Universitatea Tehnică Cluj-Napoca

QUEUE SIMULATOR

~ Tema 2 ~

Stroe Mădălina Ionela

Grupa 302210

Semigrupa2

CUPRINS

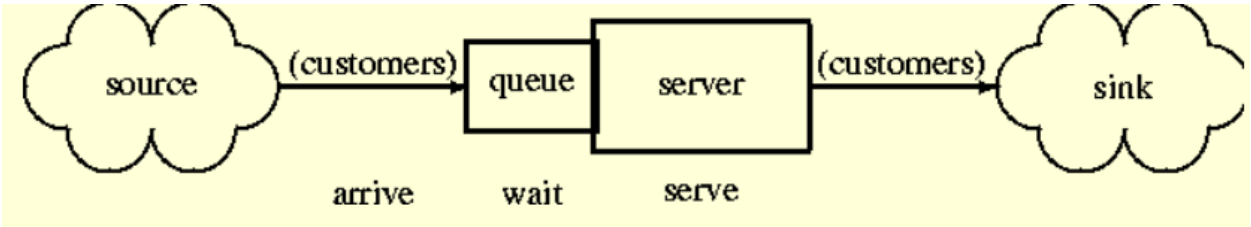
~ Introducere ~

Coadă = linie sau secvențe de itemi stocate pentru a fi accesate într-o anumită ordine, de obicei în ordinea inserării lor.

Teoria statului la coadă sau teoria așteptării este studiul matematic al liniilor de așteptare sau cozilor, care își are originea în studiile lui Agner Krarup Erlang, atunci când a creat modele pentru a descrie schimbarea telefoanelor în Copenhaga.

Obiectivul unei cozi este de a oferi un spațiu pentru un ”client” unde poate să aștepte înainte de a primi un ”serviciu”.

În management-ul unui sistem bazat pe cozi se dorește minimizarea timpului de așteptare în cozi înainte ca aceștia să fie serviți.

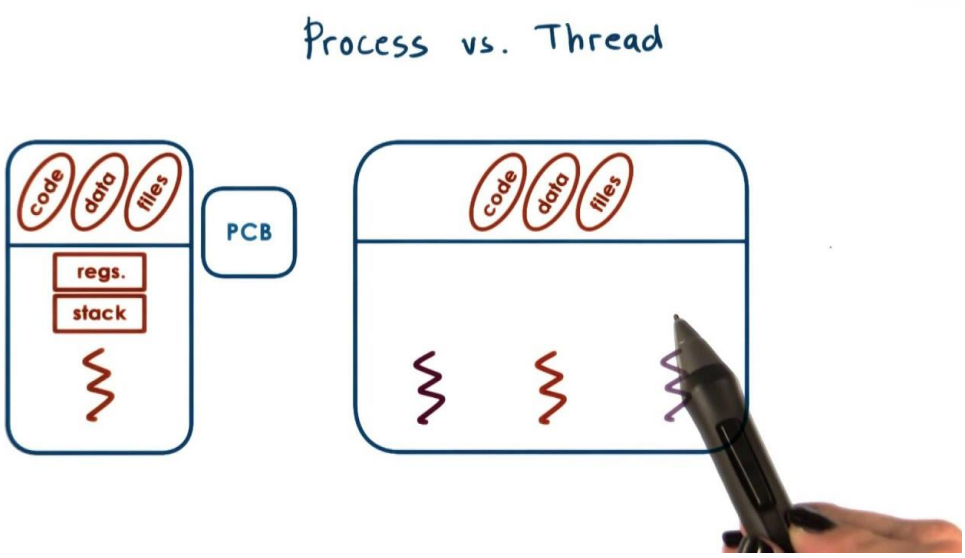


~ Obiectivul temei ~

Obiectivul acestei teme este de perfecționare a lucrului cu threaduri și aprofundare a cunoștințelor de programare orientate pe obiecte dobândite până acum, utilizare a AtomicInteger și, de asemenea, a listelor de tip ArrayBlockingQueue.

