TD2 Programmation concurrentielle

Développement Logiciel (L2-S4)

Lundi 10 février 2014

Exercice 1 (Au bar...)
Trois clients se trouvent au bar. Il est tard, ils sont fatigués, et ils s'endorment sur le bar. Chacun se réveille après s'être brièvement assoupi pendant une durée aléatoire comprise entre 0 et 1000 millisecondes (que vous pouvez obtenir avec `Thread.sleep((int)(Math.random()*1000))`).

Au réveil, chaque client demande ce qu'il veut : un “Java” (beurk), un “Expresso” (miam) ou un “Cappucino” (miam miam). Et parce qu'il est si fatigué, il s'assoupit à nouveau pour une durée aléatoire entre 0 et 1000 millisecondes. Malheureusement, le serveur s'est également assoupi, et chaque client va abandonner le bar, profondément frustré, à la cinquième demande.

Le compte-rendu d'une soirée se présente de la manière suivante :

```
Java!
Cappuccino!
Java!!
Cappuccino!!
Espresso!
Cappuccino!!!
Java!!!
Cappuccino!!!!
Espresso!!
Cappuccino!!!!!
Java!!!!
Espresso!!!
Espresso!!!!
Java!!!!!
Espresso!!!!!
```

Ecrivez un programme qui modélise ces clients (fatigués et assoiffés) et le serveur (peu coopératif).

Solution :

Client.java

```
package bar;

class Client extends Thread {
  String text;
```

```
  public Client(String text) {
    this.text = text;
  }

  public void run() {
    for(int i = 0; i < 5; i++) {
      try {
        sleep((int)(Math.random()*1000));
      }
      catch(InterruptedException e) {
      }
     this.text = this.text + "!";
     System.out.println(text);
    }
  }
}
```

Client.java

```
package bar;


public class Bar {

  public static void main(String args[]) {
    Client java, espresso, capuccino;

    java = new Client("Java");
    espresso = new Client("Espresso");
    capuccino = new Client("Cappuccino");
    java.start();
    espresso.start();
    capuccino.start();
  }

}
```

Exercice 2 (Une banque pas très fiable...)
Une banque très (trop) simple à été implémentée avec la classe suivante :

```
class SimpleBank {
  static int[] account = {30, 50, 100};

  public void transfer(int from, int to, int amount) {
    int namount;

    namount = account[from];
    namount -= amount;

    // ... difficult task of non-deterministic length

    account[from] = namount;

    namount = account[to];
    namount += amount;
    account[to] = namount;
  }

  public void balance() {
    for(int i = 0; i < account.length; i++)
      System.out.println("Account "+ i +": " + account[i]);
  }
}
```

Des employés qui travaillent dans la banque (une autre classe `Employee`). Le travail d'un employé consiste à transférer une somme d'argent `amount` du compte `from` vers le compte `to`.

1. Définissez ces deux classes, ainsi qu'une classe `SimpleBankDemo` (contenant la méthode `main`) qui crée la banque avec trois employés et qui organise un transfert cyclique de 20 EUR entre les différents comptes de la banque (chaque employé est responsable pour le transfert entre deux comptes particuliers).
2. Réalisez trois différentes exécutions de `SimpleBankDemo.main`, et observez les différentes balances résultant de cette implémentation.
3. Que constatez-vous ? Comment corriger ce problème ?

Solution :

SimpleBank.java

```
class SimpleBank {
  static int[] account = {30, 50, 100};

  public void transfer(int from, int to, int amount) {
    int namount;

    namount = account[from];
    namount -= amount;
```

```
    // race condition, since new amount is not yet booked
    try {
      Thread.sleep((int)Math.random()*1000);
    }
    catch(InterruptedException e) {
    }
    account[from] = namount;

    namount = account[to];
    namount += amount;
    // dito
    account[to] = namount;
  }

  public void balance() {
    for(int i = 0; i < account.length; i++)
      System.out.println("Account "+ i +": " + account[i]);
  }
}
```

