TD nº 4 - Synchronisations et sémaphores

Dans ce TD, nous allons voir comment les *moniteurs* et *sémaphores* peuvent être utilisés en *Java* pour réaliser l'exclusion mutuelle et / ou pour synchroniser les tâches.

Exercice 1. *Exlusions mutuelles avec "Synchronized" et coopérations*

Nous allons partager deux informations : un nombre `n` et son carré `carre` entre deux `threads`. Le premier incrémente `n` et calcul son carré dans `carre` ; le second thread se contente d'afficher `n` et le contenu de `carre`.

Ici, les deux informations (`n` et `carre`) sont regroupées dans un objet `nomb` de type `Nombres`. Cette classe dispose de deux méthodes :

— void `calcul()` qui incrémente `n`, fait une pause de 100 msec et calcul la valeur de `carre`,
— void `affiche()` qui affiche les valeurs de `n` et de `carre`.

On crée de deux `threads` de deux classes différentes :

— `calc` de la classe `Calcul` qui appelle, à son rythme (`sleep(100)`), la méthode `calcul()` de `Nombres`,
— `aff` de la classe `Affiche` qui appelle à son rythme (différent de celui de `calc`, par exemple `sleep(150)`), la méthode `affiche()` de `Nombres`.

Les deux `threads` sont lancés par le `main()` et s'arrêtent après 10 itérations.

1. Implémentez le programme décrit ci-dessus. Expliquez les résultats.

2. Pour rÃl'aliser des exclusions, mettez les méthodes `calcul()` et `affiche()` de `Nombres` en "synchronized". Essayez les classes et expliquez les résultats.

Jusqu'à présent, les deux `threads` n'étaient pas vraiment synchronisés. On modifie maintenant le code pour que les deux `threads` soient coordonnés, malgré leurs rythmes différents (i.e. on effectue alternativement une incrémentation et un calcul). Pour ce faire, on utilisera les méthodes `wait()` et `notifyAll()`, ainsi qu'un indicateur booléen `pret` permettant aux deux `threads` de communiquer entre eux.

3. Modifiez le code précédent afin de synchroniser les deux `threads`.

Exercice 2. *Sémaphores pour la synchronisation*

Pour cette exercice, on va prendre la classe simple suivante :

```
class ATache implements Runnable  { // implÃl'mentation de l'interface Runnable
  int nom;   // nom de la tÃćche
  int index; // index de la boucle d'affichage
  /** constructeur
  * @param ident le nom de la tÃćche, entier >= 1  */
  public ATache (int ident){
    this.nom = ident;
    this.index = 1;
  }
  public void run () {
    System.out.println(" dÃl'but tÃćche T"+this.nom);
          Random rand = new Random();
    int pause;
          while (index <= 30){
      try{
    System.out.println("indice: " + index + ", tÃćche T" + this.nom);
                pause = rand.nextInt(max - min + 1) + min;
```

```
    Thread.sleep(100);
    index++;
    } catch(InterruptedException e) {
        System.out.println("Interrupted Exception caught");}
     }
    System.out.println("Fin tÃćche T"+this.nom);
  }
 }
```

On veut contrôler les exécutions des trois tâches T1, T2 et T3 réalisées par trois instances distinctes de la classe `ATache` pour garantir que pour chaque valeur de l'indice, les tâches sont exécutées dans l'ordre T1, T2, T3 : (T1, $i = 1$), (T2, $i = 1$), (T3, $i = 1$), (T1, $i = 2$), (T2, $i = 2$), (T3, $i = 2$), (T1, $i = 3$), etc.

Pour réaliser cet objectif de synchronisation, nous allons utiliser la classe `Semaphore` [1].

Les méthodes `acquire()` et `release()` sur un objet de la classe `Semaphore` sont équivalentes aux primitives `P()` et `V()` vues en premiére année.

Le constructeur de la classe `Semaphore` prend deux paramétres : un entier pour la valeur initiale du sémaphore et un booléen. La valeur `true` permet une gestion FIFO des tâches en attente sur ce sémaphore.

