Programmation Concurrente en $JAVA^{TM}$

Document en cours d'élaboration issu de plusieurs versions élaborées principalement par J-P. Rigault, puis par M. Cosnard, J-F. Lalande, et F. Peix, et enfin aujourd'hui par I. Sau et M. Syska.

TP2 : Exclusion mutuelle et sémaphores en Java

14 Février 2007

1 Gestion des entrées/sorties d'un parking

Dans cet exercise on souhaite compter les entrées et sorties des véhicules dans un parking pour afficher le nombre de places occupées. Ici, pour simplifier on ne comptera que les entrées. Le fonctionnement du parking est simple:

- Il y a 2 portes d'entrée au parking (P dans le cas général).
- Le parking a capacité $2N$ ($P \cdot N$ dans le cas général).
- N voitures entrent par chaque porte du parking.

Le but de cet exercice est de compter les voitures qu'il y a dans le parking en utilisant **une seule variable partagée**. On va utiliser l'**algorithme de Peterson** pour garantir l'exclusion mutuelle entre les threads.

Idée de l'algorithme de Peterson (avec 2 threads):

```
// Chaque thread a 3 "flags" entiers (partagés) et son identificateur:
// mon_flag, flag_autre_thread, dernier, mon_id

private void entrer_section_critique (...){

    mon_flag <- 1
    dernier  <- mon_id
    while (flag_autre_thread==1  &&  dernier==mon_id) {}

}

private void sortir_section_critique (...){

    mon_flag <- 0

}
```

Corps du programme:

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


// On definit une classe pour pouvoir partager le compteur:

public class m_integer {

    private int value;

    public m_integer(){
        value=0;
    }

    public void set_value(int new_value){
        this.value = new_value;
    }

    public int get_value(){
        return this.value;
    }
}


public class Porte extends Thread {

..

}


public class Parking_Peterson {

    public static void main(String[] args) {

    ...

    }

}
```

2 Le coiffeur dormeur

Il s'agit encore d'un de ces problèmes de synchronisation mis sous une forme "plaisante". Mais celui-ci est encore plus sérieux que le problème classique des philosophes[1], car on en trouve une application presque directe dans certains mécanismes des systèmes d'exploitation (comme l'ordonnancement des accès disque).

Description du problème:

- Un coiffeur possède un salon avec un siège de coiffeur et une salle d'attente comportant un nombre fixe F de fauteuils.
- S'il n'y a pas de client, le coiffeur se repose sur son siège de coiffeur.
- Quand un client arrive:
  - s'il trouve le coiffeur endormi, il le réveille, s'assoit sur le siège de coiffeur et attend la fin de sa coupe de cheveux.
  - si le coiffeur est occupé lorsque le client arrive, le client s'assoit et s'endort sur une des C chaises de la salle d'attente.
  - si le coiffeur est occupé lorsque le client arrive et la salle d'attente est pleine, le client repasse plus tard.
- Lorsque le coiffeur a terminé une coupe de cheveux, il fait sortir son client courant et va réveiller un des clients de la salle d'attente.
- Si la salle d'attente est vide, le coiffeur se rendort sur son siège jusqu'à ce qu'un nouveau client arrive.

Le but de de cet exercice est d'associer une thread au coiffeur ainsi qu'à chaque client et de programmer une séance de coiffeur dormeur en Java, et d'utiliser les **sémaphores** pour garantir l'exclusion mutuelle parmi les processus.

2.1 Avec des sémaphores

Pour cet exercice on utilisera les sémaphores de la JDK 1.5. On écrira une classe pour le coiffeur (*BarberSemaphore*) et une classe pour les clients (*CustomerSemaphore*). L'idée est que la communication se fasse au travers de sémaphores, bloquant l'exécution des threads quand cela est nécessaire.

[1] http://fr.wikipedia.org/wiki/Dîner_des_philosophes

```
import java.util.concurrent.Semaphore;


// ======================================================================
// BarberSemaphore in Java
// ------------------------------------------------------------------
// Usage:
// javac BarberSemaphore.java
// java BarberSemaphore nbChairs nbCustomers
// ======================================================================

// Semaphore semaphore = new Semaphore(capacite);
// Method P on Semaphore "semaphore":  semaphore.acquire();
// Method V on Semaphore "semaphore":  semaphore.release();

// Simulate barber behaviors with Semaphore:


class BarberSemaphore extends Thread {

...

}


class CustomerSemaphore extends Thread {

...

}

public class Seance {

     public static void main(String[] args) {

     ...

     }
}
```

ANNEXE: Solution de la partie 4 du TP1

```
package TD1;

//=====================================================================
//Showing that synchronization between threads is needed
//---------------------------------------------------------------------
//Usage:
//javac NonSynchro_Threads.java
//java NonSynchro_Threads nloop nsize | grep BAD
//=====================================================================

// Write the same number in a given array:

class MyThreadNonSynchroWrite extends Thread {
    int id; // this writer id

    int nloop; // number of iterations

    int[] tab; // a shared array

    public MyThreadNonSynchroWrite(int id, int nloop, int[] tab, String name) {
        super(name);
        this.id = id;
        this.nloop = nloop;
        this.tab = tab;
    }

    public void run() {
        try {
            for (int i = 0; i < nloop; i++) {
                for (int j = 0; j < tab.length; j++) {
                    tab[j] = id;
                    sleep(50);
                    // yield();
                }
            }
        } catch (Exception e) {
            System.err.println("Exception in MyThreadNonSynchroWrite" + e);
        }
        System.out.println("End of " + this.getName());
        return;
    }
}
```

```
// Verify that all numbers of a given array are equal:


class MyThreadNonSynchroRead extends Thread {
    int nloop; // number of iterations

    int[] tab; // a shared array

    public MyThreadNonSynchroRead(int nloop, int[] tab, String name) {
        super(name);
        this.nloop = nloop;
        this.tab = tab;
    }

    public void run() {
        try {
            for (int i = 0; i < nloop; i++) {
                int sum = 0;
                for (int j = 0; j < tab.length; j++) {
                    sum += tab[j];
                }
                if (sum % tab.length != 0) {
                    System.out.println("BAD at i = " + i + " sum = " + sum);
                }
            }
        } catch (Exception e) {
            System.err.println("Exception in MyThreadNonSynchroRead" + e);
        }
        System.out.println("End of " + this.getName());
        return;
    }
}
```

```
public class Non_Synchro_Threads {

    public static void main(String[] args) {

        try {

            // read command-line arguments

            if (args.length != 2) {
                System.err.println("usage: NonSynchro_Threads nloop nsize");
                System.exit(1);
            }
            int nloop = Integer.parseInt(args[0]);
            int nsize = Integer.parseInt(args[1]);
            System.err.println("Starting NonSynchro_Threads for nloop = "
                    + nloop + " and nsize = " + nsize);
            int[] tab = new int[nsize];


            // thread creation

            Thread th1 = new MyThreadNonSynchroWrite(0, nloop, tab, "Writer 0");
            Thread th2 = new MyThreadNonSynchroWrite(1, nloop, tab, "Writer 1");
            Thread th3 = new MyThreadNonSynchroRead(nloop, tab, "Reader 0");


            // starting threads

            th1.start();
            th2.start();
            th3.start();

        } catch (Exception e) { // report any exceptions
            System.err.println("Exception in NonSynchro_Threads.main" + e);
        }
    }
}
```