CMPT 330: Laboratory Three

Discrete Event Simulation — A Study of a CPU Scheduler 9%

Due Date: *Source Code, Executable, & Answers to Questions:* Friday, April 8th, 2022

For this lab create a discrete CPU scheduler simulation in Java to study CPU scheduling. Your program must compile and run on a lab computer in N221. Failure to compile or run may result in a grade of zero. Your task is to implement a discrete event simulation to model a simple operating system. Processes arrive in the system and take turns executing on one or more CPUs, possibly spending some time waiting for I/O. Each process remains in the system until its predetermined computational needs are met. There are a small number of events that can occur within the system, these events will drive the discrete event simulation.

Discrete Event Simulation

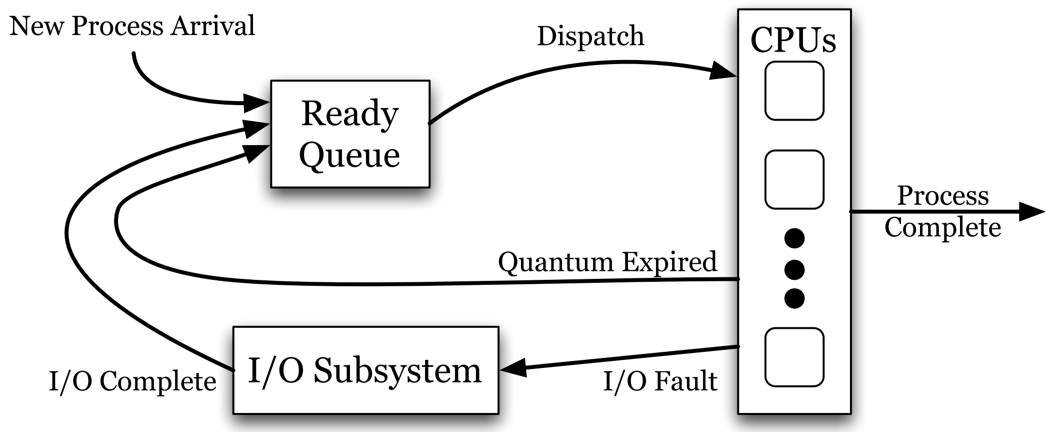
Discrete event simulation is a modeling technique used to simulate real-world systems that can be decomposed into a series of logic *events*, each of which occur at specific times. Discrete event simulation is a useful modeling technique for any system where prior events are independent whose outcomes cannot be affected by the current event. In our simple model, a clock tick will initiate zero or more processes in the system and advance existing processes forward. That is, a simulated clock will tick, and your simulation will determine which parts of the system have to change.[1]

System Description

Processes enter the system onto the end of the *ready queue* and wait for a CPU to become available. The ready queue in this simulation is a round-robin approach. The processes run on a CPU, possibly being preempted by the scheduler, or for I/O service, until their entire service time on the CPU has elapsed at which point they leave the system. A logical clock is used to coordinate all events in the system. Each *tick* of the clock measures a single arbitrary *time unit*.

The *ready queue* (a FIFO queue) contains the processes that are ready to run on a CPU but do not currently have a CPU to run on. When a CPU becomes available the next process in the ready queue is selected for execution (round-robin scheduling).

[1] Your system will use a clock that increments 1 step each time the system loops. Sometime there may be no change in the system other than the clock incrementing. Alternatively, you could use an event queue, but don't use this approach for this assignment. In discrete event simulations that use a queue each event is given a *timestamp* and is stored in a priority queue— the *event queue*. The *event queue* will ensure that the next event removed from the queue has the lowest timestamp, i.e., it is the next event that will occur. The simulation will maintain a *logical clock*, a representation of the current time within the simulation. After an event is processed, the next event to be processed is the event at the front of the event queue since there are no other events that occur between now and the time of the event at the front of the queue. Whenever an event is processed, the system removes the next event from the priority queue, updates the logical clock to the timestamp of the event, processes the event then repeats. We are guaranteed not to have missed an event because the event queue always returns the event with the soonest timestamp. It is possible for two or more events to have the same timestamp modeling concurrent events within the simulation. Although the simulator processes simultaneous events sequentially, they occur at the same logical time.