Employee.java

```
class Employee extends Thread {

  SimpleBank bank;
  int from, to, amt;

  public Employee(SimpleBank bank, int from, int to, int amount) {
    this.bank = bank;
    this.from = from;
    this.to = to;
    this.amt = amount;
  }

  public void run() {
    // do money transfer
    bank.transfer(from, to, amt);
    // print balance
    System.out.println("After:");
    bank.balance();
  }
}
```

SimpleBankDemo.java

```
public class SimpleBankDemo {

  public static void main(String[] args) {
    Employee A1, A2, A3;
    SimpleBank b = new SimpleBank();
```

```
    System.out.println("Before:");
    b.balance();

    // A cyclic transfer of 20 bucks ...
    A1 = new Employee(b, 0, 1, 20);
    A2 = new Employee(b, 1, 2, 20);
    A3 = new Employee(b, 2, 0, 20);

    A1.start();
    A2.start();
    A3.start();
  }
}
```

Traces :

```
Before:
Account 0: 30 Account 1: 50 Account 2: 100
After:
Account 0: 10 Account 1: 70 Account 2: 100
After:
Account 0: 10 Account 1: 30 Account 2: 120
After:
Account 0: 30 Account 1: 30 Account 2: 80

Before:
Account 0: 30 Account 1: 50 Account 2: 100
After:
Account 0: 10 Account 1: 70 Account 2: 100
After:
After:
Account 0: 30 Account 1: 50 Account 2: 100
Account 0: 30 Account 1: 50 Account 2: 100

Before:
Account 0: 30 Account 1: 50 Account 2: 100
After:
Account 0: 30 Account 1: 30 Account 2: 120
After:
Account 0: 50 Account 1: 30 Account 2: 80
After:
Account 0: 10 Account 1: 50 Account 2: 80
```

Traces : *Reparation : Soit introduction d'un LOCK-object qui met la section critique — l'intérieur de la methode transfer — dans une partie de code non-interlacé :*

```
class SimpleBank {
  static int[] account = {30, 50, 100};
  final static Object LOCK = new Object();

  public void transfer(int from, int to, int amount) {
    int namount;
    synchronized(LOCK) {
```

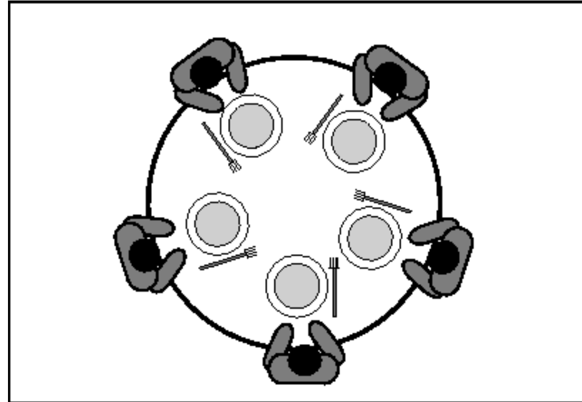
```
      namount = account[from];
      namount -= amount;
      // race condition, since new amount is not yet booked
      try {
          Thread.sleep((int)Math.random()*1000);
      }
      catch(InterruptedException e) {
      }
      account[from] = namount;

      namount = account[to];
      namount += amount;
      // dito
      account[to] = namount;
    }
  }

  public void balance() {
    for(int i = 0; i < account.length; i++)
      System.out.println("Account "+ i +": " + account[i]);
  }
}
```

ou tout simplement :

`public synchronized void transfer(int from, int to, int amount) {` *turns the method into a monitor, i.e. some un-interuptable, non-interleaveable atomic instruction sequence!*

Par contre, la dernière version a l'inconvénient suivant : on ne peut pas redéfinir la méthode sans avoir le core globalement atomique...

