*Technical University of Cluj-Napoca*

*Faculty of Automation and Computers*

*Department if Computer Science*

*2nd Semester 2015-2016*

*Programming Techniques*

*Homework 3*

Student: Hitter András

Group: 30424

Contents

[Processes 3](#_Toc448832786)

[Threads 4](#_Toc448832787)

[1. Example of working 6](#_Toc448832788)

[2. Design 7](#_Toc448832789)

[2.1. Relational Diagram 7](#_Toc448832790)

[3.3 Packages and Interfaces 12](#_Toc448832791)

[3. Using and testing the application 14](#_Toc448832792)

[4. Conclusions 15](#_Toc448832793)

[5. References 16](#_Toc448832794)

Abstract

In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads — but concurrency is possible even on simple systems, without multiple processors or execution cores.

## Processes

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.

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support *Inter Process Communication* (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a ProcessBuilder object. Multiprocess applications are beyond the scope of this lesson.

## Threads

Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to create additional threads, as we'll demonstrate in the next section.

Homework 3

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

Description Queues are commonly seen both in real world and in the models. The main objective of a queue is to provide a place for a "client" to wait before receiving a "service". The management of queue based systems is interested in minimizing the time amount its "clients" are waiting in queues. One way to minimize the waiting time is to add more servers, i.e. more queues in the system (each queue is considered as having an associated processor) but this approach increases the costs of the supplier. When a new server is added the waiting clients will be evenly distributed to all current available queues. The application should simulate a series of clients arriving for service, entering queues, waiting, being served and finally leaving the queue. It tracks the time the clients spend waiting in queues and outputs the average waiting time. To calculate waiting time we need to know the arrival time, finish time and service time. The arrival time and the service time depend on the individual clients – when they show up and how much service they need. The finish time depends on the number of queues, the number of other clients in the queue and their service needs. 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.

# Example of working

Assumed the entered values are correct (for example 3/50/1/5) the program takes these values and processes this way:

* Sets the number of servers to 3 and also creates a new thread for each server
* Sets the stopping value (50 -> after 50 ”time units” just stops)
* Sets the bounds for the random processing time generator, so that we know that the minimum value which a process (customer in a shop) can get is 1 (in this case); and so with the maximum value (5).
* The **Scheduler** is the class which creates new **Server**s and **Thread**s. Each Server gets one thread. The servers are organized in a List, so we can check on all of them at the same time. It also controls the clients, which means that it does not let for a Server to have more than 10 (in this case) clients at the same time.
* The **Simulator** creates a new Scheduler and generates the processing times for each “client” and calls the Display method
* The SimulatorFrame it simply checks the number of servers and visualizes the processes and clients as it is necessary.
* The Handler Class controls what happens when we push the START or STOP button and reads the user inputs.

# Design

## Relational Diagram

To resolve the problem, I’ve chosen to use six different classes, so that the design of application would be easier.

* Simulator + (main())
* Task
* HandlerClass
* Scheduler
* Server
* SimulatorFrame

To get a better view related to the attributes of each class, there are below the UML diagrams for each class. Thus, we can see every class with objects and their methods.

D:\Letöltések\HW3_threads asd.png

Class Design

1. *SimulatorFrame*: **public** **class** SimulatorFrame **extends** JFrame. This class is designed to create a graphical user interface so the application would be easier to use. We consider the Interface class as a subclass of the predefined class "JFrame" so we can use objects of type "button", "frame" or "panel".

In this class there is only a constructor and a displayData method, by wich is realisied the creation of a container. To this container there are added more panels: inputData, panel. The previous one is responsible for containing -4- JTextFields in which the user may enter the parameters. The last one is responsible for containing buttons, for activating different methods of the program: start, stop. The program will not accept the command start again after the stop was pressed. In case of not filling all fields correctly an error message appears and says: Fill all the fields!!!.

The attributes if the SimulatorFrame Class:

To achieve the desired SimulatorFrame I needed many attributes of different types:

* A JFrame: it is created by extending the JFrame
* Panels:
  + Private panel inputData – contains all the text boxes necessary to maintain the communication with the user
  + Private panel - contains all the buttons necessary which by the user interacting with the program can understand what he or she wants and execute the corresponding operation(s).
* 2 Buttons for executing the desired operations by the user:
  + Public static JButton start;
  + Public static JButton stop;
* Textfields:
  + Public static JTextField noOfServers;
  + Public static JTextField finish;
  + Public static JTextField min;
  + Public static JTextField max;
* Constructor:
  + Public SimulatorFrame() {…} – the initializations, coloring, placing(layout) are done here.

The following elements will be created: frame, textfields, buttons, labels.

