CSCI 204 – Introduction to Computer Science II

Lab 11 - Queue

1. Objectives

In this lab, you will:

- Get experience with Queues through completing a basic Queue ADT
- Get an appreciation of how important queuing algorithms are by use a queue to simulate a simple resource management algorithm

2. Introduction

In most systems today, computational resources are shared among many processes or users. With most modern operating systems, resource sharing and allocation is a critical task that is managed within the operating system. For example, consider a cluster of servers that are configured to share computational resources in order to run numerous CPU-intensive jobs simultaneously. It is quite possible that those servers are sharing one RAID for its shared file system. How do each of the nodes on the cluster write to the shared file system? A special system is reserved to manage requests to read/write from the single file system. Or, consider a multi-threaded application running on a modern quad-core system. Each CPU is a resource that needs to be managed among hundreds of threads and processes running in the system. How are those cores divided to that they are shared equally among the all threads of execution on the system? Or, consider a web server running a highly-visible website such as amazon.com, or google.com. Surely their servers consist of more than a single desktop computer! They likely have a *very* large array of systems all sharing the load of processing requests coming in simultaneously from millions of browers worldwide!

There are numerous scenarios that can be used to illustrate the general picture of the **client-server model** in computer systems. In this model, servers are configured to manage resources that are available for clients to use. If a client wants to use a resource, it sends the request to the server, and the server adds the request to a **queue**. When the resource becomes available, the request is removed from the queue and processed, thereby freeing up a slot in the queue for other requests.

There are a wide range of algorithms in use that determine how queues are managed. However, usually the goal is the same – we want to achieve a balanced load among all resources that are serving requests. No one server should be carrying the burden of serving the majority of the requests, and no servers should be starving for attention! How can you handle processing these requests? Keep in mind that in many scenarios, requests do not take equal amounts of time to process. This problem is complicated, and one that has been studied extensively in OS design research, specifically tapping much that has been learned from queuing theory.

1. Our simulation

We are going to simulate a simple queue strategy for handling a pool of S servers. A server will have nothing more than a single queue with a fixed-size capacity. The simulation will have 1 client sending thousands of requests randomly at a capped rate to allow requests to vary in intensity. Each

request will be sent through one method, which will handle the all-important task of determining which server to send the request to.

You will observe the overall load balance among all of the queues. Ideally, we want to keep the queues uniformly busy. You will first implement a simple round robin strategy, and then a random sampling strategy. You will find that it doesn't really take a very difficult strategy to achieve a reasonably balanced load.

3. Getting Started

As usual, you should set up your Eclipse workspace for today's lab. Import your files you need from

`~csci204/2011-spring/student/labs/lab10`

You should have four files, `QueueArray.java`, and a JUnit test file, `QueueArrayTest.java`. You will also have a `ResourceSim.java` which is the main simulation program, and `GUI.java` which provides a graphical interface to the simulation.

1. Adding the JUnit Library to Your Project

Add the JUnit library to your path by right-clicking on the project name lab11, and select **Properties**. In the right hand pane, select **Java Build Path**, then select the **Libraries** tab. Select **Add Library**, and then select **JUnit**. Specify version 4 of the JUnit library, then click **Finish** when complete.

Create a `readme.txt` file to store some answers to questions you will be asked throughout this lab. Put the `readme.txt` file in the main project folder so that it gets committed properly.

Be sure to commit everything into SVN, and then recommit your work as you are completing your exercises.

4. Part 1 – Finish the QueueArray<T> implemention

Take a moment to explore QueueArray<T>. In this queue implementation, the queue size will be fixed. If the queue is filled up, it stayed filled until an object is removed. As you browse the current QueueArray<T> implementation, you will notice that it is far from complete. In particular, pay close attention to any comment that has a **TODO** tag with the corresponding comment, as this will indicate that you have work to do. (You should notice that Eclipse places a special marker in the margin next to any TODO comment.

