Laboratorul 5

Fire de execuţie în *Java SE* – Pachetul *java.util.concurent* - Partea 1 -

1. Obiectivele laboratorului

- prezentarea cuvântului rezervat *volatile;*
- prezentarea, înţelegerea şi utilizarea următoarelor aspecte ale pachetului *java.util.concurrent:*
  - colecţiile *thread-safe,*
  - obiecte de tip *Lock, ReentrantLock,*
  - clasa *Semaphore*.

2. Consideraţii teoretice. Exemple

Pachetul *java.util.concurrent* a fost introdus în versiunea *JDK 5.0.*

2.1. Cuvântul rezervat volatile

Cuvântul cheie *volatile* se foloseşte pentru a arăta că o variabilă poate fi modificată de mai multe fire. Această variabilă se va afla în memoria principală (eng. *main memory*). Folosind cuvântul cheie *volatile* se reduce riscul erorilor de consistenţă a memoriei: dacă o variabilă *volatile* se modifică, această modificare este vizibilă pentru toate firele. Folosirea acestor variabile atomice este mai eficientă decât accesarea variabilelor prin metode sincronizate, dar necesită mai multă atenţie din partea programatorului pentru evitarea erorilor de consistenţă a memoriei [MAN, 2008].

Exemplu:

```
volatile int primitiv; //declararea atributelor volatile
volatile String referinta;
```

2.2. Colecţii thread-safe

O secvenţă de cod este *thread-safe* dacă lucrează cu date comune într-o manieră ce poate garanta siguranţa datelor atunci când aceasta este accesată de mai multe fire de execuţie.

Ierarhia de clase (eng. *framework*) care implementează interfaţa *Collection* (introdusă în *JDK 1.2*) este foarte flexibilă, beneficiind de cele trei modele (interfeţe) de bază *List, Set* şi *Map*. Unele dintre clasele *framework*-ului sunt deja *thread-safe* (*Hashtable* şi *Vector*), iar pentru altele există modalităţi de încapsulare (eng. *wrappers*).

Exemplu:

```
//colectii nesigure
List unsafeList = new ArrayList();
Set unsafeSet = new HashSet();
Map unsafeMap = new HashMap();

//colectii sigure
List threadSafeList = Collections.synchronizedList(new ArrayList());
Set threadSafeSet = Collections.synchronizedSet(new HashSet());
Map theadSafeMap = Collections.synchronizedMap(new HashMap());
```

Aşadar, în cazul în care o colecţie poate fi accesată de mai multe fire de execuţie, este nevoie ca aceasta să fie *thread-safe,* fie prin încapsularea acesteia prin intermediul unei clase

Factory ce implementează mecanisme de sincronizare, fie prin alegerea unei implementări sigure (care este deja *thread-safe*).

Toţi iteratorii colecţiilor din pachetul *java.util* eşuează dacă lista este modificată (se adaugă sau se şterg elemente) în timpul iterării, aruncând o excepţie de tipul *ConcurrentModificationException*.

Această problemă a fost rezolvată (pentru liste şi seturi) prin introducerea a două noi implementări: *CopyOnWriteArrayList* şi *CopyOnWriteArraySet* (din JDK 5.0). Aceste clase creează o copie a colecţiilor (listă, respectiv set) în curs de iterare, de fiecare dată când se adaugă sau se şterge un element.

Referitor la colecţiile de tip *Map,* s-a introdus clasa *ConcurrentHashMap,* cu performanţe mult mai bune decât variantele mai vechi (*Hashtable* sau *Collections.synchronizedMap(new HashMap())*). *ConcurrentHashMap* permite executarea citirilor, scrierilor şi ştergerilor simultan fără a pune zăvor pe colecţie (spre deosebire variantele sigure anterioare).

Iteratorii introduşi în pachetul *java.util.concurrent* nu asigură consistenţa datelor în timpul iterării, în sensul că s-ar putea să nu "remarce" obiecte şterse sau adăugate (în timpul iterării), dar acest lucru este de preferat eşecului (aruncarea excepţiei *ConcurrentModificationException*). Oricum, în general, parcurgerile unei colecţii sunt mult mai frecvente decât adăugările şi ştergerile elementelor [GOE, 2008].

Cozile neblocante (descrise de interfaţa *Queue*) sunt implementate în *Java* prin:

- *PriorityQueue* – nu este *thread-safe.*
- *ConcurrentLinkedQueue* – este *thread-safe,* rapidă, FIFO.

