CMPS 311 - Operating Systems

Process, IPC, and Threads

Chapters 3 & 4

Today's Topics

- Complete OS Design Discussion
- C Tutorial/Refresher
- Introduction to Processes
- Creating Processes - POSIX

Dual Mode Execution

- Implemented using Trap/Interrupt and Mode-Bit
  - Must be supported by hardware
- Privileged Instructions cannot be executed by user code
  - Allows OS to make sure programs “play nice”.

Designing an OS

- OS is still quite complex and difficult to write!
- Design impacts both functionality and quality
- Simple Design: Monolithic
  - MS-DOS
  - Original versions of UNIX
  - Why?

Varying Priorities

- Many implementation choices for:
  - Scheduler
  - Memory Management
  - Filesystems
  - Security/Communication
  - etc.

Modern Design Principles

- Layered
  - Advantages: Layers interchangeable, easy debugging
  - Disadvantage: Interdependencies
- Micro-Kernels:
  - Advantages: Flexible, Secure
  - Disadvantage: Takes Discipline, Inefficient
- Modules:
  - Advantages: Flexible, Secure, Efficient

C Tutorial

- Different includes
  - Ex. <stdio.h>, <stdlib.h>, <math.h>

- Variable Declarations - you must declare variables at the beginning!

- C-Style I/O - File Descriptors
  - printf, fprintf
  - fgets
    - `fgets (char * str, int num, FILE * strm );`
  - fopen, fclose

C Data Structures

- Declaring a data stucture:
  - `struct myStructure x;`
  - `struct myStructure *x;`

    `x = malloc(sizeof(myStructure));`

- Defining structures:

```
typdef struct myStructure {
   int field1;
   int field2;
} myStructure;
```

Other C “Quirks”

- No native pass by reference
  - use pointers
- no `const`, use #define
- POSIX uses structures extensively.

C-Strings

- POSIX was written **long** before the C++ STL libraries and other extensions.
  - It will **not** play nicely with the string datatype.
  - Always use c-strings

c	-	s	t	r	i	n	g	\0		

http://www.cplusplus.com/reference/clibrary/cstring/

Some Rules

- Your programs can be C++, but should **not contain** the following:
  - C++/STL string classes (use cstrings)
  - No STL (vector, map, etc.)

- You **can** use
  - C++ or C input/output (i.e. cout)
  - classes (unless otherwise specified)

Program 1

- C-Demo.c
  - *Examine C-style I/O and Structures*

Processes

- Process v.s Program
  - A Process is a *running* program, with data, stack, heap, etc.
    - What does this look like in memory?

- Process States:
  - new
  - ready
  - running
  - waiting
  - terminated

Implementation

- Each Process represented by a **P**rocess **C**ontrol **B**lock
- Scheduling Queues implemented using linked lists
  - A schedule manages several queues
- Context Switch
  - Requires PCB to be saved/restored
  - What's in the PCB?

