IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

CONCURENTA IN JAVA

Ioana Leustean

https://docs.oracle.com/javase/tutorial/essential/concurrency/

https://docs.oracle.com/javase/specs/jls/se23/jls23.pdf

Overview (Java SE 23 & JDK 23) (oracle.com)

➢ Clasa Thread

```
public class Thread
extends Object
implements Runnable
```

- Metodele ale instantelor:
  - **run()**
  - **start()**
  - **join()**
  - **join(long millisecunde)**
  - **interrupt()**
  - **boolean isAlive()**

- Metode statice
  (se aplica thread-ului current):
  - **yield()**
  - **sleep(long milisecunde)**
  - **currentThread()**

➢ Mecanismul de sincronizarea thread-urilor prin lacatul intern

- Lacatul este pe obiect.
- Accesul la toate metodele sincronizate este blocat . Accesul la metodele nesincronizate nu este blocat.
- Numai un singur thread poate detine lacatul obiectului la un moment dat.
- Un thread detine lacatul intern al unui obiect daca:
  - executa o metoda sincronizata a obiectului,
  - executa un bloc sincronizat de obiect ,
  - daca obiectul este Class, thread-ul executa o metoda static sincronizata .
- Un thread poate face aquire pe un lacat pe care deja il detine (reentrant synchronization):

```
public class reentrantEx {
    public synchronized void met1{}
    public synchronized void met2{  this.met1() ;}
}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

➢ Modele de interactiune concurenta

Modelul Producator-Consumator

https://docs.oracle.com/javase/tutorial/essential/concurrency

➢ Modelul Producator-Consumator

Doua threaduri **comunica prin intermediul unui buffer** (memorie partajata):

- thread-ul Producator creaza datele si le pune in buffer
- thread-ul Consumator ia datele din buffer si le prelucreaza

Probleme de coordonare:

- Producatorul si consumatorul nu vor accesa bufferul simultan
- Producatorul va astepta daca bufferul este plin
- Consumatorul va astepta daca bufferul este gol
- Cele doua thread-uri se vor anunta unul pe altul cand starea buferului s-a schimbat

➢ Modelul Producator-Consumator

Doua threaduri **comunica prin intermediul unui buffer** (memorie partajata):

- thread-ul Producator creaza datele si le pune in buffer
- thread-ul Consumator ia datele din buffer si le prelucreaza

```
public class ProducerConsumer {

   public static void main(String[] args) {

      PCDrop drop = new PCDrop();

      (new Thread(new PCProducer(drop))).start();

      (new Thread(new PCConsumer(drop))).start();
   }}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

- Modelul Producator-Consumator


```
public class PCDrop {

   private String message;
   private boolean empty = true;

   public synchronized String take() {
                …
                return message;  }

  public synchronized String put(String message) { …   }
```

implementarea buffer-ului:
accesul se face prin metode sincronizate

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

➢ Thread-ul **producator**

```
import java.util.Random;

class PCProducer implements Runnable {

    private PCDrop drop;
    public PCProducer(PCDrop drop) {this.drop = drop;}

   public void run() {
      String importantInfo[] = {   "m1", "m2", "m3", "m4"};
      Random random = new Random();

      for (int i = 0;  i < importantInfo.length; i++) {
                                                    drop.put(importantInfo[i]);
                                                    try {
                                                          Thread.sleep(random.nextInt(5000)
                                                      } catch (InterruptedException e) {}
                                                    }
      drop.put("DONE");   }}
```

metoda sincronizata a obiectului **drop**

https://docs.oracle.com/javase/tutorial/essential/concurrency

➢ Thread-ul **consumator**

```
class Consumer implements Runnable {

   private PCDrop drop;
   public  Consumer(PCDrop drop) {   this.drop = drop;}

   public void run() {
      Random random = new Random();

      for (String message = drop.take();  ! message.equals("DONE"); message = drop.take())
                  {
                      System.out.format("MESSAGE RECEIVED: %s%n", message);
                      try {
                            Thread.sleep(random.nextInt(5000));
                           } catch (InterruptedException e) {}
      }}}
```

Metoda sincronizata a obiectului **drop**

https://docs.oracle.com/javase/tutorial/essential/concurrency

➢ Metode ale obiectelor

Sincronizarea accesului la buffer se face folosind metodele obiectelor:

