Homework 2: Comparing Scheduling Algorithms

The purpose of this homework is to build a scheduler to compare performance of a First In First Out (FIFO) versus Round Robin algorithms. The algorithms will be tested assuming a single processor and multi-processor configurations of 2-4 processors. You can use Java or another programming language (with threads) using producer/consumer logic to emulate an actual O.S. implementation.

Start Code: Producer/Consumer

The idea behind the homework is that an operating system receives jobs to process. We will use Producer/Consumer code to execute the logic: Producers generate jobs and Consumers are processors which process the jobs. You may obtain starting code from the directory: /home/student/Classes/Cs370/BoundedBuffer. This code includes various classes:

- Factory: This class generates all the Producers and Consumers.
- Producer: This is the application that generates Jobs.
- Consumer: This is the processor which processes Jobs.
- BoundedBuffer: This is the FIFO queue where Producers issue jobs to Consumer Processors.
- Job: This class you must create to contain the information about each job.
- SleepUtilities: This is where you can generate the random interarrival and service times, and sleep() for the necessary duration.

The Consumer/Scheduler class shall contain a buffer of Jobs, protected with producer/consumer logic. A Job object can track each job request. The Job must track the required processing time, the job class, and the time the job was requested.

Step 1: Develop a Single-Processor Simulation

A Producer thread generates jobs from an application program (or jobs that request time from the O.S.) A Producer simulates a 'think time', when the user is busy preparing their next terminal input or the program is waiting completion for an I/O; and a Service time, which is the duration that the program expects to be executed. This corresponds to the Blocked (on I/O) and Running states, respectively.

The above can be re-stated as follows: Jobs arrive to be processed every N time units. Each job has a service or processing time of M average time units. We simulate this with two Producer threads, where each generate a number of application jobs by generating requests every N time units. Each Producer Sleep()s for an interval of time (according to their defined interarrival time) then issues a job to the BoundedBuffer (or Ready Queue) to be queued and processed. Each ‘job’ lists its defined service time. We will use a producer/consumer buffer and logic to ensure that the scheduler's buffer of ‘jobs’ never overflows. The buffer can be a list data structure.

Initial Simulation Configuration

There are 2 job types. One Producer will generate I/O bound jobs, and the second Producer will generate the CPU bound jobs:

- Job Class 1: Short I/O bound jobs which arrive every 10 units and require 5 units processing (or service) time.
- Job Class 2: CPU bound jobs which arrive every 100 units and require 50 units processing time.

A 'Scheduler' (consumer thread) processes all job requests according to the selected scheduling algorithm: FIFO and Round Robin. (Get FIFO working, then enhance your code to run RR also.) The Round Robin logic should assume a time quantum of 7 units.

The Scheduler gets jobs off the Bounded Buffer (Ready Queue). The Scheduler alternates between selecting a job from the Ready Queue buffer(s) according to the selected scheduler algorithm, and Sleep()ing for the duration of the processing or 'service' time. The time that a job spends sitting in the Bounded Buffer is the time the job spends sitting in the Ready state/queue, and counts as the 'Wait' Time. In a multiprocessor configuration, multiple jobs can execute simultaneously.

Report Statistics on Jobs and Processors

To compare effectiveness of different algorithms, we will need to gather statistics on a per job-class basis. Modify your program to collect the following statistics (so they can be collected for both algorithms):

- average service time per job class: time the server will actually process the job,
- maximum service time per job class,
- average turnaround (or response) time per job class: end_time – creation_time,
- maximum turnaround (or response) time per job class,
- average wait time per job class: turnaround_time – service_time,
- maximum wait time per job class,
- processor (or CPU) utilization = total consumer busy time / total time,
- processor throughput (jobs/second) = total # of jobs / total # of seconds.

Note that the service time measures the average time jobs are in the Running state, and the wait time is the average time that a job waits in the Ready state/queue. The processor utilization is the % of time the processor(s) are busy, while the throughput is the number of jobs processed over the total run time, in seconds. The processor utilization is also equal to service_time/interarrival_time, summed for all applications.

When you print statistics, generate service time, turnaround time and wait time for each job class (e.g., I/O-bound or CPU-bound). Processor utilization is normally generated per processor.

