CS-2303, System Programming Concepts, C-term 2017

Project 4 (45 points)
Assigned: Friday, February 3, 2017
Due: Monday, February 13, 2017, 6:00 PM

Programming Assignment #4 — Event Driven Simulation

Abstract

Write a *C*++ program that simulates the activity of customers in queues at a bank.

Outcomes

After successfully completing this assignment, you should be able to:–

- Develop a non-trivial *C*++ program
- Write a program that simulates some real-world activity
- Use linked lists in *C*++
- Use a random number generator

Before Starting

Read Chapter 6 of *Absolute C*++, which reviews **`structs`** from *C* and introduce classes in *C*++. Also read §7.1, which describes *constructors* and their use, and read §10.1 regarding **`new`**, **`delete`**, and *destructors*. Classes are, of course, at the heart of the *C*++ language, and **`new`**, **`delete`**, constructors, and destructors are the principal means of creating and destroying objects of those classes.

Also read §17.1 of *Absolute C*++, which is a good introduction to the general concept of linked list.

Event-driven Simulations

An *event-driven simulation* is a computer program that mimics the behavior of people or objects in a system in response to events that occur at certain times. The program must maintain a data object for each person or object (called an *actor*) and place it in a queue according to the time of its event. It then reads the queue in the order of the events and, for each event, causes the corresponding actor to do its actions scheduled for that time. The action of the actor may be to change its own state, change the state of the system, do something on behalf of another actor, or something else. Sometimes, the action will cause the actor to rejoin the event queue for a subsequent action. Sometimes, the action may add some other actor to the event queue or to another queue.

> *Example:* In a simulation of a radar system, an event might represent the transmission of a pulse at a certain time t. When simulated time t arrives, the simulator invokes the action (i.e., method) of the pulse event. This action enumerates the targets for the pulse and, for each target, computes the round trip time τ of the pulse and adds a new event to the event queue for time $t + \tau$ representing the return of the reflected signal.[1]

[1] This example is based on an MQP carried out in *C*++ at MIT Lincoln Laboratory by WPI students in 2010. The simulator developed by these students is now in production use throughout Lincoln Laboratory and is used for testing the software of major defense radar systems.

In this assignment, you will simulate customers arriving at a bank and standing in line in front of one of the tellers. People arrive at random intervals. Each person waits in his/her selected line until reaching the head of that line. When a person reaches the head of his line, the teller provides service for a random amount of time. After the service is completed, the person leaves the bank. The purpose of the simulation is to measure the average amount of time people spend between arriving at the bank and leaving the bank.

Assume that when there is a separate line for each teller, a newly arrived person joins the shortest line (or selects randomly among the set of equally short lines) and stays in that line until served. That is, no person leaves a line without being served, and no person hops from one line to another.

If a teller has finished serving a customer and there are no other customers waiting in its line, the teller selects the first customer from the line of another, randomly-chosen teller and serves that customer. If there are no customers waiting at all, the teller does other duties for a (small) random amount of time before checking the lines again.

The entire purpose of this simulation is to compare the performance of a single line serving all tellers *versus* separate lines for each teller.

Implementing your program

Your program should be called **`qSim`**. It needs to do several things:–

- Get and interpret the program parameters from the command line.
- Create a class object for each customer indicating his/her arrival time at the bank. Arrival times are determined from a uniform random number generator and the input parameters of the simulation. Also create a class object for each teller, with a random idle time in the range 1-600 seconds. *All constants in this simulation must be defined symbolically.*[2]
- Create a single *event queue* in the form of a linked list. The members of the linked list may be customers or tellers.
- Place each object in the *event queue* sorted according to the time of its event. That is, the event with the earliest time is at the head of the queue, and the event with the latest time is at the tail of the queue.
- Play out the simulation as follows:– take the first event off the *event queue*, advance a simulated *clock* to the time of that event, and invoke the action method associated with that event. Continue until the event queue is empty.
- Print out the statistics gathered by the simulation.

For this assignment, you will need to play the simulation twice — once for a bank with a single queue and multiple tellers and once for a bank with a separate queue for each teller. Draw some comparison about the average time required for a person to be served at the bank under each queue regime.