Each process can only spend up to, but not more than, one *quantum* on a CPU before it is switched out. The quantum is a constant for the entire system. A process leaves the CPU in one of three ways:

1. If the time remaining for the process to complete is less than the time to the next I/O fault or the quantum the process will leave the system. Remember to keep track of how many processes leave the system.
2. If the time remaining until the process has an I/O fault is less than the quantum the process will leave for I/O service (into the I/O queue)
3. If the processes quantum expires return the process to the ready queue.

Each process may execute on a CPU for a set number of clock ticks between each I/O fault until the process completes. The time between each I/O fault is determined by the process' *burst time* and is constant for the lifetime of each process.[2] The process burst time is set when the process is created. Although the burst time is constant for the life of a given process, each process will have a randomly assigned burst-time. See below for how to set the burst time for interactive and batch processes.

Like the burst time, the time needed to service I/O faults varies for each process. Each process is randomly assigned an I/O service time that remains constant for the lifetime of the process. Clearly this is an unrealistic simplification of the problem as it assumes infinite I/O capacity with no loss in throughput.

The Command Line Interface

The Java "Simulator" program accepts five command-line parameters. If fewer than five parameters are provided, then provide a usage instruction message: All parameters are positive integers (greater than zero). In order, they are as follows:

1. Interactive Chance: specifies x for a 1 in x chance of creating an interactive process at each time step
2. Batch Chance: specifies x for a 1 in x chance of creating a batch process at each time step
3. Simulation Time: The number of time steps before the simulation terminates
4. Quantum: The number of time steps a process may spend in a CPU before it is preempted
5. Number of CPUs: The number of CPUs in the simulation

[2] Obviously, this is an approximation since in the real-world processes may have variable burst lengths before they block for I/O.

The simulation can create two types of processes: interactive and batch. The processes are created as follows, although you may find it useful to have additional data members in your process class.

<u>**Interactive**</u>:	<u>**Batch**</u>:
Type: "Interactive" (a String)	Type: "Batch" (a String)
Burst: 10 + / - 20%	Burst: 250 + / - 20%
CPU Time: 20 + / - 20%	CPU Time: 500 + / - 20%
IO Block Time: 5 + / - 20%	IO Block Time: 10 + / - 20%
Creation Time Stamp: time of creation	Creation Time Stamp: time of creation

Other than type, the properties are all positive integers. Note that the burst time is 10 + / - 20%. 20% of 10 is 0.2 * 10 = 2. Thus, the burst time should be randomly selected from 8 to 12 (8, 9, 10, 11, or 12 time steps).

At each time step your simulation should randomly create zero, one, or two processes (none, a batch, an interactive, or a batch and an interactive) based on the probabilities passed in via the command line arguments.

Program Outline

Your program should follow the following approach:

1. Read command line arguments and set parameters
2. Create an empty ready queue, an array of CPUs, an empty I/O queue, and optionally an empty completed queue. You may use a completed queue if you want to keep track of completed processes this way.
3. Loop until the simulation time has expired, at each time step:
    a. Create new processes (zero, one, or two)
        - New processes are put into the ready queue
    b. Process the CPUs
        - Increment the processes in the CPU (clock tick)
        - Remove processes that have completed
        - Move processes that have blocked to the I/O queue
        - Move the processes that have reached their quantum to the ready queue
    c. Process the Ready Queue
        - Move processes into the CPUs if there are available CPUs
    d. Process the I/O Queue
        - Increment the processes in the I/O Queue (clock tick)
        - Move unblocked processes into the ready queue
4. Report statistics – use the following format (including the words specified at the beginning of the line)
    - Ready Processes: number of processes in the ready queue when the simulation ends
    - Processes in CPU: number of processes in the CPU array when the simulation ends
    - Blocked processes: number of processes in the I/O queue when the simulation ends

- Completed processes: number of processes that have completed when the simulation ends
- Accounted for (*ready+CPU+blocked+completed*) of *created* processes
- Number of CPUs: number of CPUs used in the simulation
- Exiting at simulation time: number of simulation steps
- "Simulation Result", CPU quantum, number of completed processes, number of time steps in the simulation, sum of the turnaround time for all completed processes

For example, your program may display output as follows (but unlikely since each execution is different):

```
Ready Processes: 128
Processes in CPU: 2
Blocked processes: 1
Completed processes: 944
Accounted for 1075 of 1075 processes
Number of CPUs: 2
Exiting at simulation time: 10000
Simulation Result,100,944,10000,654741
```

You will likely want additional output while debugging your simulation, but these are the outputs your program should produce when you submit your simulation for marking.

