

Assignment 2

Md. Monjur Ul Hasan

201175890

Question 1:

Code:

Attached (6 .java files, 5 .bat files)

Execution Instruction:

Quick Run:

The code can be compile and execute from the command line and also from the IDE. For running it from IDE, package information may needed to be added to each of the .java files, depending on the type of the IDE. For running it in the command line you need to use standard java commands. The main class for the program is QueueSimulator. The following command will run the corresponding queue simulations.

```
java QueueSimulator 120 grocery 3 2
java QueueSimulator 120 bank 3 2
```

The following batch files are also provided with the code that can help you quickly compile and execute the code.

```
runGroceryQueue.bat
runBankQueue.bat
```

For executing the code with the batch files, you may need to configure the java compiler path (JDK path), if it is not already added to your OS environment. You can do that by giving the path in the setJava.bat file¹.

Run with animation:

Since the simulation process use a simulated clock, the two hour time is actually goes very fast. However, the solution can be slow down by using an additional argument in the command as shown below.

```
java QueueSimulator 120 grocery 3 2 10
java QueueSimulator 120 bank 3 2 10
```

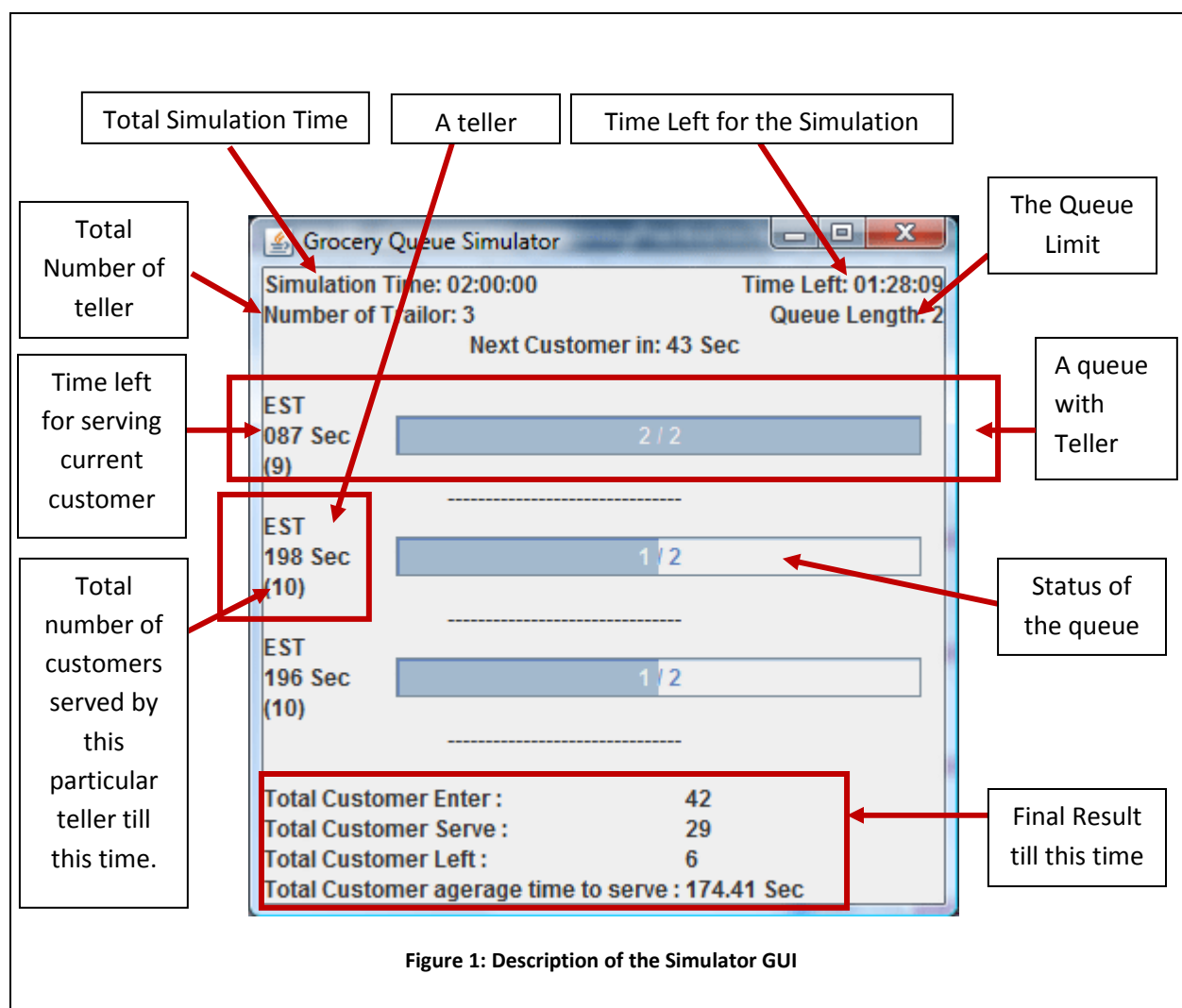
The last parameter indicates a waiting time after simulating each second of activities. The unit of this waiting time is in nanosecond. So, if you put 1000 in the last parameter, you will able to see a real time (a total of 120 minutes) simulation. The following batch file is also provided for the execution with animation.

```
runGroceryQueueWithAnimation.bat
runBankQueueWithAnimation.bat
```

