Processes

CS439: Principles of Computer Systems

January 28, 2015

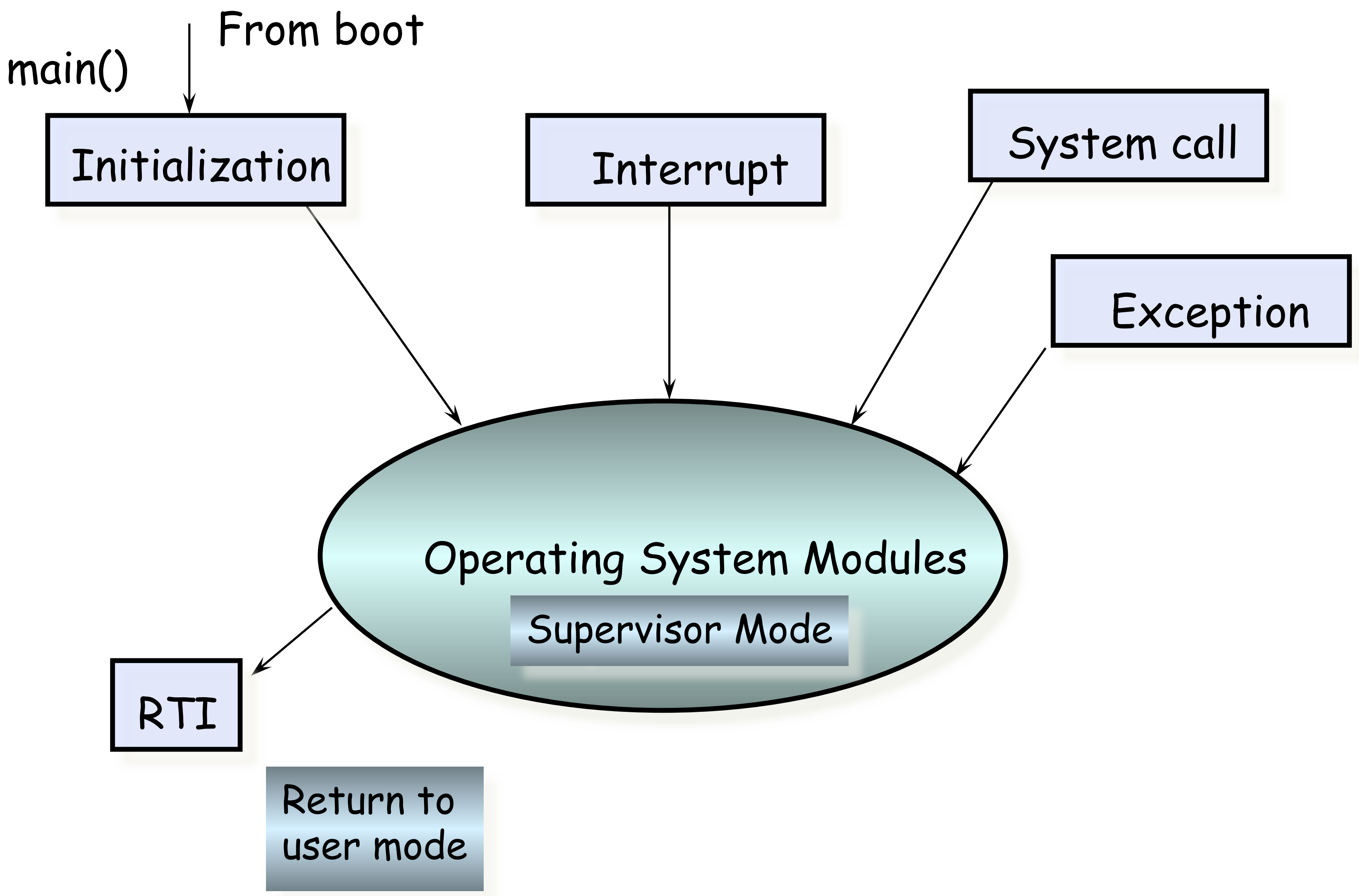
From Architecture to OS to Application and Back

Hardware	**Example OS Services**	**User Abstraction**
Processor	Process management, Scheduling, Traps, Protections, Billing, Synchronization	Process
Memory	Management, Protection, Virtual memory	Address space
I/O devices	Concurrency with CPU, Interrupt handling	Terminal, Mouse, Printer, (System Calls)
Disk	Management, Persistence	Files, File system
Distributed systems	Network security, Distributed file system	RPC system calls, Transparent file sharing

Last Time

- History Lesson
  - Hardware expensive, humans cheap
  - Hardware cheap, humans expensive
  - Hardware very cheap, humans very expensive
- Dual-mode execution helps provide protection
  - Applications execute a subset of instructions in user mode
  - OS has privileges to execute other instructions in kernel mode
  - OS gains control through exceptions, interrupts, and system calls (traps)
- A process is a unit of execution

Control Flow in an OS



Today's Agenda

- Processes
  - What are they? (again)
  - Possible execution states
  - How are they represented in the OS?
- Process Management
  - Shells, fork & exec, signals
- Summary, Announcements
- Project 0

What is a Process?