Cozile blocante (descrise de interfaţa *BlockingQueue*) determină un fir de execuţie să scrie într-o listă plină sau să citească dintr-o listă goală (spre deosebire de cele neblocante). Implementările acestei categorii de liste sunt (toate *thread-safe*):

- *LinkedBlockingQueue,*
- *PriorityBlockingQueue,*
- *ArrayBlockingQueue,*
- *SynchronousQueue,*
- *DelayQueue.*

2.3. Sincronizarea de tip zăvor (Lock)

Obiectele de tip zăvor se obţin prin implementarea interfeţei *Lock* şi lucrează într-o manieră similară cu monitoarele implicite folosite de către blocurile sincronizate (*synchronized*). La fel ca şi în cadrul monitoarelor implicite, doar un singur fir poate achiziţiona şi deţine un zăvor la un moment dat. Acestea suportă, de asemenea, metodele *wait()* şi *notify(), notifyAll().*

Unul dintre principalele avantaje a obiectelor *Lock* este posibilitatea de a verifica disponibilitatea unui monitor înainte de a încerca achiziţionarea acestuia, prin apelul metodei *tryLock()*. În cazul în care zăvorul este luat, se poate executa o logică alternativă.

Această metodă are şi o formă cu parametri *tryLock(long time, TimeUnit unit)* prin intermediul căreia se poate preciza timpul maxim de aşteptare pentru eliberarea zăvorului. Dacă firul este întrerupt în timp ce încearcă achiziţionarea zăvorului, se aruncă o excepţie de tipul *InterruptedException* care trebuie tratată.

Metoda *lockInterruptibly()* achiziţionează zăvorul, mai puţin în cazul în care firul este întrerupt, în acest caz aruncă o excepţie de tipul *InterruptedException* [ORA, 2011].

Metoda *newCondition()* returnează un obiect de tip *Condition* asociat zăvorului. Prin apelul metodelor *await()* şi *signal()* pentru obiectul de tip *Condition* se poate implementa un mecanism de tipul *wait/notify* [ORA, 2012].

Principalele diferenţe dintre *Locks* şi blocurile sincronizate:

- blocurile sincronizate nu garantează că secvenţele în care firele aşteaptă să intre în bloc vor primi accesul;
- nu se pot pune parametrii la intrarea în blocurile sincronizate, accesul nu poate fi monitorizat;
- neavând suportul programării pe bloc (ca şi în cazul *synchronized*), responsabilitatea apelării metodei *unlock()* este în sarcina programatorului.

Clasa *ReentrantLock* reprezintă una dintre implementările interfeţei *Lock* şi pe lângă metodele din această interfaţă merită amintite:

- *getOwner()* - returnează referinţa firului care deţine zăvorul sau *null*;
- *getQueuedThreads()* - returnează o colecţie de fire care probabil aşteaptă zăvorul;
- *getQueueLength()* - returnează numărul estimat al firelor care aşteaptă achiziţionarea zăvorului;
- *getWaitingThreads(Condition condition)* - returnează o colecţie conţinând acele fire care aşteaptă îndeplinirea condiţiei *Condition* asociată cu acest zăvor;
- *getWaitQueueLength(Condition condition)* - returnează un număr estimativ al firelor care poate aşteaptă îndeplinirea condiţiei *Condition* asociată cu acest zăvor;
- *hasWaiters(),* returnează o variabilă *true (false)* dacă există (sau nu) cel puţin un fir care aşteaptă achiziţionarea zăvorului.

Exemplu:

Cerinţe: Să se implementeze o aplicaţie simplă care să exemplifice funcţionalitatea de bază a zăvoarelor (*Lock*). Modul de funcţionare al mecanismului de sincronizare de tip zăvor este prezentat în diagrama secvenţială din Figura 5.1.