Output:

The output of the program is found in the graphical interface. The Figure 1 shows the different parts of the GUI and their respective explanations.

¹ To edit the bat file, please open the file with a text editor such as notepad for window.



Results:

The results of the simulation for grocery queue are given in Table - 1 with the progress with time. The figure (d) in the table shows a customer cannot able to find a queue to join and waiting for 10 seconds. During this waiting, it checks for a place in queues in every second and as soon as a queue gets an empty space, it occupies the space. Figure (f) shows the final result. The results for the bank queue simulation are given in Table-2.

The application was executed several time (5 times for each queue) to validate the results. Each time the sum of the customer served, customer left, currently serving customers, and customer waiting in the queues was equal to the value of total customers entered.

Table 1: Simulation Results from grocery Queue

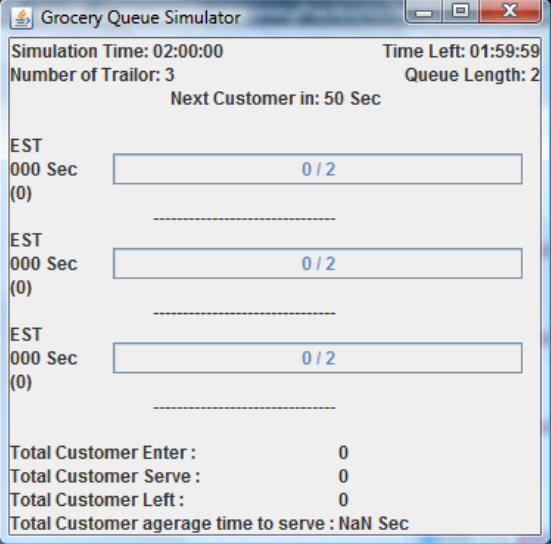
