Thread, Section critique, deadlock et executor

Exercice 1 - Modification de plusieurs variables en concurrence

On souhaite créer deux threads qui change les coordonées d'un même point

```
public class Point {
  private int x;
  private int y;

  public void move(int x,int y) {
    this.x=x;
    this.y=y;
  }

  @Override public String toString() {
    return "("+x+','+y+')';
  }

  public static void main(String[] args) {
    final Point p=new Point();

    for(int i=0;i<2;i++) {
      final int id=i;
      new Thread(new Runnable() {
        public void run() {
          for(;;) {
            p.move(id,id);
            System.out.println(p);
          }
        }
      }).start();
    }
  }
}
```

1 Vérifier en utilisant `grep` que le code affiche des points (0,1) ou (1,0).
Expliquer pourquoi.

2 Créer des sections critiques en utilisant un bloc `synchronized` là où il faut pour que seuls des points (0,0) et (1,1) puisse être affichés.

3 Expliquer pourquoi il n'est pas conseiller de synchronizé sur `this` et modifié votre code en conséquence.

4 Pourquoi les champs x et y n'on pas besoin d'être déclarés `volatile`.

Exercice 2 - Section critique et Verrou

1 Expliquer dans quels cas il est préférable d'utiliser un lock ou un bloc synchronized.

2 Créer une classe `LockedPoint` ayant la même sémantique de la classe `Point` de l'exercice précédent mais en utilisant un verrou de la classe `java.util.concurrent.locks.ReentrantLock`

Exercice 3 - Interblocage

Voici un code simplifié d'un code fournit par un étudiant :

```
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();
    }
  }
```

Exécuter le code ci-dessus et expliquer.

Où se situe l'interblocage ?

La machine virtuelle `hotspot` posséde un mécanisme de détection des interblocages taper lors de l'exéctuion Ctrl+\ sous linux (resp. Ctrl+Break sous windows) et vérifier avec ce que vous aviez pronostiqué.

Exercice 4 - Interblocage plou dur

Où se situe l'interblocage dans le code ci-dessous :

```
  public class FunnyDeadLock {
    static synchronized void m() {
      System.out.println("hello");
    }

    static {
      Thread t=new Thread() {
        @Override public synchronized void run() {
          m();
        }
      };
      t.start();

      try {
```

```
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }

      synchronized(t) {
        System.out.println("hello 2");
      }
    }
  }
```

Exercice 5 - Travail en parallèle

On souhaite écrire un programme affichant les valeurs des racines carrées de 0 à 10 000. On souhaite paralléliser le programme et permettre d'executer en même temps plusieurs calculs, chaque thread devra par exemple effectuer le calcul d'une centaine de racine carré.

1 Créer un `ExecutorService` en utilisant la classe `Executors` permettant de distribuer le calcul sur 5 threads différentes.

2 Écrire le programme qui effectue le calcul en parallèle en utilisant des `Runnable` qui effecteut l'affichage.
   On permet ici que l'affichage ne s'effectue pas dans l'ordre des racines carrés croissantes.
   Penser à utiliser la méthode `shutdown` pour indiquer qu'il n'y a plus de calcul à effectuer.

3 Modifier votre programme en utilisant l'interface `Callable` à la place de `Runnable` et le mécansime de `Future` pour effectuer l'affichage dans l'ordre.

Exercice 6 - Find en parallèle [à la maison]

On souhaite permettre à un utilisateur de rechercher un fichier dans une sous-arborescence spécifiée en utilisant une expression régulière.

1 Écrire le programme qui prend en paramètre une expression régulière et qui affiche l'ensemble des fichiers du répertoire courant dont le nom vérifie l'expression régulière.
   On utilisera pour cela les classes `java.util.regex.Pattern` et `java.util.regex.Matcher` ainsi que la méthode `listFiles` de la classe `java.io.File`.

2 On souhaite maintenant étendre le parcours à l'ensemble des sous répertoires en effectuant le parcours de chaque répertoire de façon concurrente.
   Comment doit-on faire ?

3 Discuter des différents arguments qu'il faut fournir au `ThreadPoolExecutor` pour que l'on puisse l'utiliser pour effectuer la recherche.

4 Écrire le programme qui effectueut la recherche parallèle.
   Attention, pour que le programme s'arrête il faut que :
   - Le nombre de `core pool thread` soit zéro lors de la construction.
   - Les `worker threads` meurent après un temps d'inactivité.