Technical University of Cluj-Napoca

Faculty of Automation and Computer Science

2nd Semester 2016-2017

Queuing Based System Simulation

Programming Techniques

Homework 2

Student Ieremiasș Viorel

Group 30422

Contents

[1. Problem specification 3](#_Toc479036767)

[2. Problem analysis 4](#_Toc479036768)

[2.1. Theoretical basys 4](#_Toc479036769)

[2.2. Analysis 6](#_Toc479036770)

[2.3. Scenarios, use cases 7](#_Toc479036771)

[3. Design 10](#_Toc479036772)

[3.1. UML diagram 10](#_Toc479036773)

[3.2. Class design 10](#_Toc479036774)

[3.3. Data structures 11](#_Toc479036775)

[3.4. Other classes 12](#_Toc479036776)

[4. Conclusions & further improvements 13](#_Toc479036777)

[5. Bibliography 13](#_Toc479036778)

# Problem specification

**Objective**

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

**Description**

Queues are commonly used to model real world domains. 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 their "clients" are waiting in queues before they are served. 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 service supplier. When a new server is added the waiting customers 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 customers 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 clients in the queue and their service needs.

**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;

# Problem analysis

## Theoretical basys

Multithreading is the ability of a central processing unit (CPU) or a single core in a multi-core processor to execute multiple processes or threads concurrently, appropriately supported by the operating system. This approach differs from multiprocessing, as with multithreading the processes and threads share the resources of a single or multiple cores: the computing units, the CPU caches, and the translation lookaside buffer (TLB).

Where multiprocessing systems include multiple complete processing units, multithreading aims to increase utilization of a single core by using thread-level as well as instruction-level parallelism. As the two techniques are complementary, they are sometimes combined in systems with multiple multithreading CPUs and in CPUs with multiple multithreading cores.

The multithreading paradigm has become more popular as efforts to further exploit instruction-level parallelism have stalled since the late 1990s. This allowed the concept of throughput computing to re-emerge from the more specialized field of transaction processing; even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multitasking among multiple threads or programs. Thus, techniques that improve the throughput of all tasks result in overall performance gains.

Two major techniques for throughput computing are multithreading and multiprocessing.

**Advantages**

If a thread gets a lot of cache misses, the other threads can continue taking advantage of the unused computing resources, which may lead to faster overall execution as these resources would have been idle if only a single thread were executed. Also, if a thread cannot use all the computing resources of the CPU (because instructions depend on each other's result), running another thread may prevent those resources from becoming idle.

If several threads work on the same set of data, they can actually share their cache, leading to better cache usage or synchronization on its values.

**Disadvantages**

Multiple threads can interfere with each other when sharing hardware resources such as caches or translation lookaside buffers (TLBs). As a result, execution times of a single thread are not improved but can be degraded, even when only one thread is executing, due to lower frequencies or additional pipeline stages that are necessary to accommodate thread-switching hardware.

Overall efficiency varies; Intel claims up to 30% improvement with its Hyper-Threading Technology, while a synthetic program just performing a loop of non-optimized dependent floating-point operations actually gains a 100% speed improvement when run in parallel. On the other hand, hand-tuned assembly language programs using MMX or AltiVec extensions and performing data prefetches (as a good video encoder might) do not suffer from cache misses or idle computing resources. Such programs therefore do not benefit from hardware multithreading and can indeed see degraded performance due to contention for shared resources.

From the software standpoint, hardware support for multithreading is more visible to software, requiring more changes to both application programs and operating systems than multiprocessing. Hardware techniques used to support multithreading often parallel the software techniques used for computer multitasking. Thread scheduling is also a major problem in multithreading.

**Types of multithreading**

There are multiple types of multithreading.

The simplest type of multithreading occurs when one thread runs until it is blocked by an event that normally would create a long-latency stall. Such a stall might be a cache miss that has to access off-chip memory, which might take hundreds of CPU cycles for the data to return. Instead of waiting for the stall to resolve, a threaded processor would switch execution to another thread that was ready to run. Only when the data for the previous thread had arrived, would the previous thread be placed back on the list of ready-to-run threads.

The second type is interleaved multithreading. Its purpose is to remove all data dependency stalls from the execution pipeline. Since one thread is relatively independent from other threads, there is less chance of one instruction in one pipelining stage needing an output from an older instruction in the pipeline. Conceptually, it is similar to preemptive multitasking used in operating systems; an analogy would be that the time slice given to each active thread is one CPU cycle.

