**Queues Simulator**

**Student: Flueran Robert-Denis**

**Group: 30423**

**Semigroup: 1**

**1.Homework Objectives**

The main objective of the homework is designing a simulator for assigning clients to queues according to one of multiple strategies.

Steps (objectives) in designing the queues simulator:

* Creating Java classes that describe queues and clients

Task (Client) class is going to have fields for Arrival Time, Processing Time, Id and Overall Waiting Time (how much that client is going to stay in a queue). Server (Queue) class contains a list of Tasks (Clients), a waiting period, methods for running a thread and a cleanup variable that signals when the Server thread is going to be interrupted. More details can be found at

* Creating Java classes that describe assignment strategies

ConcreteStrategyQueue and ConcreteStrategyTime implement the addTask method from the Strategy interface. ConcreteStrategyQueue assigns clients to the queue that has the smallest number of clients already assigned to it. ConcreteStrategyTime assigns clients based on the time they have to wait at each of the queues. More details can be found at

* Creating a class that schedules the addition of a client and the strategy used for the assignment

Scheduler class has a list of servers (queues), a max number of servers, max tasks per server and an assignment strategy. More details can be found at

* Creating a “Main” class in which we simulate the whole thing

SimulationManager class has fields that define arrival and processing times, number of clients and number of servers. It also contains methods for generating NoOfClients random clients. More details can be found at

* Putting the application to the test with several input files

The input files contain information regarding the number of clients, number of servers, minimum and maximum arrival time for clients, as well as the number of simulation steps the application is required to make. More details can be found at

**2.Problem Analysis**

The problem consists mainly in ensuring that all the queues are synchronized and all the simulation times are computed correctly

**“Queues Simulator” use-case**

* An input and an output file are given as command line arguments to the application
* SimulationManager class takes information from the input file with regards to the number of clients, number of servers, min and max arrival time, min and max processing time and simulation time.
* A thread is run with an object of SimulationManager that contains all the data from the input file
* A method for generating a list of noOfClients random tasks is called from the constructor of SimulationManager.
* An assignment strategy is chosen for the simulaton through the changeStrategy(SelectionPolicy policy) method of the Scheduler class.
* The run() method (implemented from Runnable interface) is being executed that will keep assigning tasks to servers according the arrivalTime of the tasks up until the simulation time has been finished or there are no more clients to assign.
* Each time a task is being assigned to an empty queue, a new thread for that queue is created which will run simultaneously with all the other queue threads to compute the remaining processing time for the clients.
* If a queue becomes empty during the simulation, it’s thread is interrupted. A thread for that queue will be created and started right after a task is assigned to that queue.

**3.Design**

I have created a host of packages that contain all the classes needed for the simulation.

