MINISTERUL EDUCAȚIEI ȘI CERCETĂRII ȘTIINȚIFICE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

DEPARTAMENTUL CALCULATOARE

ASSIGNMENT 2

QUEUES MANAGEMENT APPLICATION USING THREADS AND SYNCHRONIZATION MECHANISMS

NUME STUDENT: Boar Daniel-Ioan
GRUPA: 30223

Cuprins

1. Obiectivul temei

Obiectivul propus in rezolvarea acestei teme este de a realiza un program care simuleza o situatie reala de utilizare a cozilor intr-o institutie. In acest fel se vor simula situatii reale de sosire a clientilor la ghisee sau la alte cozi unde se asteapta procesarea comenzilor acestora.

Aplicatia doreste sa ajute clientul sa ajunga cat mai repede la inceputul cozii pentru a astepta cat mai putin la coada si pentru a fi servit cat mai repede. In momentul in care un client se asaza la o coada, noi stim ca urmatorul client care vine in institutie se va aseza la coada cu mai putini clienti (Shortest Queue Strategy).

2. Analiza problemei, modelare, scenarii, cazuri de utilizare

Pentru realizarea aplicatiei am folosit 6 clase. Aceste clase se impart in: clase care ne ajuta sa simulam evolutia cozilor (**Scheduler, Strategy, SimulationManager**), clase pentru crearea de clienti si cozi (**Server, Task**) si o clasa care ne ajuta la implementare interfetei grafice (**SimulationFrame**).

Valorile fiecarui camp din interfata (timpul total de lucru, intervalul de sosire, intervalul de servire, numarul de cozi, numarul de clienti) sunt valori intregi care se preiau din text-field-uri si sunt transmise progamului (SimulationManager) prin simpla apasare a butonului „Start”. In momentul apasarii acestui buton, ni se va simula intr-un textArea din interfata, in timp real, mecanismul de prelucrare a clientilor. Se va putea vedea la fiecare moment de timp care sunt clientii care trebuie sa ajunga in institutie, care sunt clientii care sunt deja in coada si asteapta sa fie serviti si clientii care sunt deja serviti la momentul de timp respeciv.

3. Proiectare

M-am gandit sa creez aplicatia cat de usor posibil si am lasat comentarii la aproape fiecare metoda sau operatie pe care am implementat-o. In primul rand, pentru a fi mai usor, am ordonat clientii dupa timpul de sosire pentru a-mi fi mai usor sa vad ce client urmeaza sa intre in institutie sau la coada. Alt aspect ar fi implementarea unei stategii care ne adauga un client la cea mai scurta coada.Toate testele efectuate pana acum au demonstrat faptul ca simularea functioneaza asa cum ar trebui, chiar si pe teste mai mari, mai putin calcularea timpului mediu de asteptare in coada de catre un clienti, operatie care nu stiu daca a fost implementata cum ar fi trebuit.

4. Implementare

Aplicatia mea dispune de 6 clase. Aceste clase impreuna cu metodele si functionalitatile lor vor fi descrise mai jos.

• Clasa Task

Aceasta clasa are 3 atribute: id, arrivalTime, serviceTime care reprezinta id-ul clientului, timpul de sosire si de servire al acestuia. Acestea sunt private deoarece utilizam conceptul de incapsulare a datelor.

In constructorul acestei clase, avem initializarea fiecarui client cu valorile specificate.

```
public  Task(int id, int arrivalTime1, int serviceTime1)
```

In plus, am create cate un getter pentru fiecare atribut, iar pentru serviceTime am creat si un setter. O alta metoda din aceasta clasa este aceea de toString in care am facut un pretty_print a fiecarui client in care ne afiseaza atributele sale (5 = id, 2 = arrivalTime, 5 = serviceTime).

• Clasa Server

Aceasta clasa implementeaza coada propriu-zisa. Ea contine un „BlockingQueue” de clienti (task-uri) din care se scot sau se adauga clienti pe masura ce programul ruleaza. Ca atribut mai avem si o variabila AtomicInteger in care calculam timpul pe care il petrece un client in coada.

