CS 4760 **Operating Systems**
Assignment # 4 Due Date: April 11th, 2019

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. In this project we will be using message queues for synchronization.

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 a process control table with a 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, your local simulated pid, and process priority, if any. The process control block resides in shared memory and is accessible to the children. 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.

The clock, as in previous projects, will be in nanoseconds, so use two unsigned integers for the clock, one for seconds and other for nanoseconds. `oss` will create user processes at random intervals, say every second on an average. 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 create user processes at random intervals (of simulated time), so you will have two constants, let us call them *maxTimeBetweenNewProcsNS* and *maxTimeBetweenNewProcsSecs* and will launch a new user process based on a random time interval from 0 to those constants. 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. I would suggest setting these constants initially to spawning a new process about every 1 second, but you can experiment with this later to keep the system busy. New processes that are created can have one of two scheduling classes, either real-time or a normal user process, and will remain in that class for their lifetime. There should be constant representing the percentage of time a process is launched as a normal user process or a real-time one. While this constant is specified by you, it should be heavily weighted to generating mainly user processes.

`oss` will be in control of all concurrency, so starting there will be no processes in the system but it will have a time in the future where it will launch a process. If there are no processes currently ready to run in the system, it should increment the clock until it is the time where it should launch a process. It should then set up that process, generate a new time where it will launch a process and then using a message queue, schedule a process to run by sending it a message. It should then wait for a message back from that process that it has finished its task. If your process table is already full when you go to generate a process, just skip that generation, but do determine another time in the future to try and generate a new process.

Scheduling Algorithm. Assuming you have more than one process in your simulated system, `oss` will *select* a process to run and *schedule* it for execution. It will select the process by using a scheduling algorithm with the following features:

Implement a version of multi-level scheduling. There are four scheduling queues, the highest level processes are in a round-robin queue and if a process is at this priority, the process will stay at this priority until it terminates. The other three queues are set up as a multilevel feedback queue, so we have a round-robin highest-priority queue(queue 0), a high-priority (queue 1), a medium-priority (queue 2) and a low-priority queue (queue 3). The queues will each have an associated time quantum, with queue 0 having a timeslice of that quantum, and each lower priority queue having twice the timeslice of the next highest priority. The base time quantum is determined by you as a constant, let us say something like 10 milliseconds, but certainly could be experimented with.

As we will discuss when we talk about the details on user processes, user processes will have two scheduling classes, either real-time or normal user processes. Real-time processes will enter the round-robin queue to start (queue 0), all other processes will start in queue 1. When a process has used up its timeslice, the queue that it will go into for the next round of a possible run depends on its level. Processes in the round-robin queue will stay there, while processes that were in one of the lowerlvl queues will drop in priority if they

completed their timeslice. If a process is leaving a wait queue and is going back into a ready state, then it would go into the highest priority queue of its associated class.

When `oss` has to pick a process to schedule, it will look for the highest priority occupied queue and schedule the process at the head of this queue. The process will be *dispatched* by sending the process a message using a message queue indicating how much of a time slice it has to run. Note that this scheduling itself takes time, so before launching the process the `oss` should increment the clock for the amount of work that it did, let us say from 100 to 10000 nanoseconds.

User Processes

All user processes are alike but simulate the system by performing some work that will take some random time. The user processes will wait on receiving a message giving them a time slice and then it will start to run.

Every time a process is scheduled, it should have some probability to terminate, this probability determined by some constant. I would suggest this probability be fairly small to start. If it would terminate, it would of course use some random amount of its timeslice before terminating. It should indicate to `oss` that it has done so.

Once it has decided that it will not terminate, then we have to determine if it will use its entire timeslice or if it will get blocked on an event. This should be determined by a random number. If it uses up its timeslice, this information should be conveyed to master. Otherwise, the 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 p of its assigned quantum, where p is a random number in the range $[1, 99]$. As this could happen for multiple processes, this will require a blocked queue, checked by `oss` every time it makes a decision on scheduling to see if it should wake up these processes and put them back in the appropriate queues. Note that the simulated work of moving a process from a blocked queue to a ready queue would take more time than a normal scheduling decision so it would make sense to increment the system clock to indicate this.

Your simulation should end with a report on average turnaround time, average wait time and average sleep for the processes. Also include how long the CPU was idle with no ready processes.

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 message queues.

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 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 2
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: Dispatching process with PID 1 from queue 1 at time 0:5402505,
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.

I HIGHLY SUGGEST YOU DO THIS PROJECT INCREMENTALLY. A suggested way to break it down...

- Have master create a process control table with one user process (of real-time class) to verify it is working

- Schedule the one user process over and over, logging the data
- Create the round robin queue, add additional user processes, making all user processes alternate in it
- Keep track of and output statistics like throughput, idle time, etc
- Implement an additional user class and the multi-level feedback queue.
- Add the chance for user processes to be blocked on an event, keep track of statistics on this

Do not try to do everything at once and be stuck with no idea what is failing.

Termination Criteria

`oss` should stop generating processes if it has already generated 100 processes or if more than 3 real-life seconds have passed. If you stop adding new processes, the system should eventually empty of processes and then it should terminate. What is important is that you tune your parameters so that the system has processes in all the queues at some point and that I can see that in the log file. As discussed previously, ensure that appropriate statistics are displayed.

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 -p -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.