The most advanced type of multithreading applies to superscalar processors. Whereas a normal superscalar processor issues multiple instructions from a single thread every CPU cycle, in simultaneous multithreading (SMT) a superscalar processor can issue instructions from multiple threads every CPU cycle. Recognizing that any single thread has a limited amount of instruction-level parallelism, this type of multithreading tries to exploit parallelism available across multiple threads to decrease the waste associated with unused issue slots.

## Analysis

**General overview**

This application should simulate customers waiting to receive a service (e.g. supermarket, bank, etc.). Like in the real world, they have to wait in queues, each queue processing clients simulatenously. The idea is to analyze how many clients can be served in a certain simulation interval, by entering parameters in an intuitive, user-friendly, application graphical interface.

**Input and output**

The customers are generated randomly, each having it’s own service time and arrival time. However there are a lot of parameters the user of the simulation can set to observe the evolution of the system in different contexts. These parameters include:

* the maximum number of queues available to process customers. The shop, for example, is considered to be built with a large number of queues, enough to handle even the most demanding of the situations, but not all of them are necessary to run the shop in good conditions in some situations. The user can set the number of queues are opened and serve clients at the beginning of the simulation. If the parameter is not set, the maximum number of queues is implicitly assumed;
* minimum and maximum arrival interval: the delay between customers arriving to receive a service. When generating clients, the arrival time will be chosen randomly based on this values; These parameters allows the user to simulate different flows of customers coming to receive service and thus models for example the rush hour;
* minimum and maximum service time: the number of minutes needed for a client to be processed, a value is chosen randomly. The purpose of this feature is to simulate different types of businesses or services, no matter if they require more or less time to be accomplished;
* simulation interval: sets a duration for the simulation

Other useful inputs from the user include the commands to start the simulation after the parameters are set, to pause the simulate, for example to trace in detail the evolution of the queues and the possibility to resume the normal working regime after a pause.

The results of the simulation can be observed in real-time on the GUI of the application.

The user can see:

* how many customers were served (not generated), during the simulation interval;
* the average service time of the served customers (in minutes);
* the average waiting time of the served customers (in minutes): how much the customers have waited in queue to receive the service;
* the “peak hour”, when the most clients were served;
* the general evolution of each of the opened queues: how clients arrive, where they position, when are they served and when they leave

A log-type window shows detailed information about the requirements of the customers (service time) and exact time of their arrival and leave, the time spent in the queue and who they were served by. The advantage of the visual representation of the evolution of the system is that it provides a set of data that can be analyzed both analytically and intuitively.

## Scenarios, use cases

Title: Simulate queue evolution

Resume: After inserting the input parameters in the user interface (number of queues, min. and max. serving time, min. and max. arrival interval, and the simulation interval), and pessing the START button, the user will see animations for each of the queues, a short log in the left part of the application window, containing detailed information about the clients arriving and being served.

Actors: User

Scenarios:

a) Preconditions: application is ready to use

b) Normal scenario:

1) User successfully inserts the input parameters mentioned above

2) User pushes the "START" button to start the simulation;

3) Application displays the short log, animations and real-time statistics

4) The buttons for pausing and resuming the simulation are enabled.

c) Alternative scenario:. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . .

1) User types wrong data in the input fields or doesn’t input data at all in one or more of the fields. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . .

2) User pushes the "START" button attempting to start the simulation;

3) Simulation starts with some predefined parameters that automatically take the place of the parameters that are not filled in or are not given proper values. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . .

# Design

## UML diagram

. . . . . . . . . . . . . .

## Class design

As it can be seen in the UML diagram on the previous page the approach followed the real life entities (Customer, Queue, ServiceProvider) and modeled them accordingly. These entities are grouped in the “model”, since they are tightly coupled and a change in one of the Classes would result in some changes in the other classes.

Client is an individual entity, which has to be modeled separately, because each cne has it's own id and different properties, like arrival time, service time and wait time in the queue. Customers are generated randomly during the simulation, and assigned to the shortest queue (the queue with the smallest number of customers).

Server represents the entity of the worker in that specific simulation. The only field of the server, efficiency, determines how the serve time of the client is transposed to the time of the simulation (how the skill of the server decreases of increases the time needed to serve the client.

Checkout represents a queue from the real world, where customers wait in line to receive a service. Clients are added to a queue by calling the method addClient(). After they are served, they are automatically removed from the queue.

Shop is composed of a set of queues, Checkout objects. The clients arrive in the shop with the no restriction, but always choose the shortest queue.

## Data structures

The BlockingQueue is the most important data structure used in this project.

It is a Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

BlockingQueue methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future: one throws an exception, the second returns a special value (either null or false, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit before giving up.