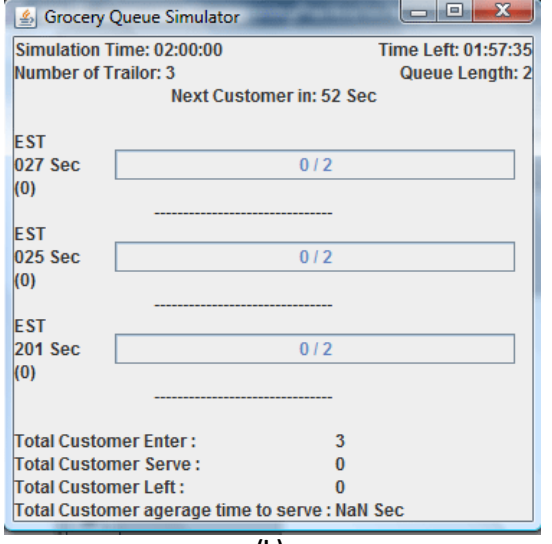
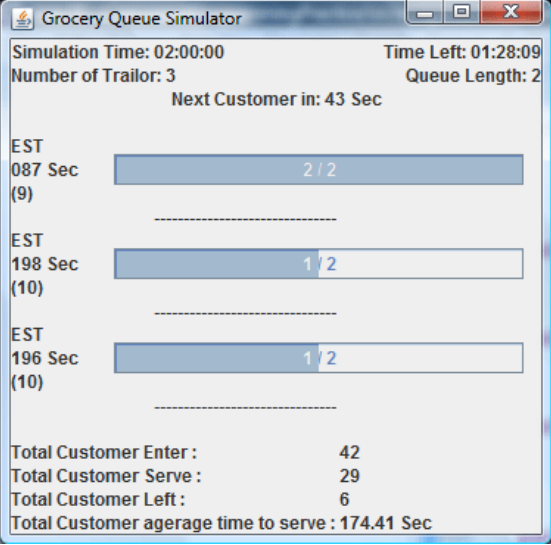
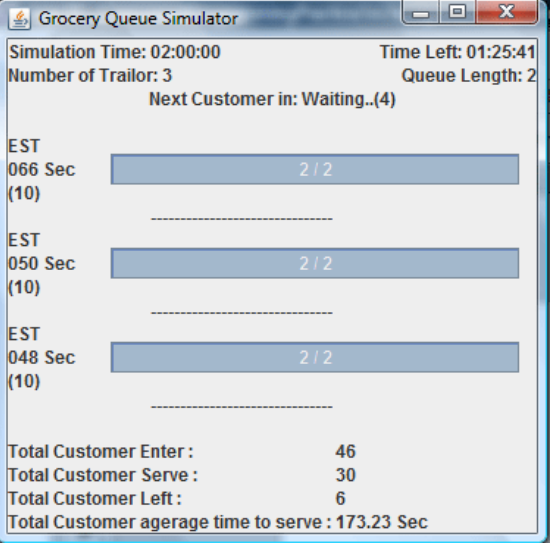
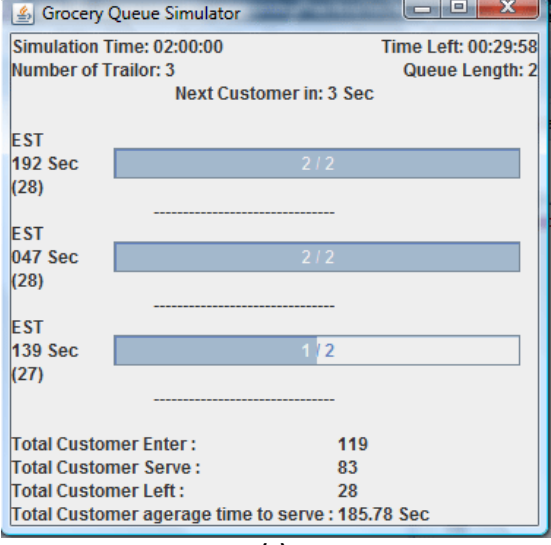
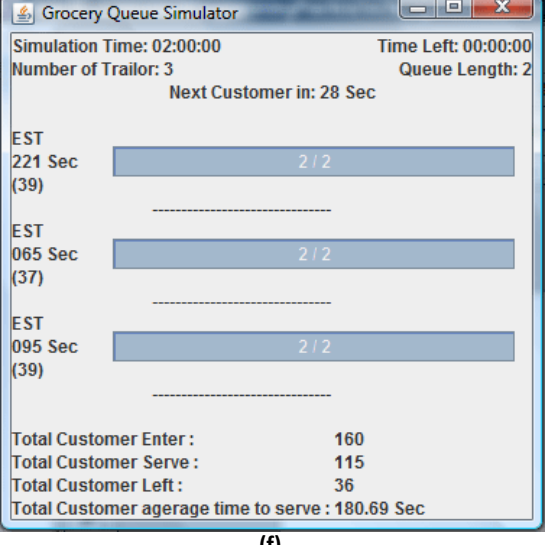
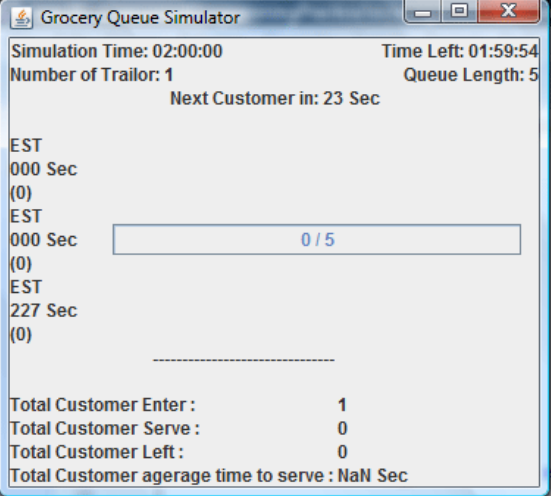
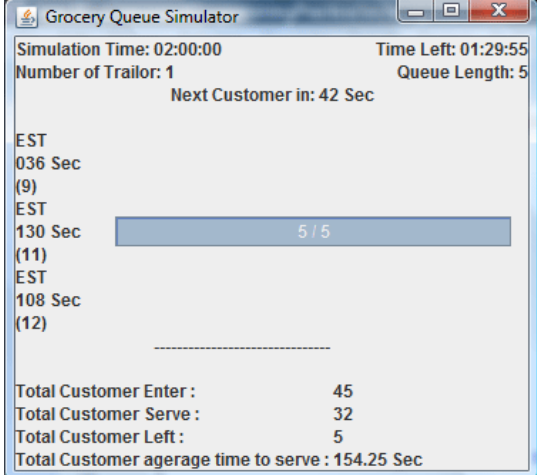
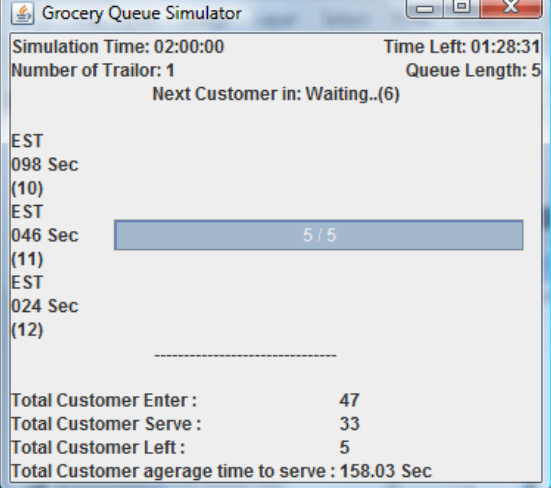
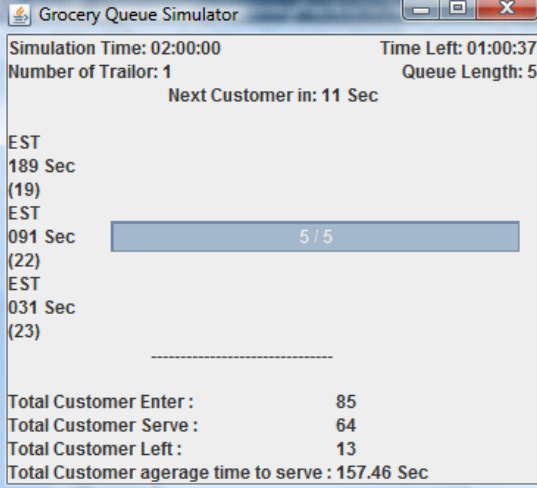
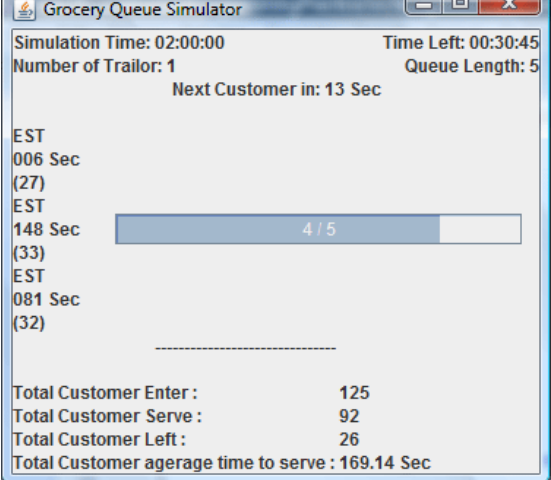
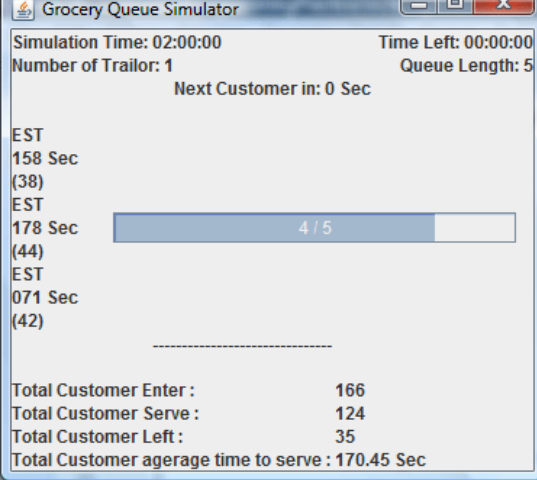
 <p>(a)</p>	 <p>(b)</p>
 <p>(c)</p>	 <p>(d)</p>
 <p>(e)</p>	 <p>(f)</p>