- A process is a program during execution.
  - Program = static file (image)
  - Process = executing program = program + execution state.
- A process is the basic unit of execution in an operating system
- Different processes may run different instances of the same program
  - E.g., my gcc and your gcc process both run the GNU C compiler
- At a minimum, process execution requires following resources:
  - Memory to contain the program code and data
  - A set of CPU registers to support execution

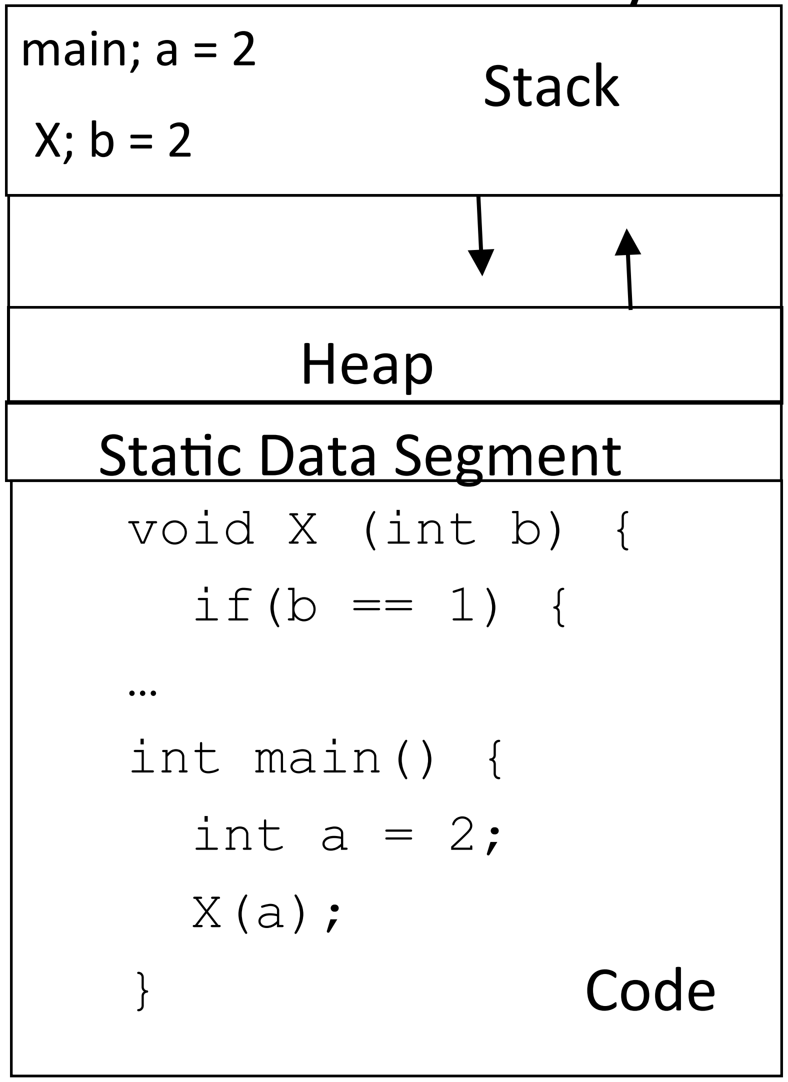
Process in Memory

What you wrote

```
void X (int b) {
  if(b == 1) {
…
int main() {
  int a = 2;
  X(a);
}
```

How does the OS manage a process?

What is in memory



```
void X (int b) {
  if(b == 1) {
…
int main() {
  int a = 2;
  X(a);
}
```

Process State

Process state consists of at least:

- The code for running the program
- The Program Counter (PC) indicating the next instruction
- An execution stack with the program’s call chain (the stack) and the stack pointer (SP)
- The static data for running the program
- Space for dynamic data (the heap), the heap pointer (HP)
- Values of CPU registers
- A set of OS resources in use (e.g., open files)
- Process identifier (PID)
- Process execution state (ready, running, etc.)

Process Life Cycle

How does the OS track this data?

The OS uses a *Process Control Block* (PCB)

– Dynamic kernel data structure kept in memory

– Represents the execution state and location of each process when it is not executing

The PCB contains:

– Process execution state, process identification number, program counter, stack pointer, general purpose registers, memory management information (HP, etc), username of owner, list of open files…

Process Control Blocks, cont.

- When a process is created, the OS allocates and initializes a new PCB and then places the PCB on the state queue

- When a process terminates, the OS deallocates the PCB

iClicker Question

When a process is waiting for I/O, what is its scheduling state?

A. Ready
B. Running
C. Blocked
D. Zombie
E. Exited

Process Management

- Process Creation
- System Calls
  - What they are, how they work, `fork()`, `exec()`, `wait()`, `kill()` (and `ps`)
- Shell

Creating a Process: What Happens

- When a program begins running, the loader:
  - reads and interprets the executable file
  - sets up the process's memory to contain the code & data from executable
  - pushes `argc` and `argv` on the stack
  - sets the CPU registers properly and calls `_start()`
- Program starts running at `_start()`

  ```
  _start(args) {
    ret = main(args);
    exit(ret)
  }
  ```

  we say "process" is now running and no longer think of "program"
- When `main()` returns, OS calls `exit()` which destroys the process and returns all resources