Here are some of the actions that can occur when an *event* reaches the head of the *event queue*:–

- If the event represents a newly arrived customer at the bank, add that person onto the end of a teller line — either the common line (in the case of a bank with a single line for all tellers) or to the shortest teller line. If there are several equally short teller lines, choose one at random.
- If the event represents a customer whose service at the bank has been completed, collect statistics about the customer:– in particular, how long has the customer been in the bank, from arrival

[2] It is recommended that you use the preferred *C++* way of defining symbolic constants, i.e., as **`const`** declarations.

time to completion of teller service. After collecting the statistics, the customer leaves the bank and its **Event** object is deleted.

- If the event represents a teller who has either completed serving a customer or has completed an idle time task, gather statistics about that teller. If there is no customer waiting in any line, put a teller event back into the *event queue* with a random idle time of 1-150 seconds.

  If there is a customer is waiting in line, remove the first customer from its line, generate a random service time according to the input parameters of the program, and add *two* events to the event queue, sorted by time. One is a customer event and represents the completion of that service. The other event is a teller event representing completion of a service and to look for the next customer (or to idle).

Class Hierarchy

You must define one or more classes that allow you to represent *Events*, *Customers*, and *Tellers*. It is suggested that the most important class of your simulation should be **Event**. How you distinguish between events associated with customers and events associated with tellers is your choice.[3]

Two methods of an **Event** are to add it to the **event queue** and to remove it from the **event queue**. In addition, each **Event** has an **Action** method that is to be invoked when an **Event** is removed from the **event queue**. The **Action** method should somehow distinguish between *customers* and *tellers* and invoke the appropriate *action* function for that kind of event. These should perform the appropriate action for a customer or teller, depending upon the type of the object that this action represents.

Each line in front of a teller should be implemented by an instance of a class called **tellerQueue**. For this assignment, do *not* attempt to use the **queue** container class from the Standard Library. Implement this class using a linked list, similarly to what you would have done in *C*. You will need to write methods to add customers to the end of the linked list and to remove them from the head of the list. In addition, include a *static variable* in the **tellerQueue** class that indicates which line (i.e., instance of the **tellerQueue** class) is the shortest. If more than one line is equally short, select one at random.

The **eventQueue** itself should also be implemented by a linked list. You will need to write a method to add an **Event** to the **eventQueue** in time order. This method should iterate through the list until it finds an **Event** with a time greater than the **Event** being inserted, and then it should insert the new **Event** just before that **Event**. You will also need a method to remove an **Event** from the **event queue** and invokes the **Action** method of the event. For extra credit (see below), you may implement a **priority_queue** class from the *Standard Template Library*.

Input

The command line of your program should be of the following form:–

```
./qSim #customers #tellers simulationTime averageServiceTime <seed>
```

The numbers of customers and tellers should be integers, and the simulation and average service times should be floating point numbers in units of minutes. The seed is optional and indicates a fixed seed for starting the random number generator (see below) For example,

---

[3] An ideal representation would be to make **Event** an abstract class and to derive **Customer** and **Teller** from it. However, *abstract classes* are not scheduled to be introduced in this course until after the assignment is due. Therefore, you should choose a practical approach that enables you to complete the assignment on time. You may, of course, read ahead to learn about *base classes* and *derived classes*.

```
./qSim 100 4 60 2.3
```

should be interpreted to mean that 100 customers and four tellers should be simulated over a period of 60 simulated minutes. The service time for each teller is an *average* of 2.3 minutes. No random number seed is specified.

Random Number Generation

To generate random numbers, use the function **`rand()`**, which is described in Figures 3.2 and 3.4 and the end of §3.1 of *Absolute C++*. These are also described on pages 46 and 252 of Kernighan & Ritchie. The function **`rand()`** is a *pseudo random number generator*. It generates a different number each time it is called, and those numbers look like they are random in the range **`0 .. RAND_MAX`**. In reality, however, it can generate the exact same sequence of "random" numbers repeatedly, to make it possible for you to debug your program. To use **`rand()`**, you need to include **`<cstdlib>`** and specify the appropriate **`using`** directive.

Random number generators work by maintaining one or more internal counters and performing a contorted transformation on the most recent number generated to get a new one that appears to be unrelated to the previous one. The random sequence can be initialized by calling **`srand(seed)`** at the beginning of your program,[4] where **`seed`** is an unsigned integer value. If you did not specify a **`seed`** in the command line, use some number that is likely to be truly random, such as the time returned from **`time(NULL)`**, which is also part of **`<cstdlib>`**. This seeds the random number generator to the current time.