* Creating a new HandlerClass object:
  + HandlerClass handler = new HandlerClass();

Every button is added to and ActionListener: start.addActionListener(handler);

stop.addActionListener(handler);

1. *HandlerClass* class: public class HandlerClass implements ActionListener;

This class was created to keep track of the user activity, and do operations corresponding to the user will. This class implements ActionListener class: public class HandlerClass implements ActionListener;

This class contains:

* Simulator object( attributes ):
* Thread object
* ( attributes ) nrOfServers, finish, min, max which will let the program know if the parameters were introduced correctly or not.
* The actionPerformed(ActionEvent): void method.

This method always listens to the buttons, textfields. If the user enters something this method reacts and performs operations predefined for the case that a particular button (or enter in a textfield) is pressed.

Also checks the correctness of the input. It will give instructions if an incorrect input occurs.

1. *Scheduler* class

This class is the heart of everything in this project! The operations class contains the implementation of all the scheduling actions with the Servers and Tasks.

In this class there is:

* CONSTRUCTOR which generates the desired number of servers and assigns to each a thread
* 3 well implemented methods: getTasks, toString, dispatchOnServer
* **public** Task[] getTasks(**int** id) {

**return** *serverList*.get(id).getTasks();

}

It just simply returns all the tasks related to one Server

* **void** dispatchTaskOnServer(Task t) {

**for** (Server s : *serverList*) {

**int** size = s.getTasks().length;

**if** (size < 10) {

s.addTask(t);

s.serviceTime++;

**break**;

}

}

}

It coordinates the threads. This method says to every task that which server it should go to.

* **public** String toString() {

**return** "Thread: \n " + *serverList*.get(0).getTasks().length;

}

This method displays the tasks, so we see what is happening right now.

1. *Server* class

This class is responsible for modelling a server. It stores in a *blockingQueue* all the tasks related to it, and calculates for each the waiting time and puts it into an AtomicInteger so every task has it’s own waiting time.

In the run method takes some tasks. In the addTask method adds factually the tasks and in the getTasks method returns the tasks.

1. *Simulator and SimulatorFrame* classes

The first one in the run method adds tasks to a server and displays them until it reached the finish time or the STOP button is pressed.

The second one displays the servers and clients for the Simulator class.

## Packages and Interfaces

A Java package is a mechanism for organizing Java classes into namespaces. Java packages can be stored in compressed files called JAR files, allowing classes to download faster as a group rather than one at a time. Programmers also typically use packages to organize classes belonging to the same category or providing similar functionality. A package provides a unique namespace for the types it contains. Classes in the same package can access each other's package-access members.

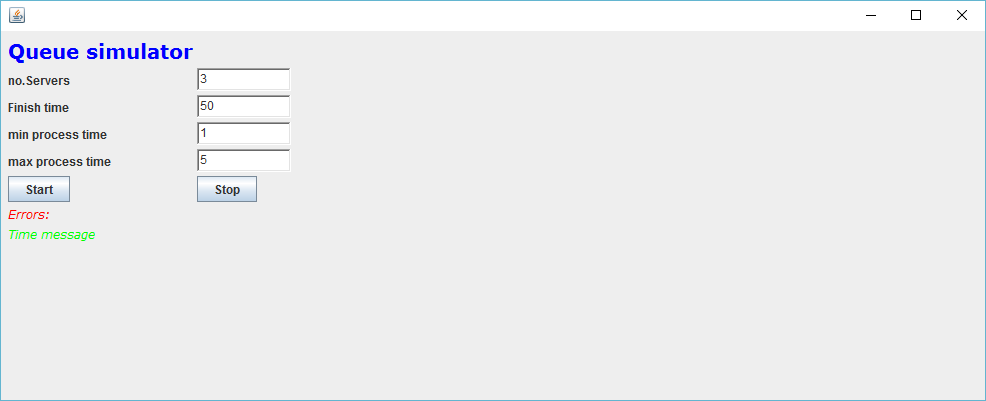
A package allows a developer to group classes (and interfaces) together. These classes will all be related in some way – they might all have to do with a specific application or perform a specific set of tasks.

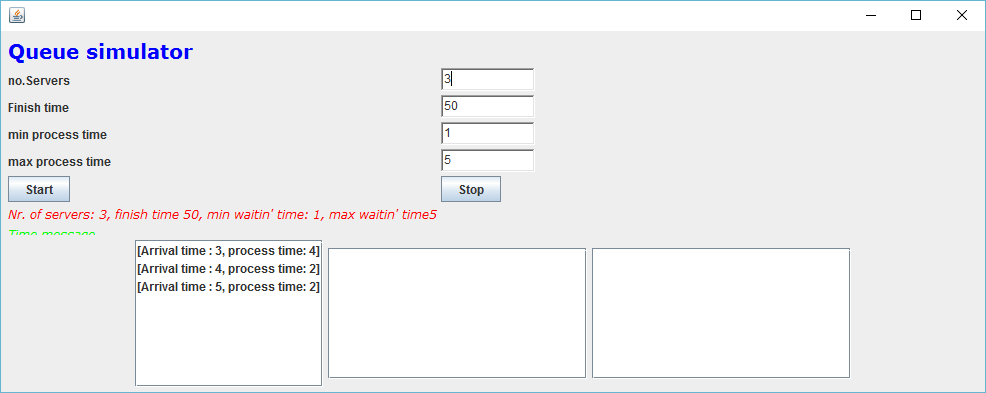
For this application the following packages are imported, each of them having a certain role for the proper working of the application. We import them in the Gui Class (most of them relate to the user interface properties):