Print Debug Info

Debug your program to the best of your ability. It is helpful to have an optional print() utility that prints all buffer entries, for debug purposes. Also, please print information about each request and service to verify that the program is executing correctly. This will allow manual checking of the simulation by both you and me. I recommend printing short statements (or writing to a file) to see when jobs are produced and serviced. This can include statements such as Created Job Class 1 job 25 at time 34335, abbreviated as:

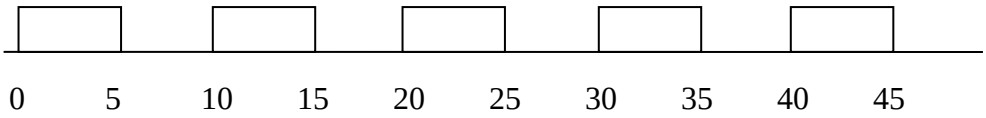
Created JC1:25 34335
Processed JC1:25 34669
Created JC1:26 34779

You may also want to print the buffer after each request and service to ensure that the entries are handled properly. After you have fully debugged and want to get accurate statistics, even minimal debugging information should be commented out to attain more reliable statistics.

Step 2: Do Manual Calculations to Determine Expected Statistics:

You should do an analysis to determine how the jobs should run. In the figure below, we see an example time line of how a single processor would process jobs, if only job class one were

introduced to the processor. You want to draw diagrams to show how processing could occur with both producer applications and a single consumer processor. Include this diagram in the results section of your paper.



Prepare a timeline and manually calculate a value for processor utilization, throughput, average turnaround time, and average wait time for each job class for 100 time units (assuming non-random times.) Do two timelines for FIFO: One assuming 1 processor; the second assuming 2 processors. Do one timeline for Round Robin, assuming 1 processor. *Include the manually-drawn diagrams in your appendix and discuss them in the Analysis section of your paper*. One interesting statistic you can calculate is:

offered rate = service_time/inter-arrival_time .

This statistic provides you the expected load, which is equivalent to the processor utilization. If your processor utilization exceeds the number of processors, then jobs will end up queuing and the queue will get longer and longer (unless you assume a second processor).

Step 3: Fix Problems in Your Program

Now that you know what statistics your program SHOULD produce, we need to get your program to produce these statistics (to a fairly high degree of accuracy). Generally there are two types of errors that you must guard against (but of course other types of errors can also occur):

- The OS timing is not precise enough to run simulations. This can be minimized by multiplying your Sleep time by a given factor in order to obtain accuracy. If a job is to sleep for n units, call Sleep(n*100) or Sleep(n*1000); Do this for both producer and consumer sleeping. The larger the multiple is, the more accurate your result will be – but the longer the simulation will run.
- When your generated output is lengthy, the output tends to delay the simulation and throw off statistics. Keep your output to a short and concise minimum.
- When you have sufficient results and you want to stop your program, you will need to coordinate all your threads to print statistics. There are multiple ways to do this, but one way is to interrupt all the threads using the following interrupt() capability at the group level (causing exceptions for each thread):

  Thread.currentThread ().getThreadGroup ().interrupt();

For debug purposes, run the simulation for 200 units (or two full iterations). When you have your manually calculated results matching your programmed results, print out a copy of the output for the two suggested simulations: 1) with 1 processor; and 2) with 2 processors, for both scheduling algorithms: FIFO and Round Robin. *Include the generated output as an appendix to your paper*. Generate your report statistics using a much longer run.

Step 4: Use Random Data instead of Fixed Data

Once you have debugged your logic, perform simulations for random type simulations for a much longer interval. Usually an 'exponential distribution' is used for the inter-arrival and service times. Use a random-number generator to generate each interarrival time and service time using the following Java equation:

time = avgTime * -Math.log(Math.random());

Run this simulation for 10000 units, at least. If your average service time is not where it should be, probably the simulation is not running for long enough. With enough time, your average should be very close to the expected average service time, but the maximum service time may be quite different due to randomness. Your average turnaround time and wait times should be close to the manually calculated, but will be slightly larger. Why?

Once you have the program running correctly for both algorithms, *print out a copy of your program(s) and include it as an appendix to your paper.*

Now you can run the actual simulations that will be used to compare the algorithms. *Include the generated program statistics in the appendix of your paper, labeled properly as to which test was run. (Only the last page of Arriving/Departing job output is necessary if your statistics are close!) Also include the statistics in a table in the Results section of your paper, showing both your manually-calculated (for certain simulations) and program-generated results, for each simulation.* Run the following simulations for both algorithms: FIFO and Round Robin.