Constructorul acestei clase initializeza coada la variabila intreaga primita ca parametru si va initializa waitingPeriod-ul cu 0.

```
public Server(int i)
```

Metoda numita „addTask” este o metoda sincronizata cu ajutorul careia adaugam un client in coada si ii incrementam waitingPeriod-ul.

```
public void addTask(Task newTask)
```

Metoda implementata de runnable este folosita pentru rularea thread-ului. Totul se intampla intr-un while(true) care este o bucla infinita. Mai intai verificam daca avem clienti in coada, iar pe urma punem pe sleep, 1 secunda, coada respectiva. La fiecare moment de timp vom decrementa serviceTime-ul clientului pana va ajunge la 0, iar apoi clientul va fi scos din coada.

```
public void run()
```

O ultima metoda din aceasta clasa este un getter pentru fiecare coada.

```
public BlockingQueue<Task> getTasks()
```

• Clasa Scheduler

Ca atribute avem o lista de cozi, un numar maxim de servere (cozi), un numar de pe clienti intr-un server (coada) si o instanta a unui obiect de tipul Strategy.

In constructorul acestei clase vom initializa lista de cozi, numarul de servere si numarul de clienti. Un alt aspect ar fi ca vom parcurge toate cozile si vom crea cate un obiect de tipul coada, vom adauga obiectul creat la lista de cozi si vom crea si porni un thread cu coada respeciva.

```
public Scheduler(int nrsv, int maxTasksPerServer)
```

In metoda „dispatchTask” vom avea un client pe care il vom adauga in coada in functie de o strategie aleasa. Eu am ales strategia in care ne adauga clientul in coada cea mai mica ca dimensiune.

```
public void dispatchTask(Task t)
```

Pe langa aceste metode, mai avem si un getter pentru lista de cozi.

```
public List<Server> getServers()
```

• Clasa Strategy

In aceasta clasa avem o singura metoda implementa, metoda „addTask” care are ca parametrii o lista de cozi si un client. In interiorul acesteia vom parcurge toate cozile si vom afla coada cu marimea cea mai mica, adica coada care are cei mai putini clienti. La finalul metodei vom adauga clientul la coada cu cei mai putini clienti.

```
public void addTask(List<Server> sv, Task t)
```

• Clasa SimulationManager

Aceasta clasa este cea mai stufoasa dintre toate si cea mai complicata de implementat. In ea avem mai multe atribute, un obiect de tip Scheduler, o lista de clienti, un fisier in care vom afisa simularea programului, un TextArea pe care il legam de interfata pentru a afisa in el simularea programului si mai avem acele atribute pe care le-am prezentat si la inceputul prezentarii (timeLimit, intervalul de sosire, intervalul de servire, numarul de clienti si numarul de cozi).

Constructorul acestei clase initializa fiecare atribut in parte, creeaza si initializeaza un obiect scheduler, apeleaza o meteoda de generare random a clientilor in functie de timpii de sosire si de servire si creeaza fisierul txt.

```
public SimulationManager(int time, int nrc, int nrs, int mina, int maxa, int mins, int maxs, JTextArea a)
```

Prima metoda implementata este o metoda care ne sorteaza lista de clienti in functie de timpul de sosire.

```
private static void sortByArrivalTime(List<Task> t)
```

Urmatoarea metoda este cea de generare random a clientilor. Aici calculam un numar random de sosire si de servire si apoi adaugam in lista de clienti, o persoana cu aceste valori calculate. Urmatorul pas este sa apelez functia de sortare dupa timpul de sosire si apoi va fi o mica afisare a tuturor clientilor.

```
private void generateNRandomTasks()
```

O alta metoda este una booleana care ne verifica daca se termina executia programului nostru. Mai intai verific daca mai sunt clienti care trebuie sa ajunga in instituite iar apoi verific daca mai sunt clienti care asteapta in coada.

```
public boolean end(List<Server> sv)
```

Urmatoarea metoda este un pretty print care primeste ca parametrii lista de cozi si timpul curent si va afisa in timp real, in interfata si in fisierul txt, timpul la care suntem, clientii care urmeaza sa intre in institutie si cozile cu clientii fiecareia. Daca coada este goala, se va afisa „closed”.

```
public void prettyPrint(List<Server> sv, int currentTime)
```

In metoda implementata de runnable avem lucrurile mai importante ale acestei clase. Totul se intampla intr-un while mare care ne verifica daca se termina executia programului nostru. In primul rand, vom parcurge toti clientii si vom alege acei clienti care au timpul de sosire egal cu timpul curent si ii vom adauga intr-o lista noua de clienti, numita „tasksRemoved” sau pe scurt, clientii care vor fi stersi din lista mare generata random. Tot aici se va calcula media timpului de servire si media timpului de asteptare a fiecareui client.