The list of the methods that need completion:

- `isEmpty()`
- `isFull()`
- `enqueue()`
- `dequeue()`
- `peek()`

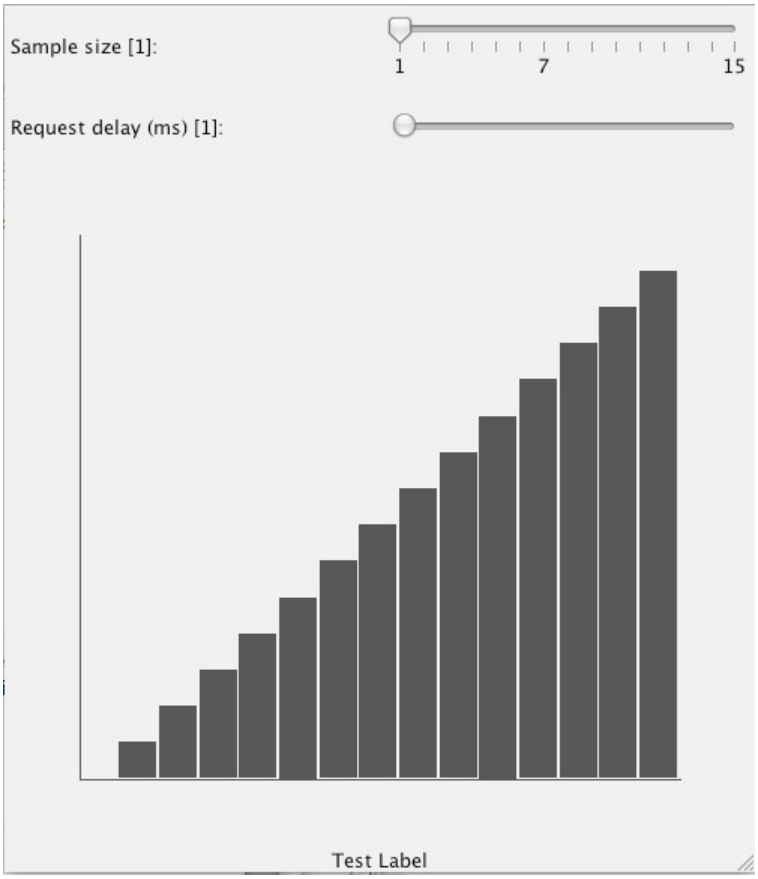
Follow the comments in each method to determine how to finish the implementation

1. Testing your implementation with JUnit

You have been provided a *complete* JUnit test suite to test every method of importance. You do not need to add or modify this method. However, you do need to use it to verify that your Queue implementation is complete. You will not have a complete QueueArray implementation until this test fully passes. Do not go any further until this passes.

5. The `GUI` class

There are two important classes remaining. The first class, `GUI`, handles all of the graphical elements for the user interface. You are not required to do anything with this class. However, before you go on, you should test it out by running the main method in the GUI class. You should see something that looks like this:



Notice that there are two slider bars at the top. One will control the number of servers that you can sample from when scheduling the next server, and the second slider controls how fast you want the client to generate requests to the pool of servers. The second slider will provide for allowing you to obtain a reasonable delay to try to maintain a roughly 50% load average. Notice the label at the bottom, since this will output the current load average and the variance observed among all of the individual server loads.

6. This `ResourceSim` class

First, read this description, and then go through the code to fix the **TODO** tasks.

This is the class that handles running the simulation. There are `NUM_SERVER` servers created, each with its own queue of fixed size of `QUEUE_SIZE` length. A server has its own abstraction managed by the `Server` nested class. The `ResourceSim` constructor handles instantiating each of these servers, and starting them up by invoking the server's `start()` method (which is inherited from the `Thread` class. Threads are cool, but not important for this assignment.) If you explore the `run()` method in the `Server` class, this handles the server-side simulation. So, where is the client-side simulation? (i.e. where are requests to the servers being generated? The single client is simulated by the `main` program itself. It is in a loop that just keeps generating requests, and delaying some amount of time in between requests. A request is sent to the server pool by calling the `addJob(Integer)` method of `ResourceSim`. The integer represents the amount of time (in ms) that the job will take to complete.

