CMP SCI 4760 | **Operating Systems**
Project 4 | Due Date: October 30, 2023

Process Scheduling

Purpose

The goal of this homework is to learn about process scheduling inside an operating system. You will work on the specified scheduling algorithm and simulate its performance.

Task

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 particular, this project will involve a main executable managing the execution of concurrent user processes just like a scheduler/dispatcher does in an OS. You may use message queues or semaphores for synchronization, depending on your choice.

Operating System Simulator

The operating system simulator, or OSS, will be your main program and serve as the master process. You will start the 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`. As we are simulating a scheduler, only one process (either `oss` or one of the children/user processes) will be in running state at any given time, while all the other processes wait to receive a message, or for a semaphore, for them to be scheduled.

In the beginning, `oss` will allocate shared memory for system data structures, including a process table with a process control block for each user process. The process control block is a fixed size structure and contains information to manage 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. However, 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 currently in use.

`oss` simulates the passing of time by using a simulated system clock. The clock is stored in two unsigned integers in shared memory, one which stores seconds and the other nanoseconds. `oss` will create user processes at random intervals, say every second on an average of the simulated clock. While the simulated clock (the two integers) is viewable by the child processes, it should only be advanced (changed) by `oss`, and only in strict ways as described in the rest of this document.

Note that `oss` simulates the passing of time in the system by adding time to the clock and it is the only process that would change the clock. At various times, a user process will simulate doing some work by running. When this happens, after it cedes control back to `oss`, `oss` would update the simulated system clock by the appropriate amount. This is achieved by generating a random number within the child that will be sent back to `oss` through either shared memory or a message queue. In the same fashion, if `oss` does something that should take some time if it was a real operating system, it should increment the clock by a small amount to indicate the time it spent.

In this project, there will be two types of user processes: normal user processes that run on a normal priority and real-time processes that are time sensitive. While they can be simulated by the same executable, the type will be randomly determined at launch of process.

`oss` will create user processes at random intervals (of simulated time), so you will have two constants; let us call them `maxTimeBetweenNewProcsNS` and `maxTimeBetweenNewProcsSecs`. `oss` 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 spawn a new process about every 1-3 seconds, but you can experiment with this later to keep the system busy.

There should be a constant representing the percentage of time a process is launched as a normal user process or a real-time process. While this constant is specified by you, it should be heavily weighted to generate mainly user processes.

`oss` will be in control of all concurrency. In the beginning, 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 time when 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. Make a note of the process table being full in your log file.

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). The time can be generated by generating two random numbers in the interval [0,2) for sec and [0,`MAXINT`) for ns.

`oss` acts as the scheduler and so will *schedule* a process by sending it a message using a message queue or semaphore. When initially started, there will be no processes in the system but it will have a time in the future when it will launch a process, generated by the random process mentioned previously. If there are no processes currently ready to run in the system, it should increment the clock until it is the time when it should launch a process. It should then set up that process, generate a new time when it will create a new process and then using a message queue or semaphore, 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 multi-level feedback queue. There are three scheduling queues, each having an associated time quantum. The base time quantum is determined by you as a constant, let us say something like 10 milliseconds, but certainly could be experimented with. The highest priority queue has this base time quantum as the amount of time it would schedule a child process if it scheduled it out of that queue. The second highest priority queue would have twice that, the third highest priority queue would have four times the base queue and so on, as per a normal multi-level feedback queue. If a process finishes using its entire timeslice, it should be moved to a queue one lower in priority. If a process comes out of a blocked queue, it should go to the highest priority queue.

In addition to the naive multi-level feedback queue, you should implement some form of *aging* to prevent processes from starving. This should be based on some function of a processes wait time compared to how much time it has spent on the CPU. This is on you to create, but I want you to document your algorithm for aging in your `README` and in the code. I give you broad latitude here, but there must be some algorithm for this or you will lose points.

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 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 simulate running. They do not do any *actual* work but instead send a message to `oss` saying how much time they used and if they had to use i/o or had to terminate.