How to Create a Process

- One process can create other processes
  - The created processes are the *child* processes
  - The creator is the *parent* process
- In some systems, the parent defines (or donates) resources and privileges to its children
- The parent can either wait for the child to complete or continue in parallel

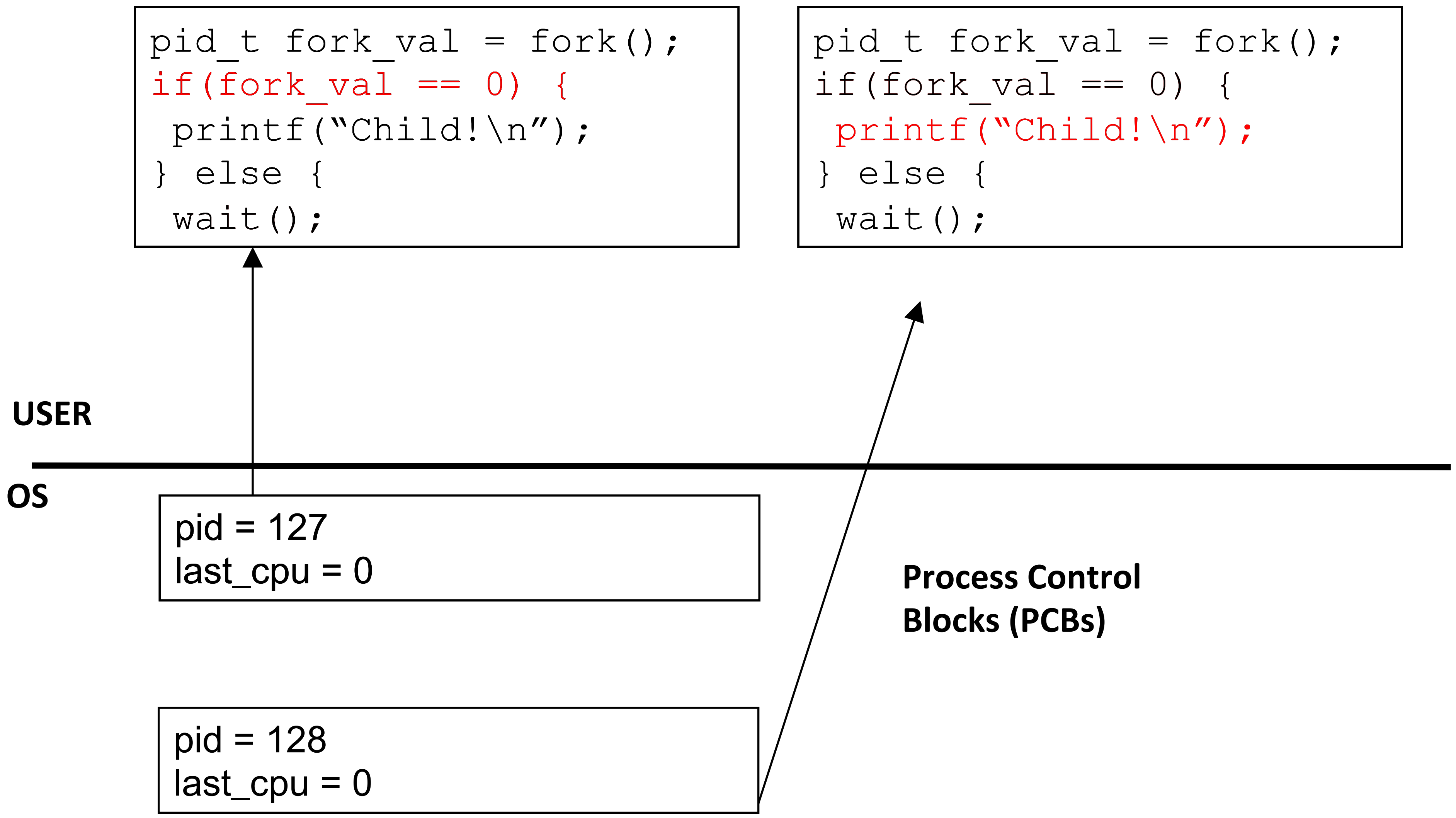
`fork()`

- In Unix, processes are created by `fork()`
- `fork()`: copies a process into an (identical) process
  - Copies variable values and program counter from parent to child
  - Returns twice: once to the parent and once to the child
  - Return value is different in the parent and child (This is the only difference!)
    - In parent, it is child process id
    - In child, it is 0
  - Both processes begin execution from the same point
    - Immediately following the call to `fork()`
  - Each process has its own memory and its own copy of each variable
    - Changes to variables in one process are not reflected in the other!