Conceptul de thread(fir de execuție) definește cea mai mica unitate de procesare ce poate fi programată spre execuție de care sistemul de operare. Este folosit în programare pentru a eficientiza execuția programelor, executând porțiuni distincte de cod în paralel în interiorul aceluiași proces.

Threadurile sunt diferite față de clasicele procese gestionte de sistemele de operare ce suportă multitasking, în principal prin faptul că, spre deosebire de procese, toate threadurile asociate unui proces folosesc același spațiu de adresare. Procesele sunt în general independente, în timp ce mai multe threaduri pot fi asociate unui unic proces. Procesele stochează un număr semnificativ de informații de stare, în timp ce threadurile dintr-un proces impart aceeași stare, memorie sau alte resurse. Procesele pot interacționa numai prin mecanisme de comunicare interproces speciale oferite de sistemul de operare (semnale, semafoare, cozi de mesaje și altele asemenea). Cum împart același spațiu de adresare, threadurile pot comunica prin modificarea unor variabile asociate procesului și se pot sincroniza prin mecanismele proprii. În general este mult mai simplu și rapid schimbul de informații intre threaduri decât între procese.



~ Analiza problemei ~

Cerințele problemei sunt de a implementa o aplicație de simulare a cozilor cu scopul de a determina și minimiza timpul de așteptare al clienților.

Considerăm că un client va avea un indice ca număr întreg, un timp de sosire și un timp de procesare.

Vom avea o listă cu toate cele un indice crescător ordonat de la 1 la n, iar celelalte 2 atribute vor fi generate random.

Acești clienți vor intra în cozile de procesare în funcție de timpul minim de așteptare, adică vor intra direct daca este coada goală, pentru ca timpul de așteptare al cozii este 0, altfel, dacă sunt clienți în cozi, vor intra în coada în care valoarea de procesare este minimă, valoarea fiind suma tuturor timpilor de procesare ai clienților aflați deja în coadă.

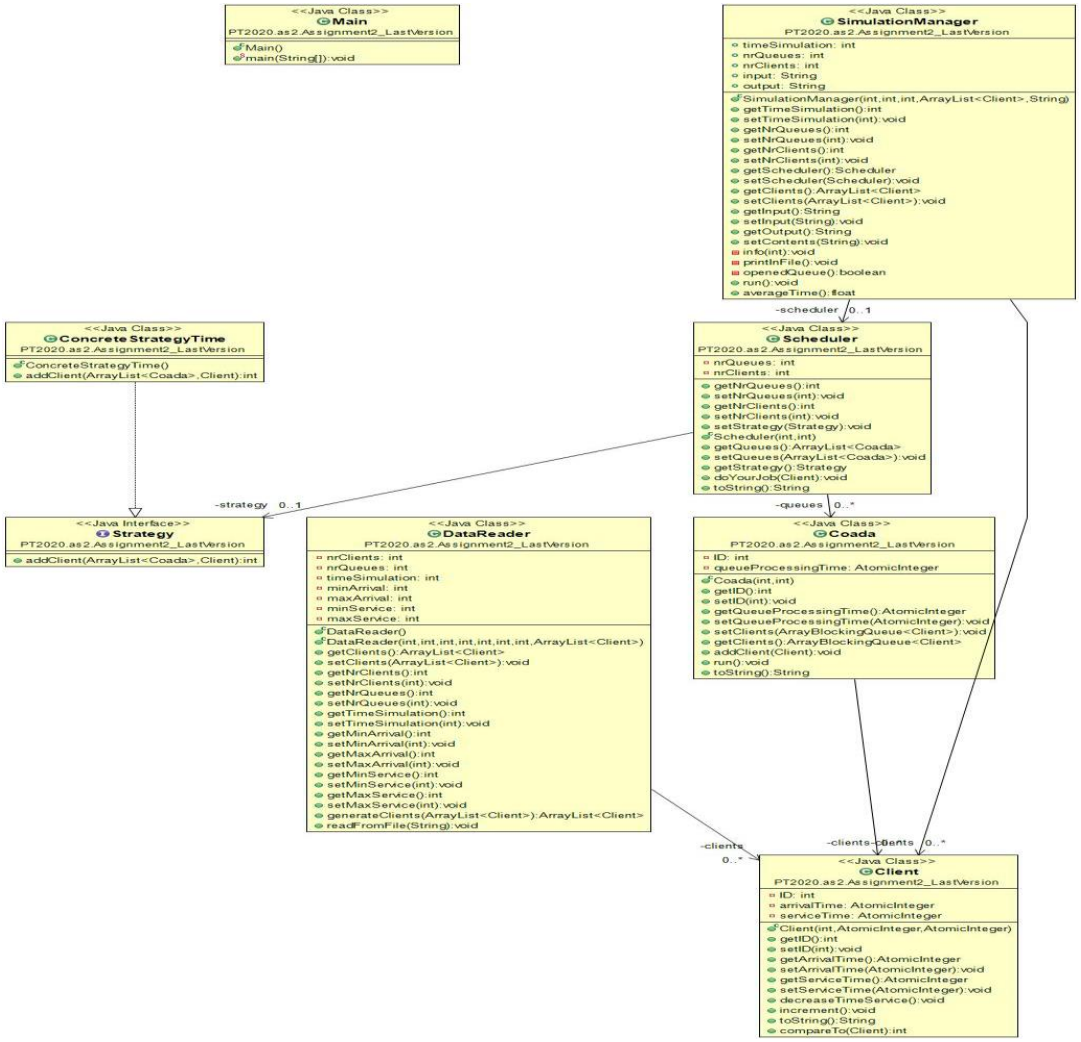
Cu cât timpul general al programului crește, cu atât timpul de procesare al clientului scade, până când acesta iese din coadă, adică atunci când timpul de procesare al clientului este 0.

