CS 4760 **Operating Systems**
Assignment # 4 Due Date: October 26, 2017

Process Scheduling

In this project, you will simulate the process scheduling part of an operating system. You will implement time-based scheduling, ignoring almost every other aspect of the OS.

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 at random times. The randomness will be simulated by a logical clock that will also be updated by `oss`.

In the beginning, `oss` will allocate shared memory for system data structures, including process control block for each user process. The process control block is a fixed size structure and contains information on managing the child process scheduling. Notice that since it is a simulator, you will not need to allocate space to save the context of child processes. But you must allocate space for scheduling-related items such as total CPU time used, total time in the system, time used during the last burst, and process priority, if any. The process control block resides in shared memory and is accessible to the child. Since we are limiting ourselves to 20 processes in this class, you should allocate space for up to 18 process control blocks. Also create a bit vector, local to `oss`, that will help you keep track of the process control blocks (or process IDs) that are already taken.

`oss` will create user processes at random intervals, say every second on an average. The clock itself will get incremented in terms of nanoseconds. I'll suggest that you have two unsigned integers for the clock; one will show the time in seconds and the other will show the time in nanoseconds, offset from the beginning of a second. The clock will be accessible to every process and hence, in shared memory. It will be advanced only by `oss` though it can be observed by all the children to see the current time.

`oss` will run concurrently with all the user processes. After it sets up all the data structures, it enters a loop where it generates and schedules processes. It *generates* a new process by allocating and initializing the process control block for the process and then, `fork`s the process. The child process will `execl` the binary.

Advance the logical clock by 1.xx seconds in each iteration of the loop where xx is the number of nanoseconds. xx will be a random number in the interval [0,1000] to simulate some overhead activity for each iteration.

A new process should be generated every 1 second, on an average. So, you should generate a random number between 0 and 2 assigning it to time to create new process. If your clock has passed this time since the creation of last process, generate a new process (and `execl` it). If the process table is already full, do not generate any more processes.

Scheduling Algorithm. `oss` will *select* the process to be run and *schedule* it for execution. It will select the process by using a scheduling algorithm with the following features:

Your scheduler will consist of two round-robin queues, one being a high-priority queue and another a low-priority queue. Jobs running in the high-priority queue should have a time quantum $q/2$ specified by you (could be hardcoded, but make it a constant). Jobs in the low-priority queue should have a time of q.

Everytime a process is created, it will be created with either high or low priority. If it is high priority, it will always remain high priority and vice-versa. The scheduler will always pick from the highest priority queue that has a job in it, so if there is a high priority job somewhere in the high-priority queue, no low priority job will get to run until it is complete. Due to this, make the chance that a job is picked as high priority to be very low.

Every time a process is scheduled, that user process will generate a random number in the range $[0, 3]$ where 0 indicates that the process terminates, 1 indicates that the process terminates at its time quantum, 2 indicates that process starts to wait for an event that will last for $r.s$ seconds where r and s are random numbers with range $[0, 5]$ and $[0, 1000]$ respectively, and 3 indicates that the process gets preempted after using a particular percent p of its assigned quantum, where p is a random number in the range $[1, 99]$. The probability of each of these should be hardcoded, but does not need to be even (ie: Could be more likely to use its time quantum than to be interrupted).

The process will be *dispatched* by putting the process ID and the time quantum in a predefined location in shared memory. The user process will pick up the quantum value from the shared memory and schedule itself to *run* for that long a time.

User Processes

All user processes are alike but simulate the system by performing some tasks at random times. The user process will keep checking in the shared memory location if it has been scheduled and once done, it will start to run. It should generate a random number to check whether it will use the entire quantum, or only a part of it (a binary random number will be sufficient for this purpose). If it has to use only a part of the quantum, it will generate a random number in the range [0,quantum] to see how long it runs. After its allocated time (completed or partial), it updates its process control block by adding to the accumulated CPU time. It joins the ready queue at that point and sends a signal on a semaphore so that `oss` can schedule another process.

While running, generate a random number again to see if the process is completed. This should be done if the process has accumulated at least 50 milliseconds. If it is done, the message should be conveyed to `oss` who should *remove* its process control block.

Your simulation should end with a report on average turnaround time and average wait time for the processes. Also include how long the CPU was idle.

Make sure that you have signal handing to terminate all processes, if needed. In case of abnormal termination, make sure to remove shared memory and semaphores, if you used them.

I HIGHLY SUGGEST YOU DO THIS PROJECT INCREMENTALLY. Have master create the PCB and work with just one child user process and get the clock working. Once this is working, then start working on maybe a one queue scheduler doing round-robin. Then start adding to the system. Do not try to do everything at once and be stuck with no idea what is failing.

Log Output

Your program should send enough output to a log file such that it is possible for me to determine its operation. For example:

```
OSS: Generating process with PID 3 (Low priority) and putting it in queue 1 at time 0:5000015
OSS: Dispatching process with PID 2 from queue 1 at time 0:5000805,
OSS: total time this dispatch was 790 nanoseconds
OSS: Receiving that process with PID 2 ran for 400000 nanoseconds
OSS: Putting process with PID 2 into queue 1
OSS: Dispatching process with PID 3 from queue 1 at time 0:5401805,
OSS: total time this dispatch was 1000 nanoseconds
OSS: Receiving that process with PID 3 ran for 270000 nanoseconds,
OSS: not using its entire time quantum
OSS: Putting process with PID 3 into queue 1
OSS: Generating process with PID 5 (High priority) and putting it in queue 0 at time 0:5402505
OSS: Dispatching process with PID 5 from queue 0 at time 0:5548375
OSS: total time spent in dispatch was 7000 nanoseconds
etc
```

I suggest not simply appending to previous logs, but start a new file each time. Also be careful about infinite loops that could generate excessively long log files. So for example, keep track of total lines in the log file and terminate writing to the log if it exceeds 10000 lines.

Note that the above log was using arbitrary numbers, so your times spent in dispatch could be quite different.

Deliverables

Handin an electronic copy of all the sources, `README`, Makefile(s), and results. Create your programs in a directory called *username*.4 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.4

cp -r username.4 /home/hauschild/cs4760/assignment4
```

Do not forget `Makefile` (with suffix rules), version control, and `README` for the assignment. If you do not use version control, you will lose 10 points. Omission of a `Makefile` (with suffix rules) will result in a loss of another 10 points, while `README` will cost you 5 points.