`fork()`: Pseudocode

```
pid_t fork_val = fork();                  //create a child
if((fork_val == FORKERR)                  //FORKERR is #define-d to -1
   printf("Fork failed!\n");
   return EXIT_FAILURE;
else if(fork_val == 0)                    //fork_val != child's PID
  printf("I am the child!");              //so child continues here
  return EXIT_SUCCESS;
else
  pid_t child_pid = fork_val              //parent continues here
  printf("I'm the parent.");
  int status;
  pid_t fin_pid = wait(&status);          //wait for child to finish
```

Example: `fork()`



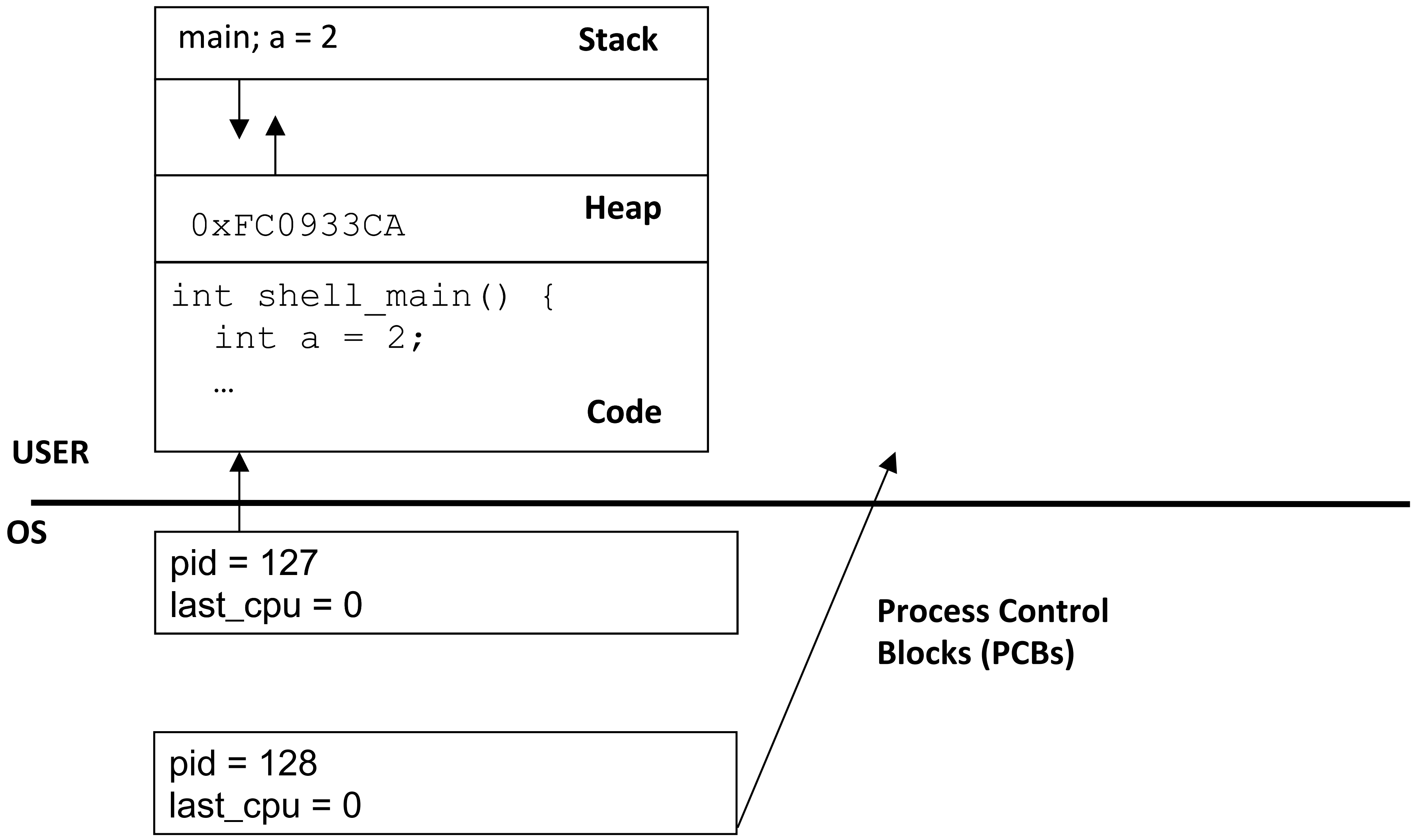
```
pid_t fork_val = fork();
if(fork_val == 0) {
 printf("Child!\n");
} else {
 wait();
```

```
pid_t fork_val = fork();
if(fork_val == 0) {
 printf("Child!\n");
} else {
 wait();
```

pid = 127
last_cpu = 0

pid = 128
last_cpu = 0

Example: `fork()`



Ummm, okay, but….

Why do I want two copies of the same process?

What if I want to start a different process?

How do I do that?

`exec()`

- Overlays a process with a new program
  - PID does not change
  - Arguments to new program may be specified
  - Code, stack, and heap are overwritten
    - Sometimes memory-mapped files are preserved
- Child processes often call `exec()` to start a new and different program
  - New program will begin at `main()`
- If call is successful, it is the *same* process, but it is running a *different* program!

