EECE.4810/EECE.5730: Operating Systems

Spring 2019

Programming Project 3

Due **<u>11:59 PM</u>, Monday, 4/22/19**

1. Introduction

This project uses simulation to explore the effectiveness of the different scheduling algorithms described in class. While your program will not schedule actual processes running on your machine, it will model how each algorithm behaves when making scheduling decisions about a set of processes.

This assignment is worth a total of 150 points. The grading rubric given in Section 4 applies to students in both EECE.4810 and EECE.5730.

This assignment was adapted, with permission, from an assignment written by Prof. Vinod Vokkarane for CIS 370 at UMass Dartmouth.

2. Project Submission and Deliverables

Your submission must meet the following requirements:

- Your solution may be written in C or C++.
- Your code must run on the Linux machines in Ball 410, but you may write it elsewhere.
- You must submit a .zip file containing the following files:
  - All source/header files you write. You should have at least three such files:
    - A source file containing your `main()` function and global variables
    - A header file for any structure/class definition(s) and function prototypes (in the spec below, I'll refer to this file as `sched_sim.h`)
    - A source file for your function definitions (in the spec below, I'll refer to this file as `sched_sim.c`). Global variables declared in your main file should be additionally declared as `extern` in this file—see the Project 2 specification for a brief description of the `extern` keyword.
  - A makefile that can be used to build your project
  - A README file that briefly describes the contents of your submission, as well as directions for compiling and running your code. Note that, if these directions do not follow the specification, you will lose points.
- Programs must be submitted via Blackboard.
- You may work in groups of up to 3 on this assignment. If you work in a group, please list the names of all group members in a header comment at the top of each file in your submission, as well as in the email you use to submit your code.

3. Specification

General overview: Simulators allow a user to evaluate the effectiveness of a particular design or algorithm without fully implementing that hardware or software. In this case, you do not have to schedule actual processes to test how scheduling algorithms will handle different process workloads. Your simulator will model each of the following algorithms covered in class:

- First come, first served (FCFS)
- Shortest job first (SJF)
- Shortest time to completion first (STCF)
- Round robin (RR) with time quantum 2
- Non-preemptive priority scheduling

Input: Your program reads input from two sources:

Command line arguments: Your executable should take the following command line arguments:

- The name of an input text file containing information about the processes to be simulated
    - A "text file" does not require a .txt extension. That description simply implies that the file's contents are human-readable. C/C++ programs can read or write files in two general formats—text or binary.
- The name of a text file that will contain your program's output
- The interval at which your program generates a snapshot of the scheduler state

For example, if your executable name is `prog3`, the following command runs the program with `test1.dat` as the input file, `out1.txt` as the output file, and a 10 cycle interval between scheduler snapshots: `./prog3 test1.dat out1.txt 10`

Input file: The input file contains the following information about all processes:

- CPU burst time
- Priority: a lower number indicates higher priority
- Arrival time: a process arriving at time T may start execution at time T + 1

Each line represents a single process. Your program must read every line in the file and store the information about each process. The file size is not given, so your program must read until it reaches the end of the file.

For example, a file describing the processes used in the example on slide 33 of Lecture 18 would have the following contents. Process ID numbers are not in the file; you should assign a unique number to each process, starting with process 0 or 1. Processes are listed in order of arrival:

```
10   1    0
3    4    0
7    2    2
1    2    4
5    3    6
```

3. Specification (continued)

Output: This program should print all output to the text file named in the command line arguments. The output takes three general forms:

Scheduler snapshots: A "snapshot" shows the state of the scheduler at a given time point. Each snapshot should be separated by the interval specified in the command line arguments and should contain the following information:

- Current time (for example: `t = 0`)
- Current CPU action, which can take one of three forms:
  - Loading (preparing to start) a new process: List the full burst time of the process
  - Running a process: List the remaining burst time of the process as it runs
  - Finishing a process: If a snapshot is taken in a cycle in which a process finishes, describe both which process is finishing and which one is next to run
- The current state of the ready queue, shown in the order processes will run next. If there are no processes in the ready queue, indicate that it is empty.

For example, given the example file contents on the previous page and assuming FCFS scheduling and a time interval of 5, your first four snapshots would be as follows. Note that, as shown, a brief message naming the algorithm should be printed before the first snapshot:

```
***** FCFS Scheduling *****
t = 0
CPU: Loading process 0 (CPU burst = 10)
Ready queue: 1

t = 5
CPU: Running process 0 (remaining CPU burst = 5)
Ready queue: 1-2-3

t = 10
CPU: Finishing process 0; loading process 1 (CPU burst = 3)
Ready queue: 2-3-4

t = 15
CPU: Running process 2 (remaining CPU burst = 5)
Ready queue: 3-4
```

3. Specification (continued)

Output (continued): The other two forms of output are:

Algorithm summaries: For each scheduling algorithm, generate a summary that shows:

- A table of all processes, where each line includes a process ID, wait time, and turnaround time. The last line of the table should list the average wait and turnaround times.
- A list of how the processes were run—essentially a text-based version of the process schedules shown in example problems.
    - This list is relatively simple for non-preemptive scheduling algorithms; for preemptive schedulers, this list should show every time a given process gains access to the CPU.
    - So, as you might guess, your round robin list will be the longest of the five.
- The total number of context switches

