

## Universitatea Tehnică Cluj-Napoca

## Facultatea de Automatică şi Calculatoare

## Secţia: Calculatoare, engleză

## Programming Techniques

## ~ Queues ~

Student: Ban Bogdan Marian

Group id: 30425, second year

Professor: Salomie Ioan

Table of Contents

1. Task Objectives. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 03

2. Problem analysis, modelling, scenarios, use cases. . . . . . . . . . . . . . . . . .03

3. Projection (UML diagrams, data structures, class projections, . . . . . . . .08

Interface, relations, packages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .09

Sequence ). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .09

4. Implementation and testing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11

5. Results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11

6. Conclusions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11

7. Bibliography. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11

1. Task Objectives

Requirement: 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 clients

Minimum and maximum service time

Number of queues

Simulation interval

Other information you may consider necessary

Minimal output:

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

Log of events and main system data

Queue evolution

Peak hour for the simulation interval

Barem:

Documentatie - 1p

Cel putin un thread sau timer care sa ruleze logica aplicatiei - 1p

Log of events (Logger, Output File sau TextArea pe fereastra) - 1p

Queue evolution (console) - 1p / Interfata grafica – Queue evolution - 2p

Input de la tastatura / fisier (nu harcoded) – 1 punct

Multithreading: 1 thread pentru fiecare Queue – 3 puncte

Average of waiting time (0,33p), service time (0,33p), peak hour (0,33p) - afisate la finalul simularii

2. Problem analysis, modelling, scenarios, use cases

Problem analysis

Queues can be found both in the real and in models. The main objective of the queue is to provide a place waiting for a "client" before being served. Management systems based on queues is interested in minimizing customer waiting time in queues. One way to minimize waiting times is to add more servers, meaning more queues in the system (it is considered that each processor has an associated tail).

The system must simulate the arrival of a set of customers to be served, entrance queue, waiting, serving and leaving the tail. It will track customer waiting time in queues and will determine the average waiting time. To calculate waiting times should be known arrival time, termination and serving. Arrival time and service time depends on the client - when it arrives and how long the service he needs. End time depends on the number of queues, the number of other customers in the queue and serving their needs.

Input: the number of clients; minimum and maximum range arrival; Minimum and maximum service time; Number of queues.

Scenarios and Use cases

The problem involves the simulation of a system of queues, using inputs: the number of customers, the time of arrival minimum and maximum service time minimum and maximum number of queues, we have to generate for each customer service time and arrival between intervals read.

Then we simulate the arrival of customers waiting in the queue, serving them and leaving the tail.

The indications received for calculation times that were not randomly generate and use one thread for each queue. To better understand the problem we need to know the concept of tail, and also use threads.

Queue

The queue is linear list that is adding new elements at one end, and extracting an element can only be done through the other end of the list. Queue as everyday element is a known element. It can be the " queue " at a desk for the purchase of tickets at a store to buy some products, to board a means of transport, etc.

From the definition follows that always extracted from a queue the oldest item. It is said that access to a tail type FIFO (First In First Out). As with the stack, basic operations on a queue are two: adding an element remove an element in the queue and queue.

What are threads?

A Java application runs inside an operating system process. The process consists of segments of code and data segments mapped to a virtual address space. Each process has a number of resources allocated by the operating system, such as open files, dynamically allocated memory areas or threads. The allocated resources are released at the end of process execution process.

A thread is a process execution unit. Each thread corresponds to a sequence of instructions, a set of CPU registers and a stack. The process does not execute instructions, it is a shared address space for one or more threads. Threads are those that executes instructions.

Threads resembles processes can be as planned execution. The main difference is that the thread is running in the process address space which belongs and can change values ​​that are seen by the other threads belonging to the same process. This leads to the need for wires to communicate with each other, that have synchronized access to shared data. Synchronize ensure data security, ie the synchronization avoids that one thread to modify a variable that is just utilazata by another thread.

Thread = sequence of a set of instructions + CPU + stack REGIST

Java is an interpreted language, holds the interpreter code and Java binary code (bytecode) is treated as a data area by interpreter. So excute wires are actually created by the Java interpreter. The execution of a Java application is automatically created and a first thread, called the main thread. It can create other threads, which in turn can create other threads, and so on.

Why we need threads?

Applications that use multiple threads can run in parallel multiple tasks. For example an application that performs an animation in a window and into another window displays the result of a database query.

Navigation program achieved at least two things in parallel: parallel brings network data and displays the data. That facilitates scheduling threads to be closer to human thinking. Man usually handle multiple things at once. A teacher while teaching must be careful and at students who listen to him / her.