Mai mult decât atât, la început cozile vor fi închise, urmând ca mai apoi să fie clienții procesați.

Când nu mai sunt clienți în coadă, aceasta se va închide iar și se va afișa un mesaj corespunzător: is Closed, spre exemplu.

Simularea va fi finalizată atunci când coada în care au fost clienții generați random este închisă **și** nu mai sunt clienți de procesat nici în celelalte cozi.

~ Proiectare ~



În figura de mai sus putem vedea relațiile dintre clase.

Unified Modeling Language (prescurtat UML) este un limbaj standard pentru descrierea de modele si specificatii software. Diagrama de clase UML Este folosită pentru reprezentarea vizuală a claselor și a interdependențelor, taxionomiei și a relațiilor de multiplicitate dintre ele. Diagramele de clasă sunt folosite și pentru reprezentarea concretă a unor instanțe de clasă, așadar obiecte și a legăturilor concrete dintre acestea.

De primirea datelor se va ocupa clasa DataReader, care citeste datele dintr-un fisier de intrare .txt.

Mai departe, după ce datele au fost citite ca string-uri, acestea vor fi separate și folosite ulterior ca tipul de care vom avea nevoie în program.

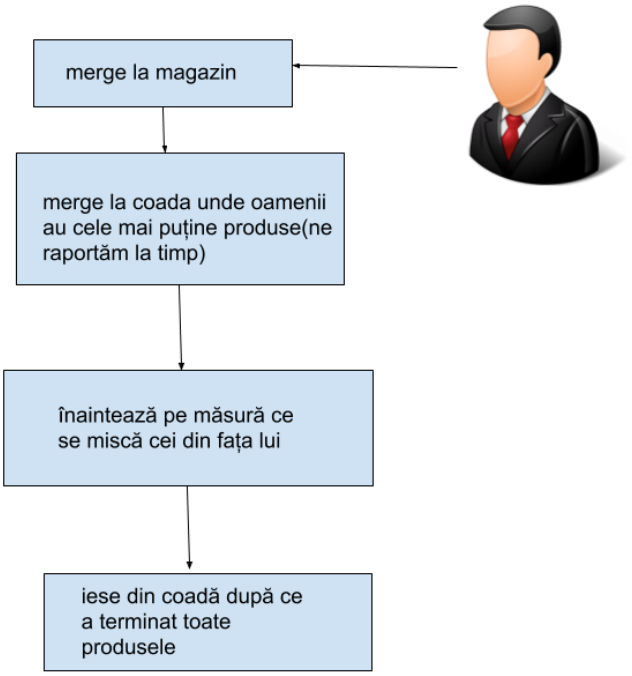
Mai întâi avem nevoie de numărul de clienti, de tip int. Urmează să generăm n clienți, folosindu-ne de metoda de generare a clienților.

