CS 4760 **Operating Systems**
Assignment 3 Due Date: October 10th, 2018

Semaphores and Operating System Simulator

In this project you will be creating an empty shell of an OS simulator and doing some very basic tasks in preparation for a more comprehensive simulation later. This will require the use of fork, exec, shared memory and semaphores. While this project shares some similarities with the previous project it is different. Be leery of simply copying your previous project and trying to change it, as it can be easy to leave artifacts around. Start fresh, copying sections of code (preferably whole functions) that you want to reuse.

Operating System Simulator

This will be your main program and serve as the master process. You will start the operating system simulator (call the executable `oss`) as one main process which will start by forking off a number of starting processes and will then fork off new children as some terminate.

As we will be doing a simulatore, we will be simulating a system clock, but unlike the previous project only the master process, in this case `oss` , will be incrementing it. The child processes should be able to view this memory but will not increment it. This shared memory clock should be two integers, with one integer to hold seconds, the other integer to hold nanoseconds (instead of milliseconds as before). So if one integer has 5 and the other has 10000 then that would mean that the clock is showing 5 seconds and 10000 nanoseconds. This clock should initially be set to 0.

In addition to this shared memory clock, there should be an additional area of shared memory allocated to allow the child processes to send information to the master. Let us call this area `shmMsg`.

With the clock at zero, `oss` should fork off the appropriate number of child processes to start. Then it should enter into a loop. Every iteration of that loop, it should increment the clock (simulating time passing in our system). While in this loop, `oss` should be checking `shmMsg`. If it has been sent a message from the child, it should output the contents of that message (which should be a set of two integers, a value in our clock) to a file. If there is a message in `shmMsg`, that means that the child process is terminating, so `oss` should then wait() for that particular child to end, clear `shmMsg` and then fork off another child. This process should continue until 2 seconds have passed in the simulated system time, 100 processes in total have been generated or the executable has been running for the maximum time allotted (in real seconds). At that point the master should terminate all children and then itself, while making sure to clean up any shared memory and semaphores it is using.

Note that I did not specify how much `oss` should increment the clock on each iteration. This will depend on your implementation. Tune it so that your system has some turnover. By that I mean you want many processes to be terminating and others being launched.

The log file should at least have the following information, but can contain additional information (it would be a very good idea on initial testing to put more information):

```
OSS: Child pid is terminating at my time xx.yy because it reached mm.nn in user
OSS: Child pid is terminating at my time xx.yy because it reached mm.nn in user
OSS: Child pid is terminating at my time xx.yy because it reached mm.nn in user
...
```

with xx.yy being the time in the simulated system clock when `oss` received the message and mm.nn is the time that the user process put in `shmMsg`. Note that this log should contain the pid's of the child and not simply the placeholder word pid!

User Processes

The child processes of the `oss` are the user processes. These should be a separate executable from master, run with exec from the fork of `oss`.

This process should start by reading the simulated time system clock generated by `oss`. It should then generate a random duration number from 1 to 1000000 and add this to the time it got from the clock to give it a termination deadline. This represents the time when this process should terminate itself. So for example if it notices that the system clock is 1.37 and then it generates a number and gets a termination deadline of 1.82 then if the clock ever contains a time over 1.82 it should decide that its time is up.

It should then loop continually over a critical section of code. This critical section should be enforced through the use of semaphores.

Each iteration over the critical section, the user processes should examine the `oss` managed clock and see if its duration has passed. If while in the critical section it sees that its duration is up and there is nothing already in `shmMsg`, it should send a message to `oss` that it is going to terminate. Once it has put a message in `shmMsg` it should terminate itself, making sure to cede the critical section to any other user processes before doing so. If `shmMsg` is not empty the user process should cede the critical section and then on the next time it enters the critical section try and terminate at that time.

This checking of duration vs `oss` clock and putting a message in the `shmMsg` for master should only occur in the critical section. If a user process gets inside the critical section and sees that its duration has not passed, it should cede the critical section to someone else and attempt to get back in the critical section.

Note: Make sure that you have signal handing to terminate all processes, if needed. In case of abnormal termination, make sure to remove any resources that are used.

Your main executable should use command line arguments. You must implement at least the following command line arguments using `getopt`:

```
-h
-s x
-l filename
-t z
```

where x is the maximum number of user processes spawned (default 5) and filename is the log file used. The parameter z is the time in seconds when the master will terminate itself and all children (default 2).

Implementation

The code for `oss` and `user` processes should be compiled separately and the executables be called `oss` and `user`. The program should be executed by

```
./oss
```

Hints

I HIGHLY SUGGEST YOU DO THIS PROJECT INCREMENTALLY. Test out the command line options, then spawn the user processes but just have them all terminate. Then encode the shared memory and termination after a specified time. Then insert semaphores and enforcement of critical region. Then check to see if the child processes are able to communicate with oss. Lastly try and get oss to spawn new children as others terminate.

What to handin

Handin an electronic copy of all the sources, `README`, `Makefile`(s), and results. Create your programs in a directory called *username*.3 where *username* is your username on `hoare`. Once you are done with development and debugging, *remove the executables and object files*, and issue the following commands:

```
% cd
% ~hauschild/bin/handin cs4760 3
```

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.