* Resource

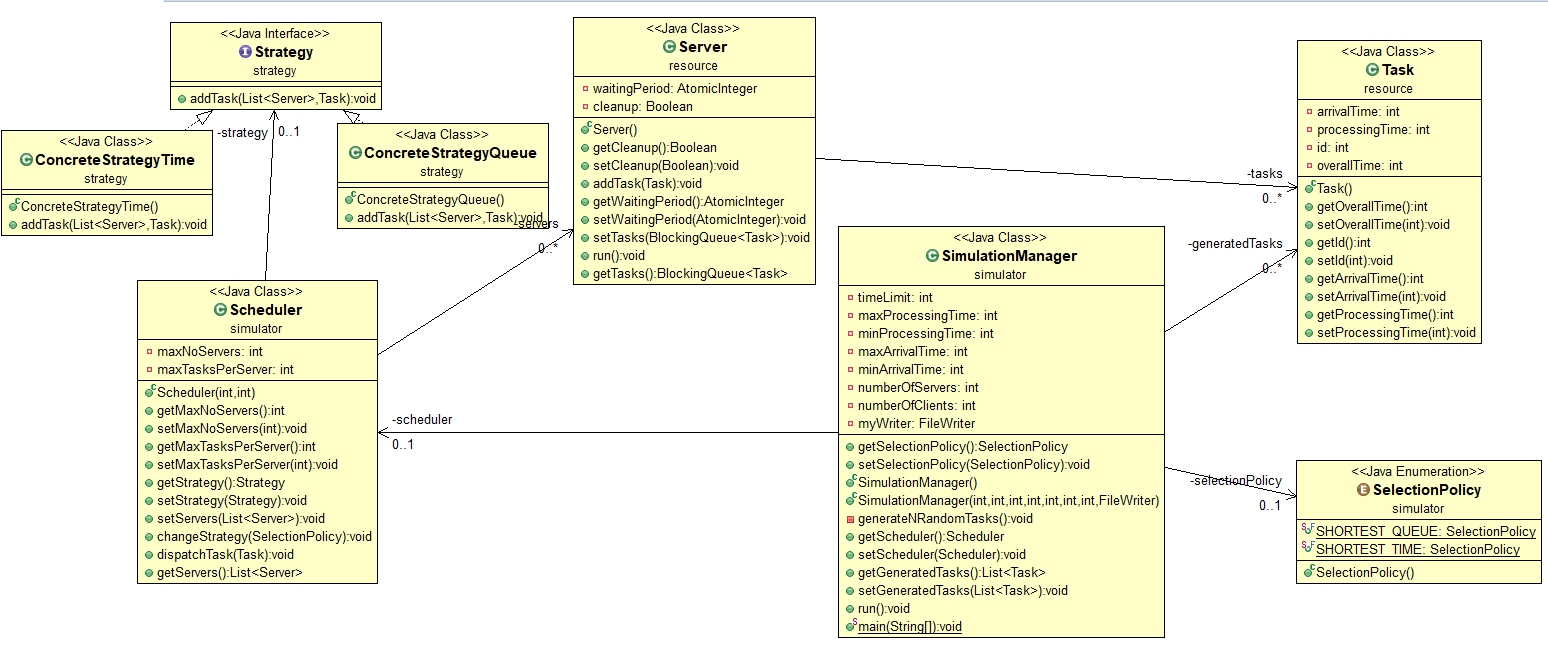
It contains the classes the simulation is working with (Tasks – clients, Servers – queues) and has run() method implemented in Server class such that we run a thread for each queue.

* Strategy

It contains the Strategy interface and two ways of implementing the interface: according to the shortest queue (ConcreteStrategyQueue), or according to the shortest waiting period (ConcreteStrategyTime).

* Simulator

It contains the Scheduler class (which takes care of assigning clients to queues according to a chosen strategy by calling a method in either one of the two Strategy classes) and SimulationManager class (which takes data from an input file, generates random clients and runs the main thread that actually does the simulation).

**UML Class Diagram**

**4.Implementation**

Let’s start with the Resource package. Here, we’ve got two classes:

* Task

Task simply defines the “characteristics” of a client. A client is defined by its id, its arrival time and processing time. In addition to this, I created a new field, overallPeriod, which we use to compute how much that client is actually going to stay in that queue (the times it takes for him to get in front of the queue + processingTime), as we need it for computing the average waiting time. I’ve only used id for system prints, to make sure the right task is removed.

Methods:

This class has no methods except for getters and setters for all the private variables.