To generate random arrival times with a uniform distribution, the following is suggested:–

```
float arrTime = simulationTime * rand()/float(RAND_MAX);
```[5]

It is useful to generate all customer arrivals at the beginning of the program and put them into the **`event queue`** in order of arrival time.

To generate random service times, the following is suggested:–

```
float serviceTime = 2*averageServiceTime*rand()/float(RAND_MAX);
```[6]

Output

After your simulation has completed for both types of queuing regimes, you should print out a summary with the following information:–

- Total number of customers served and total time required to serve all customers
- Number of tellers and type of queuing (one per teller or common)
- Average (i.e., mean) amount of time a customer spent in the bank and the standard deviation
- Maximum wait time from the time a customer arrives to the time he/she is seen by a teller.
- Total amount of teller service time and total amount of teller idle time.

The information that you need to print should determine the statistics that you gather during the simulations.

---

4 Call **`srand()`** exactly once, preferably in the **`main()`** function after interpreting the command line arguments.

5 The "function" **`float(…)`** is the constructor for an object of type **`float`** and converts the value of its argument into floating point number. It replaces the concept of *cast* from *C*.

6 In a professional situation, you would probably generate service times with a Gaussian distribution. However, that is not necessary in this project, and it would be an added complication to an already difficult assignment.

Teams

You may optionally work in two-person teams. If you already were registered as a team in Programming Assignment #3, and if you wish to carry that team forward, you do not need to do anything to re-register for this assignment.

If you were not part of a team, or if you wish to change teammates, you need to "register" your team in *Canvas* by sending an e-mail to cs2303-staff@cs.wpi.edu. We assume that both teammates share the workload approximately equally, and both teammates will receive the same grade.

Deliverables

This project *must* be carried out on the course virtual machine using your favorite development environment. It must be named **`PA4_username`**, where **`username`** is replaced by your WPI login username. You must provide the following:–

- The exported project files of your project, including **`.h`** files and **`.cpp`** files to implement your simulation, and the **`makefile`**. The target of the makefile should be called **`qSim`**.
- At least three different test cases that show the behavior of the bank under both queuing regimes. Show the command line and the output.
- A document called **`README.txt`**, **`README.pdf`**, **`README.doc`**, or **`README.docx`** summarizing your program, how to run it, and detailing any problems that you had. It must explain how you represent *events*, *customers*, and *tellers*. Also, if you borrowed all or part of the algorithm for this assignment, be sure to cite your sources *and* explain in detail how it works.
- An analysis of your results — i.e., under what circumstances a single queue is better or worse than a queue per teller. You may include this analysis in your **`README`** document.

Before submitting your assignment, *clean* your project to get rid of extraneous files. Export your files as a single zip file from *Eclipse*, as explained at the end of *Lab 2*.

Submit to *Canvas*. This assignment is named *Programming Assignment* #4.

Grading

This assignment is worth forty-five (45) points. *Your program* must *compile on the course virtual machine without errors in order to receive* any *credit*. If you develop on a platform other the course virtual machine, please export it to the course virtual machine for testing, in order to avoid surprises.

- Correct compilation using **`make`** and **`g++ -Wall`** without warnings – 5 points
- Correct use of random number generator and seed – 5 points
- Satisfactory program organization into an understandable set of functions and modules (i.e., **`.cpp`** and **`.h`** files) and satisfactory use of symbolic constants – 5 points (subjective).
- Satisfactory class definitions to represent *Events*, *Customers*, and *Tellers* – 5 points
- Satisfactory implementation of **`event queue`**, including insertion in time order – 5 points
- Satisfactory implementation and use of **`action`** methods for **`Tellers`** and **`Customers`** – 5 points
- Evidence of satisfactory testing, including output from test cases – 5 points
- Correct execution with graders' test cases – 5 points
- Satisfactory **`README`** file describing your class definitions and including a discussion of the merits of common or per-teller queuing – 5 points

Extra Credit

For ten points of extra credit, define and implement the **`eventQueue`** class using the **`priority_queue`** class from the *Standard Template Library*. The priorities are the event times of the **`Event`** objects representing the customers and the tellers.

Note: A penalty of ten **points** will be assessed if your project is not *clean* before creating the zip file or for submitting your programs in some format other than a *zip* file.

Note 2: If your program does not compile correctly on the course virtual machine using the **`makefile`**, the graders will attempt to contact you via e-mail. You will have 24 hours from the time of the graders' email to resubmit a corrected version, and a penalty of 25% will be assessed (in addition to other penalties)