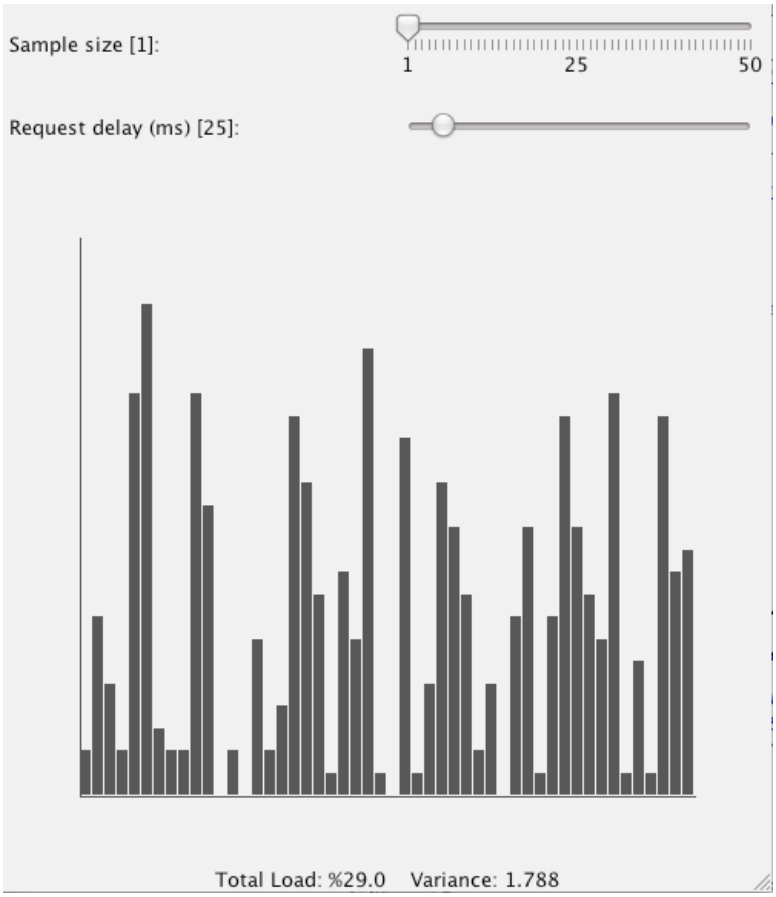
Finally, after a request is sent, there is a delay between requests that is determined by the second slider in the GUI. The simulation runs infinitely until you terminate the program.

Now, take a moment to go through the file as is, paying CLOSE attention to the places where there is a **TODO** indicator. The code and comments contain all of the help that you need to complete it. Each TODO mark is commented with instructions. First, take care of the two marks that prevent compilation. Run the simulation, just to make sure that a GUI is brought up. And then finish the remaining TODO tasks. You will need extra help with the `scheduleNextServer()` method. This is described next.

1. scheduleNextServer()

The most important method you need to write is how to schedule the next server. You will do this in two steps. First, you will ignore the slider value and simply schedule the next server using a round-robin scheduler. This involve just returning the index of the next server in order. If the last server in the list was scheduled last, then start from the beginning of the server list again.

You should be able to run the simulation and observe it running in real time. Adjust the delay so that you can stabilize the total load balance to be somewhere around 30% - 40%. Your screen should look something like this after running it for 30-60 seconds…



Record a rough estimate of the variance in the load among all of the servers. What is the delay required between requests? Did you still have servers rejecting requests? At what overall load did servers start to reject requests? Report your results along with your interpretation of these results in the readme.txt file.

You will notice, by simply observing the current load histogram, that the servers are hardly balanced! This is not ideal.

2. Improve the scheduler

Next, implement the improved scheduler that selects some number of servers at random, and then selects the server from those randomly selected with the smallest load as the server that will be schedule. Clearly, if you sample the entire suite of servers every time, and select the smallest load, that gives us the best possible load we can ask for! However, that is too much in real-world situations. Implement the random sample solution. Read the comment carefully. It tells you how to get the value from the slider so that you can vary the sample size in real time.

Finally, start your experiments. Start with a sample size of 1, then 2, then 3…. Keep observing the behavior of the overall load and variance as you increase the sample size until you don't see any significant change. What sample size did you need before you obtained a good variance? Any other observations?

Start your experiment with 50 servers, than 100 servers, then 250 servers. Record your results.

Store all of your results in `readme.txt`.

7. When you are done

Submit everything to SVN when complete