Process Creation/Termination

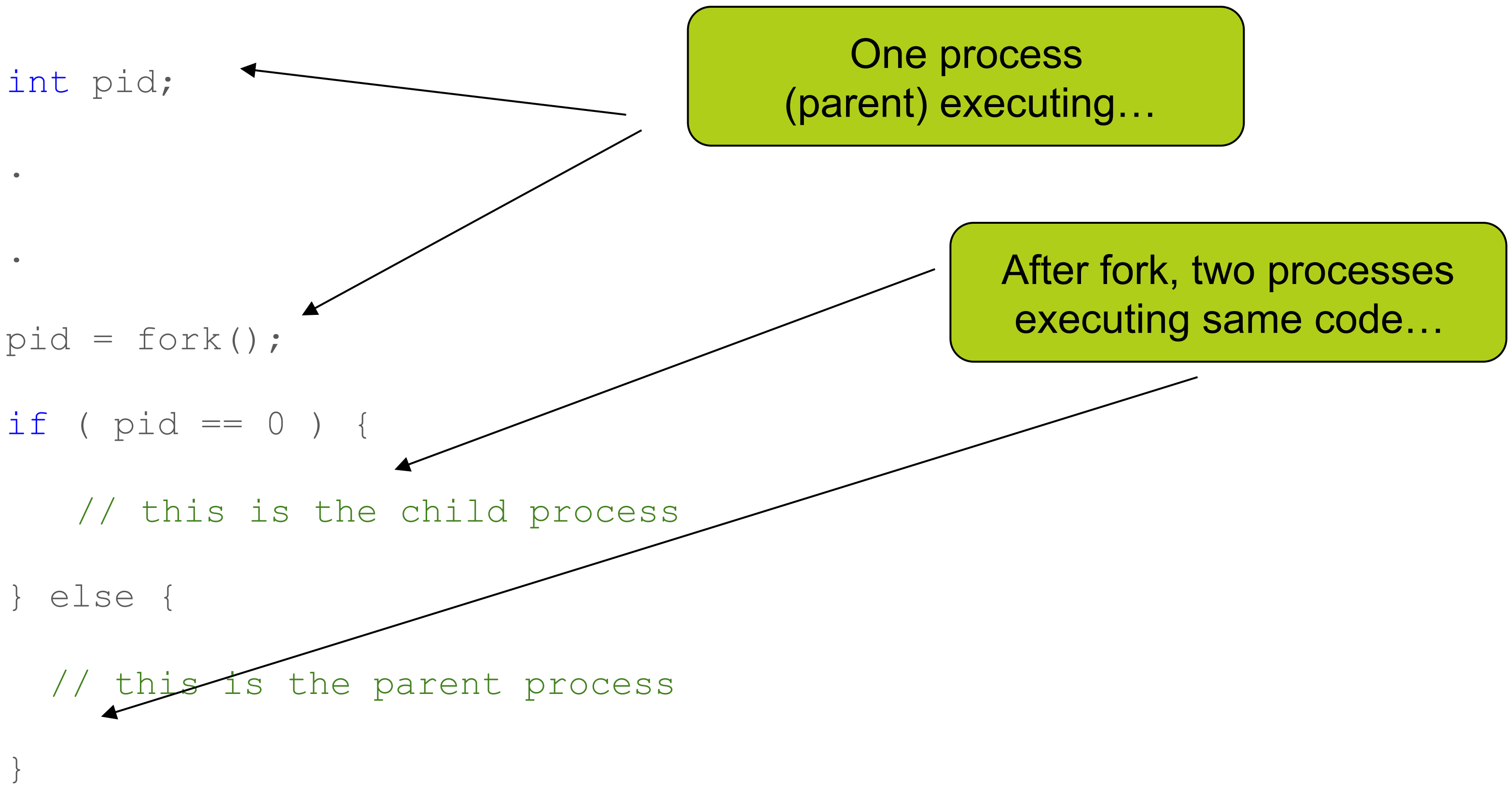
- Processes are organized in parent/child relationships
- Each process has a unique ID (integer)
- Parent may:
  - Kill off children (implicitly or explicitly)
  - Wait for children to terminate
  - Orphan its children

POSIX Process Management

- Program can create child process using `fork()` system call.

- Address space of parent is *completely* copied into child's.
- Child begins execution immediately after fork
- Parent resumes execution after fork call
- Different return values

Understanding Fork



```
int pid;
.
.
pid = fork();
if ( pid == 0 ) {
   // this is the child process
} else {
  // this is the parent process
}
```

One process (parent) executing…

After fork, two processes executing same code…

Program 2

forks.c

Examine behavior of multiple process programs.

