CS 4760 **Operating Systems**
Assignment 2 Due Date: Feb 26, 2018

Concurrent UNIX Processes and shared memory

The goal of this homework is to become familiar with using shared memory and creating multiple processes.

Problem: In this project you will have two executable files. The first executable file, `oss`, will be in charge of launching a specific number of child processes at various times using a `fork` followed by an `exec`. `oss` should then keep track of how many children have finished executing and terminate itself only when all of its children have finished.

When you run `oss`, it should take in several command line options. First, -h, to describe how it should be run. Then a -n x option to indicate the maximum total of child processes it will ever create. Another option, -s, will indicate how many children should be allowed to exist in the system at the same time. The last two options are for input and output files, -i and -o respectively. These files are used only by `oss` and should have defaults of input.txt and output.txt. Please indicate with the -h option what the default running environment of `oss` is. I suggest the default being of a -n of 4 and a -s of 2.

So for example, if called with -n 10 -s 5 then `oss` would want to fork/exec off 5 child processes but then not create any more until one of them terminated. Once one had terminated, it would create another, continuing this until it had created a total of 10. Then at that point it would wait until all of them had terminated.

As in this project we want to get familiar with shared memory, `oss` will also be responsible for creating shared memory to store two integers and then initializing those integers to zero. This shared memory should be accessible by the children. This shared memory will act as a "clock" in that one integer represents seconds, the other one represents nanoseconds. Note that this is our "simulated" clock and will not have any correlation to real time. `oss` will be responsible for advancing this clock by incrementing it at various times as described later. Note that child processes do not ever increment the clock, they only look at it.

Implementation: When `oss` starts, it should start by parsing command line options and then allocating shared memory. It should then open the input file. The input file starts with a number, which is our default increment value for the clock (in nanoseconds). The rest of the file should consist of rows with two numbers, as the following:

```
20000
0 64867 375891
0 563620 5000000
0 2758375 375620
1 30000 35000
...
```

This file should be read as indicating that the first child should be launched no earlier than at time 0 seconds and 64867 nanoseconds in our clock. The third column is the duration which we will discuss later. The second child should be launched no earlier than at time 0 seconds and 563620 and so forth.

Your main program will then be a loop. In this loop, it should start by incrementing the clock by the constant specified in the file. It should then check if the current simulated time is high enough that it should launch a child process. If so, it should fork and then exec off the child, passing it as a command line argument the duration (third number in the file). Note that it should only launch this child process if it does not violate the -n and -s command line options. If we already have too many child processes launched, then it should just not launch a child process and do another iteration of the loop. When it launches a child process, it should write in the output file the simulated time it launched the child process, what its PID was and what duration it passed to the child. Each iteration of the loop, it should also check to see if any child processes have terminated. If any terminated, it should write to the output file the PID of the terminated process and what simulated time it terminated at.

Termination Criteria: There are several termination criteria. First, if all the children have finished and the input file does not specify needing to launch more, master should free up shared memory and terminate. It should also terminate if the maximum number of processes to launch have terminated.

In addition, I expect your program to terminate after 2 real life seconds. This can be done using a timed signal, at which point it should kill off all currently running child processes and terminate. It should also catch the ctrl-c signal, free up shared memory and then terminate all children. No matter how it terminated, master should also output the value of the shared clock to the output file. For an example of a periodic timer interrupt, you can look at p318 in the text, in the program periodicasterik.c.

The children: The task in the child executable (let us call it user) is fairly straightforward. They will be given a command line argument when they were execed off, this should be their duration. They should then read the system clock and add to it the duration (which is in nanoseconds). Then then should go into a loop, constantly checking the simulated clock in shared memory. When that time has passed, they should output to stdout their PID, the value of the system clock, and a message that they are terminating. Then they should terminate.

I suggest you implement these requirements in the following order:

1. Get a makefile that compiles two source files
2. Have master allocate shared memory, use it, then deallocate it. Make sure to check all possible error returns.
3. Get Master to fork off and exec one child and have that child attach to shared memory and read the clock. Have the child output the duration it was passed from master and then the value of the clock to test for correct launch. Master should wait for it to terminate and then output the value of the clock.
4. Put in the signal handling to terminate after 2 seconds. A good idea to test this is to simply have the child go into an infinite loop so Master will not ever terminate normally. Once this is working have it catch Ctrl-c and free up the shared memory, send a kill signal to the child and then terminate itself.
5. Lastly start setting up code to fork off child processes until the specific limits.

If you do it in this order, incrementally, you help make sure that the basic fundamentals are working before getting to the point of launching many processes.

Make sure you never have more than 20 processes in the system at any time, even if the program is invoked with n being more than 20.

Hints

You will need to set up shared memory in this project to allow the processes to communicate with each other. Please check the man pages for `shmget`, `shmctl`, `shmat`, and `shmdt` to work with shared memory.

You will also need to set up signal processing and to do that, you will check on the functions for `signal` and `abort`. If you abort a process, make sure that the parent cleans up any allocated shared memory before dying.

In case you face problems, please use the shell command `ipcs` to find out any shared memory allocated to you and free it by using `ipcrm`.

What to handin

I will want you to hand in an electronic copy of all the sources, `README`, Makefile(s), and results. Create your programs in a directory called *username*.2 where *username* is your login name on hoare. Once you are done with everything, *remove the executables and object files*, and issue the following commands:

```
chmod 700 username.2

cp -p -r username.2 /home/hauschild/cs4760/assignment2
```

Do not forget `Makefile` (with suffix or pattern rules), `RCS` (or some other version control like Git), and `README` for the assignment. If you do not use version control, you will lose 10 points. I want to see the log of how the file was modified. Omission of a `Makefile` will result in a loss of another 10 points, while `README` will cost you 5 points. Make sure that there is no shared memory left after your process finishes (normal or interrupted termination). Also, use relative path to execute the child.