Threads Java programmers provides a number of facilities, but there are a number of differences from other packages threads. Compared to these packages we believe it gives us less, but they also have an advantage simplicity. There is a standard POSIX threads P10033.4a excute a package that can be implemented in any operating system.

Any package Java (Sun) is "thread safe", meaning it is ensured that the methods contained in the package can be called simultaneously by multiple threads.

Using threads in Java

Creation and launch

In Java there are two ways to create a thread:

-By Defining a class that inherits from predefined class Thread (derived from the Thread class)

-By Implementing interface Runnable

Thread class defined in java.lang package with a number of methods. The main method is the method run (). This should include all activities that thread has to execute.

public class Thread extends Object implements Runnable

Thread class is a class that has as its superclass and Object class implements the interface Runnable. Runnable interface declares a single method run () and Thread class implements it. Java Virtual Machine allows an application to run multiple threads in parallel. Each thread has a priority and each thread can be marked as a daemon thread. When creating a thread is setting these properties, ie newly created thread will have the same priority as a parent and will be wire type only if the thread parent demon is a demon.

When interprets byte code of an application, Java virtual machine first creates a thread that runs the main method. Java Virtual Machine continues to execute this newly created thread or other threads until all threads nedemon destroyed.

Start () method is used for the execution of a new thread created. Start () method is used only once in the life cycle of a thread. The thread destroy method destroy (), and it can be called only once. The method sleep () call is an argument that represents the time in milliseconds that the thread waits (sleep). sleep () is a static method calls are prefixed with the class name. If the thread is interrupted InterruptedException exception is generated. So the method call must be done in a context of treating the exceptions.

try {

Thread.sleep (1000)

}

catch (InterruptedException e) {

// Exception handling code

}

The thread can be pulled from sleep by calling method interrupt (). The method further calls to interrupt execution of a thread is blocked waiting for an operation long / O. The method join () also utilizes the binding thread. By getName () and setName () can get or set the name of the thread.

Priority threads

Java virtual machine uses wires priorities in planning for execution threads. In Java there are 10 priority levels. We have three constants in the Thread class that can be used:

Thread.MIN\_PRIORITY with value 1

Thread.MAX\_PRIORITY with value 10

Thread.NORM\_PRIORITY having a value of 5

Priority wire method is set setPriority ()

Thread synchronization.

Each thread has a life and what is not interested in what the other threads. If your computer is equipped with multiple processors, the various threads of the same process can be scheduled for execution on different processors. The application for that project, this is insignificant because all these things are the responsibility of the virtual machine and not to the programmer. However a problem arises when applications that run on multiple threads, namely access shared resources. That problems will arise, access to shared resources used to interlock. Synchronization is based on the concept of monitor introduced by C.A. Hoare. A monitor is basically a padlock attached a resource to prevent the use of resources in parallel.

A thread calls the deal a monitor if a synchronized method. If a thread handle a monitor, another thread that tries occupation same monitor is blocked and waiting to release the monitor.

Any object that contains one or more methods synchronized has a monitor attached. Synchronized methods are defined in the following way:

my\_method synchronized public void () {...}

Any Java class has a monitor attached. This monitor is different monitors attached objects are used when call the synchronized static methods of the class.

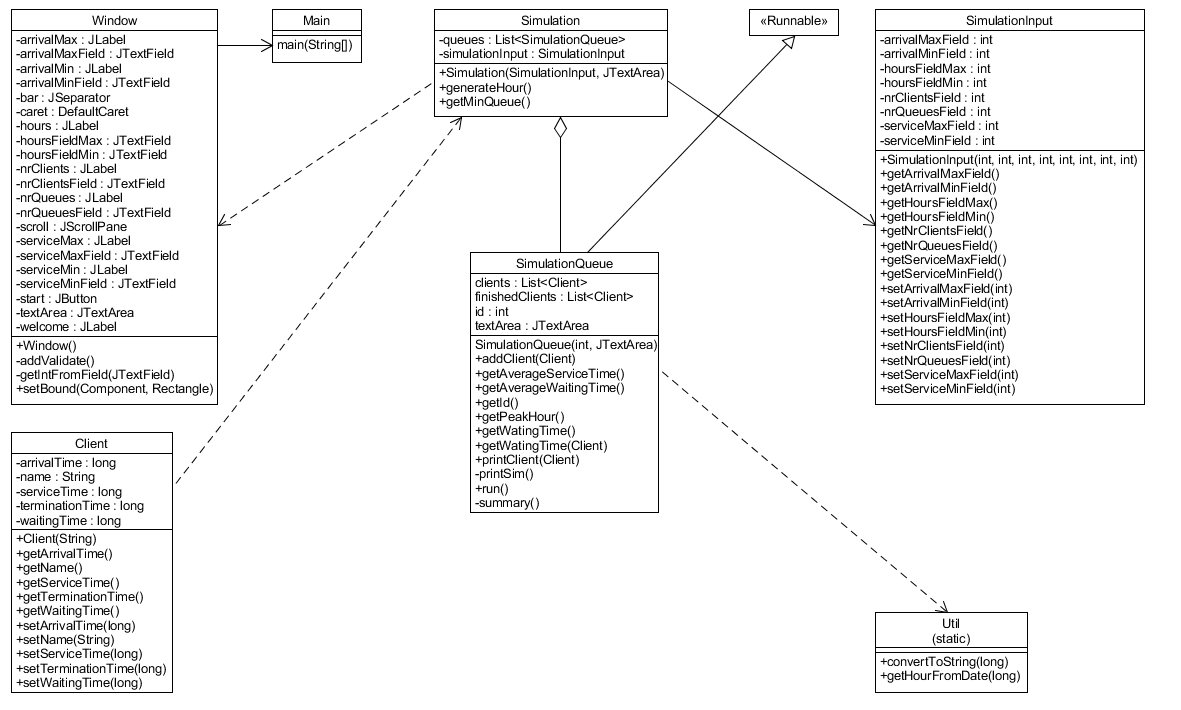
public static void my\_static\_method synchronized () {...}