Table 2: Simulation Results for Bank Queue

 <p>(g)</p>	 <p>(h)</p>
 <p>(i)</p>	 <p>(j)</p>
 <p>(k)</p>	 <p>(l)</p>

Generalization:

While the results are shown as per the question, this program is prepared for general purpose. All the arrays are created dynamically. As a result, the user is free to put any number in the command line. The following figure is an example of running a 8 grocery queues each have a queue limit of 15 persons. (The time constrain of arriving and serving customers are different in this simulation).

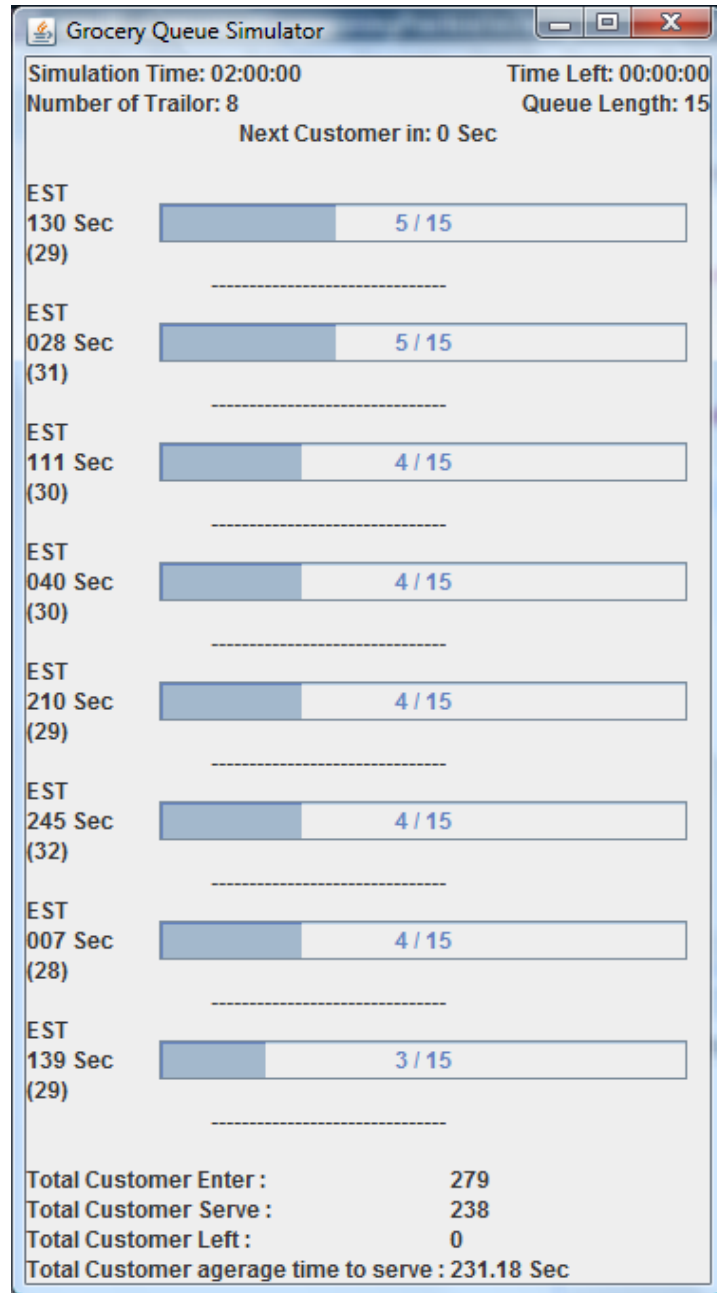


Figure 2 Running a bigger Grocery Queue

Beside modifying the parameters in the command line, this program can be modify easily from the code to achieve a robust simulation. For example, the randomness of the customer arrival time and serving time can easily be changed from a single point in the code called MyRandom.java as shown in the next page:

```

public class MyRandom
{
    private static int rangeRandom(int s,int e)
    {
        return s+((int)(Math.random()*1000))%(e-s);
    }

    public static int customerArrival(){
        return rangeRandom(20, 30);
    }

    public static int serviceTime(){
        return rangeRandom(160, 300);
    }
}

```

According to the question, the queues for grocery have only one teller per queue, and bank has multiple tellers per queue. However, making a simple change in one place in the code, we can run simulation on multiple queues each having multiple tellers. The change can be done in queue.java within the groceryQueue's static variable nTeller as shown follows.

```

class GroceryQueue extends CustomerQueue{
    public static int nTeller=2;
}

```

Then run the code with the following command line and you can see a simulation as Figure-3 in the next page. The above code indicates that two tellers per queue and 3 queues (line up) for the customer. Each queue has a limit of 15 customers.

```

java QueueSimulator 120 grocery 3 15 10

```

Concurrent Features:

The following concurrent programming features are used in this assignment.

Threads:

The program uses several threads as follows:

Main threads:

This is java main thread, that running the application. All initializations of all the objects and including UI are done here. An EDT thread and background thread are also creates in this thread.

The EDT Thread:

Since this program used an GUI, an EDT thread is created and maintained by JVM. Using a background thread and a publish() method, this GUI has been updated in a regular interval during the simulation.

The Background Thread:

The background thread as the main synchronizing thread of the simulation. It maintains the clock (using the CountdownTimeCounter class). It also adds customer as per the description of the assignment.

