Programmation et modélisation orientées objet
(USAL5A & USAL2H)

Concurrence et parallélisme

Considérons un programme qui doit exécuter deux tâches. Si ces tâches sont indépendantes (aucune des deux tâches n'a besoin du résultat de l'autre), elle peuvent être exécuter de manière « concurrente » : dans un ordre arbitraire, ou en « entrelaçant » les deux exécutions, ou encore de manière « parallèle » sur deux cœurs, ou deux processeurs ou deux ordinateurs différents.

En Java, les « activités » ou « fils d'exécution » (*threads*) font partie du langage, et permettent donc d'implanter des applications concurrentes portables. De plus, les activités Java reposent sur celles du système d'exploitation, et elles profitent donc des multi-processeurs et multi-cœurs. Elles permettent donc aussi de parallèliser les calculs (pour gagner ainsi en temps d'exécution).

Dans ce TP, nous étudions comment créer des activités en Java, et comment profiter facilement de cette l'exécution parallèle en utilisant le concept de calcul asynchrone, appelé *future* quand il a été introduit dans les années 70 pour le langage Lisp (et qui correspond aussi à la notion de *promise* de JavaScript).

1 Interface Runnable et classe Thread

Pour créer une activité concurrente en Java, il faut d'abord implanter l'interface `Runnable` de la JDK, qui contient une seule méthode `run` :

```
public interface Runnable {

   void run();

}
```

La méthode `run` sera celle exécutée quand l'activité sera lancée. Supposons que l'on ait deux classes `Task1` et `Task2` qui implantent l'interface `Runnable`, dont on crée des instances :

```
Runnable task1 = new Task1();
Runnable task2 = new Task2();
```

Il est alors possible de créer une activité (un *thread*) pour chacune de ces tâches en utilisant la classe `Thread` de la JDK ainsi :

```
Thread thread1 = new Thread(task1);
Thread thread2 = new Thread(task2);
```

et de les lancer de manière concurrente :

```
thread1.start();
thread2.start();
```

L'exercice suivant a pour but d'observer cette exécution concurrente.

1. Implanter la classe `Task1` où la méthode `run` affiche 5000 fois le message `"1"`.

2. Implanter la classe `Task2` où la méthode `run` affiche 5000 fois le message `"2"`.

3. Compléter la fonction `main` et tester ce code. Qu'observez-vous ?

4. Regrouper les classes `Task1` et `Task2` en une seule classe `Task` dont le constructeur prend le message à afficher en paramètre.

2 Interface Future et parallélisme

L'interface `Future<T>` de la JDK représente un calcul « asynchrone », qui doit ultimement (à un moment dans le futur) renvoyer un résultat de type `T`. L'interface `RunnableFuture<V>` de la JDK étend en plus l'interface `Runnable`, et la classe `FutureTask<V>` implante cette interface.

Si `e` est une expression de type `V`, on peut créer un tel calcul en utilisant simplement le constructeur ainsi : `new FutureTask<V>(() -> e)`. La syntaxe `() -> e` correspond à une fonction anonyme sans argument (qui implante donc l'interface `Callable<T>`). Ce calcul peut alors être exécuté de manière concurrente.

On peut aussi stocker ce calcul en cours d'execution, par exemple dans une variable `x`, et il est possible ensuite de récupérer le résultat avec l'expression `x.get()`, en précisant éventuellement le temps que l'on est prêt à attendre pour obtenir le résultat. La méthode `get()` est une opération bloquante.

1. Pour faire simuler des tests avec de « longs » calculs, nous donnons l'implantation volontairement naïve suivante de la suite de Fibonacci :

   ```
   static Integer fib(Integer n) {
       if (n <= 1) return 1;
       else return fib(n - 1) + fib(n - 2);
   }
   ```

   Le plus grand terme de la suite calculable avec des entiers 32 bits signés est `fib(45)` et le temps de calcul est de quelques secondes.

2. Implanter une méthode statique qui crée une `FutureTask<V>` puis lance son calcul de manière asynchrone en utilisant un nouveau `Thread` par tâche :

   ```
   static <V> Future<V> futureTask(Callable<V> callable)
   ```

3. Faire le test simple suivant :

   ```
   Future<Integer> x = futureTask(() -> fib(45));
   Integer r = x.get();
   ```

4. Faire le test suivant, qui calcule une liste de valeur en parallèle :

   ```
   SimpleList<Future<Integer>> l = SimpleList.of(
               futureTask(() -> fib(43)),
               futureTask(() -> fib(44)),
               futureTask(() -> fib(45)));

   for (Future<Integer> f : l) System.out.println(f.get());
   ```

3 Application : tri fusion parallèle

L'objectif de cette partie est d'implanter un tri fusion parallèle (où, récursivement, les deux moitiés de la liste sont à chaque fois triées en parallèle).

1. Importer les méthodes statiques `take` et `drop` déjà vues auparavant :

   ```
   static <A> SimpleList<A> take(SimpleList<A> l, Integer i)

   static <A> SimpleList<A> drop(SimpleList<A> l, Integer i)
   ```

2. Implanter une méthode statique `merge` qui fusionne deux listes d'entiers supposées triées en préservant l'ordre des éléments et retourne donc une liste triée.

```
static SimpleList<Integer> merge(SimpleList<Integer>l1, SimpleList<Integer> l2)
```

3. Implanter le tri fusion de manière récursive.

   ```
   static SimpleList<Integer> mergeSort(SimpleList<Integer> l)
   ```

4. Modifier votre tri fusion récursif pour obtenir une version parallèle (où, récursivement, les deux moitiés de la liste sont à chaque fois triées en parallèle).

   ```
   static SimpleList<Integer> mergeSortPara(SimpleList<Integer> l)
   ```

5. (*optionnel*). Donner une version imperative de votre code qui utilise les listes de la JDK.