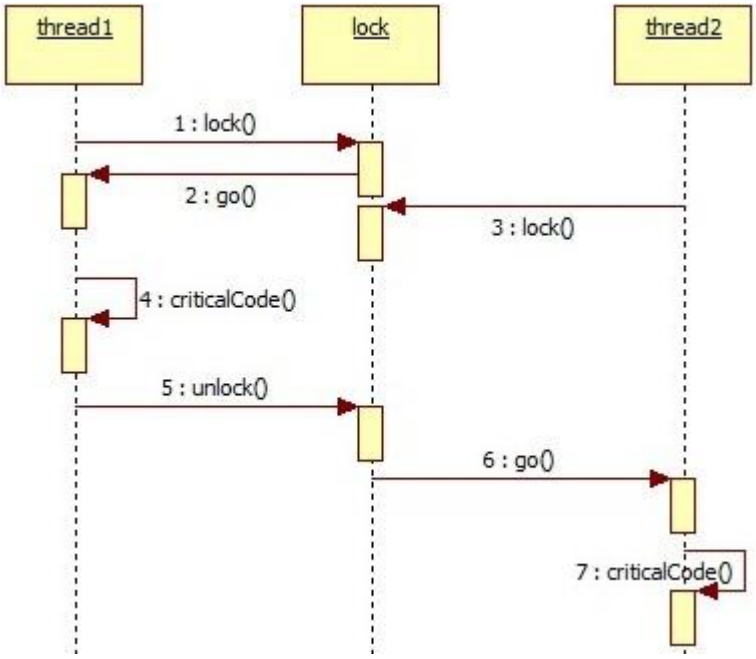

Figura 5.1 Diagrama secvenţială pentru exemplul cu zăvoare

Merită menţionat faptul că în realitate, obiectul de tip *Lock* nu apelează nicio metodă *go()*. Singurul scop pentru care această metodă apare în aplicaţie este acela de a prezenta mecanismul într-un mod cât mai intuitiv.

Implementarea aplicaţiei este prezentată în Secvenţa de cod 1.

Secvenţa de cod 1: Exemplu de implementare a excluderii mutuale prin zăvoare

```
class Fir extends Thread {
	int nume;
	Lock l;

	Fir(int n, Lock la) {
		this.nume = n;
		this.l = la;
	}

	public void run() {
		this.l.lock();
		System.out.println("Fir " + nume + " a pus zavorul");
		regiuneCritica();
		this.l.unlock();
		System.out.println("Fir " + nume + " a eliberat zavorul");
	}

	public void regiuneCritica() {
		System.out.println("SE EXECUTA REGIUNEA CRITICA!");
		try {
			Thread.sleep(3000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}

public class Main {
	public static void main(String args[]) {
		Lock l = new ReentrantLock();
		Fir f1, f2;
		f1 = new Fir(1, l);
		f2 = new Fir(2, l);
		f1.start();
		f2.start();
	}
}
```

2.4. Sincronizarea de tip semafor (Semaphore)

În esenţa, un semafor este un număr întreg care poate fi incrementat, respectiv decrementat de către două sau mai multe procese prin intermediul unor funcţii speciale. Aceste funcţii asigură de asemenea şi blocarea sau deblocarea proceselor în momentul în care valoarea semaforului atinge o anumită limită.

Clasa *Semaphore* din *Java SE* posedă două metode de bază: *acquire()* şi *release()*.

Metoda *acquire(int permits)* blochează firul de execuţie care o apelează până în momentul în care valoare internă a semaforului este cel puţin egală cu numărul specificat ca parametru al metodei.

Metoda *release(int permits)* incrementează valoarea internă a semaforului cu o valoare egală cu numărul specificat ca parametru al metodei.

Metodele *aquire()* şi *release()* sunt supraîncărcate astfel încât să nu aibă niciun parametru. Folosind această variantă a metodelor şi iniţializând semaforul cu valoarea 1, se obţine un mecanism similar zăvoarelor (semafor binar).

Constructorul acestei clase este supraîncărcat (*Semaphore(int permits)* şi *Semaphore(int permits, boolean fair)*). Când parametrul *fair* este *true*, semaforul garantează că firele de execuţie blocate primesc acces în ordinea invocării metodei *acquire()*, după modelul *FIFO*. În general, semafoarele sunt folosite în controlul accesului la resurse pentru a nu se ajunge la înfometarea firelor de execuţie [ORA, 2010].

Exemplu:

Cerinţe: Să se implementeze o aplicaţie *Java SE* care modelează reţeaua *Petri* din Figura 5.2.

