Pràctica. Problemàtica bàsica de la concurrència. Exclusió mútua.

Josep Cotrina, Marcel Fernández, Jordi Forga, Juan Luis Gorricho, Francesc Oller

Introducció a l'exclusió mútua. Accions atòmiques

A partir d'ara l'objectiu de la pràctica serà veure la problemàtica que apareix quan dos o més processos concurrents accedeixen a una variable (memòria) compartida. Es veurà que per dissenyar algorismes correctes es fa necessari algun mecanisme d'*exclusió mútua*: *mentre un dels processos està accedint a la variable compartida cap altre procés pot accedir a aquesta variable*.

Les accions (operacions, sentències o instruccions) dels processos concurrents s'entrellacen de forma no determinista, és a dir, no es pot predir l'ordre d'execució. A més, és clar que les solucions als problemes han de ser independents de l'entrellaçament, o en tot cas, s'ha de limitar els possibles entrellaçaments, evitant entrellaçaments que porten a resultats erronis.

En aquesta situació és important conèixer les accions atòmiques o indivisibles de Java. La característica bàsica d'una acció atòmica és la impossibilitat d'interrompre el procés durant l'execució d'aquesta acció.

El problema és determinar quines són les accions atòmiques en Java. Per exemple l'acció

$$x = x + 1;$$

és una acció atòmica?

Si resseguim l'execució d'aquesta acció veurem que està formada per les següents accions:

1. Lectura del valor de la variable x en un registre del processador.
2. Increment del valor del registre.
3. Escriptura del valor del registre en l'espai de memòria corresponent a la variable x.

Per tant, es pot concloure que no és una acció atòmica.

Així doncs, a partir d'ara les úniques operacions de Java que considerarem atòmiques són les operacions de lectura i escriptura d'una variable i per tant suposarem que els processos poden entrellaçar-se en qualsevol punt.

Memòria compartida

El problema que es proposa és el següent. Es disposa d'una zona de memòria compartida (un objecte que encapsula un valor de tipus `int`) per dos threads. Cadascun dels threads incrementa N vegades el valor de la variable compartida, per tant al final s'hauria d'obtenir un valor de $2N$.

Esquema de l'aplicació

Primer s'implementa una classe que representarà el comptador (memòria) compartit (completar les parts que falten).

```
public class Menter{

    ...
```

```
    // Mètode per modificar el valor

    // Mètode per recuperar el valor
}
```

El següent pas és implementar els algorismes dels threads. Per fer-ho es defineix una classe `Proces`, que simplement incrementa N vegades el valor del *comptador* compartit.

Per tant, aquesta classe ha de conèixer el valor N i a més ha de tenir accés al comptador compartit, que serà un objecte de tipus `Menter`. Un esquema de la classe és el següent:

```
public class Proces ... {

    ...

    public Proces( ... ){ ... }

    ...

}
```

Finalment queda per implementar el programa principal, que és l'encarregat de crear els diferents objectes i arrencar la simulació.

```
public class Principal{

    public static void main(String[] args){
        // Crear comptador compartit

        // Crear i iniciar 2 threads

        // Esperar que els threads acabin

        System.out.println("El␣resultat␣final␣es␣=␣" +
                           valor comptador compartit);
    }
}
```

Avaluació dels resultats

La part més important d'aquesta solució és la implementació del mètode `run()` de la classe `Proces`.

Depenent de la solució que hageu implementat pot passar que el resultat obtingut sigui correcte ($2N$) o no. En cas de no ser correcte hauríeu de justificar la raó d'aquest comportament.

Independentment de que el resultat final sigui correcte, el que s'hauria d'assegurar és que el disseny de l'algorisme és correcte (i no només el resultat calculat), és a dir, que sempre obtindrem el resultat desitjat sigui quin sigui l'entrellaçament entre els dos processos concurrents.

Provar això normalment és prou complicat, ara bé, sempre es poden estudiar els entrellaçaments més conflictius. Per fer-ho es poden forçar diferents entrellaçaments afegint crides al mètode `sleep()` o `yield()`.

El punt més delicat de l'algorisme és l'accés i modificació de la variable compartida (el comptador). Ja s'ha vist que incrementar el valor d'una variable no és una acció atòmica i per tant no es pot assegurar que s'executi en exclusió mútua. Així doncs, si l'algorisme és correcte, si es desglossa l'acció d'incrementar el comptador en les tres accions següents l'algorisme hauria de seguir sent correcte. A més, podem forçar que els processos s'entrellacin justament en aquest punt (és a dir, assegurar que l'acció d'incrementar no es realitza de forma atòmica) afegint crides al mètode `sleep()` o `yield()` on calgui.

```
public void run(){
    // Iteració
        // Obtenir el valor del comptador

        // Incrementar aquest valor

        // Guardar el valor del comptador
}
```

Per tant, si la solució proposada és correcta aquests canvis no haurien d'afectar al resultat final obtingut. Proveu què passa si realitzeu aquests canvis a la vostra solució. **Afegiu les crides a `sleep()` o `yield()` on calgui**.

