# CS 350 Lab 3

For Lab 3, you'll write a simple program that uses shared memory and pipes to accomplish a simple task. The purpose of this assignment is to familiarize you with some unix IPC syntax. Your program will be invoked as follows:

```
./lab3 <num-procs>
```

The program should create a shared memory segment large enough for exactly <num-procs> process ids, where <num-procs> will be a well-formed integer between 1 and 32, inclusive. The processes should form a "linear" process tree, where the top-level process creates one child, that child creates one child, and so on, until <num-procs> processes exist, including the top level process that you run from the command line. Each process should write its own process id into the shared memory object, which should be used as an array of integers, *not* as a character buffer of concatenated strings that happen to be numbers. The top-level process writes its own pid into the first array location, that process's child writes its own pid into the 2nd location, and so on. Once the leaf process is created and has written its pid into the last spot in the shared memory object, the leaf process should send a message to the top-level parent process on a named pipe. A process should realize that it is a leaf process by reading an integer from an unnamed pipe it shares with its parent. So the top-level process should share an unnamed pipe and pass <num-procs>-1 on the pipe to its child. That child (if it is not the leaf process) should decrement the value and send it to its child, and so on, until it reaches 1 in the leaf.

The message on the named pipe from the leaf child back to the top-level parent should trigger the top-level parent to print out all the process ids in order, one per line, from the shared memory object. It should then terminate all of the child processes *in the order they were created*. Look up the simple kill() system call for killing the processes. Processes should print the ALIVE and EXITING strings from Lab 2; to do so, each process will have to pause and catch a signal sent from the top-level parent. You should place the "EXITING" print statement in the signal handler, along with a call to exit(). Once the other processes have been terminated, the top-level parent should remove the shared memory segment, and exit.