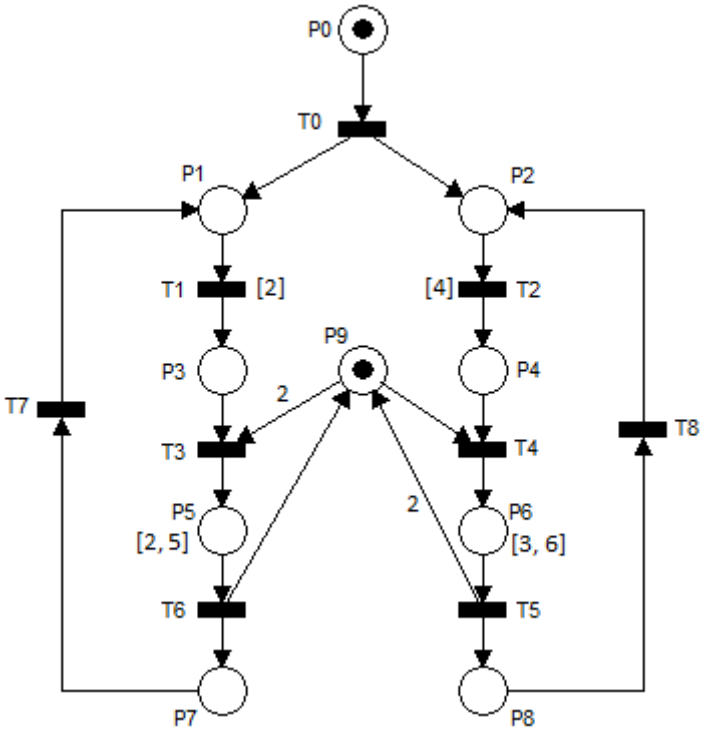

Figura 5.2 Exemplu de aplicaţie pentru semafoare

Specificaţii:

Aplicaţia va fi implementată în *Java SE*, folosind un semafor pentru elementul de sincronizare din locaţia *P9*.

Temporizările locaţiilor *P5* şi *P6* corespund unor activităţi de forma:

```
int k = ...;//nr. aleator în
            //intervalul specificat
for (int i = 0; i < k * 100000; i++) {
      i++;
      i--;
}
```

Tranziţiile temporizate *T1* şi *T2* reprezintă întârzieri şi vor fi implementate prin apelul metodei *Thread.sleep(x)*.

Implementarea aplicaţiei este prezentată în Secvenţa de cod 2.

Secvenţa de cod 2: Exemplu de implementare a excluderii mutuale folosind semafoare

```
class Fir extends Thread {
      int nume, intarziere, k, permise;
      Semaphore s;

      Fir(int n, Semaphore sa, int intarziere, int k, int permise) {
            this.nume = n;
            this.s = sa;
            this.intarziere = intarziere;
            this.k = k;
            this.permise = permise;
      }

      public void run() {
            while (true) {
                  try {
                        Thread.sleep(this.intarziere * 500);
                        this.s.acquire(this.permise); // regiune critica
                        System.out.println("Fir " + nume
                                    + " a luat un jeton din semafor");
                        for (int i = 0; i < k * 100000; i++) {
                              i++;
                              i--;
                        }
                        this.s.release(); // sfarsit regiune critica
                        System.out.println("Fir " + nume
                                    + " a eliberat un jeton din semafor");
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
      }
}

public class Main {
    public static void main(String args[]) {
        Semaphore s = new Semaphore(1);
        Fir f1, f2;
        f1 = new Fir(1, s, 2, (int) Math.round(Math.random() * 3 + 2), 2);
        f2 = new Fir(2, s, 4, (int) Math.round(Math.random() * 3 + 3), 1);
        f1.start();
        f2.start();
    }
}
```

3. Dezvoltări şi teste

3.1. Aplicaţia 1

Se vor testa şi se vor înţelege aplicaţiile demonstrative puse la dispoziţie în timpul orei de laborator.

3.2. Aplicaţia 2

Pentru exemplul de aplicaţie cu semafoare (subcapitolul 2.4) se vor întocmi: diagrama claselor şi diagrama secvenţială.

3.3. Aplicaţia 3

Enunţul problemei:

Să se implementeze aplicaţia *producător-consumator,* folosind o colecţie care nu este *thread-safe* (de exemplu *ArrayList*).

Specificaţii:

Aplicaţia va fi compusă dintr-un fir *producător* şi trei fire *consumator*. Firul *producător* va genera numere aleatoare pe care le va depozita într-o colecţie care nu este *thread-safe*, cu o perioadă de o secundă. Firele *consumator* vor extrage aceste numere şi le vor afişa în consolă (împreună cu numele firului curent).