fork() and exec(): Pseudocode

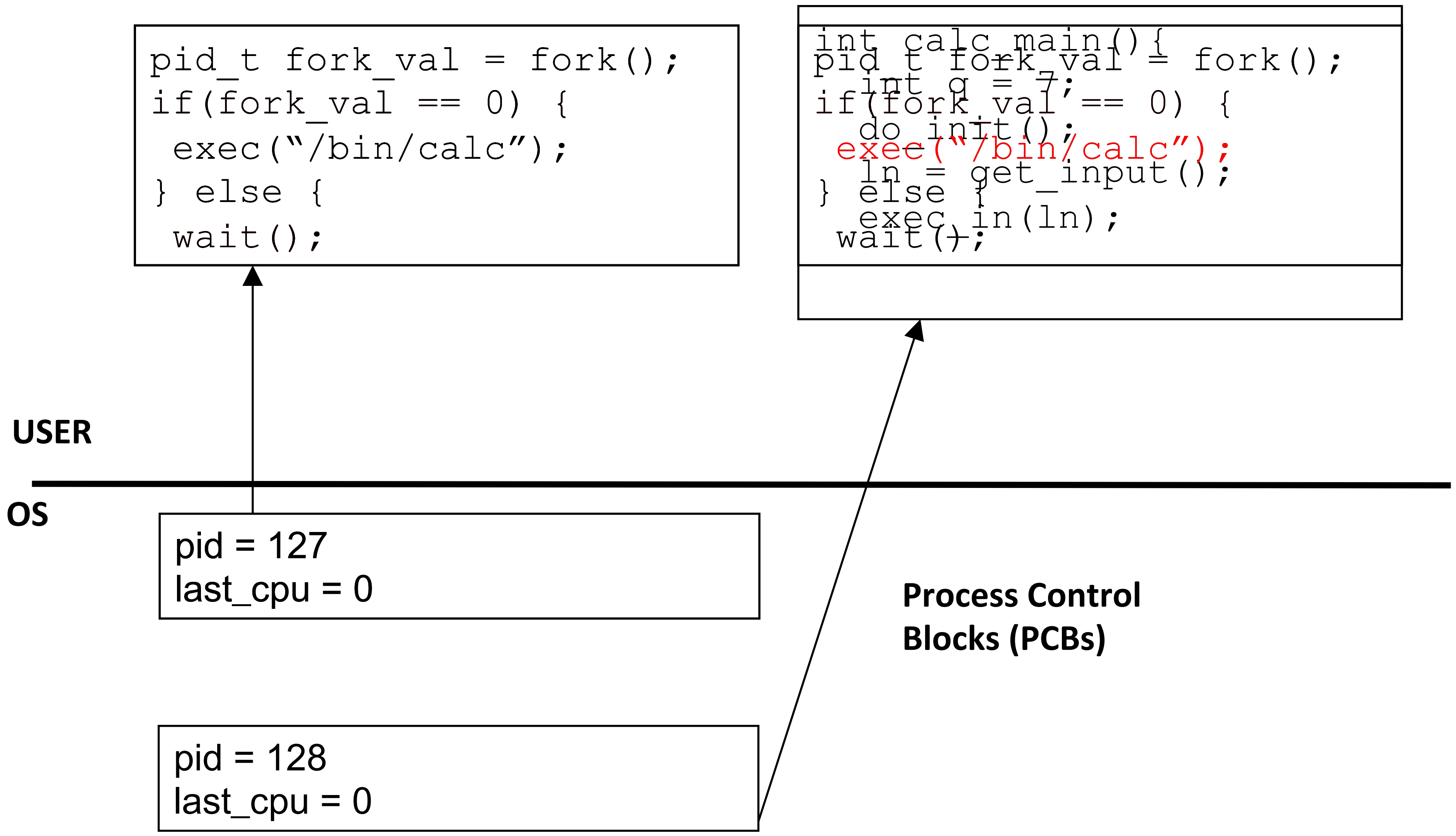
```
pid_t fork_val = fork();                        //create a child
if((fork_val = fork()) == FORKERR)
   printf("Fork failed!\n");
   return EXIT_FAILURE;
else if(fork_val == 0)                          //child continues here
  exec_status = exec("calc", argc, argv0, argv1, …);
  printf("Why would I execute?");               //should NOT execute
  return EXIT_FAILURE;
else
  pid_t child_pid = fork_val                    //parent continues here
  printf("I'm the parent.");
  int status;
  pid_t fin_pid = wait(&status);                //wait for child to finish
```

iClicker Question

What creates a process?

A. `fork()`
B. `exec()`
C. both

Example: `fork()` and `exec()`



```
pid_t fork_val = fork();
if(fork_val == 0) {
 exec("/bin/calc");
} else {
 wait();
```

```
pid_t fork_val = fork();
if(fork_val == 0) {
 exec("/bin/calc");
} else {
 wait();
```

