# CAPITOLUL 5

# 5. Realizarea problemei producatorului și a consumatorului

# 5.1 Semafoare

**Semaphore** - implementează un semafor clasic, care are un număr dat de permisii ce pot fi cerute și eliberate. Acesta este folosit pentru a restricționa numărul de thread-uri ce pot avea simultan acces concurent la o resursă.

Următoarea metodă de sincronizare a firelor este utilizând Semaphore. Folosind acest construct numit semafor, accesul la o resursă partajată este controlat printr-un contor. Semnalele sunt trimise între fire, astfel încât să putem proteja secțiunea critică și, de asemenea, să evităm semnalele ratate.

Un semafor poate fi definit ca o variabilă care este utilizată pentru a gestiona procese simultane prin sincronizarea acestor procese. Semaforele sunt, de asemenea, utilizate pentru a sincroniza accesul la resursa partajată și, astfel, a evita o condiție de rasă. Permisiunea acordată unui thread pentru accesarea resurselor partajate prin semafor se numește și permis.

**În funcție de funcțiile pe care le îndeplinesc, semaforele pot fi împărțite în două tipuri:**

**# 1) Semafor binar:** Un semafor binar este utilizat pentru sincronizarea proceselor concurente și implementarea excluderii reciproce. Un semafor binar presupune doar două valori, adică 0 și 1.

**# 2) Numărarea semaforului:** Semaforul de numărare are o valoare care indică numărul de procese care pot intra în secțiunea critică. În orice moment, valoarea indică numărul maxim de procese care intră în secțiunea critică.

***Deci, cum funcționează un semafor?***

**Funcționarea unui semafor poate fi rezumată în următorii pași:**

1. Dacă numărul de semafor este> 0, înseamnă că firul are permisiunea de a accesa secțiunea critică, iar apoi numărul este decrementat.
2. În caz contrar, firul este blocat până la obținerea permisului.
3. Când firul se termină cu accesarea resursei partajate, permisul este eliberat și numărul semaforului este incrementat, astfel încât un alt fir să poată repeta pașii de mai sus și să obțină permisul.

În Java, nu trebuie să ne implementăm semaforul, dar oferă un **Semafor** clasă care implementează funcționalitatea semaforului. Clasa Semaphore face parte din **java.util.concurrent** pachet.

**Clasa Semaphore oferă următorii constructori cu ajutorul cărora putem crea obiect semafor:**

*Semaphore (int num\_value) Semaphore (int num\_value, boolean how)*

**Aici,**

*num\_value* => valoarea inițială a numărului de permise care determină numărul de fire care pot accesa resursa partajată.

*cum =>* setează ordinea în care firele vor fi acordate permise (cum = adevărat). Dacă cum = fals, atunci nu se urmează o astfel de ordine.

Pentru a obține permisiunea de la un semafor, trebuie să apelați metoda acquire(), care are două forme:

 1. void acquire() throws InterruptedException

2. void acquire(int permits) throws InterruptedВxception

Pentru un singur permis se aplică prima opțiune, iar pentru mai multe permise, a doua opțiune. După apelarea acestei metode, până când firul primește permisiunea, se blochează. După ce ați terminat de lucrat cu resursa, permisiunea obținută anterior trebuie eliberată folosind metoda release():

 1. void release()

2 . void release(int permits)

Prima variantă a metodei eliberează un permis, iar a doua variantă eliberează numărul de autorizații specificat în autorizații.

**Exemplu de utilizarea clasei Semafor**

import java.util.concurrent.Semaphore;

 public class CountSem extends Thread {

 static volatile int n = 0;

 static Semaphore s = new Semaphore(1);

 public int getN() {

 return n;

 }

 public void run() {

 int temp;

 for (int i=0; i<10000000; i++) {

 try {

 s.acquire();

 temp = n;

 n = temp+1;

 }

 catch (InterruptedException e) {}

 finally { s.release(); }

 }}}

import java.io.\*;

public class MainCountSem {

 public static void main(String[] args) {

 CountSem p = new CountSem();

 CountSem q = new CountSem();

 p.start();

 q.start();

 try {

 p.join();

 q.join();;

 }

 catch (InterruptedException e) {}

 System.out.println("Final counter: "+p.getN());

 }

}

# 5.2 Sincronizarea prin barieră

Sincronizarea prin barieră presupune existența mai multor procese care execută aceeași secvență de cod care conține o instrucțiune cu rol de barieră. Poate fi vazută ca un set de thread-uri care se așteaptă reciproc să ajungă la un anumit punct comun, o barieră, după care se deblochează și își continuă execuția. Clasa **CyclicBarrier** permite programarea simplă a acestui tip de sincronizare. Se numește ciclic deoarece poate fi reutilizată după ce s-a ajuns la punctul dorit. Constructorul clasei poate avea două forme: **CyclicBarrier(int num)**, **CyclicBarrier(int num, Runnable barierAction)** în care num reprezintă numărul firelor de execuție, iar la atingerea acestui prag se execută metoda run() din obiectul barrierAction transmis.

