TP4 Threads

Développement Logiciel (L2-S4)

Vendredi 14 février 2014

Le but de ce TP est de vous familiariser avec différents aspects des threads en Java.

Préambule Récupérez les sources à l'adresse suivante :
`http://www.lri.fr/~garcia/devlog/tp4.zip`
Prenez le temps de lire toutes les explications, de vous référer à vos polycopiés de cours et au wiki (`https://www.lri.fr/~anab/teaching/DevLog`).

Très important : Ce TP est noté. Avant de partir, vous serez noté par votre chargé de TP sur les critères suivants : Questions réussies (questions qui fonctionnent et qui respectent le sujet du TP) ; Utilisation des concepts du cours (visibilité des variables et méthodes, héritage, exceptions, Threads, entrées sorties ...) ; Clareté du code (Lisibilité, commentaires).

Vous avez également la possibilité de terminer le TP chez vous si ce n'est pas le cas durant la séance. Dans ce cas, vous devez envoyer les sources dans une archive NomPrenomTP4.zip à votre chargé de TP. Le titre du mail doit être [DevLog] TP4 de Nom Prenom

Exercice 1 : Ordre de passage

Le programme donné dans les sources créé deux instance de la classe TwoThread qui affichent chacun 20 fois une chaine de caractère.

- 1. Pouvez vous prévoir la sortie console ? Lancer le programme plusieurs fois, qu'observez vous ?
- 2. La classe TwoThread hérite de Thread pour fonctionner. Cependant, on préfère souvent utiliser l'interface Runnable afin de pouvoir hériter d'une autre classe que Thread. Programmer une version de la classe TwoThread avec l'interface Runnable et modifier le main pour que cela fonctionne. Aide : la classe Thread dispose d'un constructeur *Thread (Runnable runable)*
- 3. La méthode *Thread.yield()* permet à un thread de passer la main à un autre thread de même priorité. Utiliser cette fonction pour que les threads alternent dans l'exemple précédent. L'alternance est elle bien faite ?

Exercice 2 : Modification de plusieurs variables en concurrence

On souhaite créer deux threads qui change le même champs d'un même objet.

```
public class Test {
  int value;

  public static void main(String[] args) {
    final Test test=new Test();

    for(int i=0;i<2;i++) {
```

```
      final int id=i;
      new Thread(new Runnable() {
        public void run() {
          while(true) {
            test.value=id;
            if (test.value!=id)
              System.out.println("id "+id+" "+test.value);
          }
        }
      }).start();
    }
  }
}
```

1. Qu'affiche le code ? Pourquoi ?
2. Pourquoi l'affichage n'évolue plus au bout d'un laps de temps ? Peut-on en déduire qu'il n'y a plus de problème de concurrence ?
3. Créer des sections critiques en utilisant un bloc synchronized là où il faut pour que chaque thread voit les modifications effectuées sur une variable par l'autre thread.

Exercice 3 : Producteur consomateur

On s'intéresse maintenant au problème du producteur consommateur dans lequel un thread produit des ressources et un autre thread les consomme.
Le producteur écrit entre 3 et 14 caractères au hasard dans un *String*. Uniquement lorsque le String n'est pas "plein", cela veut dire : n'a plus que 40 caractères. Le consommateur affiche le contenu ce String à l'écran, uniquement lorsqu'il est plein, et le vide ensuite. La classe *ProdConso* donnée ci-dessous lance le programme.

```
public class ProdConso {
     public static void main(String[] args) {
     MonitoredBuffer mon = new MonitoredBuffer();
     Consumer conso = new Consumer(mon);
     Producer prod = new Producer(mon);

     Thread c = new Thread(conso);
     Thread p = new Thread(prod);

     System.out.println("Démarrage");
     p.start();
     c.start();
     }
}
```

Voici un exemple d'exécution :

```
Démarrage
Producteur ajoute 1b|g NbElements:4
Producteur ajoute b?9)@@3{] NbElements:9
Producteur ajoute db?hg NbElements:5
Producteur ajoute O_j‘F+y NbElements:7
Producteur ajoute Wn^6(Sl2 NbElements:8
Producteur ajoute whcY=> NbElements:6
```

```
Producteur ajoute LZ@;^" NbElements:6
Consommateur prends:1b|gb?9)@@3{]db?hgO_j‘F+yWn^6(Sl2whcY=>LZ@;^"
Producteur ajoute ~@aAo^[ NbElements:7
Producteur ajoute 1Uylszxy# NbElements:9
Producteur ajoute |!o%? NbElements:5
Producteur ajoute cJw\T NbElements:5
Producteur ajoute ,1P_o< NbElements:6
Producteur ajoute ?iQ\e* NbElements:6
Producteur ajoute J7o(R% NbElements:6
Consommateur prends:~@aAo^[1Uylszxy#|!o%?cJw\T,1P_o<?iQ\e*J7o(R%
```

– 1. Créez les classes *Producer* ; *Consumer* et *MonitoredBuffered* afin de faire fonctionner le programme.

– AIDE : La classe *MonitoredBuffered* doit disposer d'une méthode *put(String texte)* qui remplit la chaine de caractère et d'une méthode *get(String texte)* qui lit puis efface la chaine de caractère.
Ces méthodes doivent disposer d'un *lock*, c'est à dire d'un membre *final Object LOCK = new Object() ;* de la classe qui servira a indiquer si les fonctions peuvent être appelées.
En particulier on utilisera les méthodes *LOCK.wait()* qui fait attendre un thread si besoin et *LOCK.notify()* signale la fin du blocage aux autres threads en attente.

Exercice 4 (Bonus) Bloqué ?

On vous fournit le code ci-dessous :

```
public class Oups {
  private int value;

  public void setValue(int value) {
    synchronized(readLock) {
      synchronized(writeLock) {
        this.value=value;
      }
    }
  }

  public int getValue() {
    synchronized(readLock) {
     return value;
    }
  }

  public void performs() throws InterruptedException {
    Thread t=new Thread() {
      @Override public void run() {
        setValue(12);
      }
    };

    synchronized(writeLock) {
      t.start();
      Thread.sleep(1000);

      System.out.println(getValue());
```

```
    }
  }

  private final Object readLock=new Object();
  private final Object writeLock=new Object();

  public static void main(String[] args) throws InterruptedException {
    Oups  oups=new Oups();
    test.performs();
  }
}
```

– 1. Executez le code et expliquer ce qu'il se passe. Où se situe l'interblocage ? Proposez une solution.