Project 2: CPU Scheduling Simulator

CSCI 442, Spring 2017

Assigned Date: March 2, 2017
Intermediate Deliverable 1 Due: March 10, 2017 @ 11:59pm
Intermediate Deliverable 2 Due: March 24, 2017 @ 11:59pm
Final Deliverable Due: April 7, 2017 @ 11:59pm

The goal of this project is to develop a CPU scheduling simulation that will complete the execution of a group of multi-threaded processes. It must support several different scheduling algorithms such that the user can specify which one to use via a command-line flag and will need to calculate and display many of the performance criteria by which scheduling algorithms are typically judged.

Your project must be implemented in C++, AND it must execute correctly on the Linux machines in the Alamode lab (BB136).

1 Simulation Constraints

The program will simulate process scheduling on a hypothetical computer system with the following attributes:

1. There is a single CPU, so only one process can be running at a time.
2. There are an infinite number of I/O devices, so any number of processes can be blocked on I/O at the same time.
3. Processes consist of one or more kernel-level threads (KLTs).
4. Threads (not processes) can exist in one of five states:
    - `NEW`
    - `READY`
    - `RUNNING`
    - `BLOCKED`
    - `EXIT`
5. Dispatching threads requires a non-zero amount of OS overhead.
    - If the previously executed thread belongs to a different process than the new thread, a full process switch occurs. This is also the case for the first thread being executed.
    - If the previously executed thread belongs to the same process as the new thread being dispatched, a cheaper thread switch is done.
    - A full process switch includes any work required by a thread switch.

6. Threads, processes, and dispatch overhead are specified via a file whose format is specified in the next section.
7. Each thread requires a sequence of CPU and I/O bursts of varying lengths as specified by the file.
8. Processes have an associated priority, specified as part of the file. Each thread in a process has the same priority as its owning process.
    - 0: `SYSTEM` (highest priority)
    - 1: `INTERACTIVE`
    - 2: `NORMAL`
    - 3: `BATCH` (lowest priority)
9. All processes have a distinct process ID, specified as part of the file. Thread IDs are unique only within the context of their owning process (so the first thread in every process has an ID of 0).
10. Overhead is incurred only when dispatching a thread (transitioning it from `READY` to `RUNNING`); all other OS actions require zero OS overhead. For example, adding a thread to a ready queue or initiating I/O are both free.
11. Threads for a given process can arrive at any time, even if some other process is currently running (i.e. some external entity is responsible for creating threads).
12. Threads get executed, not processes.

2 Simulation File Format

The simulation file provides a complete specification of a scheduling scenario. Its format is as follows:

```
num_processes  thread_switch_overhead  process_switch_overhead

process_id  process_type  num_threads
thread_0_arrival_time  num_CPU_bursts
cpu_time  io_time
cpu_time  io_time
...                          // repeat for the number of CPU bursts
cpu_time                     // the last CPU burst can NOT have I/O

thread_1_arrival_time  num_CPU_bursts
cpu_time  io_time
cpu_time  io_time
...                         // repeat for the number of CPU bursts
cpu_time                    // the last CPU burst can NOT have I/O

... (repeat for the number of threads)
```

```
... (repeat for the number of processes)
```

Here's an example. Note that the comments won't be in the actual files.

```
2 3 7          // 2 processes, thread/process overheads are 3 and 7

0 1 2          // process 0 (type 1: interactive) has 2 threads
0 3            // the first thread arrives at time 0 and has 3 bursts
4 5            // CPU burst of 4 and I/O of 5
3 6            // CPU burst of 3 and I/O of 6
1              // last CPU burst takes 1 unit; thread terminates

1 2            // the second thread arrives at time 1 and has 2 bursts
2 2            // CPU burst of 2 and I/O of 2
7              // last CPU burst takes 7 units; thread terminates

1 0 3          // process 1 (type 0: system) has 3 threads
5 3            // the first thread arrives at time 5 and has 3 bursts
4 1            // CPU burst of 4 and I/O of 1
2 2            // CPU burst of 2 and I/O of 2
2              // last CPU burst takes 2 units; thread terminates

6 2            // the second thread arrives at time 6 and has 2 bursts
2 2            // CPU burst of 2 and I/O of 2
3              // last CPU burst takes 3 units; thread terminates

7 5            // the third thread arrives at time 7 and has 5 bursts
5 7            // CPU burst of 5 and I/O of 7
2 1            // CPU burst of 2 and I/O of 1
8 1            // CPU burst of 8 and I/O of 1
5 7            // CPU burst of 5 and I/O of 7
3              // last CPU burst takes 3 units; thread terminates
```

