ECE 222 System Programming Concepts
Lecture notes – Process System Calls

Before we get to the process system calls, we need to cover a few basic concepts, system commands, and shell commands.

A process is a running program (in execution). There are a few system programs available to help monitor processes:

- ps – shows all processes
- top – monitor resource usage of processes
- kill – send a terminating signal to a process

For example:

```
ps –eaf | more
```

Each row in the listing represents a process. Each column provides information about the process. The rightmost column shows the shell command used to start the program. Commands in square brackets [] were not started in a shell; they were started by some other process. The second column shows the process ID, or PID. Every process has a unique PID that is used to identify it. For example:

```
ps –eaf | more      [start in one terminal]
ps –eaf | more      [start in second terminal]
kill -9 ####        [replace #### with the PID of the first "ps-eaf" command]
```

The main kernel process is listed as "init". It is owned by the root user, and has a process ID of 1. It's parent process, meaning the process that "spawned it", or started it, is 0, which is the original boot process. As you can see, a lot of other processes have a PPID of 1, meaning that init started them. Other fields show the time the process started, what terminal (tty) the process is running in (if any), and the time the process has spent actually running on the computer.

The system program "top" shows some of the same information, but sorts it by resources used, and continually updates it:

```
top
```

Probably the most important resources it monitors are memory and CPU usage.

Within a shell, there are a few commands that can be used to alter how a program is run. They all affect how the process connects its stdin stream:

&	run program in “background”; stdin stream is disconnected
CTRL-Z	suspend process currently connected to stdin
bg	restart suspended process in “background”; stdin still disconnected
fg	reconnect suspended/background process to stdin

These shell commands are handy for running programs that do not need to interact with a user; in other words, for programs that have no keyboard input. For example, suppose one wanted to start a program that was going to sort a database of 10 billion numbers. There is no need for this program, which may take hours, to use up the terminal keyboard and screen streams while it is running. We can run it in the “background”, and only concern ourselves with it after it finishes (the shell will tell us whenever a background process terminates).

If a process is running that is disconnected from stdin, and reaches a point where it is trying to read bytes from stdin (e.g. a scanf()), then the process will fault and terminate.

Now we can talk about process system calls. They answer the following question: How does a program run another program? How does a shell run a program you wrote? How does the main kernel process “init” run a terminal, which runs a shell? All these scenarios use the same three system calls:

1) fork() – create copy of current program, executing both starting at next line

[see forkdemo1.c example]

First, note that the processes started in forkdemo1 do not terminate. How can we terminate them? Using the system programs “ps” (to find the PID) and “kill”.

Second, note that the fork() function call returned different values for each process. For the parent, fork() returned the PID of the child; for the child, fork() returned 0. If the fork() fails, then -1 is returned in the parent process (the only one running since the fork failed). Understanding the return values from fork(), what do you expect to see in:

[see forkdemo2.c example]

The fork() function can be called iteratively, meaning that one process can start up multiple new processes, which can in turn start multiple new processes, etc.:

[see forkdemo3.c example]

2) exec family – replace currently executing code with called code

[see execdemo.c example]

Note that when we run the execdemo example, we do not see the line “Did it work?”. This is because the execvp() function call replaced the code being executed in the current process with the program “ls –l”. Once the program “ls –l” is done, our program is done, because that is the code it is now executing. You can think of the process as a container that is running some code (any code). When we call an exec() function, we replace the existing code with new code of our choice. It’s like giving our process a lobotomy followed by a brain transplant. Note that there are several variations on how exactly the code is replaced in the exec family of functions.

3) wait() – parent process suspends, waiting for child process to finish

[see waitdemo1.c example]

In this example, the program forks to create two processes. The main process then calls wait(), while the child process does some work (sleeping for a small delay). Once the child process finishes, the main process wakes up and returns from the wait(), finishing.

In addition to waiting for the child to finish, the wait() function returns information about the child process. The return value for wait() is the child’s PID. The single parameter passed back is the address of an int (32 bits), portions of which provide the exit()/return() value of the child process, whether or not it dumped core, and the signal ID if a signal was used to terminate the child process. We will discuss signals more next time.

In the C standard library, there is a library function that puts these things together in a convenient function that is a bit more intuitive to use:

system() – pause calling program until called program complete

[see sysdemo.c example]

It is often convenient to use a system() library call to call another program, pausing your program until the called one is done.