CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Project 1 (document version 1.6)
CPU Scheduling Simulation

Overview

- This homework is due by 11:59:59 PM on Friday, March 30, 2018.
- This homework will count as 10% of your final course grade.
- This project is to be completed either individually or in a team of at most three students. Teams may consist of both undergraduate and graduate students. Do not share your code with anyone else.
- Note that students registered for CSCI 6140 will be required to submit additional work for this project, as described on page 8.
- You **must** use one of the following programming languages: C, C++, Java, or Python.
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v16.04.3 LTS.
- If you use C or C++, your program **must** successfully compile via `gcc` or `g++` with absolutely no warning messages when the `-Wall` (i.e., warn all) compiler option is used. We will also use `-Werror`, which will treat all warnings as critical errors.
- Note that the `gcc`/`g++` compiler is version 5.4.0 (`Ubuntu 5.4.0-6ubuntu1~16.04.6`). For source file naming conventions, be sure to use `*.c` for C or `*.cpp` for C++. In either case, you can also include `*.h` files.
- If you use Java, name your main Java file `Project1.java`. And note that the `javac` compiler is version 8 (`javac 1.8.0_161`).
- For Python, you can use either `python2.7` or `python3.5`. Be sure to name your main Python file `project1.py`.
- For Java and Python, be sure no warning messages occur during compilation/interpretation.
- Finally, keep in mind that Project 2 may build on this initial project. Therefore, be sure your code is easily maintainable and extensible.

Project Specifications

In this first project, you will implement a rudimentary simulation of an operating system. The initial focus will be on processes, assumed to be resident in memory, waiting to use the CPU. Memory and the I/O subsystem will not be covered in depth in this project.

Conceptual Design

A **process** is defined as a program in execution. For this assignment, processes are in one of the following three states:

- **READY:** in the ready queue, ready to use the CPU
- **RUNNING:** actively using the CPU
- **BLOCKED:** blocked on I/O

Processes in the **READY** state reside in a simple queue called the ready queue. This queue is ordered based on a configurable CPU scheduling algorithm. In this first assignment, there are three algorithms to implement, i.e., first come, first served (FCFS), shortest remaining time (SRT), and round robin (RR). Note that all three algorithms will be applied to the same set of simulated processes.

In general, when a process reaches the front of the queue (and the CPU is free to accept the next process), the given process enters the **RUNNING** state and starts executing its CPU burst.

After the CPU burst is completed, if the process does not terminate, the process enters the **BLOCKED** state, waiting for an I/O operation to complete (e.g., waiting for data to be read in from a file). When the I/O operation completes, depending on the scheduling algorithm, the process either (1) returns to the **READY** state and is added to the ready queue or (2) enters the **RUNNING** state and preempts the currently running process.

First Come, First Served (FCFS)

The FCFS algorithm is a non-preemptive algorithm in which processes line up in the ready queue, waiting to use the CPU. This is your baseline algorithm.

Shortest Remaining Time (SRT)

The SRT algorithm is a preemptive version of the Shortest Job First (SJF) algorithm. In both SJF and SRT, processes are stored in the ready queue in order of priority based on their CPU burst times. More specifically, the process with the shortest CPU burst time will be selected as the next process executed by the CPU.

In SRT, when a process arrives, before it enters the ready queue, if it has a CPU burst time that is less than the remaining time of the currently running process, a preemption occurs. When such a preemption occurs, the currently running process is added back to the ready queue.

Round Robin (RR)

The RR algorithm is essentially the FCFS algorithm with predefined time slice `t_slice`. Each process is given `t_slice` amount of time to complete its CPU burst. If this time slice expires, the process is preempted and added back **(v1.2)** to the end of the ready queue. If a process completes its CPU burst before a time slice expiration, the next process on the ready queue is immediately context-switched into the CPU. **(v1.2)** Note that arriving processes and processes that have completed I/O adhere to the `rr_add` parameter.

For your simulation, if a preemption occurs and there are no other processes on the ready queue, do not perform a context switch. For example, if process `A` is using the CPU and the ready queue is empty, if process `A` is preempted by a time slice expiration, do not context-switch process `A` back to the empty queue. Instead, keep process `A` running with the CPU and do not count this as a context switch. In other words, when the time slice expires, check the queue to determine if a context switch should occur.

Simulation Configuration

The key to designing a useful simulation is to provide a number of configurable parameters. This allows you to simulate and tune a variety of scenarios (e.g., a large number of CPU-bound processes, multiple CPUs, etc.).

