CS4760 Operating Systems - Project No. 4

Due Date April 16, 2017 Max. Points : 100

Deliver your project using Mygateway to 23.59 on the due date.

Important 1: Please do all assignments on hoare

Important 2 – **for section Monday/Wednesday**: Please remember that you need to present me your program during office hours after submitting it using Mygateway.

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 be controlled by ***oss***.

In the beginning, ***oss*** will allocate shared memory for system data structures, which will include 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, process priority, and possibly other things. 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 no more than 18 process control blocks (though initially in testing, make sure this is much smaller). 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 by active processes (that is, a way for ***oss*** to determine which blocks represent currently valid processes that are still alive).

oss is the scheduler in this assignment and so will be creating new user processes at random intervals, say every second on an average (based on the logical, not system clock). The clock itself will work similarly to the last project, incremented in terms of nanoseconds. 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. Note that this time master will be incrementing the clock based on how long the user processes spent taking up their time quantum, as well as incrementing it by an amount indicating the work it had to do to perform scheduling.

You will use message passing to simulate the passing of execution context between ***oss*** and the user processes. So each of them should have a critical section that they loop over, controlled by message passing, but it should be set up so that ***oss*** and some user process swap. That is, the messages should contain information on who is allowed in. Since ***oss*** is the scheduler, it should always alternate control between any one user process.

After ***oss*** sets up the PCB, bit vector and any other data structures in shared memory, it should enter 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, forks the process. The child process will ***execl*** the binary.

Advance the logical clock by some random amount x nanoseconds in each iteration of the loop. x 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. To do this, every time ***oss*** generates a process, it should generate a time where it will create a new process. This should be (on the logical clock) between 0 and 2 seconds in the future. Once you have reached that amount of time in your logical clock, do what is necessary to generate that 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 by sending a message that lets that user process into its critical region. It will select the process by using a scheduling algorithm with the following features: Implement a version of multi-level scheduling with three queues. Most processes should start at queue 1, but if ended up blocked on io or unable to finish their quantum, they should go to the back of that level of queue. Processes that finish their quantum should be moved to the back of a queue one lower. If there are no lower queues, it should remain in that lowest queue unless it ends up not being able to finish its time quantum (had a simulated i/o event), in which case it should be put up to queue level 0.

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 (these three time slices should be different, with queue1 having twice the slice of queue0 and queue2 having twice the time slice of queue3. When a queue as no processes in it, control of the CPU is passed to the processes on the next queue in the sequence 0 ! 1 ! 2. You should maintain information about the recent CPU usage of a process in order to determine if it was interrupted by i/o or if it was allowed to finish its quantum.

The ***oss*** should dispatch a process by using message passing to signal to the appropriate user process to go into its critical region.

User Processes

All user processes are alike (same executable) but will simulate that they are performing some tasks at random times. The user process will be waiting for the appropriate message from the

master saying it has been scheduled and then will start to run by going into its critical section. 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). This choice is determining if it will be using some i/o that would cause it to block or if it is a cpu-bound process. 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 send a message to the master so that ***oss*** can schedule another process.

Each user process should also check to see if this time quantum will use up all of the cpu time that they needed to spend. If so, the message should be conveyed to ***oss*** who should *remove* its process control block and this user process should *terminate.*

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 any other resources if you used them.

OS Parameters

This project will use a broad set of parameters to determine the behavior of your operating system. The default setting of these parameters should be described in your README file and easily changed in your code (that is, defined as constants or read from a parameter file).

Some of these parameters should be the quantum length for processes, the chance on any quantum that a process terminates, the likelyhood that a process uses i/o, range of overhead done by master process, maximum number of user processes in the system, etc.

Hints

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 the 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.

What to handin

Handin an electronic copy of all the sources, README, Makefile(s), and results using Mygateway. Do not forget Makefile (with suffix or pattern rules), and README for the assignment. It will be nice when you will use some system for version control. This README should tell me how to run your project, and any problems or issues that it has running. Omission of a Makefile will result in a loss of 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). Do not forgot include in the README file your name and the date when your project was submitted. I do not like to see any extensions on Makefile and README files.