Several sample files that you can use in testing can be found on the course website.

3 Command-Line Flags

Your simulation must support invocation in the format specified below, including the following command-line flags:

```
./simulator [flags] simulation_file.txt

-t, --per_thread
```

```
  Output additional per-thread statistics for arrival time, service time, etc.

-v, --verbose
  Output information about every state-changing event and scheduling decision.

-a, --algorithm
  The scheduling algorithm to use. One of FCFS, RR, PRIORITY, or CUSTOM.

-h --help
  Display a help message about these flags and exit
```

You must use the `getopt_long(3)` method for parsing these flags. As always, check the `man` page for additional information.

Users should be able to pass the `-t` or `-v` flags separately or together. The required output formats for these flags are described in the next section.

The `-a` flag will dictate the scheduling algorithm your simulator should use. If not set, FCFS should be used as the default. Additional details about the algorithms you must implement are described in a later section.

Finally, the `--help` (or `-h`) flag must cause your program to print out instructions for how to run your program and about the flags it accepts and then immediately exit.

4 Required Output and Format

In the same way your program must read an input of a specific format, it must also output data in a specific format. This will allow my programs to read and understand your simulation's results.

In all cases, regardless of the flags a user passes, your simulation must output the following information:

- For each type of process (types 0 - 3):
  - The total number of such threads
  - The average response time
  - The average turnaround time
- The total time required to execute all threads to completion
- The time spent executing user processes (service time)
- The time spent performing I/O (sum of all I/O bursts)
- The time spent doing process and thread switches (dispatching overhead)
- The amount of time the CPU was idle

- CPU utilization
- CPU efficiency

Here is an example in the required format:

```
SYSTEM THREADS:
    Total count:                3
    Avg response time:      23.33
    Avg turnaround time:    94.67

INTERACTIVE THREADS:
    Total count:                2
    Avg response time:      10.00
    Avg turnaround time:    73.50

NORMAL THREADS:
    Total count:                0
    Avg response time:       0.00
    Avg turnaround time:     0.00

BATCH THREADS:
    Total count:                0
    Avg response time:       0.00
    Avg turnaround time:     0.00

Total elapsed time:           130
Total service time:            53
Total I/O time:                34
Total dispatch time:           69
Total idle time:                8

CPU utilization:           93.85%
CPU efficiency:            40.77%
```

If the `-t` or `--per_thread` flag is set, your program must also output the arrival time, service time, I/O time, turnaround time, and finish time for each thread.

```
Process 0 [INTERACTIVE]:
    Thread 0:  ARR: 0      CPU: 8      I/O: 11     TRT: 88     END: 88
    Thread 1:  ARR: 1      CPU: 9      I/O: 2      TRT: 59     END: 60

Process 1 [SYSTEM]:
    Thread 0:  ARR: 5      CPU: 8      I/O: 3      TRT: 92     END: 97
    Thread 1:  ARR: 6      CPU: 5      I/O: 2      TRT: 69     END: 75
```

```
    Thread 2:  ARR: 7      CPU: 23     I/O: 16     TRT: 123    END: 130
```

Finally, if the `-v` or `--verbose` flag is set, your program must also output a brief description of every state-changing event and scheduling decision.

```
At time 11:
    CPU_BURST_COMPLETED
    Thread 0 in process 0 [INTERACTIVE]
    Transitioned from RUNNING to BLOCKED

At time 11:
    DISPATCHER_INVOKED
    Thread 1 in process 0 [INTERACTIVE]
    Selected from 4 threads; will run to completion of burst

At time 14:
    THREAD_DISPATCH_COMPLETED
    Thread 1 in process 0 [INTERACTIVE]
    Transitioned from READY to RUNNING
```

You may deviate slightly from the formats outlined above, as long as the program provided on course website is able to read your simulation's output.

