Manual del laboratori de "Programació Concurrent i Distribuïda" JAVA Concurrent

Xavier Messeguer

February 27, 2008

Sessió

Processos ("threads" o fils)

Entenem per *Fil* un conjunt d'instruccions executades seqüencialment, és a dir, un algorisme seqüencial. Llavors si un algorisme està composat per varis processos que s'executen en paral·lel, caldrà implementar-los com fils.

Hi ha dues maneres de crear fils:

- Com extensió de la classe `java.lang.Thread`.
- Com implementació de la interfície `Runnable`.

Considerem separadament els dos casos.

1.1 Fils com extensió de `Thread`

La classe `Tread` té molts mètodes que heretarem i en algun cas reescriurem. N'hi ha dos de molt importants:

- `run()`: és un mètode que cal reescriure amb el codi del nostre algorisme.
- `start()`: és un mètode que invocarem quan volguem executar el fil.

L'esquema és:

```
class NostraClasse extends Thread{
   ⋮  // atributs que seran els parametres del fil
   ⋮  //metodes
   public void run(){
      ⋮//algorisme
   }
}
class Executa ...{
   ⋮
   NostraClasse t new NostraClasse();
   t.start();  // invocacio de t.run()
   ⋮
   }
}
```

El mètode `t.start` invoca el mètode `run` començant així l'execució del fil, llavors hi haurà dos fils executant-se simultàneament.

Desenvolupem un primer exemple: crearem una subclasse `PingPong` de `Thread` que escriu una cadena de caràcters cada cert temps. Això ho fa 30 vegades.

```
class PingPong extends Thread{
      // atributs
   String word;
   int delay;
      // constructor
   PingPong(String whatToSay, int delayTime) {
      word = whatToSay;
      delay = delayTime;
   }
      // el metode run
   public void run(){
      try {
         for (int i=0; i<30; i++) {
            System.out.print(word + " ");
            sleep(delay);
         }
      } catch (InterruptedException e) {
         return;
      }
   }
}
```

El mètode

- `sleep()` adorm el fil durant els mil·llisegons indicat i si és interromput per un altre fil quan dorm es desperta i s'activa la interrrupció `InterrumpedException`.

Aquesta interrrupció pot arribar en qualsevol moment de l'execució. S'ha de capturar i tractar amb l'instrucció `try`:

TryStatement:
`try` *Block* Llistade*CatchClause*
⋮

CatchClause:
`catch (` *FormalParameter* `)` *Block*

Primerament s'executen les instruccions incloses en el `try` *Block*. Si no es genera cap interrupció el bloc s'executa normalment. Si s'en genera alguna considerada en alguna *CatchClause* s'executa el bloc corresponent.

Continuem amb l'exemple. En la classe `PingPongJoc` creem els fils `Ping` i `Pong` de la classe `PingPong` i invoquem el mètode `start`.

```
import PingPong.*;
class PingPongJoc{
   public static void main(String[] args){
```

```
        PingPong Ping = new PingPong("ping", 33);
        Ping.start();
        PingPong Pong = new PingPong("PONG", 100);
        Pong.start();
    }
}
```

Quan l'executem,

• **Pregunta :** Tenen la mateixa traça diferents execucions?

1.2 Estats d'un procés

En un entorn concurrent els processos poden trobar-se en els diferents estats

- `Preparat`: un cop s'ha executat l'instrucció `start`.
  - `Execució`: un cop el planificador li ha assignat el processador.
    - `Bloquejat:` al fer una E/S. El deixa quan l'ha acabat.
    - `Adormit`: un cop s'ha executat l'instrucció `sleep`. El deixa passat el temps indicat.
    - `Espera (per synchronized)`: al sol.licitar el *lock* d'un objecte i estar ocupat. El deixa en aconseguir l'objecte.
    - `Espera (per wait)`: un cop s'ha executat l'instrucció `wait`. El deixa un cop s'executen les instruccións `notify` o `notifyAll`.
      - `Acabat`:
      - `Preparat`:

1.3 Fils com interfície de `Runnable`

Si consultem la classe `java.lang.threads` trobem la següent especificació:

```
public class thread implements Runnable {
    ⋮// Declaracions del atributs
    ⋮// i una llista de mes de 35 metodes
}
```

Llavors quan extenem aquesta classe, com hem fet en la secció precedent, heretem tots els atributs i tots els mètodes. Si el nostre programa quasi bé no utilitza aquests mètodes, podem crear fils directament de la interfície `Runnable`.