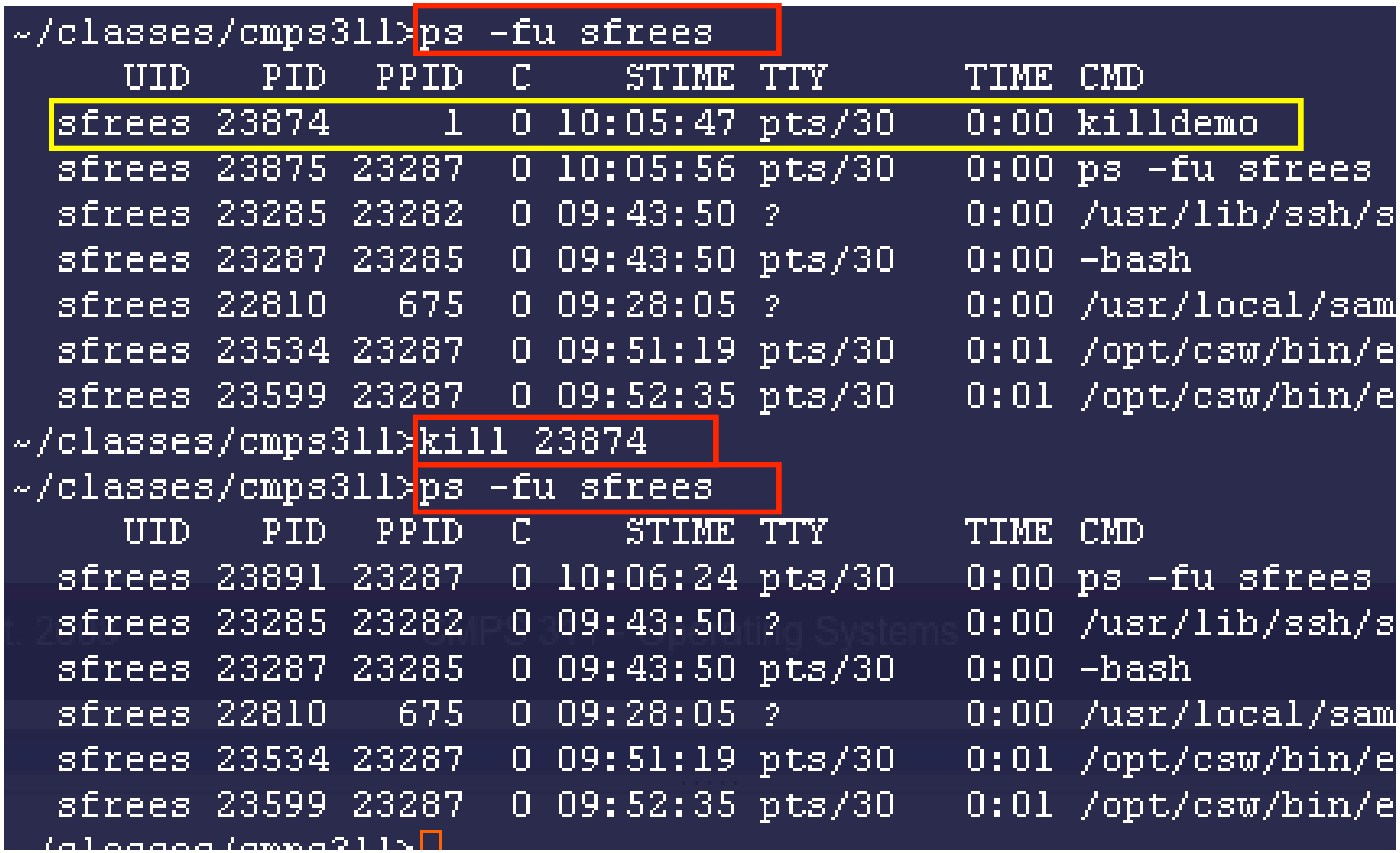
Endless Forks…

- Symptom:
  - “I can’t log into my account anymore”
  - “phobos has slowed to a crawl”
  - “Strange things are happening…”
- Common Causes:
  - You are creating too many processes(and they are not terminating)
  - You are calling fork in a loop!

Controlling Child Processes

- **Please be pro-active** - if you end up with too many rogue processes everyone suffers, you get locked out, and the sysadmin needs to fix it.
- When developing your code, always print out process creation/termination information and verify it make sense
- Use the `ps` command to monitor your active processes
- Use the `kill` command to kill off erroneous processes

Controlling Child Processes



```
~/classes/cmps311>ps -fu sfrees
     UID   PID  PPID  C    STIME TTY        TIME CMD
  sfrees 23874     1  0 10:05:47 pts/30     0:00 killdemo
  sfrees 23875 23287  0 10:05:56 pts/30     0:00 ps -fu sfrees
  sfrees 23285 23282  0 09:43:50 ?          0:00 /usr/lib/ssh/s
  sfrees 23287 23285  0 09:43:50 pts/30     0:00 -bash
  sfrees 22810   675  0 09:28:05 ?          0:00 /usr/local/sam
  sfrees 23534 23287  0 09:51:19 pts/30     0:01 /opt/csw/bin/e
  sfrees 23599 23287  0 09:52:35 pts/30     0:01 /opt/csw/bin/e
~/classes/cmps311>kill 23874
~/classes/cmps311>ps -fu sfrees
     UID   PID  PPID  C    STIME TTY        TIME CMD
  sfrees 23891 23287  0 10:06:24 pts/30     0:00 ps -fu sfrees
  sfrees 23285 23282  0 09:43:50 ?          0:00 /usr/lib/ssh/s
  sfrees 23287 23285  0 09:43:50 pts/30     0:00 -bash
  sfrees 22810   675  0 09:28:05 ?          0:00 /usr/local/sam
  sfrees 23534 23287  0 09:51:19 pts/30     0:01 /opt/csw/bin/e
  sfrees 23599 23287  0 09:52:35 pts/30     0:01 /opt/csw/bin/e
~/classes/cmps311>
```

Inter-Process Communication

- Two common types:
  - Shared Memory
  - Message Passing

- POSIX Pipes
  - Typically implemented using shared memory
  - FIFO, one way communication
  - Also partially supported on Windows

PIPES

```
int pfd[2];  // file descriptors

int result = pipe(pdf);
```

After pipe call pfd[0] contains descriptor for read end of pipe, pfd[1] contains the descriptor for write end.

Use read and write system calls to read and write pipe. These calls require an fd, which is the file descriptor (pfd[0] for read, pfd[1] for write.

```
int read(int fd, char * buffer, int maxlen)

int write (int fd, char * buffer, int numBytes);
```

Program 3

- Pipe-example.c
- *Sending information from one process to another*

Exec System Call

- Having two processes active in the same code causes confusion:
  - Rarely used.
- Normally when processes are created, they are created to run **different** programs.
- POSIX: exec*
- Win32: CreateProcess (Windows does not provide a separate fork)

Program 4

- Echo
  - *Inter-process communication with separate programs*

Next Class:

- Programming with multiple threads
  - POSIX
  - Win32

- Contact me if you have not been able to get into cs.ramapo.edu (or a linux or Mac machine)

- Homework #2 is assigned
  - Part 1 is processes (start now!)
  - Part 2 and 3 is threads (start after next class)