Programmation répartie

TD n°2 — *Mémoire partagée*

Exercice 1. Synchronized Voici un programme créant deux threads s'exécutant en parallèle.

```
public class O {

    int m=100;

    public synchronized void m1(int id){
  System.out.println("begin m1");
  for(int i=0;i<m;i++){
      System.out.println(id+" m1 "+i);
  }
  System.out.println("end m1");

    }

    public synchronized void m2(int id){
  System.out.println("begin m2");
  for(int i=0;i<m;i++){
      System.out.println(id+" m2 "+i);
  }
  System.out.println("end m2");
    }



}
```

```
class T1 extends Thread{

    O o;

    public T1(O o){
  this.o= o;
    }

    public void run(){
  o.m1(1);
    }
```

```
}
```

```
class T2 extends Thread{

    O o;

    public T2(O o){
  this.o= o;
    }

    public void run(){
  o.m1(2);
    }


}
```

```
public class Main{


    public static void main(String args[]){
  O o = new O();
  T1 t1 = new T1(o);
  T2 t2 = new T2(o);
  t1.start();
  t2.start();


    }

}
```

1.1 Les exécutions de la fonction `m1` lancée par `t1` et `t2` peuvent elles s'entrelacer ? Pourquoi ?

1.2 La méthode `run` de `T2` est modifiée en remplaçant le corps de la méthode par un appel de `o.m2()`. Les exécutions de `m1` de `t1` et `m2` de `t2` peuvent elles s'entrelacer ?

1.3 Dans la méthode `main`, on crée désormais deux objets de type `O`, `o1` et `o2`. `o1` est donné en paramètre à `t1` et `o2` à `t2`. Les exécutions de `t1` et `t2` peuvent elles s'entrelacer ?

1.4 Si l'on appelle la méthode `m2` dans `m1`. Le thread peut-il être bloqué lorsqu'il exécute `m1` ?

1.5 Les threads ont désormais deux attributs de types `O`. Modifier les méthodes, pour provoquer un potentiel interblocage.

Programmation répartie (TD n°2)

Exercice 2. Compte bancaire

Construire une modélisation d'un compte joint. Sur ce compte il est possible d'effectuer un retrait ou un dépôt, et de consulter le solde. Tous ces appels pourront être fait de façon concurrente.

Exercice 3. Producteur-consommateur

Nous souhaitons développer une application de type producteur-consommateur. Cette application partage une donnée entre deux threads : le producteur, qui crée la donnée, et le consommateur qui la récupère et la traite. La coordination est essentielle, le consommateur ne doit pas récupérer la donnée avant que le thread producteur la produise, et le producteur ne doit pas fournir une nouvelle donnée avant que l'ancienne n'ai été récupérée par le consommateur.

3.1 Implantez cette solution avec un thread pour le producteur, et un thread pour le consommateur.

3.2 Modifiez votre solution pour pouvoir gérer plusieurs consommateurs.

3.3 Maintenant il est possible d'avoir n données en même temps. Lorsque n données existent, aucune nouvelle donnée ne doit être créée par le producteur. Modifiez votre solution pour intégrer cette nouvelle contrainte.

3.4 Modifiez votre solution pour avoir plusieurs producteurs.

Exercice 4. Dîner des philosophes

Cinq philosophes se trouvent autour d'une table. Chacun a devant lui un plat de spaghetti. A gauche de chaque plat se trouve une fourchette. Il n'y a donc en tout cinq fourchettes. Les philosophes alternent les états suivant : réflexion, affamé et dégustation. Lorsqu'un philosophe est affamé il souhaite manger. Pour cela, il doit récupérer deux fourchettes, celle se trouvant à sa droite et celle se trouvant à sa gauche. Si elles ne sont pas disponibles, il reste affamé.

4.1 Identifier les threads, et les ressources

4.2 Modéliser le problème, en prenant des temps aléatoires pour la réflexion et la dégustation. Ajoutez des sorties pour visualiser les états des philosophes.