- **void wait()**
  threadul intra in asteptare pana cand primeste **notifyAll()** sau **notify()** de la alt thread
- **void wait(milisecunde)**
  threadul intra in asteptare maxim **milisecunde**
- **void notifyAll()**
  trezeste toate threadurile care asteapta lacatul obiectului
- **void notify()**
  trezeste un singur thread, ales arbitrar, care asteapta lacatul obiectului;

➢ wait() vs sleep()

- **ob.wait()**
  - poate fi apelata de orice obiect ob
  - trebuie apelata din blocuri sincronizate
  - elibereaza lacatul intern al obiectului
  - asteapta sa primeasca o notificare prin **notify()** / **notifyAll()**
  - thread-ul current (care detine lacatul obiectului) va fi in starea WAITING
    iar dupa ce primeste notificare re-incearca sa detina lacatul obiectului

- **Thread.sleep()**
  - poate fi apelata oriunde
  - thread-ul curent se va opri din executie pentru perioada de timp precizata
    (va fi in starea BLOCKED)
  - nu elibereaza lacatele pe care le detine

Metodele **wait(), sleep() si join()** pot arunca **InterruptedException**
daca un alt thread intrerupe threadul care le executa.

➢ Modelul Producator-Consumator – implementarea buffer-ului

```
public class PCDrop {

   private String message;
   private boolean empty = true;

public synchronized String take() {
       if (empty) {
          try {
             wait();
          } catch (InterruptedException e) {}
       }
       empty = true;
       notifyAll();
       return message;
}
public synchronized String put(String message) {..}}
```

- implementarea foloseste *blocuri cu garzi*
- thread-ul este suspendat pana cand o anume conditie este satisfacuta

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

➢ Modelul Producator-Consumator – implementarea buffer-ului

```
public class PCDrop {

   private String message;
   private boolean empty = true;

public synchronized String take() { … return message;}

public synchronized void put(String message) {
            if (!empty) {
                        try {
                              wait();
                        } catch (InterruptedException e) {}
                  }
      empty = false;
      this.message = message;
      notifyAll(); }}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

➢ Modelul Producator-Consumator

```
public class ProducerConsumer {

   public static void main(String[] args) {

      PCDrop drop = new PCDrop();

      (new Thread(new PCProducer(drop))).start();

      (new Thread(new PCConsumer(drop))).start();
   }}
```

```
C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java ProducerConsumer
Messace received: m1
Messace received: m2
Messace received: m3
Messace received: m4
```

➢ Modelul Producator-Consumator

```
public class ProducerConsumer {

   public static void main(String[] args) {

      PCDrop drop = new PCDrop();

      (new Thread(new PCProducer(drop))).start();

      (new Thread(new PCConsumer(drop))).start();
      (new Thread(new PCConsumer(drop))).start();
   }}
```

```
Message received: m1
Message received: m2
Message received: m2
Message received: m3
Message received: m4
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

➢ Modelul Producator-Consumator

```
public class ProducerConsumer {

   public static void main(String[] args) {

      PCDrop drop = new PCDrop();

      (new Thread(new PCProducer(drop))).start();

      (new Thread(new PCConsumer(drop))).start();
      (new Thread(new PCConsumer(drop))).start();
   }}
```

```
Message received: m1
Message received: m2
Message received: m2
Message received: m3
Message received: m4
```

comportament nedorit

https://docs.oracle.com/javase/tutorial/essential/concurrency

➢ Modelul Producator-Consumator – implementarea buffer-ului

```
public class PCDrop {

   private String message;
   private boolean empty = true;

public synchronized String take() {
      while (empty) {
         try {
            wait();
         } catch (InterruptedException e) {}
      }
      empty = true;
      notifyAll();
      return message;
}
public synchronized String put(String message) {..}}
```

- implementarea foloseste *blocuri cu garzi*
- thread-ul este suspendat pana cand o anume conditie este satisfacuta

- **testarea unei conditii** se face intotdeauna folosind **while**

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

- Modelul Producator-Consumator – implementarea buffer-ului

```
public class PCDrop {

   private String message;
   private boolean empty = true;

public synchronized String take() { … return message;}

public synchronized void put(String message) {
            while (!empty) {
                        try {
                              wait();
                        } catch (InterruptedException e) {}
                  }
      empty = false;
      this.message = message;
      notifyAll(); }}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

➢ Modelul Producator-Consumator