Exercice 3 (Les 5 philosophes...)
Dans la programmation concurrentielle, une fois le problème de l'accès synchronisé aux ressources (attributs, listes, etc.) résolu, d'autres problèmes surgissent. En voici un exemple classique :



Cinq philosophes alternativement pensent et mangent. Pour manger, un philosophe doit acquérir deux fourchettes : celui de sa gauche et de sa droite. Pour des raisons de précarité, il n'y a que 5 fourchettes, une entre chaque philosophe...

Proposez une solution qui synchronise l'accès aux fourchettes en implémentant une classe `Fork` (*fourchette* en français) et une classe `Philosopher` (*philosophe* en français). La classe `Fork` devra avoir une méthode permettant à une instance de `Philosopher` de *saisir* la fourchette. Une classe `PhilosopherDemo` mettra en œuvre ces classes dans sa méthode `main` en créant 5 instances de `Fork` et 5 instances de `Philosopher`.

Voici un exemple du résultat d'exécution :

```
There they go ...
Philosopher 0 starts thinking.
Philosopher 1 starts thinking.
Philosopher 2 starts thinking.
Philosopher 3 starts thinking.
Philosopher 4 starts thinking.
Philosopher 0 starts eating.
Fork 0 grabbed by philo 0.
Fork 1 grabbed by philo 0.
Philosopher 1 starts eating.
Philosopher 2 starts eating.
Fork 2 grabbed by philo 2.
Fork 3 grabbed by philo 2.
Philosopher 3 starts eating.
Fork 0 released by philo 0.
Fork 1 released by philo 0.
Philosopher 0 starts thinking.
```

Que constatez-vous ? Comment s'appelle ce phénomène et comment l'expliquez-vous ?

Solution :

Fork :

```
public class Fork {

final Object LOCK = new Object();
    private boolean status = false; /* ungrabbed */
private int id;

public Fork(int id) {
this.id = id;
}

    public void grab(int philoId) {
      synchronized(LOCK) {
        try {
          while (status) {LOCK.wait();}
          status=true;
          System.out.println("Fork "+id+" grabbed by philo "+philoId+".");
        }
        catch(InterruptedException e) {}
      }
    }

    public void release(int philoId) {
        synchronized(LOCK) {
            if (!status) {System.out.println("Program Error: Release of Fork not grabbed!"); Systen
            status=false;
            System.out.println("Fork "+id+" released by philo "+philoId+".");
            LOCK.notifyAll();  /* necessary since two neighbors may wait, so all philosophers
                                  holding a lock must be informed ... */
        }
       }
}
```

Philosopher :

```
public class Philosopher extends Thread{
private int id;
private Fork [] forks;

public Philosopher(int id,Fork[] forks){
this.id = id;
this.forks = forks;
}

 public void run() {
    while(true){
     // THINK
     System.out.println("Philosopher "+id+" starts thinking.");
        try {
            Thread.sleep((int)Math.random()*10000);
```

```
        }
        catch(InterruptedException e) {
        }
        System.out.println("Philosopher "+id+" starts eating.");
        forks[id].grab(id);
        forks[(id+1) % 5].grab(id);
        // EAT
        try {
            Thread.sleep((int)Math.random()*10000);
        }
        catch(InterruptedException e) {
        }
        forks[id].release(id);
        forks[(id+1) % 5].release(id);
    }
  }

}
```

PhilosopherDemo :

```
public class PhilosopherDemo {

public static void main(String[] args) {
    final Fork [] forks = {new Fork(0), new Fork(1), new Fork(2), new Fork(3), new Fork(4)};


    // Creation of Philosophers
    Philosopher A0 = new Philosopher(0, forks);
    Philosopher A1 = new Philosopher(1, forks);
    Philosopher A2 = new Philosopher(2, forks);
    Philosopher A3 = new Philosopher(3, forks);
    Philosopher A4 = new Philosopher(4, forks);

    System.out.println("There they go ...");

    A0.start();
    A1.start();
    A2.start();
    A3.start();
    A4.start();
  }

}
```

Le problème qu'on observe, sooner or later, s'appelle blocage.