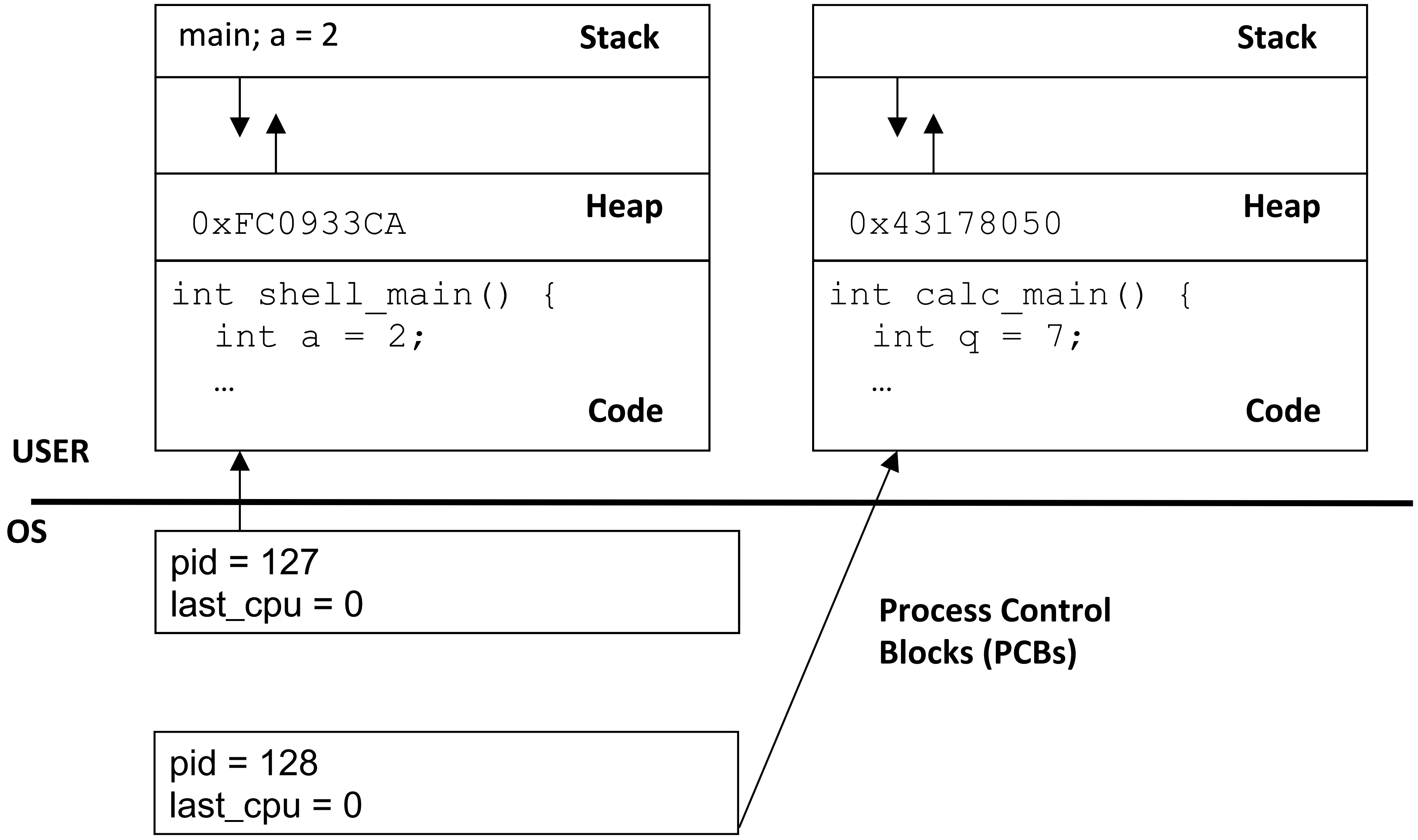
```
int calc_main(){
  int q = 7;
  do_init();
  ln = get_input();
  exec_in(ln);
```

pid = 127
last_cpu = 0

pid = 128
last_cpu = 0

Process Control Blocks (PCBs)

Example: `fork()` and `exec()`



The `wait()` System Call

`wait()` system call causes the parent process to wait for the child process to terminate

- Allows parent process to get return value from the child
- It puts parent to sleep waiting for a child's result
- when a child calls `exit()`, the OS unblocks the parent and returns the value passed by `exit()` as a result of the wait call (along with the pid of the child)
- if there are no children alive, `wait()` returns immediately
- also, if there are zombies waiting for their parents, wait() returns one of the values immediately (and deallocates the zombie)

Terminating a process: `exit()`

- After the program finishes execution, it calls `exit()`
- This system call:
  - takes the result (return value) of the program as an argument
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - *checks if parent is alive:*
    - If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the *zombie/defunct* state
    - If not, it deallocates all data structures, the process is dead
  - cleans up all waiting zombies
- Process termination is the ultimate garbage collection (resource reclamation).

Terminating a process: `kill()`

- A parent can terminate a child using `kill()`
  - `kill()` is also used for interprocess communication
- This system call:
  - Sends a *signal* to a specified process (identified by its PID)
    - SIGHUP, SIGKILL, SIGCHLD, SIGUSR1, SIGUSR2
  - The receiving process can define *signal handlers* to handle signals in a particular way
  - If a handler for that signal does not exist, the default action is taken

Signals as Virtual Interrupts

Interrupts/Exceptions	Signals
Hardware-defined	Kernel-defined
Handlers in interrupt vector in kernel	Handlers defined by user
Interrupt stack in kernel	Signal stack in user space
Interrupt masking in kernel	Signal masking in user space

Orphaned Processes

- Parent terminates before the child:
  - In some instances, the child becomes an *orphan process*
    - In UNIX, parent automatically becomes the init process
  - In other instances, all children are killed (depending on the shell)
    - Bash kills all child processes when it receives a SIGHUP
- Child can orphan itself to keep running in the background
  - nohup command (also prevents it from being killed when SIGHUP is sent)

Zombie Processes

- Process has terminated
- Parent process has not collected its status
- Dead, but not gone…

Process Control

OS must include calls to enable special control of a process:

- Priority manipulation:
  - `nice()`, which specifies base process priority (initial priority)
  - In UNIX, process priority decays as the process consumes CPU
- Debugging support:
  - `ptrace()` allows a process to be put under control of another process
  - The other process can set breakpoints, examine registers, etc.
- Alarms and time:
  - Sleep puts a process on a timer queue waiting for some number of seconds, supporting an alarm functionality

So… The Unix Shell

- When you log in to a machine running Unix, the OS creates a shell process for you to use
- Every command you type into the shell is a *child* of your shell process
  - For example, if you type "`ls`", the OS forks a new process and then execs `ls`

If you type an `&` after your command, Unix will run the process in parallel with your shell, otherwise your next shell command must wait until the first one completes.