Note that you should increment a counter every time a process is created. After creation a process should be in the ready queue, a CPU, in the I/O queue, or completed.

Use the text "Simulation Result", without quotes, to begin a summary as specified above. This should make it easy to find in your output (think of tools from laboratory one that may be useful here). The CPU quantum was specified as a command line argument. The turnaround time for each process can be determined by subtracting the creation time of a process from the completion time of a process. The sum of all process's turnaround time is printed on this line. Please note that the average turnaround time can be determined since the number of completed processes is also specified on this line.

Simulation Case Studies

You are to determine the non-zero steady state for the system given various settings. To find the steady state, determine the correct chance of creating a process such that the number of processes in the ready queue remains relatively constant regardless of the number of time steps in the simulation. If processes are created too infrequently then they will be processed nearly immediately, and the ready queue will end in a size near zero. If processes are created too often then the size of the ready queue will increase as the length of the simulation increases. Thus, if the simulation time increases, and the number of processes in the ready state remains relatively equal, you have found the steady state of the system. Find the steady state for each of the following configurations:

- 2 CPUs, all the processes created are interactive
- 4 CPUs, all the processes created are interactive
- 2 CPUs, all the processes created are batch
- 4 CPUs, all the processes created are batch
- 2 CPUs, half of the processes are interactive, and half are batch
- 4 CPUs, half of the processes are interactive, and half are batch

To help find the steady state, generate a graph of likelihood of process creation against the number of processes in the ready queue at completion. You may not necessarily find a "sharp" steady state, but you should be able to support your answer from your graph.

In additional to finding the creation percentages to create a steady state, also find the average and standard deviation for the throughput, and the average and standard deviation for the turnaround time for each configuration.

Note that you can make all the processes interactive by making a very large *x* for 1 in *x* chance of creating a batch process. While this may cause an occasional batch process, this should have little effect on the statistics.

The most representative statistics are collected after the system stabilizes, that is, after the system has been under a *load* for a while. The first processes to arrive will enter an unloaded system and their behavior will not be representative of the long-term characteristics of the system. When gathering statistics, run your simulation long enough so that the behavior of the early processes will not have a significant effect on the long-term trends that you are studying.

Remember your bash scripting skills from Lab 1, they will be valuable for quickly generating useful results by modifying input parameters, formatting output, and directing output to a file. You may use an additional program (e.g., Spreadsheet) to create your graphs.

If your simulator is not finished in time to use it for this part, you may generate your results using the one provided. **Two-thirds** of the marks, however, are for creating the simulator and **one-third** of the marks are for creating the graphs, and determining the steady states, throughputs, and turnaround times.

Please note that you do not need a full lab report, but you do need to include a pdf clearly showing your three values (creation percentage for steady state, throughput, turnaround time) for the six configurations (of CPU and process type).

Submission Summary

Put your solution in a folder containing your name and zip it up into a zip file containing your name. For example, JanzenCMPT330Lab3.zip would contain a folder named JanzenCMPT330Lab3. This folder would contain:

- Source code – java files with proper commenting
- Executable – java byte code (a.k.a. class files)
- Trace file – a text file showing a successful execution of your program, including the java command to start your program
- Bash files used to create your statistics (.sh)
- Data file(s) generated for your analysis
- PDF file, using headings/titles to denote the different configurations
    - Six creation percentages to cause steady states
    - Six graphs, one each to support your steady state
    - Six average throughputs, each with a standard deviation
    - Six average turnaround times, each with a standard deviation

Remember that graphs have labeled axis and a title.