wait () and notify ()

With the keyword synchronized execution can serialize certain methods. The methods wait () and notify () Object class extends this capability. Using wait () and notify (), a thread can release the monitor busy, waiting for a signal to its reoccupation. The methods can be used only in synchronized method or block instructions synchronized. Putting wait () in a synchronized block, the thread releases the monitor falls asleep. Usually resort to this possibility when the thread must wait occurrence of an event in another part of the application. Later, when the event occurs, the thread in which the event occurred, calling notify () to awaken sleeping thread. Thread woke occupy the monitor and then continue working from the point of interruption.

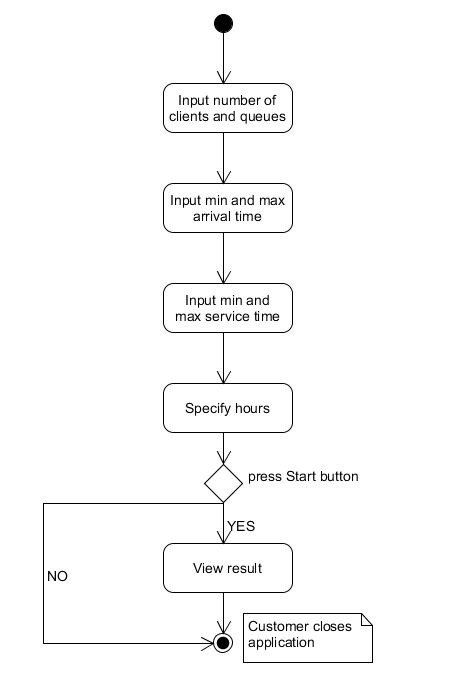
3.Projection

UML Class Diagram

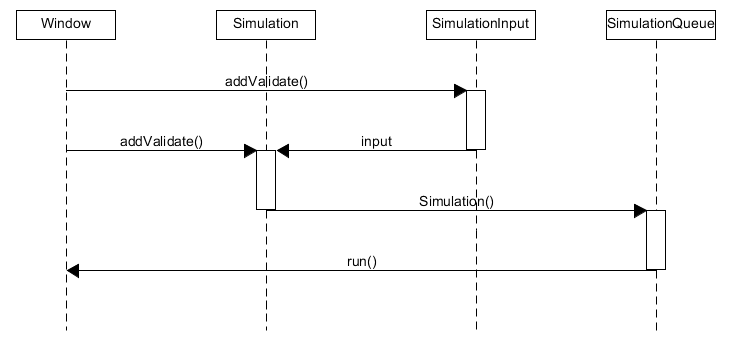


Use case Diagram

Activity diagram



Sequence diagram



A sequence diagram showing the interaction of the processes between the following objects window, simulation, simulationInput and simulationQueue.

Packages

The classes chosen for this project are specific and each one of them handles an important part of the program.

1. Model

In this package we have the Client class which imitates a real life client.

1. Queues

In this package we have the following classes:

Main – the main class of our application

Simulation – our most important class, here we create the queues , the clients , generateHours etc.