```
public class ProducerConsumer {

    public static void main(String[] args) {

        PCDrop drop = new PCDrop();

        (new Thread(new PCProducer(drop))).start();
        (new Thread(new PCConsumer(drop))).start();
        (new Thread(new PCProducer(drop))).start();
        (new Thread(new PCConsumer(drop))).start();

    }}
```

```
Message received: m1
Message received: m1
Message received: m2
Message received: m3
Message received: m2
Message received: m4
Message received: m3
Message received: m4
```

➢ Interfata Lock

interface Lock

class ReentrantLock

Metode:
lock(), unlock(), tryLock()

Lock vs **synchronized**

- **syncronized** acceseaza lacatul intern al resursei si impune o programare structurata: primul thread care detine resursa trebuie sa o si elibereze

- obiectele din clasa **Lock** nu acceseaza lacatul resursei ci **propriul lor lacat**, permitand mai multa flexibilitate

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/concurrent/locks/Lock.html

- Interfata Lock

```
interface Lock

class ReentrantLock
```

```
import java.util.concurrent.locks.*

 Lock obLock = new ReentrantLock();
    obLock.lock();
    try {
          //  acceseaza resursa protejata de obLock
} finally {
             obLock.unlock();
          }
```

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/concurrent/locks/Lock.html

➢ class ReentrantLock

```
import java.util.concurrent.locks.*;
public class Interferencelock {

public static void main (String[] args) throws InterruptedException  {
        Counter c = new Counter();
        Thread thread1 = new Thread(new CounterThread(c));
        Thread thread2 = new Thread(new CounterThread(c));
        thread1.start(); thread2.start();
        thread1.join(); thread2.join();
    }}
class CounterThread implements Runnable {
    SCounter scounter;
    CounterThread (SCounter scounter) {
                                this.scounter=scounter;}
        public void run () { for (int i = 0; i < 5; i++) {
                                    counter.performTask();}
}
```

```
class Counter{
private int counter = 0;
private Lock counter_lock = new ReentrantLock();
public  void performTask () {
     counter_lock.lock();
      try {  …
          }
   finally{counter_lock.unlock();}
   }}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

- class ReentrantLock

```
class CounterThread implements Runnable {
   SCounter scounter;
   CounterThread (SCounter scounter) {this.scounter=scounter;}
      public void run () {}
}
```

```
class Counter{
private int counter = 0;
private Lock counter_lock = new ReentrantLock();
public  void performTask () {
    counter_lock.lock();
     try {
      int temp = counter;
      counter++;
      System.out.println(Thread.currentThread()
                          .getName() + " - before: "+temp+" after:" + counter);
   }
   finally{counter_lock.unlock();}
   }}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

➢ Interface Condition

- conditiile sunt legate de un obiect Lock

```
Lock obiectLock = new ReentrantLock();
Condition cond_obiectLock = obiectLock.newCondition();
```

- pot exista mai multe conditii pentru acelasi obiect Lock.
- implementeaza metode asemanatoare cu **wait()**, **notify()** si **notifyall()** pentru obiectele din clasa Lock
    - **await()**, **cond.await(long time, TimeUnit unit)**
      thread-ul current intra in asteptare
    - **signall()**
      un singur thread care asteapta este trezit
    - **signalAll()**
      toate thread-urile care asteapta sunt trezite

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/concurrent/locks/Condition.html

➢ Exemplul Producator-Consumator cu obiecte Lock in locul metodelor sincronizate

```
public class PCDrop1 {

    private String message;
    private boolean empty = true;

    private Lock dropLock = new ReentrantLock();
    private Condition cond_dropLock =   dropLock.newCondition();

    public String take() {  …
                                return message;  }

   public String put(String message) {  …   }
```

➢ Exemplul Producator-Consumator in care folosim obiecte Lock in locul metodelor sincronizate

```
public String take() {
      dropLock.lock();
      try{
      while (empty) {
         try {
            cond_dropLock.await();
         } catch (InterruptedException e) {}
      }

      empty = true;
      cond_dropLock.signalAll();
      return message;}
      finally { dropLock.unlock(); }
   }
```

