**CS 4760** **Operating Systems**

Assignment 3 Due Date: March 6th, 2018

# Message Passing and Operating System Simulator

In this project you will be creating an empty shell of an OS simulator and doing some tasks in preparation for a more comprehensive simulation later. This will require shared memory and some message passing.

## 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 who will fork multiple children and then fork off new children as some terminate.

`oss` will start by first allocating shared memory for a clock. The child processes and master will be able to view and modify this clock, protected by a critical section enforced by message queues. This shared memory clock should be made up of two integers. One integer to hold seconds, the other integer to hold nanoseconds. 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 start at 0.

With the clock at zero, `oss` should then fork off the appropriate number of child processes. Then it should wait for child processes to send a message to it. When sent a message, it should output the contents of that message (which should be a set of two integers, a value in our clock) to a file and then go into the critical section to add 100 to the clock (the amount of time it took to do this in our simulated system). The master should then fork off another child. This process should continue until 2 seconds have passed in the simulated system, 100 processes in total have been generated or the executable has been running for the maximum time allotted. At that point the master should terminate all children and then itself.

As we will discuss later, the children when in their critical section will be incrementing the clock. This amount will be a constant defining by you to be high enough to cause turnover in the system. Tune it so that your system has some turnover (processes terminating and being generated).

The log file should look as follows:

```
Master: Child pid xxx is terminating at my time xx.xx because it reached xx.xx, which lived for time xx.xx
Master: Creating new child pid xxx at my time xx.xx
Master: Child pid xxx is terminating at my time xx.xx because it reached xx.xx which lived for time xx.xx
Master: Creating new child pid xxx at my time xx.xx
Master: Child pid xxx is terminating at my time xx.xx because it reached xx.xx which lived for time xx.xx
Master: Creating new child pid xxx at my time xx.xx
...
```

## 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 our simulated system clock. It should then generate a random duration number from 1 to some maximum number of nanosecond constant (A reasonable value might be something like 1000000). This represents how long this child should run.

It should then loop continually over a critical section of code. This critical section should be enforced through the user of message passing with msgsnd and msgrcv.

In this critical section of code done using message passing, the user process should read the `oss` clock and then add an increment of time to it. This increment is the amount of work that the process has done this round. Add that number of nanoseconds to the clock. If this additional time would put it past its duration, then that indicates that the process is going to terminate. It should then send a message to `oss` that it is going to terminate. Then the child should terminate. The message sent to `oss` should consist of the current simulated system clock time that it decided to terminate on, as well as the total amount of seconds.nanoseconds that this child incremented the clock. Note that if the random interval of work exceeds the maximum allotted lifetime, that it should not work those extra nanoseconds and so only increment the clock enough to hit that maximum time.

This checking of duration vs `oss` clock and messaging to 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. Make sure that your process automatically terminates itself with a timed interrupt.

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 slave 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 20).

## 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 slave processes but just have them all terminate. Then encode the shared memory and termination after a specified time. Then do message passing and enforcement of critical region. Lastly try and get master 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 login name on hoare. Once you are done with everything, *remove the executables and object files as well as log files*, and issue the following commands:

```
chmod 700 username.3

cp -r username.3 /home/hauschild/cs4760/assignment3
```

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.