SimulationQueue – depicts the real life queue , implementing the Runnable interface

SimulationInput – a class which has at its constructor every input we need for the simulation ,

**public** SimulationInput(**int** nrClientsField, **int** nrQueuesField, **int** arrivalMinField, **int** arrivalMaxField,

**int** serviceMinField, **int** serviceMaxField, **int** hoursFieldMin, **int** hoursFieldMax) {

**this**.nrClientsField = nrClientsField;

**this**.nrQueuesField = nrQueuesField;

**this**.arrivalMinField = arrivalMinField;

**this**.arrivalMaxField = arrivalMaxField;

**this**.serviceMinField = serviceMinField;

**this**.serviceMaxField = serviceMaxField;

**this**.hoursFieldMin = hoursFieldMin;

**this**.hoursFieldMax = hoursFieldMax;

}

Util – a util class

4. Implementation and testing

After starting the application you will be able to change the default input , or put your own. After all is ready , press the “Start” button.

On the interface , the real time clients that will stay at the queue will be shown while in the console the real life time log of events.

After the queues are empty , for each queue there will be shown , the summary for queue " + id + ":\nAverage waiting time: " + getAverageWaitingTime()

+ "\n Average service time:" + getAverageServiceTime() + "\n Peak hour: " + getPeakHour() + "\n");

For concurrency , I used multithreading. Java is amulti threaded programming language which means we can develop multi threaded program using Java. A multi threaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition multitasking is when multiple processes share common processing resources such as a CPU. Multi threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

For the clean display of the date, from the util class I created a static method :

**public** **static** String convertToString(**long** time) {

**return** **new** SimpleDateFormat("yyyy/MM/dd HH:mm:ss").format(time);

}

Which converts miliseconds into a real date.

In order to generate the hours from the input , I created this method:

**private** **long** generateHour() {

**int** hour = ThreadLocalRandom.*current*().nextInt(simulationInput.getHoursFieldMin(),

simulationInput.getHoursFieldMax() + 1);

**int** minute = ThreadLocalRandom.*current*().nextInt(0, 61);

SimpleDateFormat sdf = **new** SimpleDateFormat("dd/MM/yyyy hh:mm:ss");

Date d = **null**;

**try** {

d = sdf.parse("15/04/2016 " + hour + ":" + minute + ":00");

} **catch** (ParseException e) {

e.printStackTrace();

}

**return** d.getTime();

}

This generated hour along with current.setArrivalTime(generateHour() + Math.*round*(ThreadLocalRandom.*current*()

.nextInt(simulationInput.getArrivalMinField(), simulationInput.getArrivalMaxField() + 1)) \* 60);

The interesting part here is that I put the round math in order to output exactly the minutes shown. Otherwise some extra seconds may be added always, which would actually make the program more realistic. But I sticked to the “simulation”.

Of course we have to keep in mind that all this is long int and it is miliseconds.

Here is where the util method comes to help. When outputting, I want to output the real date so : System.***out***.println(client.getName() + " entered @ " + Util.*convertToString*(client.getArrivalTime())

+ " has been at @ queue " + **this**.id + " waiting in total for " + client.getWaitingTime() +" minute(s) "

+ " being serviced for " + client.getServiceTime() + " minute(s) , leaving at " + Util.*convertToString*(client.getTerminationTime()) + "\n");

For random generating I used ThreadLocalRandom which is a random number generator isolated to the current thread. Like the global Random generator used by the Math class, a ThreadLocalRandom is initialized with an internally generated seed that may not otherwise be modified. When applicable, use of ThreadLocalRandom rather than shared Random objects in concurrent programs will typically encounter much less overhead and contention. Use of ThreadLocalRandom is particularly appropriate when multiple tasks (for example, each a ForkJoinTask) use random numbers in parallel in thread pools.

Usages of this class should typically be of the form: ThreadLocalRandom.current().nextX(...) (where X is Int, Long, etc). When all usages are of this form, it is never possible to accidently share a ThreadLocalRandom across multiple threads.

This class also provides additional commonly used bounded random generation methods.

5. Results

The results can be seen from the application. As for this we obtained a very easy to use application that simulates a queue system based on the data generated after the interaction of the user.

6. Conclusions

Queues are widespread in everyday life, an application that can simulate the behavior of a queue can be very useful to most shops and offices to collect taxes. I managed to realize an application that simulates the success addition, expectations and leaving a queue of a group of customers. The only thing that stops us to use this simulation system in reality is given the impossibility of knowing exactly input data. Even if we make estimates and use this system we will not have exactly 100%, but we can still choose to use when we have reliable data, specifically when queuing system will set hours before.

7. Bibliography

<https://en.wikipedia.org/wiki/Thread_%28computing%29>

<http://www.tutorialspoint.com/java/java_multithreading.htm>