CSCI 204 – Introduction to Computer Science II

Lab 7 – Queue ADT

1. Objectives

In this lab, you will practice the following:

- Implement the Queue ADT using a structure of your choice, e.g., array or linked list
- Experiment with various settings of a queuing system by modifying program segments
- 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 (*redundant array of independent disks*, one type of disk storage system) for its shared file system. How does each of the nodes on the cluster write to the shared file system? A special system is used 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 so 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 browsers 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.

3. Simulation

We are going to simulate a few simple queuing strategies for handling a pool of *S* servers. Each server will have a single queue with a fixed-size capacity. The simulation will have one client sending thousands of requests randomly at a capped rate to allow requests to vary in intensity. Each request will be sent through one manager, 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 randomized load-balancing strategy. You will find that it doesn't really take a very difficult strategy to achieve a reasonably balanced load.

4. Getting started

Make a directory for this lab and copy all the files from the directory,

```
~csci204/2017-fall/student/labs/lab07/
```

You should see three files, namely

```
resourcesim.py mtTkinter.py graphics.py
```

You will create the file `myqueue.py` to implement the queue.

These files give you a starting point for your lab work. You will only be editing `myqueue.py` and `resourcesim.py`.

5. The Queue

Task 1: Write a complete Queue ADT in **myqueue.py**. You can pick any implementation. For example, you can revise the array implementation of the Stack ADT you wrote in last week's lab; or you can use linked list implementation; or you can use the Python list implementation.

You must name your file **myqueue.py**. You must name your class **Queue**.

The resource simulator uses two kinds of queues; bounded and limitless. Your Queue class will provide both at once.

a) The constructor for the queue class takes an optional parameter which indicates the max size of the queue. If no size is given, the queue is unbounded.
b) The **`__len__(self)`** method returns the current size of the queue.
c) The **`is_empty(self)`** method returns `True` if the queue is empty and `False` if it is not.
d) If **`enqueue(self, item)`** is called on a bounded queue at max capacity, return **-1** to indicate a failure and do not add the item. Otherwise (operation is successful) return the size of the queue. The method **enqueue()** adds the item to the "entering" end of the queue. Note that the `item` is the data. You will need to store the `item` in a node that is to be inserted into the queue.
e) The method **`dequeue(self)`** both removes and returns the item at the "leaving" end of the queue. If the method **dequeue()** is called on an empty queue, it returns **`None`**.
f) The **`peek(self)`** method returns the item at the "leaving" end of the queue, which is ready to be removed. If peek is called on an empty queue, it returns **`None`**.

6. The Simulation

The **ResourceSim** class in `resourcesim.py` handles the running of the simulation.

The **main()** method runs a resource management simulation which uses both bounded and unbounded queues. Run it to test that your Queue class works. (You must complete the Queue class first.) When the program is running, the GUI shows a set of queues full of items (e.g., job requests to a CPU, or a set of CPUs). Currently all the requests are put into the first queue. If you implemented the queue correctly, the first queue on the GUI should fill up very quickly and no exceptions should occur. You should see the message

```
Server 0 too full and failed to enqueue.
```

repeated over and over as the first server becomes too full. Close the simulator's window to quit the simulation. (If you run the program from the command line, you may also need to type *Control-C* to stop the program.)

When running the program, the GUI (Graphics User Interface) displays two sliding bars at the top. One controls the number of servers from which you can sample when picking the next server for the incoming job, and the second slider controls the rate at which the client can generate requests. The second slider allows you to set a reasonable job arrival rate to maintain a roughly 50% load average across the servers. Notice the label at the bottom, which indicates the current load average and the variance observed among all of the individual server loads.

Open the file `resourcesim.py`. The class specifies values such as the `NUM_SERVERS` servers created, each with its own queue of fixed size of `QUEUE_SIZE` length. The *ResourceSim* class constructor handles instantiating and starting each of these servers. If you explore the *run()* method in the *ResouceSim* class, you will find that the method handles the server-side simulation. So, where is the client-side simulation? That is, where are requests to the servers being generated? The single client is simulated by the *run()* method in *ResourceSim*. It is in a loop that just keeps generating requests, and delaying some amount of time in between requests (`time.sleep()`). A request is sent to the server pool by calling the `add_job()` method of the *ResourceSim* class. The integer parameter to the `add_job()` method 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.

