Queue Simulation

Fundamental Programming Techniques (PT)

Homework #2

Motioc Alexandru

G: 30424

Contents

1. Objective
2. Problem analysis
3. Diagram
4. Code
5. Results / Testing
6. Conclusion
7. Objective

Design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients’ waiting time.

Input data:

- Minimum and maximum interval of arriving time between customers;

- Minimum and maximum service time;

- Number of queues;

- Simulation interval;

- Other information you may consider necessary;

Minimal output:

- The average of waiting time, service time and empty queue time for 1, 2 and 3 queues for the simulation interval and for a specified interval (other useful information may be also considered);

- Log of events and main system data;

- Queue evolution;

- Peak hour for the simulation interval;

Secondary Objectives:

* Making the program abstract
* Usage of data structure
* Division into classes
* Algorithm implementation
* Poison pill implementation

1. Problem Analysis

The user has to enter the input data of the queue simulation, which is refrained by: maximum arriving interval, minimum arrival interval, maximum processing time, minimum processing time, number of queues and simulation time. After the data has been entered, the graphic interface will showcase the simulation of generated clients entering and leaving the queues. Additionally, a log will be shown in the console where we can see exactly the number of clients entering or leaving the queue, and a timer presenting every count of the simulation.

Success scenarios:

1. User enters the necessary data inputs correctly

2. User presses start button

3. The graphic user interface is opened

4. The program will execute the simulation for the selected simulation time.

Failure scenario:

1. User enters the data inputs incorrectly

2. Error window will pop up

3. Problem Analysis

This project is going to contain 5 classes:

1. Client
2. Queue (A queues is considered a Linked List of clients)
3. Simulation (This class contains the generation of clients and simulation thread)
4. Gui
5. Main

Our classes are divided into 3 packages:

1. model

This package contains the classes which contribute to the logic of our program, thus it contains the simulation and status log of the queues.

2. gui

This package contains only the Gui class, which contains the display of data inputs and the simulation frame.

3. main

It contains the Main class of the program.

Definitions

A thread, in the context of Java, is the path followed when executing a program. All Java programs have at least one thread, known as the main thread, which is created by the Java Virtual Machine (JVM) at the program’s start, when the main() method is invoked with the main thread.

In Java, creating a thread is accomplished by implementing an interface and extending a class. Every Java thread is created and controlled by the java.lang.Thread class.

Java is a multi-threaded application that allows multiple thread execution at any particular time. In a single-threaded application, only one thread is executed at a time because the application or program can handle only one task at a time.

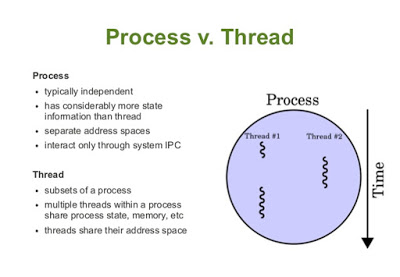
For example, a single-threaded application may allow for the typing of words. However, this single thread requires an additional single thread allowing for the recording of keystrokes in order to type the words. Thus, a single-threaded application records the keystrokes, allowing the next single-threaded application (the typing of words) to follow.

However, a multi-threaded application allows for the handling of both tasks (recording and typing the keystrokes) within one application.

When a thread is created, it is assigned a priority. The thread with higher priority is executed first, followed by lower-priority threads. The JVM stops executing threads under either of the following conditions:

* If the exit method has been invoked and authorized by the security manager
* All the daemon threads of the program have died

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.



What is poison pill pattern?

Poison pill pattern is a fancy term to notify the consumer that producer is done with producing messages & there are no more tasks for the consumer to wait for.

