EECE.4810/EECE.5730: Operating Systems

Spring 2018

Programming Project 3
Additional Hints

Here are some additional points to hopefully help you complete this program. These points are based, to some degree, on my implementation, so remember you're welcome to implement your solution any way you want. This project has no implementation requirements, just input and output requirements.

Structures/classes: I found the following structures (or classes, if you're using C++) to be useful in this program:

- A general queue structure that simply points to the first and last nodes in the queue
    - The most obvious use of a queue is the ready queue, but you can use this type to track processes in other ways (arriving processes, finishing processes …)
    - Having pointers to the first and last nodes gives you the ability to easily dequeue a node from the front of the queue and add one to the back …
    - … although it doesn't always make sense to add a new node to the back of a queue (see the discussion of the enqueue function below)
    - Creating a queue data type makes it easier to pass queues to functions, too.
    - While you're welcome to use any queue implementation you want, this project defines no boundary on the size of the queue, so make sure your queue (and its nodes) support the ability to grow and shrink
- A single queue node to track all necessary information about each process
    - "Necessary information" might include the process ID, data from the input file (burst time, priority, arrival time), and statistics about the process (wait time, turnaround time).
    - You'll likely also need some additional "helper" data to help calculate statistics or make sure that process loading is handled appropriately
- A data type to store per-algorithm stats (average wait time, average turnaround time, context switches)
    - This type helps to generate the final summary at the end of the output file

Functions: I implemented the following functions; you may, of course, use others:

- Queue operations
  - Enqueue/add: adds a new node to a queue
    - A typical, simple queue always enqueues a new node as the last node.
    - However, you'll want the ability to sort your queue in different ways—look at how the ready queues are built for each different algorithm.
    - The easiest way to maintain a sorted queue is to ensure each new node is placed in the right spot when it's added
  - Dequeue/remove: remove the first node from a queue
    - Nothing out of the ordinary about this function—no matter how a queue is ordered, whatever's at the front of the queue is always the node you want
  - Print: print the queue contents
  - Clean: if your queue contains dynamically allocated data, you'll want to deallocate all the nodes before the end of the program
- Main simulation loop
  - As noted in class, you'll need to run each algorithm separately, using a loop that tracks cycles, the currently running process, the ready queue, and other data. The loop ends when the last process finishes.
  - However, the majority of the work you do for each algorithm is the same.
  - I therefore wrote a function to handle the simulation of each algorithm, with conditional blocks to implement different behaviors required in some algorithms
  - The parameters to this function should help you get the function to behave in different ways for different algorithms
- Stat sorter
  - I've got a list of structures that contain the stats about each algorithm (FCFS, SJF, etc.) that I can sort by any of the three stats printed in the final summary (average wait time, average turnaround time, and context switches)
  - This function also prints the ordered table for whatever stat it's sorted by.

Things I found particularly tricky: You may, of course, run into different problems, but here are some relatively small points I found difficult to account for:

- Accounting for multiple processes arriving at the same time and ensuring they all enter the ready queue at the appropriate time
- Ensuring that a process that arrives at time T is forced to wait until time T+1 to start executing, rather than immediately beginning execution at time T
    - You must ensure that a process that's marked as "loading" in one cycle actually starts executing in the next cycle, regardless of whether a new arrival is a "better" choice according to a given scheduling algorithm
    - In preemptive schemes, that "better" choice can replace the recently loaded process after just one cycle.
        - Look at the STCF output from the first test case I posted. In cycle 3, process 1 finishes and process 2 is loaded. In cycle 4, process 3 arrives and preempts process 2 after just one cycle.
- Tracking the total wait time for each process in preemptive algorithms
    - Determining wait time in a non-preemptive algorithm is relatively straightforward, since each process completes its CPU burst once it starts
    - In a preemptive algorithm, however, you must track wait time whenever a process returns to the ready queue after executing part of its CPU burst