Si consultem la classe `java.lang.Runnable` trobem la següent especificació:

```
public interface Runnable {
    public abstract void run();
}
```

Per tant la nostra implementació ha de ser:

```
class Exemple implements Runnable{
   public void run(){
      .
      .
      .//algorisme
   }
}
```

I el classe que l'executi:

```
class Execucio {
   public static void main(String[] args){
      Runnable r = new Exemple(...);
      Thread t = new Thread(r);
      t.start();
      .
      .
      .
   }
}
```

Les dues últimes instruccions poden substituir-se per

```
new Thread (r).start();
```

Desenvolupem l'exemple `PingPong`:

```
class PingPongRun implements Runnable {
   String word;
   int delay;
   PingPongRun(String whatToSay, int delayTime) {
      word = whatToSay;
      delay = delayTime;
   }
   public void run(){
      try {
         for (int i=0; i<30; i++) {
            System.out.print(word + " ");
            Thread.sleep(delay);
         }
      } catch (InterruptedException e) {
         return;
      }
   }
}

class PingPongJocRun{
   public static void main(String[] args){
      Runnable ping = new PingPongRun("ping", 33);
      new Thread(ping).start();
      Runnable Pong = new PingPongRun("PONG", 100);
      new Thread (Pong).start();
   }
}
```

1.4 Exclusió mutua dels mètodes d'un objecte

En aquesta secció veurem els mecanismes que ofereix JAVA per evitar l'accés concurrent a les dades que pugui donar lloc a errors o a la seva inconsistència. Concretament veurem com declarar els mètodes d'un objecte en exclusió mutua. Definim la classe `DosValors`,

```
class DosValors{
   private int x,y,s,t;
   //{ s = x + y, t = x + y}
   public void SetVal (int a,int b){
      x = a;y = b ; t = a + b ; s = a + b ;
      System.out.println (x+"+"+y+"="+t+"="+s);
   }
}
```

els processos que la modifiquen concurrentment,

```
class DosValorsFil extends Thread{
   private DosValors v;
   public DosValorsFil(DosValors h){ v = h; } // Igualtat de variables
   public void run(){
      for (int i =0; i <20; i++){
         v.SetVal(i,2*i);
      }
   }
}
```

i la classe executable,

```
public class DosValorsExec{
   public static void main(String[] args){
      DosValors h = new DosValors();
      DosValorsFil d = new DosValorsFil(h);
      DosValorsFil e = new DosValorsFil(h);
      d.start();
      e.start();
   }
}
```

En executar el programa no es probable que veiem cap inconsistència en les dades, la cual cosa pot succeir tal i com ara comprovarem. Modifiquem la classe `DosValors` afegint un bucle entre les actualitzacions d'`s` i `t`,

```
class DosValors{
      ⋮
      t = a + b ;
      for (int i=0;i<10000000;i++){;}
      s = a + b ;
      ⋮
}
```

o bé, si no es reflecteix, feu

```
class DosValors{
      ⋮
      t = a + b ;
      try {
         Thread.sleep(1000);
      } catch (InterruptedException e) {
         return;
      }
      s = a + b ;
      ⋮
}
```

En executar el programa podem veure que els dos processos que l'actualitzen intercalen el mètode `SetVal`, la cual cosa comporta la inconsistència de les dades. Per evitar-ho Java ofereix el *MethodModifier* `synchronized` que actua de la forma següent: per a tots els mètodes declarats `synchronized` crea una cua de processos que els criden i fins que l'última crida no ha acabat no deixa entrar una nova crida, és a dir, dòna els mètode en exclusió mútua al primer procés d'una cua associada a aquest objecte. Per tant si fem,

```
class DosValors{
   ⋮
   public synchronized void SetVal (int a,int b){
   ⋮
   }
}
```

podem comprovar que les dades conserven la seva integritat.

Cal remarcar que l'exclusió mútua és fa sobre els métodes d'un objecte, no de la classe. Es a dir que podrien executar-se concurrentment dos mètodes `SetVal` si pertanyessin a objectes diferents.

1.5 Exclusió mútua sobre un objecte

Ens pot interessar que la part d'exclusió mútua no sigui tot el mètode sino només unes quantes instruccions, llavors la cua ha d'associar-se a un objecte qualsevol. En el següent exemple s'associa al propi objecte:

```
class DosValors{
   private int x,y,s,t;
   //{ s = x + y, t = x + y}
   public void SetVal (int a,int b){
      synchronized (this){
         x = a;y = b ; t = a + b;
         // amb el for o l'sleep
```

```
            s = a + b ;
        }
        System.out.println (x+"+"+y+"="+t+"="+s);
    }
}
```

1.6 Monitors: comunicació entre processos

Un monitor és una classe que controla l'accés a unes dades per part de diferents processos per tal de conservar la seva consistència i establir un protocol d'accés. Veurem com es dissenya un monitor en el següent exemple: dissenyarem un procés escriptor `EscriuEnter` i un procés lector `LlegeixEnter` que s'aniran alternant en l'accés a una variable entera.