Cele mai importante dintre metodele acestei clase sunt:

• **int await()** – suspendă thread-ul până la atingerea pragului corespunzător obiectului **CyclicBarrier**

• **int getNumberWaiting()** – returnează numărul de threaduri suspendate la barieră

• **int getParties()** – returnează valoarea parametrului num

• **void reset()** – reinițializează bariera

Un exemplu de utilizare al clasei **CyclicBarrier** :

public class TechLead extends Thread{

CyclicBarrier cyclicBarrier;

public TechLead(CyclicBarrier cyclicBarrier, String name)

{super(name);

this.cyclicBarrier= cyclicBarrier; }

public void run()

{

try {

Thread.sleep(3000);

System.out.println(Thread.currentThread().getName()+ " recruited developer");

System.out.println(Thread.currentThread().getName()+ " waiting to complete ...");

cyclicBarrier.await();

System.out.println("All finished recruting, " + Thread.currentThread().getName() + " gives offer letter to candidate");

} catch (Exception e) {

e.printStackTrace();} }}

public class HRManager {

public static void main(String[] args) {

 CyclicBarrier cyclicBarrier = new CyclicBarrier(3);

 TechLead techLead1 = new TechLead(cyclicBarrier,"John TL");

 TechLead techLead2 = new TechLead(cyclicBarrier,"Doe TL");

 TechLead techLead3 = new TechLead(cyclicBarrier,"Mark TL");

 techLead1.start();

 techLead2.start();

 techLead3.start();

 System.out.println("No work");}}

## Lucrarea de laborator 5

* 1. **Tema lucrării:**

Problema producătorului și consumatorului.

* 1. **Scopul lucrării:**
* Însuşirea modalităţilor şi a mecanizmelor de sincronizare;
* Însuşirea modalităţilor de realizare a mecanizmelor de sincronizare prin clase;

**3. Etapele de realizare:**

* sincronizarea firelor de execuţie la realizarea problemei;
* realizarea metodelor de sincronizare a firelor de execuţie prin clase cpeciale de sincronizare;
* prezentarea lucrării.
	1. **Exemplu de realizare:**

import java.util.LinkedList;

import java.util.concurrent.Semaphore;

import java.util.concurrent.CyclicBarrier;

 class Stock {

 private LinkedList<Character> stock = new LinkedList<>();

 Semaphore producer\_sem = new Semaphore(1, true);

 Semaphore consumer\_sem = new Semaphore(1, true);

 boolean response = false;

 public boolean store (char n, String name) {

 try {

 producer\_sem.acquire();

 } catch (InterruptedException e){

 e.printStackTrace();

 }

 if (response) {

 producer\_sem.release();

 return response;

 }

 if (stock.size() <= 8) {

 stock.add(n);

 producer\_sem.release();

 } else if (stock.size() == 9) {

 response = true;

 System.out.println("Stock is full");

 producer\_sem.release();

 } else {

 System.out.println("Error");

 producer\_sem.release();

 }

 return response;

 }

 public Character load (String name) {

 Character character = null;

 try {

 consumer\_sem.acquire();

 } catch (InterruptedException e){

 e.printStackTrace();

 }

 if ((response)) {

 if (stock.isEmpty()){

 response=false;

 System.out.println("Stock is empty");

 consumer\_sem.release();

 return character;

 }

 character = stock.getFirst();

 stock.removeFirst();

 }

 consumer\_sem.release();

 return character;

 }

 }

class Consumer extends Thread {

 CyclicBarrier barr;

 private Stock data;

 private String name;

 static int max = 0;

 public Consumer(Stock data, CyclicBarrier barr, Strin name) {

 this.data = data;

 this.name = name;

 this.barr = barr;

 }

 public void run() {

 Character tmp=null;

 int cons = 0;

 while(true) {

 try {

 if (max < 54) {

 tmp = data.load(name);

 if (tmp == null) {

 barr.await();

