cs3157 lecture #8 notes.

mon 10 mar 2003

http://www.cs.columbia.edu/~cs3157

- news
  - homework #2 is due today
  - homework #3 will be posted today
- today's topic
  - unix processes
  - threads
  - sockets

what is a process? (1).

- fundamental to almost all operating systems
- it's a program in execution
- usually has its own address space
- also has
  - program counter (PC)
  - stack pointer (SP)
  - hardware registers

what is a process? (2).

- simple computer: *one program, never stops*
- timesharing system: *alternates between processes, interrupted by OS*
  - runs on CPU
  - clock interrupt happens
  - saves process state
    - registers (PC, SP, numeric)
    - memory map
    - memory (core image) → possibly swapped to disk
    - → process table
  - continues some other process

process relationships.

- process tree structure is hierarchical (i.e., parent and child processes)
- children 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* happens when CPU goes from one process to another

process creation.

- processes are created:
  - at system initialization
  - by another process
  - by user request (from shell)
  - by a batch job (timed, Unix at or chron)
- 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 300
    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/sshd
    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 -s
    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`)
- unix `ps` command is like the windows task manager
  - options:
    `-e` = select all processes
    `-a` = select all with a tty except session leaders
    `-f` = show full listing
    `-u` = select by effective user ID (e.g., to see your processes)
  - and many more
  - try "`man ps`" for the complete list

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:/bin/bash
```

- a few commands to try:

```
bash# who
galil      pts/11       Mar  5 10:25    (dynamic-72-230.dyn.columbia.edu)
cs3157     pts/7        Mar  9 22:41    (miles.cs.columbia.edu)
dennis     pts/12       Feb 20 16:23    (gomel.cs.columbia.edu)
cs3101     pts/14       Mar  5 12:02    (miles.cs.columbia.edu)

bash# whoami
cs3157

bash# who am i
cs3157     pts/7        Mar  9 22:41    (miles.cs.columbia.edu)

bash# id
uid=8420(cs3157) gid=98(guest)

bash# groups
guest
```


file permissions.

- unix C function: `chmod`

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
```

S_ISUID	04000	set user ID on execution
S_ISGID	02000	set group ID on execution
S_ISVTX	01000	sticky bit
S_IRUSR	00400	read by owner
S_IWUSR	00200	write by owner
S_IXUSR	00100	execute/search by owner
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

- or at unix prompt:

```
bash# chmod 777 hw1.java
```


process identifiers.

- get process ID of current or parent process

```
pid_t getpid( void );
pid_t getppid( void );
```

- get real or effective group ID of current process (i.e., real corresponds to ID of calling process and effective corresponds to set ID bit of file being executed)

```
gid_t getgid( void );
uid_t getegid( void );
```

- get real or effective user ID of the current process (i.e., real corresponds to ID of calling process and effective corresponds to set ID bit of file being executed)

```
uid_t getuid( void );
uid_t geteuid( void );
```

- all of the above require the following include files:

```
#include <sys/types.h>
#include <unistd.h>
```


unix process creation: forking.

- `fork()` creates a child process

```
#include <sys/types.h>
#include <unistd.h>

pid_t  fork( void );
```

- differs from parent in PID and PPID
- file locks are not inherited
- signals are not inherited
- function call returns 0 to child, PID of child to parent
- returns -1 (to parent) if there is an error

unix process creation: forking example.

```
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
extern int errno;
pid_t fork( void );
int v = 42;
if (( pid = fork() ) < 0 ) {
  printf( "error in fork: %s\n",strerror( errno ));
  exit( 1 );
}
else if ( pid == 0 ) { /* inside the child! */
  printf( "child %d of parent %d\n", getpid(), getppid() );
  ...
}
else ... /* inside the parent */
```


using `errno`.

- many unix C system calls use the `errno` value
- inside your program, do the following:

```
#include <errno.h>
extern int errno;
```

- then you will have access to this value which is set by various system functions (like `fork()`)
- `errno` is set to indicate something descriptive (other than -1, which is often the return value)
- use `char *strerror( int errno );` to turn the numeric error message into a text description
- you need to `#include <string.h>` to use this

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

waiting for a child to terminate.

- two functions:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait( int *status );
pid_t waitpid( pid_t pid, int *status, int options );
```

- where

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

`status` = if not null, location to store status information
`options` = OR of zero or more of the following constants:
  `WNOHANG` to return immediately if no child has exited
  `WUNTRACED` to also return for children which are stopped and whose status has not been reported

waiting for a child to terminate – example.

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

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


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

exec: running another program.

- replace current process by new program
  - text, data, heap, stack
- multiple ways of calling `exec`:

```
#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
- 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

thread functions (1).

- to create a new thread of control:

```
#include <pthread.h>
int pthread_create( pthread_t *tid,
                    const pthread_attr_t *attr,
                    void *(*start_routine)(void *),
                    void *arg );
```

- where
  `tid` = identifier of new thread
  `start_routine` = function that thread calls when it first starts, using `arg` as its first argument
  `attr` = thread attributes (=NULL for default)

- returns 0 if call is successful; non-zero otherwise
- default thread attributes: joinable (not detached) and has non-real-time scheduling policy
- see `pthread_attr_init()` for more on attributes

thread functions (2).

- thread terminates using:
  `void pthread_exit( void *retval );`

- where
  `retval` is the return value of the thread

- this function never returns!

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...

references.

- you can execute the unix `man` command on all of the unix C functions described herein
- http://www.cs.rpi.edu/courses/sysprog/sockets/sock.html