```
public  void put(String message) {
    dropLock.lock();
     try{
       while (!empty) {
          try {
             cond_dropLock.await();
          } catch (InterruptedException e) {}
       }
       empty = false;
       this.message = message;
       cond_dropLock.signalAll();
     }
     finally {dropLock.unlock();}
    }
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

➢ Exemplul Producator-Consumator cu doua obiecte Condition pentru acelasi obiect Lock

```
public class PCDrop {

    private Queue<String> drop = new LinkedList<>();
    private static int Max = 5;

    private Lock dlock = new ReentrantLock();
    private Condition cond_empty = dlock.newCondition();
    private Condition cond_full = dlock.newCondition();

    public String take() {  …
                              return message;  }

   public String put(String message) {  …   }
```

buffer cu capacitate

cond_empty semnaleaza ca *exista spatiu* pentru a pentru a produce

cond_full semnaleaza ca *exista produse* care pot fi consumate

➢ Exemplul Producator-Consumator cu doua obiecte Condition si coada cu capacitate limitata

```
public String take() {
     dlock.lock();
      try{
      while (drop.size() == 0) {
         try {
            cond_full.await();
         }
         catch (InterruptedException ex) {}
      }
      String message  = drop.poll();
      System.out.format("Buffer items: %d%n", drop.size());

      cond_empty.signalAll();
      return message;
      }       finally { dropLock.unlock(); }
   }
```

cond_empty semnaleaza ca *exista spatiu* pentru a pentru a produce

cond_full semnaleaza ca *exista produse* care pot fi consumate

- Exemplul Producator-Consumator cu doua obiecte Condition si coada cu capacitate limitata

```
public String put() {
      dlock.lock();
       try{
       while (drop.size() == Max) {
          try {
             cond_empty.await();
          }
          catch (InterruptedException ex) {}
       }
       drop.offer(message);
       System.out.format("Buffer items: %d%n", drop.size());

       cond_full.signalAll();
        }       finally { dropLock.unlock(); }
   }
```

cond_empty semnaleaza ca *exista spatiu* pentru a pentru a produce

cond_full semnaleaza ca *exista produse* care pot fi consumate

- Exemplul Producator-Consumator cu doua obiecte Condition si coada cu capacitate limitata

```
class PCProducer implements Runnable {
   private PCDrop drop;

   public PCProducer(PCDrop drop) {
      this.drop = drop;
   }

   public void run() {
      Random random = new Random();

      while (true) {
         drop.put("Message" + random.nextInt(50));
         try {
            Thread.sleep(random.nextInt(50));
         }
         catch (InterruptedException ex) {

         }}}}
```

```
class PCConsumer implements Runnable {
   private PCDrop drop;

   public PCConsumer(PCDrop drop) {
      this.drop = drop;}

   public void run() {
      Random random = new Random();
      while (true) {String message = drop.take();
         System.out.format("Message received:
                                          %s%n", message);
         try {
               Thread.sleep(100);
         }
         catch (InterruptedException ex) {
         }}}
```

Vrem sa verificam ca bufferul nu va depasi capacitatea maxima

- Exemplul Producator-Consumator cu doua obiecte Condition si coada cu capacitate limitata

```
public class ProducerConsumerlockcond {

public static void main(String[] args) {
        PCDrop drop = new PCDrop();
        (new Thread(new PCProducer(drop))).start();
        (new Thread(new PCProducer(drop))).start();
        (new Thread(new PCConsumer(drop))).start();
        (new Thread(new PCConsumer(drop))).start();
        (new Thread(new PCConsumer(drop))).start();
    }}
```

```
Buffer items: 1
Buffer items: 2
Buffer items: 1
Message received: Message49
Buffer items: 0
Message received: Message34
Buffer items: 1
Buffer items: 0
Message received: Message44
Buffer items: 1
Buffer items: 2
Buffer items: 3
Buffer items: 4
Buffer items: 5
Buffer items: 4
Message received: Message46
Buffer items: 3
Message received: Message14
Buffer items: 4
Buffer items: 5
Buffer items: 4
Message received: Message42
Buffer items: 5
Buffer items: 4
Message received: Message34
Buffer items: 3
```

Pe săptămâna viitoare!

https://docs.oracle.com/javase/tutorial/essential/concurrency