Sigui la classe que enregistra un enter

```
class ValorEnter{
   private int Enter;
   public synchronized void SetEnter (int val){
      Enter = val;
   }
   public synchronized int GetEnter(){
      return Enter;
   }
}
```

i les classes `EscriuEnter` i `LlegeixEnter`,

```
class EscriuEnter extends Thread{
   private ValorEnter valor;
   public EscriuEnter(ValorEnter h){
      valor = h; // Valor inicial
   }
   public void run(){
      for (int i =0; i <20; i++){
         valor.SetEnter(i);
         System.out.println("Escriptor escriu'' + i);
         //adormim el proces
         try {
             sleep( (int)(Math.random() * 3000) );
         }
         catch ( InterruptedException e ) {
             System.err.println("Exception "+ e.toString());
         }
      }
   }
}

class LlegeixEnter extends Thread{
   private ValorEnter valor;
   public LlegeixEnter(ValorEnter h){
      valor=...;
```

```
    };
    public void run(){
       int val;
       for(int i=0;i<20;i++)){
          val = valor.GetEnter();
          System.out.println("Lector llegeix " + val);
          try {
              sleep(...);
          }
          catch(...)  {
              ...;
          }
       }
    }
}
```

Finalment fem

```
public class UsaValorEnter{
   public static void main(String[] args){
      ValorEnter h = new ValorEnter();
      EscriuEnter p = new EscriuEnter(h);
      LlegeixEnter c = new ...;
      p.start();
      ...;
   }  }
```

Quan executem `UsaValorEnter` podem comprovar que lectura i escriptura no s'alternen tal i com haviem exigit. Per aconseguir-ho transformarem `ValorEnter` en un monitor fent el següent:

- Afegint la variable `llegible`, anomenada variable de condició del monitor, que permetrà alternar els mètodes.
- Completant l'esquema

```
while (!condicio) wait();
   ⋮ // instruccions a executar quan la condicio es certa
```

Cal remarcar que l'esquema només és vàlid en mètodes `synchronized` (o en conjunts d'instruccions declarades `synchronized`). Cal heretar els següents mètodes de la classe `java.lang.Object`:

- `wait()` que adorm l'execució del fil,
- `notify()` que desperta només un altre fil,
- `notifyAll()` que desperta tots els altres fils.

Ara ja podem dissenyar l'accés alternat:

```
class ValorEnter{
   private int Enter;
   private boolean llegible = false;

   public synchronized void SetEnter (int val){
      while(llegible){
          try{ wait(); }
          catch (...)  {...}
      }
      Enter = val;
      llegible = true;
      notify();
   }

   public synchronized int getEnter(){
      while(!llegible){
          try{...}
          catch (...)  {...}
      }
      llegible = false;
      notify();
      return Enter;
   }
}
```

Recordeu que `synchronized` indica que el mètode es té en exclusió mútua, pero al entrar en estat `wait` es perd aquesta exclusivitat. Quan rebi un `notify` s'evaluarà la condició del monitor i si és certa es tornarà a demanar el mètode en exclusió mútua.

1.7 Exercicis

1. Dissenyar un programa *Productors/Consumidors* que emmagatzemin i extreguin els elements en una pila de la classe `stack` de java de `MAX` elements. És clar que els Productors empilen elements mentre la pila no estigui plena i que els Consumidors en consumeixen mentre no estigui buida. Visualitzeu d'alguna forma l'evolució de la pila.

2. Dissenyeu un programa amb quatre processos que escriuen respectivament les lletres a, b, c, d 100 vegades, sota les següents condicions:

   - Sense determinar cap tipus d'ordre, és a dir sense monitoritzar-les.
   - Escrivint de forma consecutiva *abcdabcd*....
   - Les lletres a i d s'han d'anar intercal.lant a l'igual que les lletres b i c, pero cada parell de forma independent com per exemple *adabdadcb*.

   Codifiqueu-lo de tal manera que si en lloc de ser quatre lletres fossin 10 lletres només calgués afegir unes poques línies més de codi.