Exemple d'utilisation :

```
Semaphore sem1;
sem1 = new Semaphore(0, true);  // arret imm\'ediat de la tache au premier P()
sem1.acquire();                 // P() une valeur peut etre donn\'ee en param\'etre
sem1.release();                 // V() une valeur peut etre donn\'ee en param\'etre
```

Les fonctions `acquire()` et `release()` peuvent également recevoir un entier comme argument. `acquire(2)` ne passe que si la valeur du sémaphore est au moins 2 (la valeur est alors décrémentée de 2), `release(2)` ajoute 2 á la valeur courante du sémaphore. Si l'on ne donne pas d'arguments les fonctions se comportent comme si l'argument était 1.

Remarque : Plusieurs tâches peuvent partager un même objet créé préalablement si celui-ci est passé en argument á la création de l'instance correspondant á la tâche. Les données d'un tel objet ne sont pas perdues lorsque la tâche se termine.

Deux paramétres vont être ajoutés au constructeur de la classe `ATache`. Le premier sémaphore (`prive`) sera celui que la tâche doit surveiller pour savoir quand elle a le droit de poursuivre son exécution. Le second (`voisin`) sera celui que la tâche suivante attend pour continuer.

```
public ATache (int nom, Semaphore prive, Semaphore voisin)
```

1. En utilisant des sémaphores, modifiez votre programme pour garantir que l'exécution des trois tâches T1, T2 et T3 se fasse de maniére réguliérement intercalée.

Indication : Il faut créer trois sémaphores dans la fonction `main` (un pour chaque tâche), puis donner á chaque tâche son sémaphore et celui de la tâche suivante au moment de sa création.

á chaque itération de la boucle dans la fonction `run` de la classe `ATache`, il faut commencer par demander si le sémaphore est disponible, exécuter une itération de la boucle puis libérer le sémaphore de la tâche suivante (pour qu'elle puisse á son tour effectuer une itération de la boucle).

Attention á l'intialisation de chacun des trois sémaphores...

2. Modifiez le programme pour que les tâches s'alternent dans l'ordre inverse : T3, T2 puis T1.

3. Que se passe t-il si le groupe de tâches est initialisé avec une seule tâche exécutable á la fois ?

4. Ajoutez un sémaphore `semfin` pour que l'affichage « fin tâche principale » se trouve aprés les affichages des trois tâches.

Indication : Il y a plusieurs solutions. Soit on utilise le fait que les exécutions des tâches sont maÃőtrisées et que l'on sait donc celle qui doit se terminer en dernier pour faire en sorte que ce soit cette

1. Pour utiliser la classe `Semaphore` :
`import java.util.concurrent.Semaphore;`

derniére tâche qui libére le sémaphore `semfin`, soit on initialise le sémaphore de telle sorte qu'il ne devienne passant qu'aprés avoir été libéré par chacune des trois tâches.

Exercice 3. *Pertes de mises à jour, exclusion mutuelle*

Dans ce dernier exercice, nous allons examiner les problèmes causés par l'utilisation concurrente des ressources. Pour cela, on prend un exemple simple : plusieurs threads d'un programme utilise les objets (une chaîne et un entier) statiques partagés. Voici le code correspondant :

```
public class partage extends Thread {
    private static String chaine = "";
    private static int cpt = 0;
    private String nom;
    partage ( String s )  { nom = s;   }
    public void run() {
       for (int i = 0; i<10; i++) {
          maj(nom);
        try {
             Thread.sleep(100);  // milliseconds
            } catch(InterruptedException e) {}
        }
    }
   public void maj(String nn){   // mises a jour
        chaine = chaine + nn;
        cpt++;
   }
    public static void main(String args[]) {
        Thread T1 = new partage( "T1 " );
        Thread T2 = new partage( "T2 " );
        Thread T3 = new partage( "T3 " );
        T1.start();
        T2.start();
        T3.start();
        try {
            T1.join();
            T2.join();
            T3.join();
           } catch (InterruptedException e) {     }
        System.out.println( chaine );
        System.out.println( cpt );
    }
}
```

1. Copiez et compilez le code. Que se passe t-il quand vous exécutez ce programme ? Expliquez les résultats.

2. Modifiez le programme en assurant l'exécution exclusive de la section critique en utilisant une méthode `synchronized`.

3. Réalisez maintenant l'exclusion mutuelle avec des sémaphores.