1 Primera aproximació: flag d'ocupat

El primer que s'ha de fer és determinar la zona crítica (ha d'incloure el menor número de sentències possible), i després fer que els threads hi accedeixin seqüencialment.

La manera immediata que proporciona un llenguatge de programació de fer esperar un thread (quan sigui necessari) serà amb una *espera activa*, és a dir, s'atura el procés en una iteració fins que la condició d'aturada deixa de ser certa.

```
while(/*condicio*/){}
```

A partir d'ara treballarem solucions al problema d'Exclusió Mútua amb dos processos. És convenient encapsular els protocols d'entrada i sortida en dos mètodes d'una classe Mutex: `entraZC` i `surtZC` respectivament. D'aquesta manera s'aconsegueix que els clients de `Mutex`, en el nostre cas `Proces`, tant sols facin servir aquests mètodes, despreocupant-se de cóm estan implementats. L'algorisme proposat tindrà un esquema com el que segueix:

```
class Mutex{

    ...

    protected void entraZC(){ ... }

    protected void surtZC(){ ... }
}

public class Process ...{
    protected Mutex m;

    ...

    public void run(){

        for( ... ){
            // Entrada a la Zona Critica
            m.entraZC();

            //Zona critica
            ...

            // Sortida de la Zona Critica
            m.surtZC();
        }
```

```
    }
}
```

Quan un thread vol entrar, espera a que el flag d'ocupat sigui fals i a continuació el posa a cert. Al sortir posa el flag a fals. L'esquema de l'algorisme és:

```
protected void entraZC(){
    // Entrada ZC
        // Esperar fins que el flag d'ocupat valgui fals
        // Posar el flag d'ocupat a cert
}

protected void surtZC(){
    //Sortida ZC
        // Posar el flag d'ocupat a fals
}
```

Decidiu conceptualment si l'algorisme és correcte, es a dir busqueu algun entrellaçat que faci que entrin els dos threads alhora. Si el trobeu indicarà que l'algorisme és incorrecte. Poseu crides d' `sleep()` o `yield()` on calgui per tal de reproduir l'entrellaçat incorrecte. Que s'observa en l'execució? Perquè?

2 Segona aproximació: Avís (flags) d'intenció

Els threads han d'avisar de intenció d'entrar (però encara no estan dins). Aquest comportament es pot aconseguir amb dos flags. Es poden definir dues variables d'accés `flag1` i `flag2`, de tipus `boolean`—o millor, un array— de manera que cada procés té associat un dels flags. Cada procés podrà modificar el seu flag (valor true per indicar que vol entrar a la zona crítica) i com a molt consultar el flag de l'altre procés.

Llavors, un procés abans d'entrar a la zona crítica posa el valor del seu flag a true i espera que el valor del flag de l'altre procés valgui false. L'esquema de l'algorisme és:

```
protected void entraZC(){
    // Entrada ZC
        // Posar al meu flag el valor true
        // Esperar fins que el flag de l'altre valgui false
}

protected void surtZC(){
    //Sortida ZC
        // Posar el meu flag a false
}
```

Sorgeix una petita dificultat tècnica: cóm discrimina `entraZC` el flag propi i l'aliè? Dues solucions:

- Passar l'identificador de procés—0 o 1— com a paràmetre a `entraZC`. Aquest identificador és paràmetre del constructor `Proces`. D'aquesta forma el mètode `entraZC` s'implementa amb sentències selectives, lo qual no és gaire elegant. Ens les podríem estalviar? Cóm?
- L'identificador de procés és un atribut de dades de `Proces` de nom, suposem, `id`. Referenciar-lo dins de Mutex com `((Proces)Thread.currentThread()).id`

Posar crides d' `sleep()` o `yield()` on calgui. Que s'observa en l'execució? Perquè?

3 Trencar el bloqueig

Finalment, per evitar el problema del bloqueig, es pot fer que mentre un procés està esperant a que el flag de l'altre valgui 0, posi transitòriament el seu flag a 0 (permet que l'altre procés pugui avançar). L'esquema del protocol d'entrada és ara:

```
...
protected void entra_zc(){
    /* Protocol d'entrada:
         Posar al meu flag el valor 1
         Esperar fins que el flag de l'altre valgui 0 {
             posar el meu flag a 0
             --> en aquest punt el flag esta desactivat.
             posar el meu flag a 1
         }
    */
}
...
```

4 Algorisme de Peterson

Implementeu l'algorisme de Peterson que es descriu a continuació i explica el perquè és correcte.

```
    // algorisme de Peterson o de Tie-break (1981)
    protected void entraZC(){
        // Entrada ZC
            // Posar al meu flag el valor true
            // Dona prioritat a l'altre
            // Esperar fins que el flag de l'altre valgui false o la prioritat sigui per mi
    }

    protected void surtZC(){
        //Sortida ZC
            // Posar el meu flag a false
    }
```