Therefore, define the following simulation parameters as tunable constants within your code:

- Define `n` as the number of processes to simulate. Note that this is determined via the input file described below.
- Define `t_cs` as the time, in milliseconds, that it takes to perform a context switch (use a default value of `8`). Remember that a context switch occurs each time a process leaves the CPU and is replaced by another process. Note that the first half of the context switch time (i.e., half of `t_cs`) is the time required to remove the given process from the CPU; the second half of the context switch time is the time required to bring the next process in to use the CPU.
- For the RR algorithm, define the time slice (i.e., `t_slice`) value, measured in milliseconds, with a default value of `80`.
- Also for the RR algorithm, define whether processes are added to the end or the beginning of the ready queue when they arrive **(v1.2)** or complete I/O. Use `rr_add` and define distinct values `BEGINNING` and `END` (with `END` being the default behavior). An optional third command-line argument can specify either "`BEGINNING`" or "`END`" to set this parameter.

Input File

The input file to your simulator specifies the processes to simulate. This input file is a simple text file that adheres to the following specifications:

- The input file to read must be specified on the command-line as the first argument.
- Any line beginning with a `#` character is ignored; these lines are comments.
- All blank lines are also ignored, including lines containing only whitespace characters.
- Each non-comment line specifies a single process by defining the process (designated as a uppercase letter), the initial arrival time, the CPU burst time, the number of CPU bursts to perform before process termination, and the I/O wait time (for each I/O burst); all fields are delimited by `|` (pipe) characters. Note that times are specified in milliseconds and that the I/O wait time is defined as the amount of time from the end of the CPU burst (i.e., from the end of the first half of the given context switch) to the end of the I/O operation.

An example input file is shown below.

```
# example simulator input file
#
# <proc-id>|<initial-arrival-time>|<cpu-burst-time>|<num-bursts>|<io-time>
A|0|168|5|287
B|0|385|1|0
C|190|97|5|2499
D|250|1770|2|822
```

In the above example, processes `A` and `B` both arrive at time `0`. Further, process `A` has a CPU burst time of `168ms`. Its burst will be executed `5` times, then the process will terminate. After each CPU burst is executed, the process will be blocked on I/O for `287ms`. Further, process `B` has a CPU burst time of `385ms`, after which it will terminate (and therefore has no I/O to perform).

Processes `C` and `D` arrive at times `190ms` and `250ms`, respectively. Upon arrival, each process either is added to the ready queue or immediately starts using the CPU, depending on the CPU scheduling algorithm being simulated and the state of the simulation.

Your simulator must read this input file, adding processes to the queue based on the scheduling algorithm. For FCFS and RR, processes arriving at the same time are always initially added in the process order shown in the input file. Therefore, in the above example, processes will initially be on the queue in the order `A` and `B` at time `0` (with process `A` at the front of the queue).

After you simulate an algorithm, you must reset the simulation back to the initial set of processes and set your elapsed time back to zero. In short, we wish to compare these algorithms with one another given the same initial conditions.

Note that depending on the contents of the given input file, there may be times during your simulation in which the CPU is idle because all processes are busy performing I/O. Also, when all processes terminate, your simulation ends.

(v1.1) Assumptions about the Input File

To simplify your code, you can make the following assumptions about the input file:

- You can assume that process IDs in the input file are unique; therefore, at most you will have 26 processes to simulate.
- When you read the input file, you can assume that processes are presented in alphabetical order.
- You can assume that arrival times are in non-decreasing order.

Handling "Ties"

For events that occur at the same time, use the following order: (a) CPU burst completion; (b) I/O burst completion (i.e., back to the ready queue); and then (c) process arrival. Any "ties" within these three categories are to be broken using process ID order. As an example, if processes `Q` and `T` happen to both finish with their I/O at the same time, process `Q` wins this "tie" (because `Q` is alphabetically before `T`) and is added to the ready queue before process `T`.

Be sure you do not implement any additional logic for the I/O subsystem. In other words, there are no I/O queues to implement here in this first project.

CPU Burst, Turnaround, and Wait Times

CPU Burst Time: CPU burst times are given (in the input file) for each process that you simulate. CPU burst time is defined as the amount of time a process is **actually** using the CPU. Therefore, this measure does not include context switch times.

Turnaround Time: Turnaround times are to be measured for each process that you simulate. Turnaround time is defined as the end-to-end time a process spends in executing a single CPU burst. More specifically, this is measured from the process's arrival time through to when the CPU burst is completed and the process is switched out of the CPU. **(v1.4)** Therefore, note that this measure includes the second half of the initial context switch in and the first half of the final context switch out, as well as any other context switches that occur while the CPU burst is being completed (i.e., due to preemptions).