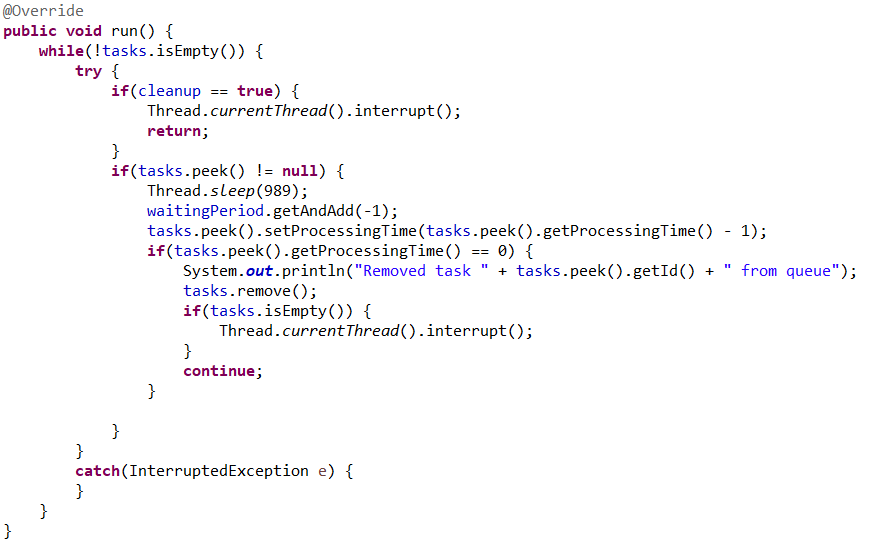
* Server

A Server consists of a queue of tasks (clients assigned to that queue), a waitingPeriod variable used for the ConcreteStrategyTime strategy and a cleanup variable (if we set it to true, the thread the server is running on is interrupted).

Methods:

Apart from getters and setters, the class also contains a method for adding a task to its queue and increasing the waitingPeriod and the run() method (implemented from Runnable interface), which decrements the waitingPeriod and the processingTime of the client in front of the queue each second and pops tasks when their processing time reaches zero. If the queue is empty after some time, then the thread is going to be interrupted.

**Most important method from the package:**



This represents the run() method from the Server class. It first checks if the Boolean cleanup flag is set to true (the cleanup flag will be set once the simulation time is over or there are no more remaining clients), which will result in the interruption of the Thread. Afterwards, it checks if there exists at least one element in the queue (if tasks.peek() == null, then there is no client in the queue).

In case there is an element in the queue, the thread is going to sleep for a second, the waitingPeriod is going to be decremented along with the processingTime of the client in the front of the queue. If the processingTime is over for the client in front, then the client is going to be removed. If the queue becomes empty after we pop the client at the peek, then we’ll interrupt the thread.

Let’s move on to the Strategy package, which also contains two classes:

* ConcreteStrategyQueue

The class implements the Strategy interface and has no fields.

Methods:

The only method it has is the only implemented from the Strategy interface. Here, addTask(List<Server> servers, Task t) assigns task t to the Server with the smallest number of tasks already assigned to it. If the Server the method assigns the task to happens to be empty, then a new thread with that server is started.

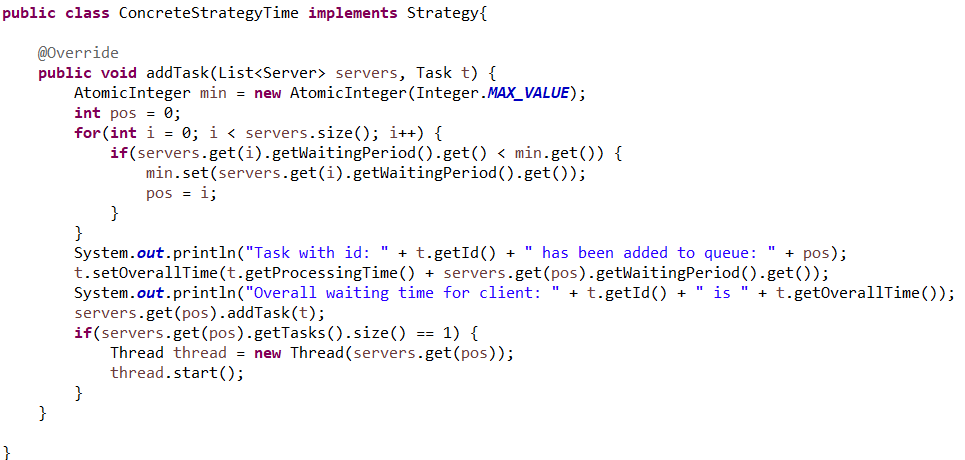
* ConcreteStrategyTime

The class implements the Strategy interface and has no fields.

Methods:

The only method it has is the only implemented from the Strategy interface. Here, addTask(List<Server> servers, Task t) assign task t to the Server with the smallest waitingPeriod. If the Server the method assigns the task to happens to be empty, then a new thread with that server is started.

**Most important method from the package:**



Since the time-related strategy has been used in the documentation, the addTask method from ConcreteStrategyTime is the most important method in the package. Here, a variable of AtomicInteger type (called min) is declared and initialized with the largest value possible to make sure we change it at least once.

The Servers List is traversed and the waiting period each server is compared to the value of min variable. In case we find a smaller value, we set that value to min and keep the position (sort of an id) of the server with that smaller waiting period.

