CS 4760 **Operating Systems**

Assignment # 4 Due Date: March 23, 2016

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: Implement a version of multi-level scheduling. There are three priority queues – a high-priority queue (queue 0), a medium priority queue (queue 1), and a low-priority queue (queue 2). Processes in queues 0 and 1 are treated according to Shortest Job First (SJF) strategy. Those in queue 2 are handled by round-robin. Since it is a simulator, you should be able to assign total CPU burst time to each process as the process is generated and keep it in its process control block. Each queue is given a time slice (the three slices may be different). When a queue has used up its time-slice, control of the CPU is passed to the processes on the next queue in the sequence $0 \to 1 \to 2 \to 0$. You should maintain information about the recent CPU usage of a process, and attempt to put I/O-bound processes in the high-priority queue and CPU-bound processes in the low-priority queue, so that on average, the high-priority queue contains processes with short CPU bursts and the low-priority queue contains processes with longer CPU bursts. The dispatcher should not always prefer the high-priority queue over the low-priority queue, but instead should attempt to allocate CPU time between different queues to get the best system performance. One scheme would be to allocate more CPU time to high-priority queue, so that the high-priority queue will tend to cycle faster than the other queues (assuming you have managed to get mostly processes with short CPU bursts in the high-priority queue and mostly processes with long CPU bursts in the low-priority queue). You should try to tune your scheduler to get the best performance under a variety of conditions. The main parameters you can adjust are the ratio of CPU time allocated to high-priority queue as opposed to the low-priority queue, and the method by which you decide into which queue a process should be placed.

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.

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:

```
% cd
% ~sanjiv/bin/handin cs4760 4
```

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.