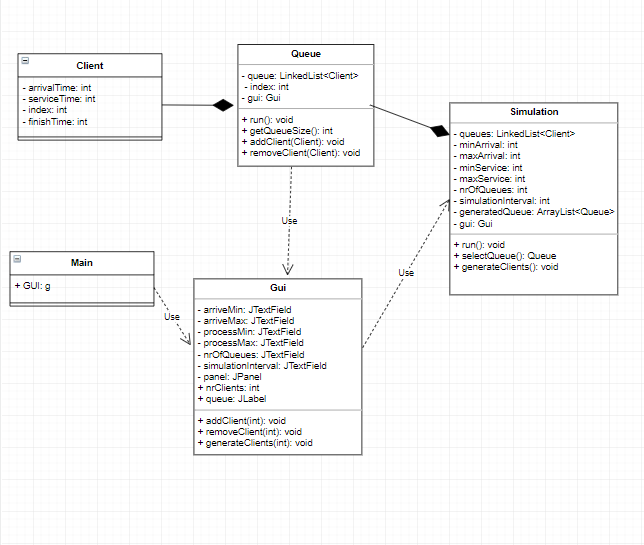
If you don’t have control over producer’s task iterator, drop the poison pill from finally block.

e.g.:

public void run() {  
 try {  
 while (TasksIterator.hasNext()) {  
 queue.put(iterator.next());  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 queue.put(PoisonPillTask);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

In the case of our project, the poison pill is the client of index -1, which is added to the queue when simulation time is over, thus giving us a interrupt condition for our run method.

UML Diagram



This UML diagram represents the structure of the project’s code. As we can see from the diagram, we have 2 threads that will run in this program: one from the Queue class, the other one from the Simulation class. The simulation thread simulates the actions of the generated clients, whilst the run method implemented in the queue class verifies if there is a client ready to be served, and if that is the case, the client will be served for the generated service interval.

4. Code Implementation

Client

The first class implemented in this project, Client, is a component of the queues that are going to be generated on the interface of the program. It’s constructer receives 3 parameters: arrivalTime, serviceTime, index. Based on the generated arrival time and service time, the finish time of a client is calculated later. Each client is assigned a certain index in order to keep track on the console log. Additionally, having implemented the Poison Pill Pattern, each queue will receive a client with index ‘-1’ when the simulation time has passed, so we can verify when we can interrupt the thread. Other than that, this class only contains getters and setters for: arrivalTime, finishTime, serviceTime, index.

Queue

Queue is the class that puts clients in a selected queue. This class implements the Runnable interface, which is overwritten in order for us to simulate multiple queues at once. Our queue is considered a Linked List of Client type objects. This class implements addClient and removeClient method which modify the contents of our queue using linked list commands (add / remove). When a thread is created, the run method will be executed, meaning that we will verify if our queue is empty, and if not we will take in the shop our next client, and keep him for the selected service time, and remove him from the queue. If the queue is empty, we will apply the wait method in order to advance in time and get a new client in the queue. This thread will be interrupted when the poison pill( the client with index -1) is introduced in the queue. This poison pill is introduced when the simulation time is over.

* Run method re-written:

**public** **synchronized** **void** run() {

**while**(**true**) {

**if**(!queue.isEmpty()) {

Client c=queue.getFirst();

**if**(c.getIndex()==-1) //poison pill

{

**this**.interrupt(); //if we find the poison pill, we know the simulation time is over

}

**try** {

**this**.wait(c.getServiceTime()\*1000);

System.***out***.println("Clientul " + c.getIndex() + " a parasit coada " + index);

gui.removeClient(index);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

queue.remove();

} **else**

**try** {

**this**.wait(100); //we verify more times per second if a client has entered the queue, because wait's are not simultaneous

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

Simulation

This class contains the core of our program’s logic, also having implemented the Runnable interface and a overwritten run method that checks when the arrival time of a client matches the current time, and if it is the case, the client is served and removed afterwards from the queue. For every time increment, 1000 milliseconds (1 second) will pass. Simulation class also implements the generateClients method, which generates for each client a random arrival and service time (that respects the boundaries declared by the user) and is added to the queue linked list afterwards. selectQueue is a method that finds the smallest queue in the simulation, which helps us determine which queue is reserved for our next client to enter. If all queues are of the same size, the client will be added in the first queue. The simulation constructor initializes a selected amount of queues(nrOfQueues) and for each queue, a thread from class Queue is ran in order to start simulation. It is worth mentioning that our run methods (from Runnable interface) are synchronized, which allows us to use the wait method on our threads.

