data, heap, stack

```
#include <unistd.h>
int execl( const char *path, const char *arg0, ...,
           const char *argn, char * /*NULL*/ );
int execv( const char *path, char *const argv[] );
int execle( const char *path,char *const arg0[], ... ,
            const char *argn, char * /*NULL*/, char *const envp[]
int execve( const char *path, char *const argv[],
            char *const envp[] );
int execlp( const char *file, const char *arg0, ...,
            const char *argn, char * /*NULL*/ );
int execvp( const char *file, char *const argv[] );
```

- file: absolute (fully qualified) path or one of the `$PATH` entries

exec example

```
char *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int main( void ) {
  pid\_t pid;
  if (( pid = fork() ) < 0 ) perror( "fork error" );
  else if ( pid == 0 ) {
    if ( execle( "echoall", "echoall", "myarg1",
                 "MY ARG2", NULL, env_init ) < 0 )
      perror( "exec" );
  }
  if ( waitpid( pid, NULL, 0 ) < 0 ) perror( "wait error" );
  printf( "child done\n" );
  exit( 0 );
}
```

another alternative: use `system()` to execute a command

```
#include <stdlib.h>
int system( const char *string );
```

- invokes command string from program
- e.g., `system( "date > file" );`
- handled by shell (`/usr/bin/sh`)

threads

- process: address space + single thread of control
- sometimes want multiple threads of control (flow) in same address space
- quasi-parallel
- threads separate resource grouping and execution
- thread: program counter, registers, stack
- also called lightweight processes
- multithreading: avoid blocking when waiting for resources
  - multiple services running in parallel
- state: running, blocked, ready, terminated

why threads?

- parallel execution
- shared resources → faster communication without serialization
- easier to create and destroy than processes (100x)
- useful if some are I/O-bound → overlap computation and I/O
- easy porting to multiple CPUs

thread variants

- POSIX (pthreads)
- Sun threads (mostly obsolete)
- Java threads

creating a thread

```
int pthread_create( pthread_t *tid,
                    const pthread_attr_t *,
                    void *(*func)(void *),
                    void *arg );
```

- start function `func` with argument `arg` in new thread
- return 0 if ok, > 0 if not
- careful with `arg` argument

network server example

- lots of little requests (hundreds to thousands a second)
- simple model: new thread for each request
  $\rightarrow$ doesn't scale (memory, creation overhead)
- dispatcher reads incoming requests
- picks idle worker thread and sends it message with pointer to request
- if thread blocks, another one works on another request
- limit number of threads

worker thread

```
while ( 1 ) {
  wait for work( &buf );
  look in cache
  if not in cache
    read page from disk
  return page
}
```


leaving a thread

- threads can return value, but typically `NULL`
- just return from function (return `void *`)
- main process exits $\rightarrow$ kill all threads
- `pthread_exit( void *status );`

thread synchronization

- mutual exclusion, locks: mutex
  - protect shared or global data structures
- synchronization: condition variables
- semaphores

sockets (1)

- the client server model
  - used by most interprocess communication (i.e., two processes which will be communicating with each other)
  - one of the two processes, the *client*, connects to the other process, the *server*, typically to make a request for information
  - e.g., a person who makes a phone call to another person
- the client needs to know of the existence of and the address of the server
- but the server does not need to know the address of the client before the connection is established, or even that the client exists
- once a connection is established, both sides can send and receive information

sockets (2)

- implementation
- system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a `socket`
- a `socket` is one end of an interprocess communication channel
- the two processes each establish their own socket
- e.g., each person in a phone call needs to have a phone

sockets (3)

- establishing a server side socket
- five steps:
  1. create a socket with the `socket()` system call
  2. bind the socket to an address using the `bind()` system call
     - for a server socket on the Internet, an address consists of a port number on the host machine
  3. listen for connections with the `listen()` system call
  4. accept a connection with the `accept()` system call
  5. send and receive data, using the `read()` and `write()` system calls

sockets (4)

- establishing a client side socket
- three steps:
  1. create a socket with the `socket()` system call
  2. connect the socket to the address of the server using the `connect()` system call
  3. send and receive data, using the `read()` and `write()` system calls

socket types (1)

- when creating a socket, you need to specify
  - address domain
  - socket type
- two widely used address domains:
  - unix domain
  - Internet domain
- each has its own address format

socket types (2)

- unix domain sockets
  - communication between two processes that share a common file system
  - address is a character string which is basically an entry in the file system
- Internet domain sockets
  - communication between two processes on the Internet
  - address consists of:
    - Internet address of the host machine
      (every computer on the Internet has a unique 32-bit address, often referred to as its IP address)
    - port number (16-bit unsigned integers; the lower numbers are reserved in unix for standard services; generally, port numbers above 2000 are available)

socket types (3)

- two widely used socket types:
  - stream sockets
  - datagram sockets
- stream sockets:
  - communication is a continuous stream of characters
  - communications protocol = TCP (Transmission Control Protocol)
- datagram sockets:
  - read entire messages at once
  - communications protocol = UDP (Unix Datagram Protocol)
    (unreliable and message oriented)
- so we'll stick with TCP...