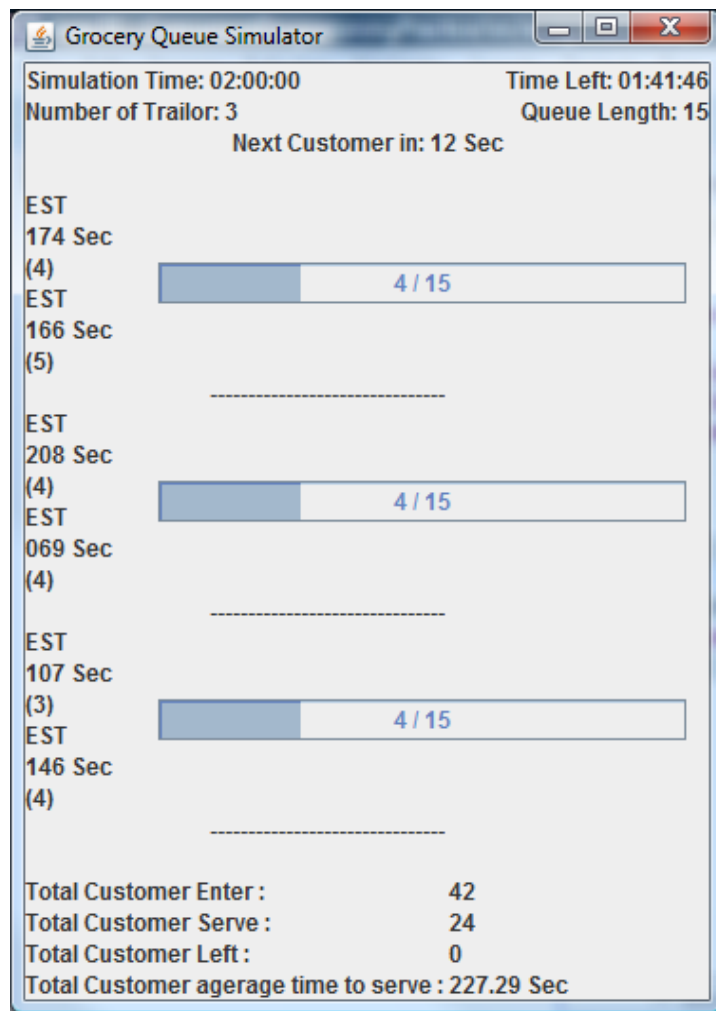


Figure 3 Simulating a robust Grocery Queue

The Teller Thread:

The program creates one thread for each of the tellers. The threads are created using the class `tellerService` which use `Thread` as super class. All of these classes are declared as demon thread. So these thread runs in an infinite loop, however, when the background thread finish counting the time, the program exits and these threads also exits with the main program.

Lock and Monitors:

This program use locks and monitors in different places to make sure the right values found by the concurrent threads. The program use `ReentrantLock` for this purpose. The class `CustomerQueue` uses a monitor by using a `ReentrantLock`. The method `addCustomer()` and `removeCustomer()` was guard against concurrent access using this monitor.

The class `customer` is also use a `ReentrantLock` to protect the change of certain values from updating concurrently.

Barrier:

This program also uses a JAVA Barrier called `Phaser` to synchronize the concurrent thread execution. The barrier used in two different places. First the barrier ensures that all threads are created and has

completed all of its initialization. Next, the barrier synchronizes the threads with the simulated time. In each simulated second all the threads wait for other threads to finish their jobs that were needed to be done by that second.

Limitation and Conclusion:

This program has one limitation. This program did not use any thread executor poll feature. It considers the number of thread will be limited. If the numbers of threads are increase with a great number, it will encounter an out of memory exception.

Question 2

The proof outline shown for the client server problem by Dr. Norvell's note as a generic proof outline for the function $f(x)$ and $g(x)$. By giving a non recursive and non conditional definition of these functions will not have any effect in the proof outline.

In this question $f(x) = x^4$, and $g(x) = \sqrt{x}$. Which makes $h(x) = f \circ g(x) = x^2$. As a result $h^n(x) = (x^2)^n$.

However, the only requirement for this function $g(x)$ is, x is required to be positive for the square root function. So the initial condition $x = X$ is required to be positive so $x = X$ and $x \geq 0$ will be the initial pre condition. The other assertions will remain same.