5 Scheduling Algorithms

Your scheduling simulator must support four different scheduling algorithms. These are as follows, with the corresponding flag value indicated in parentheses:

- First-Come, First-Served `(--algorithm FCFS)`
- Round Robin `(--algorithm RR)`
- Process-Priority Scheduling, described below `(--algorithm PRIORITY)`
- An algorithm of your own design, described below `(--algorithm CUSTOM)`

The first two should be implemented as they are described in your textbook. Your RR algorithm should use a hard-coded time slice size of your choosing. The default value of the time quantum should be 3, so that preemption events are common.

`--algorithm PRIORITY`

Your process-priority scheduling algorithm is a non-preemptive algorithm that must use four separate first-come, first-served ready queues. They consist of the following:

- Queue 0: Dedicated to threads whose processes are of type `SYSTEM`.

- Queue 1: Dedicated to threads whose processes are of type `INTERACTIVE`.
- Queue 2: Dedicated to threads whose processes are of type `NORMAL`.
- Queue 3: Dedicated to threads whose processes are of type `BATCH`.

The next thread to run is taken from the front of the highest non-empty queue (with queue 0 being the highest). For example, if queue 0 is non-empty, the next thread will be taken from it. If queue 2 is the highest non-empty queue, the next thread will be drawn from it instead. This scheduling algorithm thus preferentially chooses ready threads from higher-priority process types first.

`--algorithm CUSTOM`

This algorithm is yours to design, though it must meet the following criteria:

- Consist of 4 or more separate queues, all of which must be used somehow (e.g. demotion, aging, or some other mechanism).
- Make use of preemption (e.g. on thread arrival or using a time slice).
- Somehow prioritize processes of higher priority types (e.g. prefer SYSTEM over BATCH threads).
- Attempt to optimize for one of the performance metrics (e.g. response time, turnaround time, CPU efficiency, etc.) or to provide some sort of fairness.

Your textbook describes both HRRN and multi-level feedback queues. Either of these are good starting points for ideas on designing your own algorithm, though you are free to experiment, so long as you meet the above requirements. For example, you might choose to preferentially choose threads from the same process to minimize process switches, or conversely, you might try to opt for fairness and ensure a process with 50 threads doesn't get more CPU time than a process with 1 thread.

You'll need to write a few paragraphs describing the design choices you made for your custom algorithm. Put some thought into it before you start coding, and you'll make your life a bit easier.

6 Next-Event Simulation

Your simulation structure must follow the next-event pattern. At any given time, the simulation is in a single state. The simulation state can only change at event times, where an event is defined as an occurrence that may change the state of the system.

Since the simulation state only changes at an event, the ”clock” can be advanced to the next scheduled event–regardless of whether the next event is 1 or 1,000,000 time units in the future. This is why it is called a ”next-event” simulation model. In our case, time is measured in simple ”units”.

Your simulation must support the following event types:

- `THREAD_ARRIVED`: A thread has been created in the system.

- `THREAD_DISPATCH_COMPLETED`: A thread switch has completed, allowing a new thread to start executing on the CPU.
- `PROCESS_DISPATCH_COMPLETED`: A process switch has completed, allowing a new thread in a different process to start executing on the CPU.
- `CPU_BURST_COMPLETED`: A thread has finished one of its CPU bursts and has initiated an I/O request.
- `IO_BURST_COMPLETED`: A thread has finished one of its I/O bursts and is once again ready to be executed.
- `THREAD_COMPLETED`: A thread has finished the last of its CPU bursts.
- `THREAD_PREEMPTED`: A thread has been preempted during execution of one of its CPU bursts.
- `DISPATCHER_INVOKED`: The OS dispatcher routine has been invoked to determine the next thread to be run on the CPU.