Clasa Scheduler ne permite să adaugăm clientul într-o coadă, în funcție de strategia abordată, și anume, următorul client sosit va merge la coada cu cel mai mic timp de procesare, adică cel mai mic timp de servire al clienților.

Coada este definită de un ID, de o listă de clienți și un timp de procesare al cozii. Aici vom procesare clienții pe măsură ce va trece timpul.

Clasa SimulationManager se ocupă de verificarea cozilor și procesarea clienților în cazul în care nu sunt toate cozile închise, adică mai există cel puțin un client în oricare din cozi, fie cea a clienților generați random, fie cozile de procesare.

Pașii execuției sunt următorii:



~Implementare~

Începând de la cel mai “de jos” element al implementării programului, vor fi prezentate clasele pe rând.

Clasa Client are următoarele atribute: ID, care este numărul de identificare al clientului și nu este generat random, arrivalTime, care reprezintă timpul de sosire al clientului și serviceTime, care reprezintă timpul în care clientul va fi procesat.

Această clasă implementează interfața Comparable și ne permite să creăm metoda compareTo(Client c), care va fi folosită ulterior pentru a ordona crescător clienții în funcție de arrivalTime.

De asemenea, avem si metoda decreaseTimeService(), care va fi utilizată atunci când clientul de află în coada de procesare.

Clasa DataReader va fi folosită pentru a citi de la tastatură informații despre numărul de clienți, numărul de cozi, timpul de simulare, timpul minim de sosire, timpul maxim de sosire, timpul minim de procesare, timpul maxim de procesare. Mai întâi, liniile din fișierul de intrare vor fi citite ca string-uri, apoi vor fi transformate în integer.

Pentru liniile care au mai multe valori separate prin virgulă, am creat un vector de string-uri, apoi am folosit metoda Split() pentru a obține separat aceste elemente, transformandu-le și pe acestea în valori întregi pentru utilizările ulterioare.

Tot în clasa DataReader avem metoda generateClients, care este o metodă de generare random a unui arrivalTime și a unui serviceTime, pentru un număr de n clienți, cuprinse între minimul și maximul citite de la tastatură pentru fiecare dintre ele.

Clasa Coada are ca atribute un ID, o listă de tip ArrayBlockingQueue de clienți și un queueProcessingTime de tip AtomicInteger.

Într-o listă de tip ArrayBlockingQueue elementele sunt ordonate după principiul FIFO(first-in-first-out).

Clasa AtomicInteger oferă o variabila de tip int care poate fi citită și scrisă automat și care conține diferite metode precum, getAndIncrement, compareAndSet(),etc.

În clasa Coada, avem o metodă addClient(Client c) care permite adaugarea unui client în lista cozii, dupa acest lucru urmând ca timpul de procesare al cozii să crească cu exact valoarea timpului de procesare al clientului.

Despre metoda run() din această clasă vom vorbi în curând.

Interfața Strategy este implementată de clasa ConcreteStrategyTime. În această clasă avem metoda addClient, care funționează în felul următor. Avem o variabilă numită min, care de fapt e inițializată cu o valoare foarte mare, o altă variabilă index ce va lua o valoare mai mică decât 0.

Pentru fiecare coadă din array-ul de cozi dat ca parametru, vom verifica timpul de procesare, iar valoarea variabilei min va fi înlocuită cu cea mai mică valoarea după ce parcurgerea va fi finalizată. În același timp, în variabila index se va stoca indicele cozii. Funcția addClient va returna de fapt indicele cozii care are cel mai mic timp de procesare.