More Shell

The shell also:

- Translates `<CTRL-C>` to the `kill()` system call with SIGINT
- Translates `<CTRL-Z>` to the `kill()` system call with SIGSTOP
- Allows input-output redirections, pipes, and a lot of other stuff that we will see later

Practical Usage: `ps` and `kill`

- If you have a process running you need to kill:
  - From the command line, type:

    ```
    ps –au <login_name>
    ```

  - Find the process you would like to terminate (the name is in the CMD column) and then determine its PID. You can do this visually or use `grep`:

    ```
    ps –au <login_name> | grep <program_name>
    ```

  - From the command line, type:

    ```
    kill -9 <PID>
    ```

Summary

A process is a unit of execution

- Processes are represented as Process Control Blocks in the OS
- At any time, a process is either New, Ready, Blocked, Running, or Terminated
- Processes are created and managed through system calls
    - System calls exist for other things, too

Announcements

- Homework 1 is due Friday, 8:45a
  - Remember that it is a two-part turnin:
    - Part 1: Posted on website, follow turnin instructions (exactly!) for electronic submission
    - Part 2: Attend _your_ discussion section, be *on time*, answer written question
- Project 0 posted this evening
  - Some reminders and help…

Project 0

- 3 parts: fork and exec, signal handling, building a mini shell
- Gets progressively more difficult
- Many system calls necessary, so prepare to read the man pages!
  - Check to be certain your man page is in the correct section (you likely want 2, but maybe 3)
- Must work with a partner
- Must keep your work in a protected directory

man Pages

- Access the online information
- Most pertinent and up-to-date information
- Use `man man` to learn about how to use the man pages
- Some common uses:

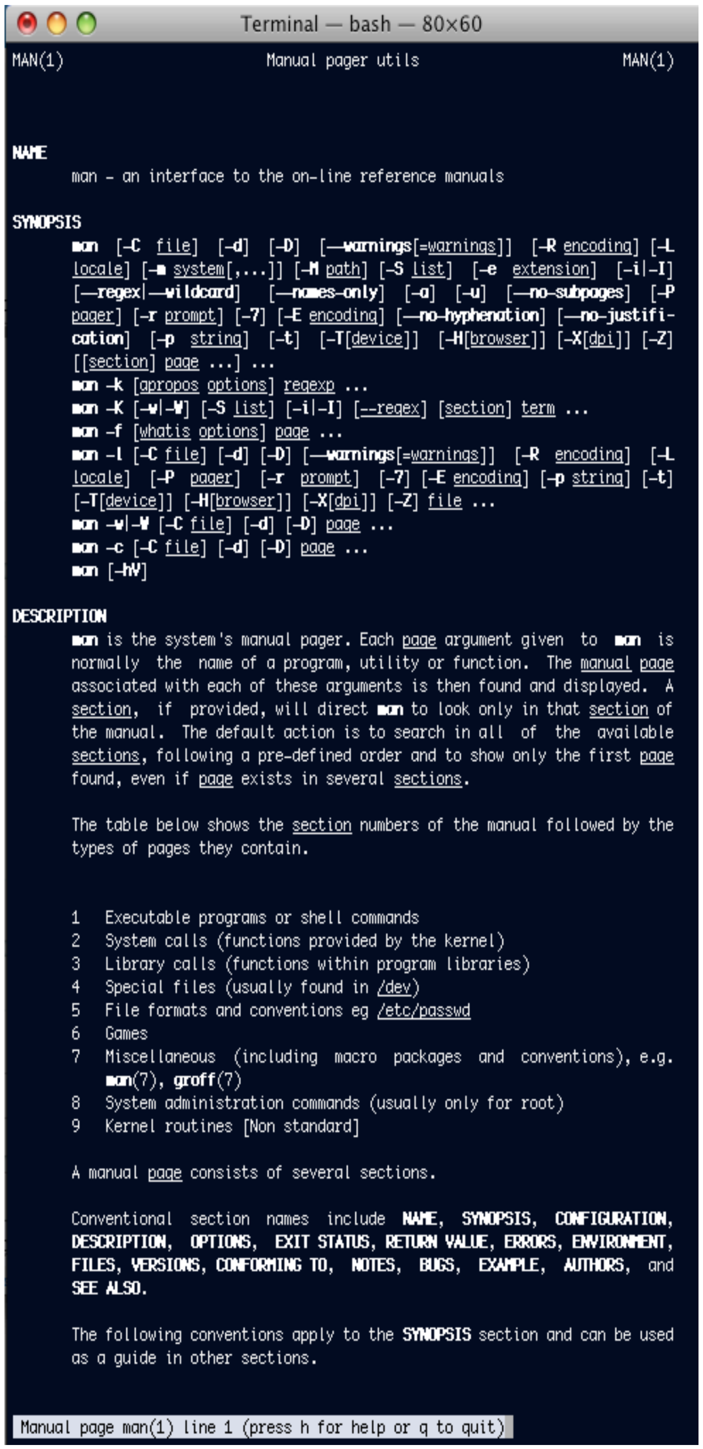
    `man 2 write`

    access the man page for the `write` system call

    `man 3 printf`

    access the man page for the `printf` command in the C library

    (note that `man printf` gets you the man page for the user command `printf`)



