IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

JAVA

Ioana Leustean

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

➢ Comunicarea intre thread-uri

- doua thread-uri care incrementeaza acelasi contor

```
public class Interference  implements Runnable {
 static Integer counter = 0;

        public void run () {
           for (int i = 0; i < 5; i++) {
              performTask();
           }}

private void performTask () {
        int temp = counter;
        counter++;
        System.out.println(Thread.currentThread()
                              .getName() + " - before: "+temp+" after:" + counter);}
public static void main (String[] args) {.. }}
```

➢ Comunicarea intre thread-uri – data race

- doua thread-uri care incrementeaza acelasi contor

```
public static void main (String[] args) {
    Thread thread1 = new Thread(new Interference());
    Thread thread2 = new Thread(new Interference());
    thread1.start(); thread2.start();
    thread1.join(); thread2.join();  }
```

```
Thread-1 - before: 1 after:2
Thread-0 - before: 0 after:1
Thread-1 - before: 2 after:3
Thread-0 - before: 3 after:4
Thread-1 - before: 4 after:5
Thread-0 - before: 5 after:6
Thread-1 - before: 6 after:7
Thread-0 - before: 7 after:8
Thread-1 - before: 8 after:9
Thread-0 - before: 9 after:10
```

```
Thread-0 - before: 0 after:2
Thread-1 - before: 1 after:2
Thread-0 - before: 2 after:3
Thread-0 - before: 4 after:5
Thread-1 - before: 3 after:4
Thread-0 - before: 5 after:6
Thread-1 - before: 6 after:7
Thread-0 - before: 7 after:8
Thread-1 - before: 8 after:9
Thread-1 - before: 9 after:10
```

data race

➢ Mecanismul de sincronizarea thread-urilor

- Fiecare obiect are un lacat intern (*intrinsic lock*, *monitor lock*).

- Un thread care acceseaza un obiect trebuie sa:
  - detina (**aquire**) lacatul intern,
  - acceseaza/modifica datele obiectului,
  - elibereaza (**release**) lacatul obiectului.

- In timpul in care un thread detine lacatul intern al unui obiect, orice alt thread care doreste sa detina (faca acquire) lacatul este blocat.

➢ Sincronizarea thread-urilor se face cu:

- Metode sincronizate

```
private synchronized void syncMethod () {
   //codul metodei
 }
```

Cand un thread apeleaza o metoda sincronizata el trebuie sa detina lacatul obiectului caruia ii apartine metoda, executa metoda apoi elibereaza lacatul.

acquire, execute, **release**

Pentru metodele statice, lactul este al obiectului *Class* asociat clasei respective.

➢ Sincronizarea thread-urilor

- Metode sincronizate

```
private synchronized void syncMethod () {
   //codul metodei
 }
```

- Instructiuni (blocuri) sincronizate

```
synchronized (object reference){
   // instructiuni
 }
```

se specifica obiectul care detine lacatul

O metoda sincronizata poate fi scrisa ca bloc sincronizat:

```
private void syncMethod () {
     synchronized (this){
       //codul metodei
 }}
```

➢ Metode sincronizate

- doua thread-uri care incrementeaza acelasi contor

```
public class Interference  implements Runnable {
 static Integer counter = 0;

       public void run () {
          for (int i = 0; i < 5; i++) {
             performTask();
          }}

private synchronized void performTask () {
       int temp = counter;
       counter++;
       System.out.println(Thread.currentThread()
                          .getName() + " - before: "+temp+" after:" + counter);}
public static void main (String[] args) {.. }}
```

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

➢ Metode sincronizate

- doua thread-uri care incrementeaza acelasi contor

```
public static void main (String[] args) {
      Thread thread1 = new Thread(new Interference());
      Thread thread2 = new Thread(new Interference());
      thread1.start(); thread2.start();
      thread1.join(); thread2.join();  }
```

```
Thread-1 - before: 1 after:2
Thread-1 - before: 2 after:3
Thread-0 - before: 0 after:1
Thread-1 - before: 3 after:4
Thread-0 - before: 4 after:5
Thread-1 - before: 5 after:6
Thread-0 - before: 6 after:7
Thread-1 - before: 7 after:8
Thread-0 - before: 8 after:9
Thread-0 - before: 9 after:10
```

➢ Instructiuni (blocuri) sincronizate

- doua thread-uri care incrementeaza acelasi contor

```
public class Interference  implements Runnable {
 static Integer counter = 0;
 static Object clock = new Object();

       public void run () {
          for (int i = 0; i < 5; i++)
            synchronized (clock) performTask();
          }

private  void performTask () {
       int temp = counter;
       counter++;
       System.out.println(Thread.currentThread()
                          .getName() + " - before: "+temp+" after:" + counter);}
public static void main (String[] args) {.. }}
```

sincronizarea se face prin lacatul obiectului **clock**

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

➢ Mecanismul de sincronizarea thread-urilor

- 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

➢ Metode ale 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()**
  - 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
  - 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

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


```
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;

➢ 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

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();
   }}
```

```
C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java ProducerConsumer
Messace received: m1
Messace received: m2
Messace received: m3
Messace 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/javase/7/docs/api/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/javase/7/docs/api/java/util/concurrent/locks/Lock.html

- class ReentrantLock

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

public class ThreadInterference{
   private Integer counter = 0;
   private ReentrantLock lock = new ReentrantLock();

public static void main (String[] args) {
       ThreadInterference demo = new ThreadInterference();
       Task task1 = demo.new Task();
       Thread thread1 = new Thread(task1);
       Task task2 = demo.new Task();
       Thread thread2 = new Thread(task2);
       thread1.start();
       thread2.start();
   }
  private class Task implements Runnable { …}
  private void perform Task() { …}
}
```

```
   private class Task implements Runnable {
      public void run () {
          for (int i = 0; i < 5; i++) {
              performTask();
          }}}
```

```
private void performTask () {
       lock.lock();
       try{
       int temp = counter;
       counter++;
       System.out.println(Thread.currentThread().getName()
                          + " - before: "+temp+" after:" + counter);
       }
       finally {lock.unlock();}
    }
```

https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/java-thread-synchronization.html

➢ Interface Condition

- 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

- Conditiile sunt legate de un obiect Lock
- Pot exista mai multe conditii pentru acelasi obiect Lock.

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

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Condition.html

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

```
public class PCDrop1 {

   private String message;
   private boolean empty = true;

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

   public String take() {
                                    …
                              return message;  }

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

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

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

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

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

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

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