You should `#define` a constant in your system to indicate the probability that a process will terminate when scheduled. I would suggest this probability be fairly small to start. Processes will then, using a random number, use this to determine if they should terminate. Note that when creating this random number you must be careful that the seed you use is different for all processes, so I suggest seeding off of some function of the process' pid (for example, time + pid). 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 decided to terminate and also how much of its timeslice it used, so that `oss` can increment the clock by the appropriate amount.

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, but CPU-bound processes should be very unlikely to get interrupted (so they will usually use up their entire timeslice without getting interrupted). On the other hand, I/O-bound processes should more likely than not get interrupted before finishing their time slices. If a process determines that it would use up its timeslice, this information should be conveyed to `oss` when it cedes control to `oss`. Otherwise, the process would have to indicate to `oss` that it should be blocked, as well as the amount of its assigned quantum that it used up before being blocked. That process should then be put in a blocked queue waiting for an event that will happen in $r.s$ seconds where r and s are random numbers with range $[0, 5]$ and $[0, 1000]$ respectively. 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 ready queue. 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 by a different amount to indicate this.

Invoking the solution

`oss` should take in several command line options as follows:

```
oss [-h] [-s t] [-l f]
```

`-h`	Describe how the project should be run and then, terminate.
`-s t`	Indicate how many maximum seconds before the system terminates
`-l f`	Specify a particular name for the log file

Report

As we have two different types of processes, we would expect that the performance of these processes would be quite different. Your simulation should end with a report on average wait time, average time in the system, average CPU utilization and average time a process waited in a blocked queue for each of the different types of 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, assuming queue 0 is our ready queue:

```
OSS: Generating process with PID 3 and putting it in queue 0 at time 0:5000015
OSS: Dispatching process with PID 2 from queue 0 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 0
OSS: Dispatching process with PID 3 from queue 0 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 blocked queue
OSS: Dispatching process with PID 1 from queue 0 at time 0:5402505,
OSS: total time spent in dispatch was 7000 nanoseconds
etc
```

I suggest not simply appending to the 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 doing this project incrementally. A suggested way to break it down.

1. Create your `Makefile` and bare executables for `oss` and the child process. [Day 1]
2. Set up a system clock in shared memory and create a single user process, testing your synchronization method to pass information and control between them. [Day 2-3]
3. Have `oss` create a process control table with one user process and create that user process, testing the message queues back and forth. [Day 4]
4. Have `oss` simulate the scheduling of one user process over and over, logging the data but have no blocked queue. [Day 5]
5. Now add the chance for your single child process to sometimes block and have `oss` wait until it is ready again to reschedule. [Day 6]
6. Create your round robin ready queue, add additional user processes, making all user processes alternate in it. [Day 7]
7. Add the distinction between real-time and user processes. [Day 8]
8. Add the chance for user processes to be blocked on an event, keep track of statistics on this. [Day 9]
9. Add additional queues. [Day 10]
10. Add your own aging algorithm to prevent any chance at starvation. [Day 11]
11. Keep track of and output statistics like throughput, wait time, etc [Day 12]

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 50 processes in total, or if more than 3 real-life seconds have passed. Keep these two parameters (max number of processes and max time) as configurables. If you stop adding new processes, the system should eventually be 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.

Criteria for success

Make sure that you implement the specified scheduling algorithm and document it appropriately. *You must clean up after yourself.* That is, after the program terminates, whether normally or by force, there should be no shared memory, semaphore, or message queue that is left allocated to you.

Grading

1. *Overall submission: 10pts.* Program compiles and upon reading, seems to solve the assigned problem in the specified manner.
2. *README/Makefile: 10pts.* Ensure that they are present and work appropriately.
3. *Code readability: 10pts.* Code should be readable with appropriate comments. Author and date should be identified.
4. *Signal handling/cleaning up after yourself: 20pts.* The code terminates appropriately and all IPC primitives are cleaned up.

5. *Proper scheduling algorithm: 30pts.* The algorithm should be correct and appropriately documented.
6. *Conformance to specifications: 20pts.* Overall proper implementation and documentation.

Submission

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:

```
% cd
% chmod 755 ~
% ~sanjiv/bin/handin cs4760 4
% chmod 700 ~
```

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.