Bank Semaphore Assignment

Overview

One interesting class of applications of concurrent programming is simulation, in which a program simulates a real-world system by using individual threads to simulate its individual components. For this project, we will be simulating a bank, having some number of tellers. Each customer arrives at the bank, waits in a single line until a teller becomes free, and then goes to that teller to transact his/her business, leaving the bank when his/her business is done. In this case, a thread simulates each individual customer.

The major parameters of this system are: the number of tellers, the rate at which new customers arrive (specified as an average interval between arrivals), and the average time it takes to service a customer.

Clearly:

1. If (average inter-arrival time) >> (average service time) / (# of tellers) then customers will seldom stand in line, but tellers will often not be productively employed.
2. If (average inter-arrival time) < (average service time) / (# of tellers) then the waiting line will grow without limit and customers will become very angry.
3. If (average inter-arrival time) = (average service time) / (# of tellers) then the system will eventually reach a steady state with a fairly constant waiting line length.

The value for each of these three parameters supplied to the assignment's program should be chosen to satisfy number 3 above so the tellers are productive and the customers don't become angry. Therefore, to test your program, choose values for two of the parameters in the equation of number 3 and then compute the value of the third parameter. You can generate several different sets of values for the three parameters to test your program.

For this project, you write a program that simulates this system. At startup, your program will accept the three parameters mentioned above plus the length of time to run the simulation. All times will be entered in seconds. To avoid having to run the program for an entire day, we will simulate ten seconds of "simulated world" time by one second of actual program run time.

The simulation will proceed as follows:

1. One thread will be responsible for creating new customers at random intervals, such that the average time between new customers is equal to the specified parameter. It will exist throughout the simulation.
2. Created when a customer arrives, a unique thread simulates each customer, works its way through the bank, and then terminates.
3. A general semaphore that uses a FIFO queue will simulate the waiting line.
4. The semaphore initializes to the number of tellers. If there are n tellers, then the first n customers doing an acquire() on this semaphore will be able to proceed without delay.
5. When each customer thread enters the bank, he/she will do a acquire() on this semaphore. When he/she finishes being served, he/she will do a release(), allowing another customer to pass the semaphore and use the teller.

Design

1. Create a driver class and make the name of the driver class **Assignment2** and it should only contain only one method:
   `public static void main(String args[]).`
   The main method receives, via the command line arguments, these four parameters in exactly this order:
   1. The number of tellers.
   2. The mean (average) time between arrivals.
   3. The mean (average) service time.
   4. The length of time to run the simulation.

   The main method itself should be fairly short creating and then starting the thread which generates the customer threads.

2. The thread that generates the customer threads can be structured as follows:

```
while (the end of the simulation has not yet been reached) do
  begin
    sleep() a random amount of time, based on inter-arrival mean;
    if (simulation end time not yet reached)
    then create a customer thread;
  end;
```

3. A customer thread can be structured as follows:
   - Get next available customer number.
   - Increment number of customers in bank.
   - Report arrival and save arrival time.
   - Get in line (i.e., wait on semaphore.)
   - Done waiting; calculate waiting time.
   - Report starting to be served.
   - Sleep() a random amount of time, based on service time mean.
   - Report leaving the bank.
   - Update global statistics.
   - Decrement number of customers in bank.
   - Exit.

4. All of the parameter time values represent simulated world seconds, with ten simulated world seconds simulated by one second of actual program run time. Therefore, the parameter time values should be a multiple of 10.

5. To obtain a random number using the mean time between arrivals (inter-arrival time) or the mean service time you'll use the random number generator implemented in this class: Random_Int_Mean.java which you'll download to include in your assignment's code. Since one second of actual program run time simulates ten simulated world seconds, be sure to divide time parameters by 10 and use that value to pass to the random_int(…) method since the return value of the random_int method is the number of real world seconds the particular thread will sleep.

6. A thread which executes a call to the Thread.sleep(n) method sleeps, does nothing, for n milliseconds. Since the value obtained by random_int is the number of seconds to put a thread to sleep then that value must be multiplied by 1,000 before passing it as the parameter to the Thread.sleep method.

7. Any customer who is present in the bank when the simulation ends must be allowed to complete his/her work and leave the bank, even if this means that some tellers work overtime a bit. This means that you must keep track of the number of customer threads that are still alive and not finish the simulation until this becomes 0 after the simulation time is up. Actually, you need to keep track of every customer thread created. Once the simulation time is up, the main thread, which created the customer threads, needs to take care of each customer thread. If the customer thread is dead then execute a join in order to properly dispose of that customer thread. If the customer thread is still alive then move on to the next customer thread. The main thread keeps doing this check until the last remaining thread has died and been properly disposed. The methods of the Thread class you'll need for this are join() and isAlive().

8. You must treat all variables which multiple threads manipulate as shared resources accessed only by one thread at a time. One example, out of several, of this kind of a variable in this assignment is the number of customers in the bank. In Java, the synchronized methods achieve this mutual exclusion. Here is an example of the use of a synchronized method for a shared variable, Example.java. Even if your code appears to work, it will receive a poor grade if you do not properly ensure mutual exclusion on accesses to these!