Aplicaţia va garanta accesul concurent la colecţie prin utilizarea sincronizărilor de tip zăvor.

Se va garanta buna funcţionare a aplicaţiei chiar şi în cazul în care colecţia este goală.

Se va limita capacitatea colecţiei la 100 de elemente.

Cerinţe:

Proiectarea aplicaţiei se va realiza cu ajutorul reţelelor *Petri* şi a diagramei claselor.

Aplicaţia va fi Implementată în *Java SE*, folosind zăvoare şi obiecte de tip *Condition*.

Testare:

Aplicaţia va fi implementată astfel încât să se poată testa sincronizările.

Se va implementa un mecanism de blocare (la cerere) a firelor *consumator*, astfel încât condiţia privind capacitatea colecţiei să poată fi testată.

3.4. Aplicaţia 4

Enunţul problemei:

Să se implementeze un simulator simplu pentru controlul traficului rutier dintr-o intersecţie ca cea din Figura 5.3.

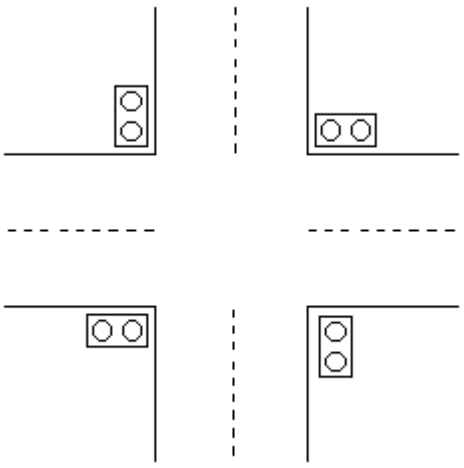

Figura 5.3 Aplicaţia 4

Specificaţii:

Intersecţia controlată are o singură bandă pe sens, iar la un moment dat, un singur semafor indică verde.

Simulatorul va fi compus din 5 fire de execuţie: un fir responsabil cu generarea maşinilor pentru cele 4 cozi de aşteptare (numărul de maşini generat va fi unul rezonabil). Celelalte 4 fire de execuţie vor încerca achiziţionarea semaforului şi vor permite extragerea a maxim 10 maşini din cozile de aşteptare.

Cele 4 semafoare vor fi implementate prin utilizarea unui singur obiect de tip *Semaphore,* astfel:

- în momentul în care apar cel puţin două maşini în coada de aşteptare se va încerca rechiziţionarea tuturor celor 10 permisiuni ale semaforului;
- semaforul va garanta accesul firelor de execuţie în ordine *FIFO*;
- timpul necesar unei maşini pentru a părăsi intersecţia este de o secundă; aşadar, semaforul va fi rechiziţionat de către un fir de execuţie pentru un număr de secunde egal cu numărul de maşini din coada de aşteptare (dar maxim 10 *sec*.).

Cerinţe:

Aplicaţia va fi modelată prin diagrama claselor şi prin maşini de stare.

Implementarea aplicaţiei se va realiza în *Java SE*. Cozile de aşteptare vor fi modelate prin simple valori numerice (accesul sincronizat trebuie garantat). Ratele de generare a maşinilor în cozile de aşteptare vor fi configurate într-un fişier text. Nu este obligatoriu ca aplicaţia să aibă interfaţă grafică. Totuşi, prezenţa interfeţei grafice va fi bonificată.

Testare:

În cazul în care aplicaţia nu dispune de interfaţă grafică, se vor depune eforturi suplimentare la proiectare / implementare pentru demonstrarea funcţionalităţii acesteia.

Se vor testa funcţional logica celor două tipuri de fire de execuţie.

Se vor testa sincronizările dintre firele de execuţie.

Se va testa accesul la fişierul text de configurare.

4. Verificare cunoştinţelor

1) Explicaţi care este scopul cuvântului rezervat *volatile.*

2) Ce înţelegeţi prin sintagma: colecţii *thread-safe?*

3) Care sunt diferenţele principale dintre implementarea excluderii mutuale prin blocuri sincronizate şi prin zăvoare?

4) Care sunt diferenţele principale dintre implementarea excluderii mutuale prin zăvoare şi prin semafoare?

5) Explicaţi cum aţi implementa un mecanism similar cu *wait/notify* folosind semafoare? Care ar fi avantajele unei astfel de implementări?