- 1 Processor: I/O Bound inter-arrival time=10, service=5; CPU bound inter-arrival=100, service=50
- 2 Processors: I/O Bound inter-arrival time=10, service=5; CPU bound inter-arrival=100, service=50
- 4 Processors: I/O Bound inter-arrival time=3, service=5; CPU bound inter-arrival=30, service=50

What results do you see and why?

Step 5: Perform Analysis and Complete Write-up

The analysis section of your lab report should focus on two major sections:

Accuracy of Simulation: Did the manually-generated statistics match the program-generated results? How far off was it and why? Was the producer/consumer logic effective in simulating this logic? How did the simulation estimates perform using random versus non-random times?

Does the Buffer size affect the accuracy of the simulation? Why?

The producer/consumer logic will tend to slow down each producer's job requests when the processor is very busy, which is realistic since requests may be received more sporadically when response time is poor. (Why will the producer/consumer logic work this way?)

Comparison of Scheduling Algorithms: Compare how the two scheduling algorithms performed: FIFO and Round Robin, with one, two, and four processors. Did the scheduler favor I/O bound or CPU bound jobs? Why? Was there any starvation? Which algorithm would you recommend?

Try to answer the questions posed in above sections as part of your analysis.

The write-up should be done in Lab Report format. Be sure to include the following in your report:

- A table in your Results section showing results for each test, including manual, when available. (Only three tests must have manual results.)

In the Analysis:

- Three manually drawn time sequence of the 2 algorithms, as described above.
- Manually calculated statistics and hopefully closely matching program-generated statistics. (Small errors are ok. Generous errors mean program defects.)
- An analysis that answers the main questions listed above.

In the Appendix:

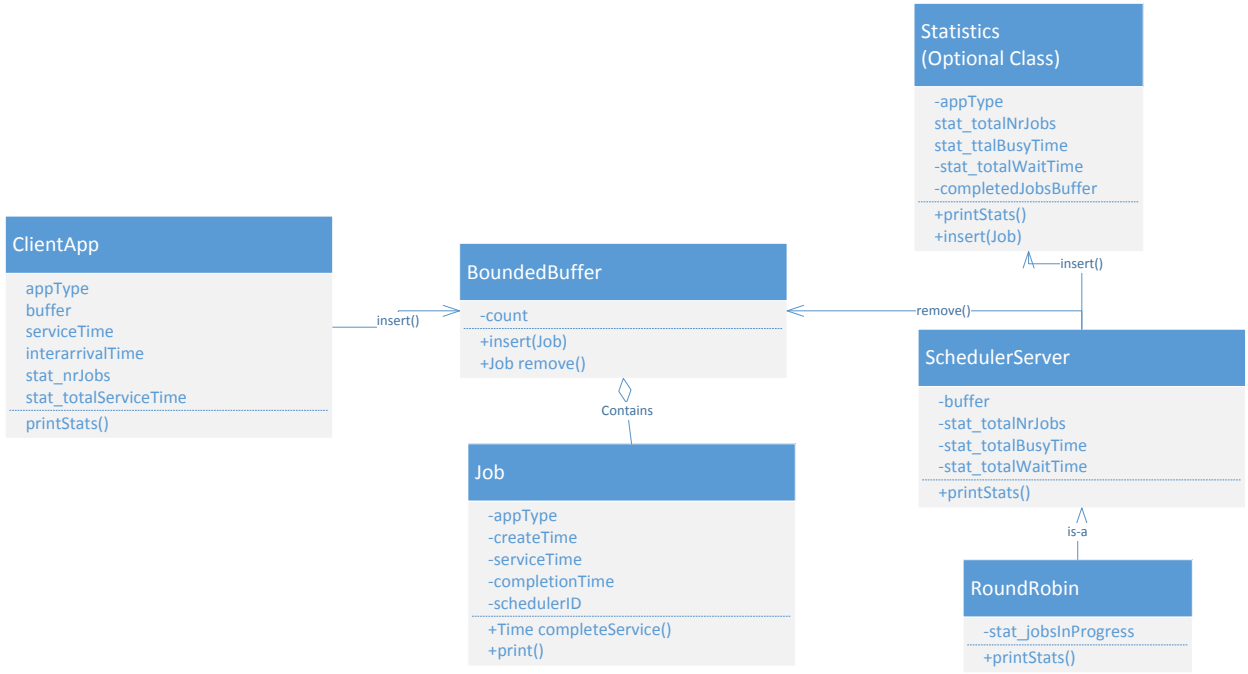
- Output showing how jobs are processed for each algorithm - for two complete cycles for non-random data. It should match your manually-drawn time scheme.
- For each random data test, the last page of output showing the output statistics generated.