Dupa aceasta iteratie, se face apelarea de pretty print pentru a afisa in timp real timpul si clientii. Se va verifica daca clientii din coada au timpul de servire si daca il au, atunci ii vom scoate din coada respectiva. Se va calcula si ora de varf din coada pentru a o afisa la sfarsitul executiei programului pentru a sti la ce moment de timp a fost cel mai aglomerat in institutie.

Dupa toate cele de sus, vom pune pe sleep thread-ul timp de o secunda, iar apoi, dupa iesirea din bucla de while vom face afisarile in fisier si in interfata a timpului mediu de servire, timpului mediu de asteptare si a momentului de timp cand institutia se afla in ora de varf.

```
public void run()
```

In aceasta clasa se afla si main-ul in care creem un obiect de tipul SimulationFrame care este de fapt o apelare a constructorului din interfata.

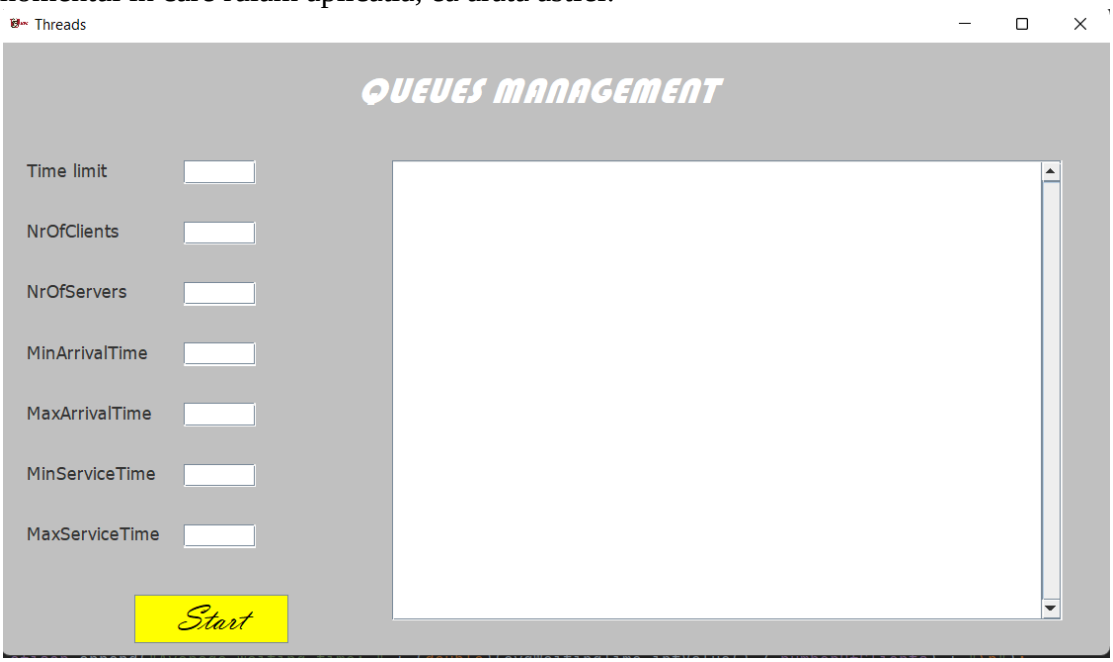
```
public static void main(String[] args) {
    SimulationFrame f = new SimulationFrame();
    f.setVisible(true);
}
```

- **Clasa SimulationFrame**

Scopul acestei clase este de a crea interfata grafica si de a face legatura cu programul nostru. Aceasta metoda preia valorine introduse de noi in campurile disponibile din interfata, le parseaza si mai apoi porneste executia propriu-zisa a programului, urmand ca rezultatele sa se afiseze intr-un textArea.