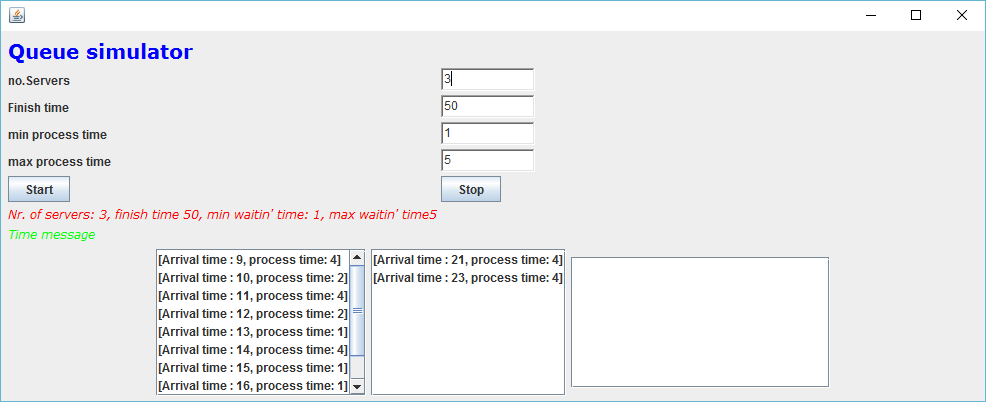
* import java.awt: Contains all of the classes for creating user interfaces and for painting graphics and images. A user interface object such as a button or a scrollbar is called, in AWT terminology, a component. The Component class is the root of all AWT components.
  + java.awt.BorderLayout: A border layout lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west, and center.
  + java.awt.Color: The Color class is used encapsulate colors in the default RGB color space or colors in arbitrary color spaces identified by a [ColorSpace](http://docs.oracle.com/javase/1.4.2/docs/api/java/awt/color/ColorSpace.html).
  + java.awt.Dimension: This encapsulates the width and height of a component (in integer precision) in a single object.
  + java.awt.GridLayout: The GridLayout class is a layout manager that lays out a container's components in a rectangular grid for a better view of all the buttons and textfields which are added to the main panel.
* import java.awt.event
  + java.awt.event.ActionEvent;
  + java.awt.event.ActionListener;
* import javax.swing: Typical Swing applications do processing in response to an event generated from a user gesture. For example, clicking on a JButton notifies all ActionListeners added to the JButton. That’s why we use this package for creating the user interface Gui.
  + javax.swing.JButton;
  + javax.swing.JFrame; javax.swing.JLabel; javax.swing.JPanel;

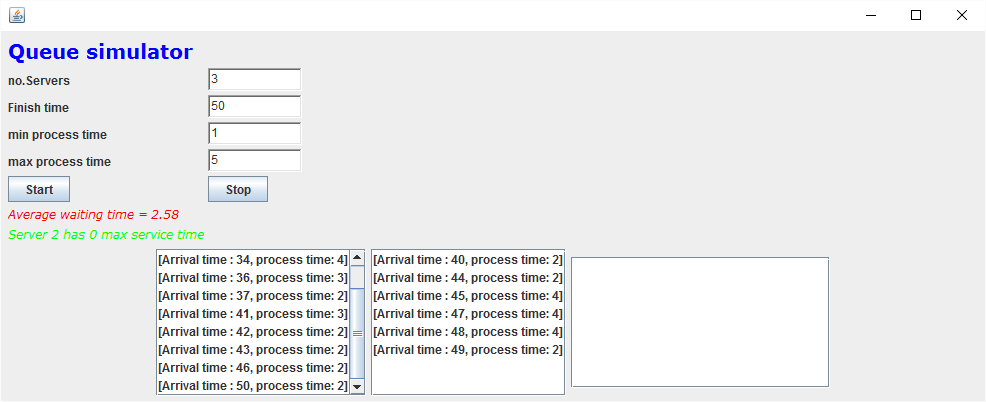
## Using and testing the application

In order to use the application open Homework3. This will open a window which generates the Gui class. Thus the user can enter the desired values and selecting start or stop buttons.









# Conclusions

If one can use threads he/she is able to do more efficiently many things, which are almost impossible otherwise.

It was a really good practice, which taught me many things about this kind of programming.

# References

[*http://stackoverflow.com/*](http://stackoverflow.com/)

[*http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html*](http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html)

*Teacher(s)*