A BlockingQueue does not accept null elements. Implementations throw NullPointerException on attempts to add, put or offer a null. A null is used as a sentinel value to indicate failure of poll operations.

A BlockingQueue may be capacity bounded. At any given time it may have a remainingCapacity beyond which no additional elements can be put without blocking. A BlockingQueue without any intrinsic capacity constraints always reports a remaining capacity of Integer.MAX\_VALUE.

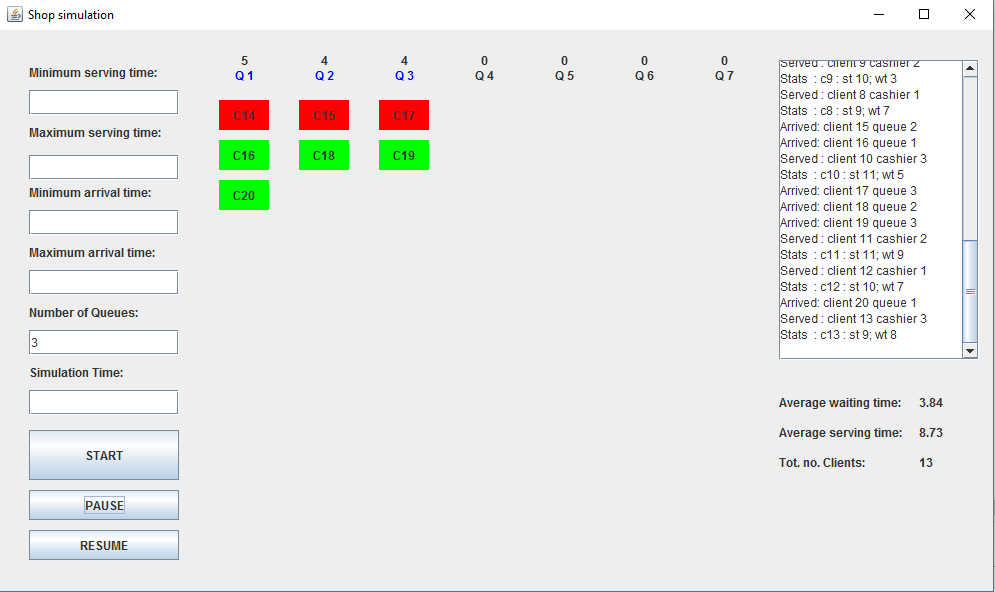
BlockingQueue implementations are designed to be used primarily for producer-consumer queues, but additionally support the Collection interface. So, for example, it is possible to remove an arbitrary element from a queue using remove(x). However, such operations are in general not performed very efficiently, and are intended for only occasional use, such as when a queued message is cancelled. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . .

The main reason for choosing BlockingQueue implementations though is because they are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control. However, the bulk Collection operations addAll, containsAll, retainAll and removeAll are not necessarily performed atomically unless specified otherwise in an implementation. So it is possible, for example, for addAll(c) to fail (throwing an exception) after adding only some of the elements in c. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . .

## Other classes

Except the above mentioned classes, that model the simulation, there are a few more auxiliary classes that perform different functions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

The Factory method in the factories package is responsible for randomly generating clients, servers and waiting times between the arrival of different clients.

The Graphical User Interface (GUI) is meant to be user-friendly and easy to use, even for non-specialists, due to the intuitive controls and very interactive. When starting the application animations of the queues present the progress of the clients that arrive and are served.

The Controller is another very important class. The project is built on the MVC model, so that the Controller notifies the View about the changes in the Model and vice-versa, facilitating the communication between them. In the case of this project, certain events in the model, like the apparition of a client are random events that cannot predicted. So, the View cannot be notified to make the required changes predictably. One solution is to setup a timer that refreshes the view periodically, 10ms being a period good enough for this purpose.

# Conclusions & further improvements

Working with multiple threads is an interesting challenge. It is on one hand, a situation that is widely spread in the domain of reality, where concurrent processes are common, but it requires, in the field of computer science a special approach, since the processes, instructions and data is designed to be operated with sequentially.

Further development: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

* keep a log of the events
* develop an algorithm that chooses when to open and close queues based on the number of clients in the simulation and the state of the system
* better enforce the simulation limits and allow more functionality and features regarding this. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . .

# Bibliography

<https://en.wikipedia.org/wiki/Multithreading_(computer_architecture)>

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

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html>

<https://www.caveofprogramming.com/java-multithreading/java-multithreading-thread-pools-video-tutorial-part.html>