For example, using STCF scheduling with the sample input file described above would yield the following summary (this information comes from slides 34 & 35 of Lecture 18, with the processes numbered slightly differently):

```
STCF Summary (WT = wait time, TT = turnaround time):

PID    WT      TT
 0     16      26
 1      0       3
 2      2       9
 3      0       1
 4      5      10
AVG   4.6     9.8

Process sequence: 2-3-4-3-5-1
Context switches: 6
```

Overall summary: After the detailed output for each algorithm, your program should print a summary showing how all the algorithms perform based on the metrics measured in your simulation. The results for each metric should be listed in a table ordered from best to worst:

- Shortest average wait time
- Shortest average turnaround time
- Fewest context switches

One example of an overall summary can be seen on the next page; other examples can be found in the test output files posted on the course website.

3. Specification (continued)

Output (continued): An example overall summary is shown below:

```
***** OVERALL SUMMARY *****

Wait Time Comparison
1 STCF           4.60
2 SJF            5.60
3 FCFS          10.40
4 Round robin   10.80
5 Priority      11.20

Turnaround Time Comparison
1 STCF           9.80
2 SJF           10.80
3 FCFS          15.60
4 Round robin   16.00
5 Priority      16.40

Context Switch Comparison
1 SJF            5
2 FCFS           5
3 Priority       5
4 STCF           6
5 Round robin   15
```

4. Grading Rubric

Your assignment will be graded according to the rubric below; partial credit may be given if you successfully complete part of a given objective. **You should not submit separate files for each scheduling algorithm**—your sole submission will be judged on which of the scheduling algorithms it simulates correctly. Algorithms must include:

- First come, first served (FCFS)
- Shortest job first (SJF)
- Shortest time to completion first (STCF)
- Round robin (RR) with time quantum 2
- Non-preemptive priority scheduling

Each algorithm is worth **30 points**, for a total of **150 points**. Note that a fundamental flaw in your simulator affecting all algorithms will result in a deduction for all sections of the program.

For this project, there will be two extra credit objectives. *If you have at least attempted either (or both) of these objectives (even if not successful), please include that information in your Blackboard submission (under Comments) and your README file when submitting the project*:

- **10 points**: Implement aging in your priority scheme, such that the priority of a waiting process increases by 1 for every 25 cycles it must wait.
    - Note that "increasing priority by 1" actually means decrementing the priority value, as lower numbers represent higher priority.
- **10 points**: Assume that every process requires a 10 cycle I/O burst in the middle of the CPU burst and incorporate that additional time into your scheduling process.
    - Adding I/O bursts will require you to implement a separate I/O queue and track not just CPU burst time but I/O burst time as well.
    - This addition would effectively make every scheduling algorithm preemptive, as a process should be "preempted" after it has executed half of its CPU cycles (rounding down if the total CPU burst is odd) and added to the I/O device queue.
    - Assume the system has only a single I/O device and that device is scheduled on a first come, first served basis.
    - Once a process finishes its I/O burst, it should return to the ready queue to complete its CPU burst.

5. Test Cases

Test cases for this assignment come in the form of files posted on the course website. These files come in pairs—an input file and the output generated based on that input file. The general form of different parts of your output—the snapshots, algorithm summaries, and final summaries—are described in Section 3.

The command lines used to generate these files were as follows (remember, the three command line arguments are input file name, output file name, and snapshot interval):

- `./prog3 testin1.dat out1.txt 1`
  (This test case is from the example process list given earlier, under "Input" in Section 3)
- `./prog3 testin2.dat out2.txt 10`
- `./prog3 testin2.dat out2_interval1.txt 1`
  (This output file is significantly longer than the one generated above, given that it contains 10 times as much output as the file with the 10 cycle interval. However, it's much easier to debug your program when you can see what happens every cycle—looking at the output every 10 cycles gives a more practical amount of output, but it doesn't give you a very good idea of what your program's doing.)

6. Hints

The following 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.

Tracking information: While this assignment does not require a specific implementation, you should, at a minimum, track the following information to generate the correct output:

- The current "time" as the simulator progresses
  - Time has no units in this program, but, for each algorithm, one iteration of your simulator loop should represent one time step
  - You will likely need to start a separate loop for each algorithm, since each algorithm's simulation is described in a separate section of your output file
- The currently running process
- The state of the ready queue
  - Remember, in preemptive algorithms, a process will return to the ready queue if it is forced to give up the processor before completing its CPU burst
- An ordered list of past processes that have run, so you can generate the final schedule to be shown in each algorithm summary
- All of the statistics described in the output

6. Hints (continued)

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

6. Hints (continued)

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.

6. Hints (continued)

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
  - Determining wait time in a non-preemptive algorithm is relatively straightforward, since each process completes its CPU burst once it starts.
    - However, you shouldn't consider the cycle in which a process arrives to be a cycle in which it waits.
  - 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.
    - And, as above, you shouldn't consider the cycle in which a process returns to the ready queue to be a cycle in which it waits.