Homework 2: Implementing a Firewall with Load Sharing

The purpose of this homework is to simulate a redundant firewall to compare performance of one versus multiple processors, and various queue sizes. The algorithms will be tested assuming a single processor and multi-processor configurations of 1-2 processors. You can use Java or another programming language (with threads) using semaphore logic to emulate an actual firewall implementation. The Barber semaphore example might be a good place to start.

Start Code: Barber

The idea behind the homework is that a firewall receives packets to process. We will use code similar to the Producer/Consumer code to execute the logic: Producers generate Packets and Consumers are processors in a firewall which process the packets. You may obtain starting code from the directory: /home/student/Classes/Cs370/BoundedBuffer and merge it with my example online of the Barber problem. This code includes various classes:

- Factory: This generates all the Producers (Network) and Consumers (Firewall Processors).
- Network or Producer: This is the network that generates packets.
- Processor or Consumer: This is the firewall processor which processes packets.
- BoundedBuffer: This is the FIFO queue where Producers (the network) issue packets to Consumer firewall processors. You will want to modify it to enable a queue count, if a processor is not immediately available. (Use the code that is in the Barber example.)
- Packet: This class you create to track information about each packet.
- SleepUtilities: This is where you can generate the random interarrival and service times, and sleep() for the necessary duration.

The Consumer/Firewall class shall contain a buffer of Packets, protected with barber logic. A Packet object tracks each packet request. The Packet must track whether the packet was created, queued or discarded, the required processing time, and the time the packet was completed processing (end time).

Step 1: Develop a Single-Processor Simulation

A Producer thread generates arriving packets (or packets that request time from the firewall.) A Network/Producer simulates a 'producing time', when the network is busy preparing their next packet; and a Service time, which is the duration that the packet is actually processed by the firewall. The network-producer will in a loop: 1) generate a packet; 2) queue the packet for the firewall processor; 3) sleep for an interarrival time N.

The above can be re-stated as follows: Packets arrive to be processed every N time units. Each packet has a service or processing time of M average time units. We simulate this with one Producer thread. The Producer Sleep()s for an interval of time (according to their defined interarrival time) then issues a packet to the BoundedBuffer (or Receive Queue) to be queued and processed. Each ‘packet’ lists its defined service time. We will use a queued buffer and semaphore logic to ensure that the firewall's buffer of ‘Packets’ hopefully never overflows. The buffer can be a list data structure.

Initial Simulation Configuration

One Producer will generate Packets:

- Packet: Packets which arrive every 10 units and require 5 units processing (or service) time.

A 'Firewall' (consumer thread(s)) processes all packet requests according to the FIFO scheduling algorithm. A firewall processor gets Packets off the Bounded Buffer (or Receive queue). The Firewall processors alternate between selecting a packet from the Receive queue buffer(s) according to a FIFO mechanism, and Sleep()ing for the duration of the processing or 'service' time. The time that a packet spends sitting in the Bounded Buffer is the time the packet spends waiting in the receive queue, and counts as the 'Wait' Time. In a multiprocessor firewall configuration, multiple Packets can execute simultaneously.

Code Hints

The purpose of the Network-Producer is to generate (or produce) Packets every interarrivalTime. The Network-Producer puts a serviceTime is put into every generated Packet, so that they can be processed properly. Each Packet is entered into the BoundedBuffer. The Network-Producer also tracks statistics on the Packets it generates and prints these at the end of the program (mainly for debug reasons.)

The Packet is what the Network-Producer produces. 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 BoundedBuffer is equivalent to the Barber semaphore example's Customer logic. Packets can only be queued if there is space for them. Packets should block for a Firewall Processor to become available, and Firewall Processors should block if there are no Packets to process.

The Firewall Server is the Consumer. It removes a Packet from the BoundedBuffer and sleeps for the packet's serviceTime. Use the Barber logic to accomplish this. It increments all its total statistics from each packet, to track the total time it is busy. This will become part of its 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 turnaround times. These can be incremented in a static array of the FirewallServer or a separate Statistics of some sort.

The Factory 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.