Terminal — bash — 80×60

```
MAN(1)                        Manual pager utils                        MAN(1)



NAME
       man - an interface to the on-line reference manuals

SYNOPSIS
       man  [-C  file]  [-d]  [-D]  [--warnings[=warnings]]  [-R encoding] [-L
       locale] [-m system[,...]] [-M path] [-S list]  [-e  extension]  [-i|-I]
       [--regex|--wildcard]   [--names-only]  [-a]  [-u]  [--no-subpages]  [-P
       pager] [-r prompt] [-7] [-E encoding] [--no-hyphenation] [--no-justifi-
       cation]  [-p  string]  [-t]  [-T[device]]  [-H[browser]] [-X[dpi]] [-Z]
       [[section] page ...] ...
       man -k [apropos options] regexp ...
       man -K [-w|-W] [-S list] [-i|-I] [--regex] [section] term ...
       man -f [whatis options] page ...
       man -l [-C file] [-d] [-D] [--warnings[=warnings]]  [-R  encoding]  [-L
       locale]  [-P  pager]  [-r  prompt]  [-7] [-E encoding] [-p string] [-t]
       [-T[device]] [-H[browser]] [-X[dpi]] [-Z] file ...
       man -w|-W [-C file] [-d] [-D] page ...
       man -c [-C file] [-d] [-D] page ...
       man [-hV]

DESCRIPTION
       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 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 eg /etc/passwd
       6   Games
       7   Miscellaneous  (including  macro  packages  and  conventions), e.g.
           man(7), groff(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, CONFORMING TO,  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.

 Manual page man(1) line 1 (press h for help or q to quit)
```

Linux Permissions

In Linux, files have permissions. You can see the permissions of a file using the `ls –al` command, which will print the "long" directory listing for all files.

```
-rw-r--r-- 1 ans grad 2476 2004-02-05 08:53 fsinfo.txt
```

The permissions are in the first column, though the first dash will be replaced by a "`d`" if the entry is a directory. The next three are the permissions for the user, then the permissions for the group, and the last are the permissions for other.

Changing Linux Permissions

Use the `chmod` command

```
chmod <permission_group><+ or -><permission type> <filename>
```

To make a file group readable:

```
    chmod g+r <filename>   //g means group
```

To make a file write-protected:

```
    chmod a-w <filename>  //a means all
```

Use `man chmod` to learn more!

Version Control

- Create a backup plan and stick to it
- Can do it by hand
- Many version control systems out there
  - Help with diffs between versions, merging
  - git, cvs, svn
- Intro to Git tomorrow
  - 3:30p-5:30p GDC 2.506 (Instructional lab)
  - In place of Robert Lynch's office hours

The Makefile

- Executes the commands it is given for its targets
  - Often, this means compilation
  - Sometimes it does not
- `make clean`
  - clean is the target typically used that says “delete all the executables”
  - when you re-compile after a make clean, every file will be recompiled

Grading

- Grading Criteria can be found on the website
  - It is linked from the project page
- 0 if it does not compile
- Pair programming required, as is log
  - Or lose half your grade
- Fill out your README! (or lose half your correctness grade)
- Create your group in Canvas (Shell groups)
- Use `make turnin` to create tarball for turnin then upload to Canvas
  - Only one partner per group turns in
  - Incorrect turnin results in a 0
- Use given command to turn in design document
  - **Must be turned in individually**
  - Name files as specified!
  - Incorrect turnin results in a 0
- Design document contains partner evaluations
  - If your score is a median of `Unsatisfactory` or less for three or more projects, your final course grade will be reduced by a letter grade

Cheating

Acceptable

- Discussing projects, homeworks, etc.
  - No writing or recording in any way
  - Use thirty minute break rule!
- Discussing debugging techniques
- Using existing public libraries
  - Citations are necessary
- Investigating OS code for public domain software (e.g., Linux)

Not acceptable

- **Looking at** (or copying and pasting) someone else's project code
  - in this class
  - in a previous version of this class
  - in similar classes around the country and globe
- Using code you wrote for this class in a previous semester unless:
  - you did not have a partner for that project, or
  - you do not have a partner for the project this semester

  (Note that neither of these options is allowed.)
- Using code that has previously been part of an academic dishonesty case

We WILL be using MOSS. And if you can find it, I can find it.

Choosing a Partner

- Scheduling!
- Work routines
  - prefer morning or night?
  - like to start early? (hint: start early)
- Priorities
  - time willing to put into projects?
- Weaknesses and Strengths
  - know Linux? know C? know systems? good at design? debugging?
- Personality Traits
  - Need at least one upbeat person
  - Need someone who will force you to begin