Lab 4

Submission deadline: 2359, ~~Sunday, September 24, 2017.~~ Monday, September 25, 2017.

Prerequisites

This lab assumes that students:

- have already attempted Lab 3
- have an understanding of the customer/server system being simulated
- are familiar with CS2030 coding style and javadoc documentation
- are comfortable with using Java Collection Framework
- are familiar with random number generation in `java.lang.Math`
- are familiar with static methods and variables
- are familiar with inheritance and interfaces

Learning Objectives

After completing this lab, students should:

- develop a better appreciation of object-oriented programming, especially in encapsulating data and behavior within classes
- develop the mentality to think about future possible extensions beyond current requirement and design for change.

Setup

There is no new skeleton code provided. You are to build from your Lab 4 solution based on your Lab 3. To setup Lab 4, do the following.

- Login to `cs2030-i`
- Copy `~/lab03` to `~/lab04`
- Rename `LabThree.java` to `LabFour.java`
- Rename the class `LabThree` to `LabFour`
- Copy the test data (`TESTDATA1.txt` .. `TESTDATA4.txt`) from `~cs2030/lab04` to `~/lab04` [1] [#fn:1]

Task

For Lab 4, you will be asked to make changes to your Labs 2 and 3, to:

- Allow multiple servers to serve customers at the same time
- Allow multiple customers waiting at the same time, with two configurations -- either everyone wait at a single queue (e.g., at SingPost Post Office) or every server has its own queue.

As a result of this, you might realize that there is a better way to encapsulate the data and the behavior of the various entities in the program. In which case, you may want to reorganize your classes, create new classes, etc. Depending on how "changeable" your Lab 3 solution is, you may have ended up with major changes. This lab is complex enough that declaring parent types (abstract classes, concrete classes, interfaces) might begin to be useful.

You are still required to - follows the CS2030 Coding Style [../style/index.html] - clearly documented with `javadoc` [../javadoc/index.html]

Grading

This lab contributes another 4 marks to your final grade (100 marks).

- 1 mark for design of multiple servers
- 1 mark for design of multiple waiting customers
- 1 mark for correctness
- 1 mark for documentation and style

Multiple Servers

The first change you need to do in this lab assignment is to model multiple servers. Places like supermarkets, banks, post offices, etc, usually provides multiple counters to serve their customers. This situation is what are want to model here. Each server serves their customer independently. The state of the servers is visible to the customers (e.g., a customer can check if any of the servers is idle).

Just like before, we assume that if there is a customer waiting, then an idle server must serve the next customer immediately. In the case where each server has its own queue, then the server only serve the next customer in its queue (i.e., it will not get a customer from neighboring queue). So it is possible for a server to be idle, with no customer in its queue, but there are still waiting customers in the system (in other queues).

We assume that every server has the same service rate μ. We might naturally think that service time is a property associated with a server. But, how long it takes to serve a customer, depends on the service needed by the customer (e.g., how many items the customer buys). Thus, for our labs, we are going to associate a randomly generated service time with each customer. As in Lab 3, the service time is exponentially distributed and can be generated with the same formula that depends on the service rate μ and a uniformly random value obtained from a `Random` object.

Multiple Waiting Customers

So far, we only model one waiting customer (at most). The next change is to expand the model to allow multiple waiting customers. Just like a real system, customers

queued up if the server is busy. We will have two possible configurations: a single queue or multiple queues.

1. All servers share a single queue. Once a server finishes serving a customer, it gets the next customer in the shared queue, or
2. Each server has its own queue. Once a server finishes serving a customer, it gets the next customer in its queue.

For this lab:

- A queue has an unlimited capacity
- A customer never leaves until he/she is served
- When a customer arrives, if there is at least one idle server, the customer go to any one of the idle servers (it can go to the first server it finds idling). If none of the servers is idle, the customer randomly chooses a queue to join in the case where each server has its own queue.
- For simplicity, once a customer joins a queue, it is committed to that queue, he/she will not switch to another shorter queue, or go to a server that is now idle.

In the next lab, we will consider different customer behavior when choosing which queue to join and when waiting in a queue (such as switching to a shorter queue or leaving if waiting for too long). So, you may want to design your program so that it is flexible to support different customer behavior.

Terminating Condition

Since Lab 2, we have been terminating the simulation based on the number of customers served, i.e., we always served a fixed number of customers. From this lab onwards, we will be terminating the simulation based on time -- i.e., we will simulate until the simulated time hits a given value, then, regardless of how many customers have been served, we will print the statistics and quit.

This change sets the stage for further separation of `Simulator` from the actual systems being simulated. Eventually, we want your `Simulator` to be a general

simulator that can simulate systems other than queueing customers.

Input Files

With this change, the input file now contains additional *simulator parameters*. It has the following lines:

- The first line is an `int` value, indicating the seed to the simulator
- The next line is an `int` value, indicating the number of servers c
- The next line contains a `double` value, indicating the service rate μ
- The next line is an `int` value, indicating the number of queues -- it must be either 1 or c
- The next line is a `double` value, indicating the arrival rate λ
- The last line is a `double` value, indicating the total simulation time.

Output

As the system being simulation become complex, it is hard to match the output of the programs, without putting too much constraint about how you implement your solution. But, if we simulate large enough number of customers, the calculated average waiting time should not be too different. As such, we will simulate for a long period of time (1,000,000), and check if your average waiting time is within a given range of our answers [1] [#fn:1]. Like in previous two labs, the last line of your program output should not be changed. It should remain as:

```
System.out.printf("%.3f %d %d", ..")
```

The two integer outputs are not as important here [2] [#fn:2], but let's keep it there and print them anyway (they will be useful again in future labs!)

TEST	Average Waiting Time	
1	1.36 - 1.42	
2	0.63 - 0.67	
3	1.31 - 1.35	
4	48000 - 52000	

Submission

When you are ready to submit your lab, on `cs2030-i` , run the script

```
~cs2030/submit04
```

which will copy all files matching `*.java` (and nothing else) from your `~/lab04` directory on `cs2030-i` to an internal grading directory. We will test compile and test run with a tiny sample input to make sure that your submission is OK.

You can submit multiple times, but only the most recent submission will be graded.

---

1. Your code might still be correct if it is slightly out of range -- try a different seed and see. If it gets consistently out of range, or is way out of range, then double check your logic.
2. With an infinite capacity queue, everyone gets served. So we will not have any lost customer, at least for this lab.