5. Rezultate

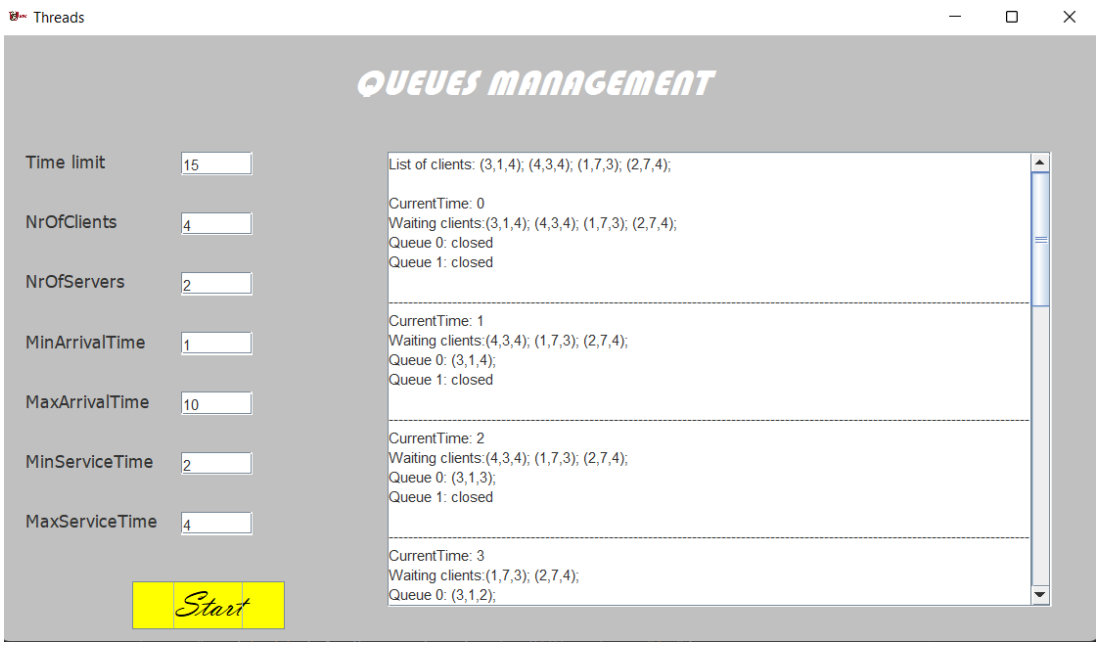
In momentul in care rulam aplicatia, ea arata astfel:



Dupa cum se vede, aplicatia asteapta ca input un timelimit, nr de clienti, nr de cozi (servers), intervalul de sosire si intervalul de servire.

In momentul in care se vor completa cele 7 campuri, se va apasa pe butonul „Start” si va incepe executia programului.

Haideti sa testam programul nostru si sa introducem ca inputr-uri, Time limit = 15, NrOFClients = 4, NrOfServers = 2, MinArrivalTime = 1, MaxArrivalTime = 10, minServiceTime = 2, maxServiceTime = 4.



S-au obtinut urmatoarele rezultate pentru timpul de servire, timpul de asteptare si ora de varf:

```
---------------------------------------
Average waiting time: 3.0
Average service time: 3.75
PeakHour:3
```

La ora 3, au fost 2 clienti in coada, si într-adevar, atunci a fost cel mai aglomerat in institutie

```
---------------------------------------
CurrentTime: 3
Waiting clients:(1,7,3); (2,7,4);
Queue 0: (3,1,2);
Queue 1: (4,3,4);
```

6. Concluzii

In concluzie, consider ca aceasta tema a adus un bonus in dezvoltarea modului de lucru in Java si mai ales in prelucrarea thread-urilor. A fost o tema grea, care la inceput a parut sa fie destul de usoara, dar din nou m-am lovit de unele neclaritati privind thread-uri si sincronizari,

Ca dezvoltari ulterioare, ar trebui sa mai modific aflarea timpului mediu de asteptare in coada deoarece nu cred ca se calculeaza corect. O alta dezvoltare ar fi sa implementez o alta strategie de introducere a clientului in coada, de exemplu, in functie de numarul minim de servire al cozii. Ca bonus in dezoltare ar fi si interfata grafica pentru a o face mai prietenoasa si mai usor de folosit.

7. Bibliografie

[1] https://stackoverflow.com/questions/
[2] https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html
[3] https://www.formdev.com/jformdesigner/doc/ides/eclipse/
[4] https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html