In a separate dropbox:

- Code for the Round Robin scenario.

Code Hints

Here is a class diagram that may help you understand the configuration.



The purpose of the ClientApp is to generate (or produce) jobs every interarrivalTime. The ClientApp includes the appType, which is a number indicating the application type. This number is important to track which application class statistic should be incremented. The appType and serviceTime is put into every generated Job, so that they can be processed properly. Each ClientApp also tracks statistics on the jobs it generates and prints these at the end of the program (mainly for debug reasons.)

The Job class is what the ClientApps produce. At the time they are generated, a createTime should be saved in each instance. Then it will be easy to calculate the turnaround time and the wait time.

The SchedulerServer is the Consumer class. It removes a Job from the BoundedBuffer and sleeps for the job's serviceTime. It increments all its total statistics from each job, to track the total time it is busy. This will become part of the processor utilization. At the end of the simulation, we need to minimally print the amount of time the processor was busy.

The other statistics that need to be printed include average and maximum service times, wait times and response times per application class. These can be incremented in a static array of the SchedulerServer or a separate Statistics class of some sort.

The Factory class generates all the necessary instances, and should shut down the simulation at the end of your scheduled time. You may use the interrupt() logic provided above to get the full simulation to abort – but be ready to print statistics in your exception handling logic.

References:

For additional information see the Operating Systems text:

- Producer/Consumer logic as discussed Synchronization Chapter.
- FCFS & Priority & Round Robin algorithms in Scheduling Chapter.
- My web pages on CPU Scheduling and their statistics.

Note: This simulation would best be performed using a simulation language. Because Sleep() is sometimes inaccurate, our simulation will be close but not precise. A simulation language implementation would give much more accurate results. However the purpose of this class is to learn about semaphores, producer / consumer problems and CPU scheduling, and this homework accomplishes that.

Producer-Consumer Lab

The idea behind the homework is that an operating system receives jobs to process. We will use Producer/Consumer code to execute the logic: Producers generate jobs and Consumers are processors which process the jobs. You may obtain starting code from the directory: /home/student/Classes/Cs370/BoundedBuffer. This code includes various classes:

- Factory: This class generates all the Producers and Consumers.
- Producer: This is the application that generates Jobs.
- Consumer: This is the processor which processes Jobs.
- BoundedBuffer: This is the FIFO queue where Producers issue jobs to Consumer Processors.
- Job: This class you must create to contain the information about each job.
- SleepUtilities: This is where you can generate the random interarrival and service times, and sleep() for the necessary duration.

Let's get you familiar with this code. Copy the .java code into your directory.

1. How many producers and consumers are currently generated? Look in Factory.java. Change it generate 2 producers and 2 consumers. Will each thread have its own object or will it share instances as currently defined?

2. SleepUtilities.java contains one or more nap() methods. What happens if there is a parameter passed, versus not?

3. In BoundedBuffer.java, three semaphores are used. What are their names and what are they initialized to?

4. How is Buffer.java a different implementation of BoundedBuffer?

5. Look in Producer.java and Consumer.java. What is produced to put into Buffer?

6. Compile and run the code in your own directory. Estimate the average number of objects in BoundedBuffer. If you can't tell, add print statements to determine the current number of objects in the buffer.

7. How many producers and consumers appear active at any point in time? Add print statements if you need to. Does this coincide with what you see produced in Factory? Why

or why not? For example, with two producers, would there be two items in the BoundedBuffer often?

8. If this were an Operating System scheduler, the Producers would be the application programs and the Consumer would be the O.S. Scheduler. The BoundedBuffer would be the Ready Queue. These could in fact be renamed as such. Do you think this might work? What might need to be changed to get this to work?

9. If this were an O.S., we would like to collect the amount of time the Scheduler was busy (as the processor utilization). Therefore, all time napping in the Scheduler should be summed and printed at the end. Make this change. What did you do?

10. To determine the processor utilization, we also need to know the total time. Have the parent run the simulation for 20 seconds and then interrupt all its children threads. Have the children threads print their statistics as they exit. Have the parent calculate and print the total run time, by subtracting the end time from the start time.