7. Round Robin Scheduling

Currently, the simulation schedules all tasks on the first server (which promptly becomes too full).
A **round robin scheduler** will pick each server in turn, one after the other so that all servers are used equally.

Task 2: Implement Round Robin Scheduling by editing the `select_next_server()` method in the **ResouirceSim** class. Comment out the old code.

The **`select_next_server()`** method decides which server to schedule next. It returns an integer that is the index of the chosen server. The default mechanism is to use the `self.lastServer` value. Comment out this line. For the moment, ignore the sample size value.

Find 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. This is the round-robin scheduling algorithm.

Task 3: Test your Round Robin Scheduler and answer the following questions. Save your answers in a text file `lab.txt`.

You should be able to run the simulation and observe it running in real time. Adjust the request delay parameter and sample size through the GUI so that you can stabilize the system. For example, try the sample size of 2 and request delay of 10 ms. Your screen should look something similar in Figure 1 after running it for 30-60 seconds, though the exact values of sample size or request delay may vary.

When the system is in stable condition, answer the following questions.

1. Record the variance in the load among all of the servers.
2. What is the delay required between requests? (Where did you set it?)
3. Do you still have servers rejecting requests? (Yes, No)
4. Lower the delay to find the (stable) point when servers just start to reject requests. What is the total load?

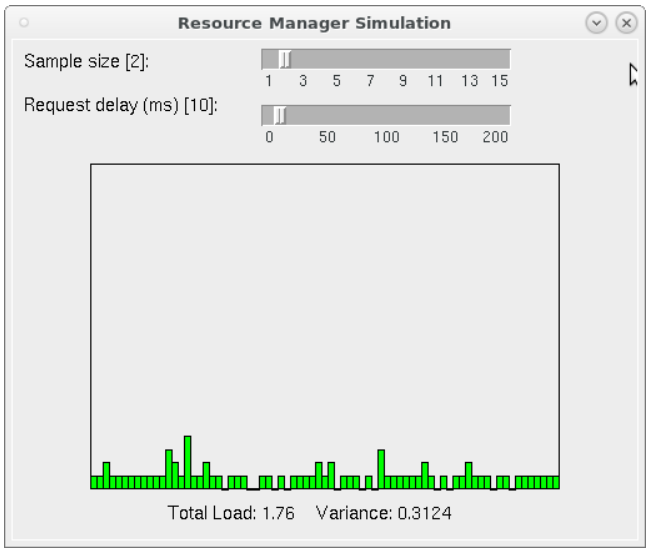

Figure 1: A sample configuration in which the system is in a stable state

8. Randomized Load-Balancing Scheduling

You will notice, by simply observing the current load histogram, that the servers using round robin scheduling are hardly balanced! We can improve this! A **randomized load-balancing scheduler** will pick the server with the lowest load from a randomly selected subset of the servers.

Task 4: Implement Randomized Load-Balancing Scheduling by editing the `select_next_server()` method.

DO NOT DELETE YOUR ROUND ROBIN CODE! Instead, comment out your Round Robin code.

Implement this scheduler so that it selects some number of servers at random and, from those, selects the server with the smallest load. 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 work in real-world situations. Instead, implement the random sample solution to see how few servers we can sample and still get good results.

To implement this, get the current number of samples from the GUI slider bar. Then, write a loop to randomly select that number of servers, and return the index of the server that has the lowest load. The Server class has a method called `get_current_load()` which returns the current load of the server. You can use this method to retrieve the server load.

Task 5: Test your Randomized Load-Balancing Scheduler and answer the following questions. Save your answers in `lab.txt`.

You should be able to run the simulation and observe it running in real time.

a) Set the sample size to 1.
b) Adjust the delay so that you bring the system to a stable state with the minimum value of delay (a boundary condition). In another word, if you reduce the delay any further, the queue length or the load will continually increase to a point when the queues will get full. Answer the following questions.
    1. Record the variance in the load among all of the servers.
    2. What is the delay required between requests? (Where did you set it?)
    3. Do you still have servers rejecting requests?
    4. Lower the delay to find the (stable) point when servers just start to reject requests. What is the total load?

d) Repeat this with sample size 3.
e) Repeat this with sample size 5.
f) Then figure out the lowest sample size you need to get the Variance under 1.

9. Final submission

Submit all your lab files to Moodle (`resourcesim.py` and `myqueue.py` and `lab.txt`) so that grader(s) can run your program directly from your folder. Double check on Moodle and make sure your files were submitted successfully.