Mai departe, avem clasa Scheduler care ne permite să avem o listă de elemente de tip Coada. Strategia cu care va lucra Scheduler este dată, de fapt, de clasa ConcreteTimeStrategy, dorind a fi găsită coada cu cel mai mic timp de procesare unde să fie adăugat clientul.

Metoda doYourJob găsește indexul cozii, apoi vom adăuga clientul.

Constructorul acestei clase are ca parametrii numarul de cozi și numărul de clienți, valori pe care le vom obține dupa ce se va face citirea din fișier.

Tot aici, în lista de cozi, iterând un for de la 0 la numărul cozilor citit din fișierul de intrare, creăm câte o coadă pe care o adăugăm în lista de cozi a clasei Scheduler.

Clasa SimulationManager este cea care oferă mișcare întregului sistem. Aceasta are ca atribute 3 variabile întregi timeSimulation, nrQueues, nrClients, 2 variabile de tip String input și output, un

Scheduler si un ArrayList de Client. În constructorul acestei clase, Scheduler va fi inițializat cu un new Scheduler(numar cozi, numar clienți), citite din fișier.

Output-ul este inițializat cu un String. Output-ul este ceea ce se va afișa în fișier și va depinde de fișierul de intrare al cărui nume este citit de la utilizator.

Pentru a afișa în fișier și a evita suprascrierea și pierderea informațiilor anterioare, am creat o metodă pentru a concatena informația în funcție de momentul de timp la care ne aflăm cu procesarea.

Mai întâi, vom afișa timpul la care ne aflăm, apoi pe rândurile următoare, cozile și indicii acestora. Prima coadă afișată este coada ce păstrează clienții generați random, care așteaptă să fie procesați, coadă purtând un nume sugestiv.

În cazul în care această coadă este goală, vom concatena un mesaj de tipul ”is closed”.

Altfel, în cazul în care coada nu este goală, vom concatena clienții ce se află în coada de așteptare pentru a fi procesați.

Mai apoi, urmează să avem cozile de procesare, fiecare afișată pe o linie nouă, alături de indicii lor.

Verificăm pentru fiecare coadă dacă este goală, iar în cazul în care este, afișăm ”is closed”.

Altfel, concatenăm toți clienții care se află în acea coadă.

Metoda printInFile() se ocupă de printarea / afișarea informațiilor în fișier.

Metoda openedQueue() va returna o valoare booleană dacă:

1. Dacă în coada inițială mai sunt clienți, variabila waitingClients este setată la true.
2. Dacă în cel puțin una din cozile de procesare a clienților mai sunt clienți, variabila isOpened este setată la true.

Daca cel puțin una din aceste condiții este true, variabila result ce urmează să fie returnată va lua valoarea true.

Metoda averageTime returnează timpul mediu de așteptare al clienților.

In metoda Run setăm o inițializăm currentTime cu 0. Cât timp mai e cel puțin o coadă deschisă, timpul curent e mai mic decât timpul la care se termină simularea și arrivalTime-ul clientului care urmează să intre in coadă este egal cu timpul curent al procesării, adăugăm clientul în coada cu cel mai mic timp de procesare.

În cazul în care coada este goală, vom crea un thread pentru acea coadă, pe care îl vom porni ulterior.

După ce clientul a intrat în coada de procesare, este eliminat din coada de așteptare, apoi așteptăm.

Clienții aflați în coada de procesare sunt luați în ordinea în care au intrat. Thread-ul execuției va fi oprit pentru puțin timp. Pe măsură ce timpul simulării crește, timpul procesării clientului și al cozii scade.

Când timpul clientului ajunge la 0, acesta este scos din coadă, lăsând loc liber pentru următorul.

Clasa Main se ocupă de organizarea lucrurilor ce țin de input și output, de asemenea si execuția programului conform cerințelor.

Citim de la utilizator unul dintre cele 3 fișiere de input sub forma ”in-test-n”, unde n este 1, 2 sau 3 pentru a putea stabili care este fișierul de output, in funcție de datele de intrare.

~ Rezultate ~

Rezultatele sunt afișate în fisierele de output, cu extensia .txt.