9. Semaphores are explicitly implemented with the java.util.concurrent.Semaphore class. In the constructor for a Semaphore object, the first parameter specifies the count (number of permits) for the semaphore and the second parameter states the fairness setting (true or false) for the threads waiting to acquire a permit for the semaphore. If fairness is true then the waiting threads are handled in a First Come First Serve order; otherwise, with the fairness setting as false then the waiting threads are not handled in a First Come First Serve order. In this assignment the semaphore count is the number of tellers and the fairness setting should be set to true. The only methods you need and can use for the Semaphore class are: acquire() and release().

10. **Tip:** Make your program as modular as possible, not placing all your code in one .java file. You can create as many classes as you need in addition to the classes described above. Methods should be reasonably small following the guidance that "A function should do one thing, and do it well."

11. You must declare every class you create public and therefore each class has to be in a separate file.

12. Do **NOT** use your own packages in your program. If you see the keyword **package** on the top line of any of your .java files then you created a package. Create every .java file in the **src** folder of your Eclipse project.

13. Do **NOT** use any graphical user interface code in your program!

14. For the output:

Each customer process should write three lines to the screen, at the following times:

1. When he/she arrives in the bank and enters the waiting line.
2. When he/she steps up to a teller and starts being served.
3. When he/she is through being served and leaves the bank.

Each line consists of the current simulated world time, identify the customer by number, and an appropriate message.

Finally, at the end of the simulation print a summary message, giving:

1. The total number of customers served.
2. The average time spent waiting in line by the customers served.

The output of a sample run of the program:

```
Mean inter-arrival time: 10
Mean service time: 30
Number of tellers: 3
Length of simulation: 70

At time       30, customer        1 arrives in line
At time       30, customer        1 starts being served
At time       30, customer        2 arrives in line
At time       30, customer        2 starts being served
At time       30, customer        3 arrives in line
At time       30, customer        3 starts being served
At time       40, customer        1 leaves the bank
At time       50, customer        4 arrives in line
At time       50, customer        4 starts being served
At time       50, customer        4 leaves the bank
At time       50, customer        5 arrives in line
At time       50, customer        5 starts being served
At time       50, customer        6 arrives in line
At time       50, customer        7 arrives in line
At time       60, customer        3 leaves the bank
At time       60, customer        5 leaves the bank
At time       60, customer        6 starts being served
At time       60, customer        7 starts being served
At time       70, customer        7 leaves the bank
At time       70, customer        8 arrives in line
At time       70, customer        8 starts being served
At time       70, customer        8 leaves the bank
At time      100, customer        6 leaves the bank
At time      110, customer        2 leaves the bank

Simulation terminated after 8 customers served
Average waiting time =     2.50
```

Grading Criteria

The total project is worth 20 points, broken down as follows:

1. If your program does not compile successfully then the grade for the assignment is zero.
2. If your program produces runtime errors which prevents the grader from determining if your code works properly then the grade for the assignment is zero.

If the program compiles successfully and executes without significant runtime errors then the grade computes as follows:

Proper submission instructions, 4 points:

- Was the file submitted a zip file.
- The zip file has the correct filename.
- The contents of the zip file are in the correct format.
- The keyword **package** should not appear at the top of any of the .java files.

Program execution, 4 points:

- Program input, the program properly reads, processes, and uses the input parameters.
- Program output, the program produces the correct results for the input.

Code implementation, 8 points:

- The driver file has the correct filename, **Assignment2.java** and contains only the method **main** performing the exact tasks as described in the assignment description.
- The code performs all the tasks as described in the assignment description.
- The code is free from logical errors.

Code readability, 4 points:

- Good variable, method, and class names.
- Variables, classes, and methods that have a single small purpose.
- Consistent indentation and formatting style.
- Reduction of the nesting level in code.

Late submission penalty: assignments submitted after the due date are subjected to a 2 point deduction for each day late.

Submission instructions

Go to the folder containing the .java files of your assignment and select all (and **ONLY**) the .java files which you created for the assignment in order to place them in a Zip file. The file should **NOT** be a **7z** or **rar** file! Then, follow the directions below for creating a zip file depending on the operating system running on the computer containing your assignment's .java files.

Creating a Zip file in Microsoft Windows (any version):

1. Right-click any of the selected .java files to display a pop-up menu.
2. Click on **Send to**.
3. Click on **Compressed (zipped) Folder**.
4. Rename your Zip file as described below.
5. Follow the directions below to submit your assignment.

Creating a Zip file in Mac OS X:

1. Click **File** on the menu bar.
2. Click on **Compress ? Items** where ? is the number of .java files you selected.
3. Mac OS X creates the file **Archive.zip**.
4. Rename **Archive** as described below.
5. Follow the directions below to submit your assignment.

Save the Zip file with the filename having the following format:

your last name,
followed by an underscore _,
followed by your first name,
followed by an underscore _,
followed by the word **Assignment2**.

For example, if your name is John Doe then the filename would be: **Doe_John_Assignment2**

Once you submit your assignment you will not be able to resubmit it!
Make absolutely sure the assignment you want to submit is the assignment you want graded.
There will be **NO** exceptions to this rule!

You will submit your Zip file via your CUNY Blackboard account.
Follow these instructions:

Log onto your CUNY BlackBoard account.
Click on the CSCI 340 course link in the list of courses you're taking this semester.
Click on **Content** in the green area on the left side of the webpage.
You will see the **Assignment 2 – Bank Semaphore Assignment**.
Click on the assignment.
Upload your Zip file and then click the submit button to submit your assignment.

Due Date: Submit this assignment by Thursday, May 17, 2018.