 } else {

 System.out.println("" + name + " consumed " + tmp);

 cons++;

 max++;

 System.out.println("Consumed: " + max);

 sleep((int) (Math.random() \* 1000));

 }

 } else {

 tmp = data.load(name);

 sleep(2000)

System.out.println(name+" consumed "+cons+" elements");

 break;

 }

 } catch (Exception e) {

 e.printStackTrace();

 }

 }

 }

}

class Producer extends Thread {

 private CyclicBarrier barr;

 private Stock data;

 private String name;

private char[] dep = new char []{'a','e','i','o','u'};

 static int max = 0;

public Producer(Stock data, CyclicBarrier barr, String name) {

 this.data = data;

 this.name = name;

 this.barr = barr;

 }

 public void run() {

 int tmp;

 boolean response;

 int prod = 0;

 while(true){

 try {

 if(max < 54) {

 tmp = (int) (Math.random() \* 5);

 response = data.store(dep[tmp], name);

 if(response) {

 barr.await();

 }

 else {

 System.out.println(name + " produced " + dep[tmp]);

 prod++;

 max++;

 sleep((int) (Math.random() \* 1000));

 }

 }

 else {

 tmp = (int) (Math.random() \* 5);

 response = data.store(dep[tmp], name);

 if(response) {

 barr.await();

 }

 sleep(2000);

System.out.println(name+" produced "+prod+" elements");

 break;

 }

 }

 catch (Exception e) {

 e.printStackTrace();

 }

 }

 }

}

public class Lab3 {

 public static void main (String[] args) {

 Stock data = new Stock();

 CyclicBarrier bar = new CyclicBarrier(6);

 Producer p1 = new Producer(data, bar, "Producer 1");

 Producer p2 = new Producer(data, bar, "Producer 2");

 Producer p3 = new Producer(data, bar, "Producer 3");

 Consumer c1 = new Consumer(data, bar, "Consumer 1");

 Consumer c2 = new Consumer(data, bar, "Consumer 2");

 Consumer c3 = new Consumer(data, bar, "Consumer 3");

 p1.start();

 p2.start();

 p3.start();

 c1.start();

 c2.start();

 c3.start();

 }

}

* 1. **Probleme propuse spre realizare:**

X producători generează aleatoriu F obiecte care sunt consumate de Y consumatori. De afişat informaţia despre producerea şi consumarea obiectelor, mesajele despre cazurile când “depozitul e gol sau plin”. Toate operaţiile se efectuează până când producătorii nu vor produce Z obiecte. Producătorii umplă depozitul și pînă cînd consumatorii nu vor elibera depozetul producătorii nu mai produc.

Fiecare producător produce câte 1 obiecte de fiecare dată și fiecare consumator consumă câte 1 obiecte de fiecare dată.

Sincronizarea producătorilor și a consumatorilor de realizat prin clasele semafoare, bariere, lock.

Dimensiunea depozitului este D. Valorile pentru X, Y, Z, D sunt indicate în Tabelul 3.

**Tabelul 3 Variantele pentru realizarea sarcinii**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Nr.** | **X** | **Y** | **Z** | **D** | **Tip Obiecte** |
| **1** | 3 | 3 | 48 | 8 | Numere pare |
| **2** | 2 | 4 | 50 | 5 | Numere impare |
| **3** | 4 | 2 | 45 | 9 | Vocale |
| **4** | 2 | 2 | 49 | 7 | Consoane |
| **5** | 3 | 5 | 33 | 3 | Numere pare |
| **6** | 4 | 3 | 35 | 7 | Numere impare |
| **7** | 5 | 3 | 54 | 9 | Vocale |
| **8** | 5 | 4 | 50 | 5 | Consoane |
| **9** | 5 | 2 | 44 | 4 | Numere pare |
| **10** | 3 | 3 | 40 | 2 | Numere impare |

**Fiecare participant a grupului utilizează diferite combinații de clase pentru sincronizare.**

**Semaphore, Lock, CyclicBarrier și altele.**

* 1. **Întrebări de verificare:**
1. Pentru ce este utilizată clasa CyclicBarrier?
2. Pentru ce este folosită metoda int await()?
3. Pentru ce este folosită metoda int getNumberWaiting()?
4. Pentru ce este folosită metoda int getParties()?
5. Pentru ce este utilizată clasa Semaphore?