STI 2e année – POO avancée: Java

TD: Thread et deadlocks

1 Les premiéres « threads »

Exercice 1 Définir une classe extension de `Thread` qui recoit en paramétre de construction un nom et un entier n. Son corps est une boucle, parcourue 20 fois, qui affiche son nom sur la sortie standard puis se met en attente pendant n secondes.

Exercice 2 Ecrire un programme principal qui recoit en argument de la ligne de commande deux entiers n_1 et n_2. Ce programme crée 2 threads du type précédents, en passant n_1 à la premiére et n_2 à la seconde. Puis il lance les deux threads et attend leur fin.

Exercice 3 Etudier les effets des variations relatives de n_1 et n_2 sur l'ordre d'exécution des deux threads.

Solution exercice 3

```
// First threads in Java
// Jean-Paul Rigault --- ESSI ---
// Usage:
// javac First_Threads.java
// java First_Threads n1 n2

class MyFirstThread extends Thread {

  final int nloop = 20;  // number of iterations
  int n = 0;   // sleep time in second

  public MyFirstThread(int n, String name) {
    super(name);
    this.n = n;
  }

  public void run() {
    try {
      for (int i = 0; i < nloop; i++) {
        System.out.println(getName());
        sleep(n*1000); // parameter of sleep() in ms
      }
    }
    catch (Exception e) {
      System.err.println("Exception in MyThread" + e);
    }
  }

}

public class First_Threads {

  public static void main(String[] args) {
    try {
      // read command-line arguments
      if (args.length != 2) {
        System.err.println("usage: First_Threads n1 n2");
        System.exit(1);
      }
      int n1 = Integer.parseInt(args[0]);
      int n2 = Integer.parseInt(args[1]);
      System.err.println("Starting First_Threads for n1 = "
          + n1 + " and n2 = " + n2);

      // thread creation
      Thread th1 = new MyFirstThread(n1, "Thread 1");
      Thread th2 = new MyFirstThread(n2, "Thread 2");

      // starting threads
      th1.start();
      th2.start();

      // waiting for threads to terminate
      th1.join();
      th2.join();

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

}
```

2 Ordonnancement des « threads »

Cet exercice simple vise à montrer que les threads Java sont effectivement ordonnancées : elles peuvent etre préemptées à tout moment par le systéme ce qui provoque un entrelacement (apparemment spontané) de leur exécution. En fait l'ordonnancement se fait par « tranche de temps »[1] : on accorde une durée d'exécution continue maximale à chaque thread ; quand cette durée est atteinte, la thread est préemptée, pour laisser leur chance à d'autres threads de priorité égale ou supérieure ; elle sera reprise plus tard, bien sur.

Exercice 4 On reprend la structure à deux threads de l'exercice précédent que l'on modifie de la facon suivante :

- le programme principal ne recoit cette fois qu'un seul argument, le nombre n de tours de boucle que chacune des deux threads doit effectuer (c'est le même nombre pour les deux) ;
- on retire l'instruction d'attente (`sleep()`) de la boucle du corps de la méthode `run()` ; les threads vont donc se dérouler en continu, sans auto-préemption.

En faisant croitre n de maniére (vaguement) exponentielle on montrera qu'à partir d'une certaine valeur, les deux threads ne s'exécutent plus l'une après l'autre mais entrelacent leur exécution.

Si l'on a pris soin de ne pas écrire sur la sortie standard autre chose que ce qui est demandé ici, on pourra compter le nombre de « changements de main » en filtrant la sortie standard du programme par le script shell **nContextSwitches** suivant :

```
#! /bin/sh

# Count the number of time when a line from stdin differs
# from the previous line

awk '{if (prev != $0) {prev = $0; print "changement"}}' | wc -l
```

1. Dans les premiéres versions de Java, ce comportement n'était pas garanti sur tout systéme d'exploitation ; depuis Java 2 (version 1.2 et supérieur du JDK) il l'est.

Solution exercice 4

Ainsi, si votre `main()` est dans la classe `TimeSliced_Threads`, l'exécution au terminal aura la forme suivante :

```
borobudur% java TimeSliced_Threads 1000 | nContextSwitches
6
borobudur%
```

La valeur 6 indique qu'il y a eu de l'ordre de 4 préemptions « spontanées » (cette commande compte deux changements « de trop », c'est-à-dire que la valeur 2 indique une absence de préemption). Notez que les résultats de ces expériences varient énormement d'une exécution à l'autre et évidemment d'une machine à l'autre, meme avec des paramétres identiques.

```
// Second threads in Java
// JFL --- ENSIB ---
// Usage:
// javac Second_Threads.java
// java Second_Threads n1

class MyFirstThread extends Thread {

  int nloop = 20;  // number of iterations

  public MyFirstThread(int n, String name) {
    super(name);
    this.nloop = n;
  }

  public void run() {
    try {
      for (int i = 0; i < nloop; i++) {
        System.out.println(getName());
      }
    }
    catch (Exception e) {
      System.err.println("Exception in MyThread" + e);
    }
  }

}

public class Second_Threads {

  public static void main(String[] args) {
    try {
      // read command-line arguments
      if (args.length != 1) {
        System.err.println("usage: Second_Threads n1");
        System.exit(1);
      }
      int n1 = Integer.parseInt(args[0]);
      System.err.println("Starting Second_Threads for n1 = "
          + n1);

      // thread creation
      Thread th1 = new MyFirstThread(n1, "Thread 1");
      Thread th2 = new MyFirstThread(n1, "Thread 2");

      // starting threads
      th1.start();
      th2.start();

      // waiting for threads to terminate
      th1.join();
      th2.join();

    }
    catch (Exception e) { // report any exceptions
```

3 Barriéres

Question 5 Testez les `CyclicBarrier` pour synchroniser (rendez-vous) vos threads précédents. Affichez un "J'attends les autres" avant la barriére et un "On est tous débloqué!" juste aprés.

4 Deadlock

Le but de cet exercice est de créer un deadlock (une fois n'est pas coutume!).

Question 6 Créez une classe `Ressource` ne contenant rien. Créez deux objets `r1` et `r2` dans votre main et passez ces deux objets à la construction des deux Threads précédents. Vous stockerez ces deux objets en attributs privés des Threads.

Question 7 Réalisez alors l'algorithme suivant dans le Thread 1 :

```
prendre un moniteur sur r1;
attendre 3s;
prendre un moniteur sur r2;
afficher "Je suis 1 et j'ai les deux ressources !"
```

Question 8 Réalisez alors l'algorithme suivant dans le Thread 1 :

```
attendre 1s;
prendre un moniteur sur r2;
attendre 3s;
prendre un moniteur sur r1;
afficher "Je suis 2 et j'ai les deux ressources !"
```

Exercice 9 Constatez le deadlock.

Exercice 10 Supprimez les temporisations. Vous constaterez que le deadlock disparait car la probabilité qu'il y ait entrelacement de la prise des deux moniteurs est faible.

Exercice 11 Faites des boucles trés longues faisant itérer la prise de ressources. Un deadlock devrait survenir à plus ou moins long terme.

Question 12 Résolvez le deadlock en changeant d'implémentation et en utilisant un objet implémentant l'interface Lock au lieu d'utiliser ***synchronized***, par exemple un Lock réentrant.