Events are scheduled via an event queue. The event queue is a priority queue that contains future events; the priority of each item in the queue corresponds to its scheduled time, where the event with the highest priority (at the front of the queue) is the one that will happen next.

The main loop of the simulation should consist of processing the next event, perhaps adding more future events in the queue as a result, advancing the clock (by taking the next scheduled event from the front of the event queue), and so on until all threads have terminated.

7 Getting Started

Unlike Project 1, you will not be provided any starter code for project 3. Related materials will be posted on the course website.

8 Requirements and Reference

- Use good design. Do not code monolithic functions. You should avoid coding practices that make for fragile, rigid and redundant code.
- Use good formatting skills. A well formatted project will not only be easier to work in and debug, but it will also make for a happier grader.
- You can develop your project anywhere you want, but it must execute correctly on the machines in the Alamode lab (BB136).
- Your final submission must contain a `README` file with the following information:
  - Your name.
  - A list of all the files in your submission and what each does.
  - Any unusual/interesting features in your programs.

  - Approximate number of hours you spent on the project.
  - A short essay that explains your custom CPU scheduling algorithm. Make sure to cover:
    * How are processes of different priorities handled?
    * What metrics did you try to optimize (e.g. throughput, response time, etc)?
    * How does your algorithm use preemption?
    * How do you make use of the required number of queues?
    * Assuming a constant stream of processes, is starvation possible in your algorithm?
    * Is your algorithm fair? What does that even mean?

- To compile your code, the grader should be able to `cd` into your code directory and simply type `make`.

9 Deliverables

You are required to submit each deliverable by 11:59 on the due date. There are no exceptions for intermediate deliverables. For final deliverables, late submissions will lose 20% of the total score per day. Any requirements not mentioned in an intermediate deliverable are due as part of the final deliverable.

All deliverables must be submitted to the Blackboard! Three entries have been created: P2-D1, P2-D2, and P2-Final.

9.1 Intermediate Deliverable 1: Due March 10, 2017 @ 11:59pm

You must submit a version of your code where:

- Your program determines the file to parse from the command line.
- You have `Process`, `Thread`, `Event`, and `Burst` classes or structs that contains appropriate instance variables.
- Your program reads a file with the required format, populates appropriate data structures (`Process`, `Event`, `Thread`, and `Burst` instances), and sets up the initial event priority queue.
- Your program iterates over the event queue in the appropriate order to output each `THREAD_ARRIVED` events in the correct format.
- You provide a makefile that can build your program by typing "`make`".

Zip or tar all your files into a single file with the name:

```
BlackboardUserName-D1.zip (or .tar)
```

Replace "BlackboardUserName" with your real username. Submit this single file to Blackboard.

9.2 Intermediate Deliverable 2: Due March 24, 2017 @ 11:59pm

You must push a version of your code where:

- You have at least a first-come, first-serve scheduling algorithm implemented.
- All required metrics are displayed on program completion.
- Your program correctly parses and handles all required flags except `--algorithm`.

Zip or tar all your files into a single file with the name:

```
BlackboardUserName-D2.zip (or .tar)
```

Replace ”BlackboardUserName” with your real username. Submit this single file to Blackboard.

9.3 Final Deliverable: Due April 7, 2017 @ 11:59pm

You must submit a completed version of your program where:

- All required algorithms are supported and can be selected via the `--algorithm` flag.
- Your custom algorithm meets all requirements and is described in your README file.

Zip or tar all your files into a single file with the name:

```
BlackboardUserName-Final.zip (or .tar)
```

Replace ”BlackboardUserName” with your real username. Submit this single file to Blackboard.