CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Project 1 (document version 1.2)
Process Simulation Framework

Overview

- This project is due by 11:59:59 PM on Thursday, October 20, 2016. Projects are to be submitted electronically.
- This project will count as 12% of your final course grade.
- This project is to be completed **individually** or in a team of at most three students. Do not share your code with anyone else.
- You **must** use one of the following programming languages: C, C++, Java, or Python. If using Java, name your main Java file `Project1.java`.
- Your program **must** successfully compile and run on Ubuntu v14.04.5 LTS or higher.
- 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.
- For Java and Python, be sure no warning messages occur during compilation and/or interpretation.
- Finally, keep in mind that Project 2 will likely 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 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, i.e. 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-serve (FCFS), shortest job first (SJF), and round robin (RR). Note that all three algorithms will be applied to the same set of simulated processes.

Once a process reaches the front of the queue (and the CPU is free to accept a process), the given process enters the **RUNNING** state and executes its current CPU burst.

After the CPU burst is completed, the process enters the **BLOCKED** state, performing some sort of I/O operation (e.g., writing results to the terminal or a file, interacting with the user, etc.). Once the I/O operation completes, the process returns to the **READY** state and is added to the ready queue, with its position within the queue based on each given CPU scheduling algorithm.

First-Come-First-Serve (FCFS)

The FCFS algorithm is a non-preemptive algorithm in which processes line up in the ready queue, awaiting the CPU.

Shortest Job First (SJF)

The SJF algorithm is also a non-preemptive algorithm. In SJF, 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 the next process executed by the CPU.

Round Robin (RR)

The RR algorithm is essentially the FCFS algorithm with a 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 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.

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 `m` as the number of processors (i.e., cores) available within the CPU (use a default value of `1`). Note that we could use this in a future project.
- 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.
- For the RR algorithm, define the aforementioned time slice (i.e., `t_slice`) value, measured in milliseconds, with a default value of `84`.

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 capital letter), the initial arrival time, the CPU burst time, the number of bursts to perform before process termination, and the I/O wait time; all fields are delimited by `|` (pipe) characters. Note that times are specified in milliseconds (ms) and that the I/O wait time is defined as the amount of time from the end of the CPU burst (i.e., before the 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
D|190|97|5|2499
C|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 `250ms` and `190ms`, 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 given process order. 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.

All "ties" 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 to be measured 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.

Note that this measure can simply be calculated from the given input data.

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 trying to complete a single CPU burst. More specifically, this is measured from the process's arrival time through to when the CPU burst is completed. This measure includes all context switch times.

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 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).

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 output each "interesting" event that occurs using the format shown below (note that 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 starts using the CPU
- Process finishes using the CPU (i.e., completes its CPU burst)
- Process is preempted (due to time slice expiration)
- 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 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). When a preemption occurs, a context switch occurs.

Note that when your simulation ends, you must display that event as follows (and skip the very last context switch):

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

In addition to the above output (which should simply be sent to `stdout`), generate an output file (with filename specified by 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. And note that averages are averaged over all executed CPU bursts.

```
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 SJF
-- 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: ##
```

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>
```

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

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

(v1.2) In both error cases above, be sure to exit with `EXIT_FAILURE` as the return value.

Submission Instructions

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