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 Packets are produced and serviced. This can include statements such as Created packet 25 at time 34335, abbreviated as:

Created P:25 34335
Processed P:25 34669
Created P:26 34779
Discarded P: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: Report Statistics on Packets and Processors

To compare effectiveness of different queue lengths and number of processors, we will need to gather statistics on a per packet basis. Modify your program to collect the following statistics (so they can be collected for both algorithms):

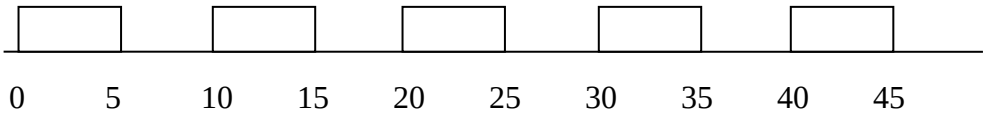
- % of discarded packets: the average percent of packets for which the queue was too long and packets were discarded.
- average service time per packet: time the server will actually process the packet;
- maximum service time per packet;
- average turnaround (or processing) time per packet : end_time – creation_time;
- maximum turnaround (or processing) time per packet;
- average wait time per packet : turnaround_time – service_time;
- maximum wait time per packet;
- processor (or CPU) utilization = total processor busy time / total time;
- processor throughput (packets/second) = total # of packets / total # of seconds.

Note that the service time measures the average time Packets are being processed, and the wait time is the average time that a packet waits in the Receive queue. The processor utilization is the % of time the processor(s) are busy, while the throughput is the number of Packets processed over the total run time, in seconds. The expected 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 average packets. Processor utilization is normally generated per processor.

Step 3: Do Manual Calculations to Determine Expected Statistics:

You should do an analysis to determine how the Packets should run. In the figure below, we see an example time line of how a single processor would process Packets, if only packets were introduced to one processor. You want to draw diagrams to show how processing could occur with one producer network and one or two consumer (firewall) processors. 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 packet for 100 time units (assuming non-random times.) Do two timelines: One assuming 1 processor; the second assuming 2 processors. *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.

Where the interarrival_time is the duration of time between when the producer generates subsequent packets, and the service_time is the time for the server to process any packet. This statistic provides you the expected load, which is equivalent to the processor utilization. If your processor utilization exceeds the number of processors, then Packets will end up queuing and the queue will get longer and longer (unless you assume a second processor). In this example, packets not fitting in the queue should be deleted.

Step 4: 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 packet 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();

  This will work if you have try-catch blocks everywhere that threads may be blocked (such as semaphores) and that as part of your try-catch blocks you exit and/or print statistics.

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. *Include the generated output as an appendix to your paper.* Generate your report statistics using a much longer run.

Step 5: 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 packet 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 the FIFO algorithm: (no Round Robin)

- 1 Processor: inter-arrival time=5, service=8; queue size=5
- 2 Processors: inter-arrival time=5, service=8; queue size = 5

- 4 Processors: inter-arrival time=2, service=5; queue size = 5
- 4 Processors: inter-arrival time=2, service=5; queue size = 1

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 barber shop logic effective in simulating this logic? How did the simulation estimates perform using random versus non-random times?

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

Comparison of Scheduling Algorithms: Compare how algorithms performed: varying queue size, with one, two, and four processors. Did the firewall process all packets? Why or why not? Was there any starvation? Which method 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.

In the Analysis:

- Two manually drawn time sequence of the 1 and 2 processors, 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 Packets 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 2-processor 5 buffer-queue scenario.

References:

For additional information see the Firewalls text:

- Producer/Consumer logic as discussed Synchronization Chapter.
- FCFS 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 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 firewall receives Packets to process. We will use Producer/Consumer code to execute the logic: Producers generate Packets and Consumers are processors which process the Packets. You may obtain starting code from the directory: /home/student/es/Cs370/BoundedBuffer. This code includes various es:

- Factory: This generates all the Producers and Consumers.
- Producer: This is the application that generates Packets.
- Consumer: This is the processor which processes Packets.
- BoundedBuffer: This is the FIFO queue where Producers issue Packets to Consumer Processors.
- Packet: This you must create to contain the information about each packet.
- 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 Firewall firewall, the Producers would be the application programs and the Consumer would be the O.S. Firewall. The BoundedBuffer would be the Receive 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 Firewall was busy (as the processor utilization). Therefore, all time napping in the Firewall 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.