Wait Time: Wait times are to be measured for each process that you simulate. Wait time is defined as the amount of time a process spends waiting to use the CPU, which equates to the amount of time the given process is actually in the ready queue. Therefore, this measure does not include context switch times that the given process experiences (i.e., only measure the time the given process is actually in the ready queue). **(v1.4)** More specifically, a process leaves the ready queue when it is switched into the CPU, which takes half of context switch time `t_cs`. Likewise, a preempted process leaves the CPU and enters the ready queue after the first half of `t_cs`.

Required Output

Your simulator should keep track of elapsed time `t` (measured in milliseconds), which is initially zero. As your simulation proceeds based on the input file, `t` advances to each "interesting" event that occurs, displaying a specific line of output describing each event.

Note that your simulator output should be entirely deterministic. To achieve this, your simulator must follow the above specifications and output each "interesting" event that occurs using the format shown below. The contents of the ready queue are shown for each event except for the end-of-simulation event.

```
time <t>ms: <event-details> [Q <queue-contents>]
```

The "interesting" events are:

- Start of simulation
- Process arrives (i.e., based on arrival time in input file)
- Process starts using the CPU
- Process finishes using the CPU (i.e., completes its CPU burst)
- Process is preempted
- Process starts performing I/O
- Process finishes performing I/O
- Process terminates by finishing its last CPU burst
- End of simulation

The "process arrival" event occurs every time a process arrives, i.e., based on the arrival time given in the input file and when a process completes I/O. In other words, processes "arrive" within the subsystem that consists of the CPU and the ready queue.

The "process preemption" event occurs every time a process is preempted by a time slice expiration (in RR) or by an arriving process (in SRT). When a preemption occurs, a context switch occurs (unless for RR there are no processes in the ready queue).

Note that when your simulation ends, you must display that event as shown below.

```
time <t>ms: Simulator ended for <algorithm>
```

Be sure that you still include the process removal time (i.e., half the context switch time) for this last process.

In addition to the above output (which should simply be sent to `stdout`), generate an output file (with the filename specified as the second command-line argument) that contains statistics for each simulated algorithm. The file format is shown below (with `#` as a placeholder for actual numerical data). Round to exactly two digits after the decimal point for your averages.

```
Algorithm FCFS
-- average CPU burst time: #.## ms
-- average wait time: #.## ms
-- average turnaround time: #.## ms
-- total number of context switches: #
-- total number of preemptions: #
Algorithm SRT
-- average CPU burst time: #.## ms
-- average wait time: #.## ms
-- average turnaround time: #.## ms
-- total number of context switches: #
-- total number of preemptions: #
Algorithm RR
-- average CPU burst time: #.## ms
-- average wait time: #.## ms
-- average turnaround time: #.## ms
-- total number of context switches: #
-- total number of preemptions: #
```

Note that averages are averaged over all executed CPU bursts. Also note that to count the number of context switches, you should count the number of times a process **starts** using the CPU.

Handling Errors

Your program must ensure that the correct number of command-line arguments are included. If not, display an error message and usage information exactly as follows on `stderr`:

```
ERROR: Invalid arguments
USAGE: ./a.out <input-file> <stats-output-file> [<rr-add>]
```

If you detect an error in the input file format, display an error message as follows on `stderr`:

```
ERROR: Invalid input file format
```

In both error cases above, be sure to exit your program by returning `EXIT_FAILURE`.

Submission Instructions

To submit your assignment (and also perform final testing of your code), please use Submitty, the homework submission server. The specific URL is on the course website.

If you are submitting a team project, please have each team member submit the same submission (to be sure everyone gets a grade). Also be sure to include all names and RCS IDs in comments at the top of each source file.

Note that this assignment will be available on Submitty a few days before the due date. Please do not ask on Piazza when Submitty will be available, as you should perform adequate testing on your own Ubuntu platform.

Graduate Section Requirements

For students registered for CSCI 6140, additional analysis is required. Please answer the questions below and submit as a PDF file called `project1-analysis.pdf`. Answer all questions below in no more than three pages.

Note that each student registered for CSCI 6140 must write up his or her own answers even if you are working on a team.

If you are registered for CSCI 4210, feel free to review these questions, but do not submit an analysis on Submitty.

1. Of the three simulated algorithms, which algorithm is the best algorithm? Which algorithm is the best for CPU-bound processes? Which algorithm is the best for I/O-bound processes? Support your answer by citing specific simulation results.
2. For the RR algorithm, how does changing `rr_add` from `END` to `BEGINNING` change your results?
3. Again for the RR algorithm, can you tune `t_slice` such that you attain the shortest average turnaround time or shortest average wait time for the given sample test input files? Explain what happens when you use different values of `t_slice`.