When we get the minimum waiting period, we can set the overall waiting of the task we want to add as the SUM of its processing time and the current waiting period of the server with the minimum waiting period (because this actually represents how much a client is going to stay in a queue). Then, we add the task to the queue found at pos position.

The printing is meant to help the user see in real time which task has been added to which queue and also see its overall waiting time (which is going to help in computing the average waiting time for clients).

If the chosen server has size equal to 1 (it means that the server didn’t have any client before the newly added task), then a thread for that server is created and started.

Let’s describe the Simulator package:

* Scheduler

The class is comprised of a list of servers and variables for maximum number of servers, max tasks per server (which I thought would be the initial number of clients, since nothing about max tasks per server was mentioned in the documentation) and strategy used for assignment.

Methods:

Apart from the standard getters and setters, the class contains a method for changing the assignment strategy and a method for adding a task to a server.

* SimulationManager

The class consists of fields for number of clients, number of servers, min and max arrival time, min and max processing time, simulation time, selection policy, a list of randomly generated clients and a FileWriter for output.

Methods:

The main methods in this class are generateNRandomTasks(), which generates noOfClients random tasks, run() (implemented from Runnable interface) in which tasks are being added to list until the simulationTime is over (we make use of a variable called currentTime, which is incremented each second..when it happens to be equal with the simulation time, we set the flag for interrupting all the Server threads and we interrupt the SimulationManager thread) and the main(String[] args) method, which reads all the input data from a file and passes it to a constructor for a SimulationManager object, and starts a thread with the new SimulationManager object.

**Most important method from the package:**



This represents the run() method from SimulationManager class. Here, we are going to need three variables: currentTime simply states how many time units have passed and is incremented each second until it reaches timeLimit or no more clients are to be assigned (noRemainingTask() checks for remaining clients); sum is the variable used to store the SUM of the overall waiting time of all clients; size variable is used to store the number of clients that have actually finished shopping and were assigned to queue during the simulation (we are going to get the average waiting time by dividing sum to size). I have decided to divide the sum to the clients actually assigned to a queue because the clients that didn’t even get to a queue don’t have a waiting time (I’ve taken into consideration the cases where the simulation time is small or the ratio between the number of clients and servers is really large).

The method firstly checks if the currentTime is smaller that the simulation time given as input. Afterwards, it checks if there are any clients left unassigned. If all clients have already been assigned and the queues are empty, the average waiting time will be printed to the output file and all the Server threads (interruptThreads() sets the cleanup flag for all of them), along with the main thread, are interrupted, and the output file is closed.

If there are still clients to be assigned, queues that aren’t empty and the simulation time isn’t over, the method proceeds by adding all the clients with their arrival time equal to currentTime to corresponding servers and removing them from the List containing remaining tasks. Afterwards, the methods calls two other methods: printTasks() (which simply prints remaining clients to the output file) and printQueues() (which prints the contents of the queues to the output file).

CurrentTime is incremented and we check if it is equal to the simulation time. If the method got to the end of the simulation, it will then print the Average Waiting Time to the output file and interrupt all threads by interrupting the current one and setting the flag for all Server threads to interrupt.

**6.Results**

I have managed to perform all the requirements for the assignment. I’ve put the application to the test with the input files given in the assignment’s documentation and added a couple other tests to make sure my app does the correct computations.

Firstly, I tested the application by assigning some values to the SimulationManager fields. Afterwards, I’ve created five input test and five output test files with different amounts of servers and tasks.

**7.Conclusions**

This has been a really interesting assignment for me, since it’s the first time I get to learn about and use threads. I admit that, given these hard times, in which feedback and teaching from university cadres is not that easy and natural when compared to face to face meetings, I didn’t get to fully master the concepts related to threads. From the tests, it seems that the application is running fine, but I am not sure that how I made it work is how it should be done. But at least it gets the job done.

I believe the “multi-threading” part of the application could be improved.

**8.Bibliography**

* Threads

<http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdf>

* Interrupting threads

<https://www.baeldung.com/java-thread-stop>

* How to use threads

https://www.youtube.com/watch?v=F-CkaU8aQZI