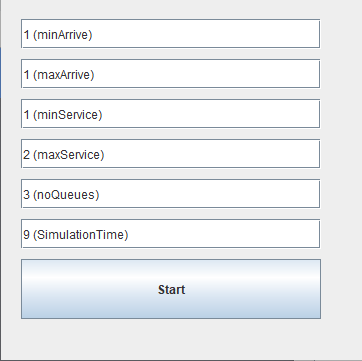
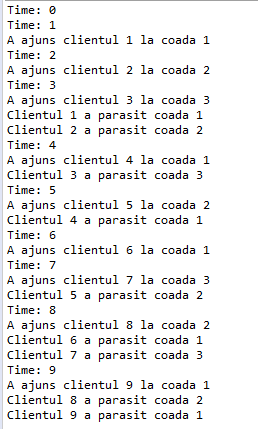
* Run method re-written:
* @Override
* **public** **synchronized** **void** run() {
* **int** time=0;
* Queue selectedQueue;
* **while**(time<=simulationInterval) {
* System.***out***.println("Time: " + time);
* **if**(generatedQueue.isEmpty()==**false**) {
* **if**(generatedQueue.peek().getArrivalTime()==time) {
* selectedQueue=selectQueue();
* selectedQueue.addClient(generatedQueue.getFirst());
* gui.addClient(selectedQueue.getIndex());
* System.***out***.println("A ajuns clientul " + generatedQueue.getFirst().getIndex() + " la coada " + selectedQueue.getIndex());
* generatedQueue.remove();
* }
* time++;
* **try** {
* **this**.wait(1000);
* } **catch** (InterruptedException e) {
* // **TODO** Auto-generated catch block
* e.printStackTrace();
* }
* }
* }
* **for**(**int** i=0;i<nrOfQueues;i++) {
* queues.get(i).addClient(**new** Client(0,0,-1)); //poison pill introduction into the queues
* }
* }

Gui

This class contains the display that pops up when our program is executed. This class extends JFrame and contains 6 JTextFields where the user can introduce the input data, a JButton that starts the simulation thread when it is pressed. As a quick mention, in this class we have to initialize a Gui type object thisGui, because the compiler won’t allow this to be called in the Simulation constructor parameters. In this class, two more methods have been implemented: addClient and removeClient. This methods are used for displaying the arrival of clients in a JLabel type matrix, and ofcourse, removing them from the display matrix when necessary. The display is refreshed using the methods: **this**.revalidate() and **this**.repaint().

4. Results

In this paragraph I will showcase a simulation created by the program and status log for the clients that enter/leave the queue.

In this scenario, the user has selected an equal value for minArrive and maxArrive parameters which happens to be the value 1. Because of this, a client will enter the queues in each second, i.e. time increment. The minService and maxService inputs are, however, different. In this scenario the service time values will vary between 1 and 2. As it can be seen from the console log, every client leaves the queue 1 or 2 turns after they have entered the queue first. Because our simulationTime is declared as 9, client nr. 9 will leave in the same turn that it entered the queue, because simulation time is lesser than the current time increment, thus introducing the poison pill into the queue and forcing the client to leave before interrupting the thread:

Client c=queue.getFirst();

**if**(c.getIndex()==-1) //poison pill check

{

**this**.interrupt();

}

The simulation will interrupt after simulationTime is passed by the current time, regardless of the content of our queues. All queues will be emptied and the simulation will be interrupted, afterwards the user can introduce other input data for further testing.

6. Conclusion

This project helps us understand how threads work in our programs. The evolving status of the queues is achievable only if we start a thread for each queue generated. Also, in order for the simulation to be managed in real time, there needs to be a thread that interrupts the thread when the simulation time condition don’t match with the time increment that starts at the beginning of the simulation thread. This project can also be a good opportunity to learn about new programming techniques such as the poison pill technique or thread management.

With all that being said, this project possesses a few flaws: there’s no error log window in case the user enters foul data inputs, there is no average waiting time calculator implemented, and the queue generating simulator display is very simplistic and perhaps a bit hard to follow by someone who doesn’t fully understand it’s purpose beforehand, which represents a weak visual presentation through graphical interface.