Exemplu fișier de intrare:

in-test-1 - Notepad

File Edit Format View Help

```
4
2
60
2,30
2,4
```

Exemplu fișier de ieșire:

output2 - Notepad

File Edit Format View Help

```
Clients: (29 , 4 , 6) (33 , 4 , 3) (4 , 7 , 7) (32 , 7 , 7) (37 , 9 , 4) (30 , 10 , 1) (40 , 10 , 2) (15 , 11 ,
Queue 0 : is closed
Queue 1 : (9 , 2 , 3)
Queue 2 : (17 , 2 , 4)
Queue 3 : (19 , 3 , 1)
Queue 4 : (47 , 3 , 2)

Time    4
Clients: (4 , 7 , 7) (32 , 7 , 7) (37 , 9 , 4) (30 , 10 , 1) (40 , 10 , 2) (15 , 11 , 3) (27 , 11 , 2) (41 , 11
Queue 0 : (29 , 4 , 6)
Queue 1 : (9 , 2 , 2)
Queue 2 : (17 , 2 , 3)
Queue 3 : (33 , 4 , 3)
Queue 4 : (47 , 3 , 1)

Time    5
Clients: (4 , 7 , 7) (32 , 7 , 7) (37 , 9 , 4) (30 , 10 , 1) (40 , 10 , 2) (15 , 11 , 3) (27 , 11 , 2) (41 , 11
Queue 0 : (29 , 4 , 5)
```

Exemplu .jar

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18362.720]
(c) 2019 Microsoft Corporation. All rights reserved.

J:\PT2020\pt_302210_stroe_madalina_assignment2\Assignment2_LastVersion\src\main\java\PT2020\as2\Assignment2_LastVersion>java -jar file.jar
Introduceti numele fisierului de input:
in-test-1.txt
in-test-1.txt
Nr. clienti:4
Nr. cozi:2
Timp simulare:60
min. Arrival:2
max. Arrival:30
min. Service:2
max. Service:4
Ordonare crescatoare clienti dupa ArrivalTime: [(1 , 11 , 3) , (2 , 15 , 3) , (3 , 22 , 2) , (4 , 25 , 4) ]
Average Waiting Time: 3.0
Indicele cozii in care adaugam clientul: 0
Indicele cozii in care adaugam clientul: 0
Indicele cozii in care adaugam clientul: 0
Indicele cozii in care adaugam clientul: 0
Output

Time    0
Clients: (1 , 11 , 3) (2 , 15 , 3) (3 , 22 , 2) (4 , 25 , 4)
Queue 0: is closed
Queue 1: is closed

Time    1
Clients: (1 , 11 , 3) (2 , 15 , 3) (3 , 22 , 2) (4 , 25 , 4)
Queue 0: is closed
Queue 1: is closed

Time    2
Clients: (1 , 11 , 3) (2 , 15 , 3) (3 , 22 , 2) (4 , 25 , 4)
Queue 0: is closed
Queue 1: is closed

Time    3
Clients: (1 , 11 , 3) (2 , 15 , 3) (3 , 22 , 2) (4 , 25 , 4)
Queue 0: is closed
Queue 1: is closed

Time    4
Clients: (1 , 11 , 3) (2 , 15 , 3) (3 , 22 , 2) (4 , 25 , 4)
Queue 0: is closed
Queue 1: is closed

Time    5
```

~ Concluzii ~

Odată cu rezolvarea acestei teme, am avut ocazia de a învăța să lucrez cu Threaduri, ArrayBlockingQueues și AtomicInteger.

De această dată, lucrul cu threadurile a fost aprofundat, înțelegând mult mai bine modul de funcționare și domeniile de aplicabilitate.

Am aprofundat, de asemenea, cunoștintele anterioare de programare orientata pe obiecte.

~ Bibliografie ~

https://www.journaldev.com/1020/thread-sleep-java#java-thread-sleep-important-points

http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdfhttps://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html

https://www.geeksforgeeks.org/arrayblockingqueue-class-in-java/

https://www.geeksforgeeks.org/split-string-java-examples/

https://www.edx.org/course/queuing-theory-from-markov-chains-to-multi-server

și altele