Process System Calls

Section 7.1, Section 7.2, and Section 7.3

System Calls

- Operating systems manage the resources of the computer
- Also coordinate the programs running on the computer
- **Kernel** is the core of the operating system

- Functions built into the operating system are **system calls**
  - stat() is a system call
- System calls vary from operating system to operating system
  - A reason why software built for one OS often cannot simply run on another OS

- **POSIX** is a standard for system calls
- Most Linux and Unix variants are POSIX compliant
- Keeps some standardization between all the different flavors of Linux and Unix

- System calls can be categorized
  - Memory management
  - Time management
  - File system calls
  - Process management
  - Signal system calls (interprocess communication)
  - Network system calls
  - Etc.

- Often library calls are built on top of system calls
  - malloc(), fread(), etc. are library calls, but probably use system calls under the hood
- Using library calls can make software more portable

Process System Calls

Section 7.3

Processes

- A program that's currently being run is a **process**
- Usually have many processes running on a computer
  - Can be seen with Task Manager in Windows, Activity Monitor in OS X, and ps in Linux
- Only one actually running on the CPU (single core) at a time
- OS manages the processes taking turns using the CPU

fork()

- **fork()** creates a clone of the program that calls it
  - So then there would be two processes of the same program running
- Sometimes the cloned process is called a **child process** and the one it was cloned from the **parent process**

- One of the main reasons fork() is used is for efficiency
- If the computer has multiple CPUs or multiple cores, a process can run on each CPU / core
- Even on a single core machine, it can increase responsiveness
  - Web server creating a process for each client that connects
  - Web browser making a process for each tab

- Each process has a **PID** that identifies
- When fork() is called, it returns an integer
  - In the parent process, the integer is the PID of the new (child) process
  - In the child process, the integer is 0
- Since the number is different in the parent and child, we can have the two behave differently

- **fork1.c**
  - Need to **#include <unistd.h>** to use fork()
  - Calls fork(), putting the integer in i
  - Prints out i

- **fork2.c**
  - Calls fork(), putting the integer in i
  - Uses an if else to choose the behavior depending on whether its the parent or child process
  - Clone gets a copy of variables that were in the parent process
  - Order that lines run may vary between runs

- **fork3.c**
  - getpid() gives back the PID of the current process
  - fork() is called twice
  - Second call happens in both the parent and first child
  - Altogether, four processes

exec()

- exec() allows us to run (execute) a Unix command from our program
- Actually swaps in the specified command in the place of the program that called exec()
- exec() is actually a family of calls
- More less do the same thing, but take slightly different arguments

- exec() can be used in conjunction with fork() to create a child process and have it run some Unix command
- exec() return -1 if they fail to run the command given to them
  - For example, if the command didn't exist
- Normally don't end up returning

- **exec1.c**
  - Uses execvp() to run the Unix date command
  - Notice the last print statement doesn't run when the execvp() succeeds
  - Try giving it a fake or misspelled command

wait()

- When we create a child process, it's unpredictable whether the child or parent will finish running first
- Can call the **wait()** function in the parent process to wait on a children processes to finish
- Basic form of synchronization

- **wait1.c**
  - Forks
  - Parent waits for child to finish