Programmation répartie. Module 4102C
TP 1 : les threads
année universitaire 2018-2019

Samuel Delepoulle – Franck Vandewiele

Janvier 2019

Les Threads en java

Introduction

Les threads permettent de concevoir des programmes "multi-tâches", c'est à dire des programmes qui réalisent plusieurs actions simultanément. Plusieurs threads peuvent être exécutés en même temps, sur la même machine virtuelle. La notion de thread est proche de celle de processus Unix. Attention cependant car les thread partagent la même mémoire ce qui n'est pas forcément le cas pour les processus. Il est possible de lancer un thread sans pour autant arrêter les autres, de leur donner une priorité, de les faire attendre, de les synchroniser . . .Mais ceci pose également des problèmes d'accès concurrents aux même données.

Sur une machine mono-processeur, les threads ne s'exécutent pas exactement en même temps. Dans la pratique, chaque thread va accéder à une partie du temps du processeur et des ressources de la machine. Si le découpage du temps processeur est fin, cela donnera l'impression que les threads s'exécutent en parallèle. Mais il ne faut pas oublier ce fonctionnement car un thread qui a une priorité importante peut bloquer le fonctionnement des autres : un thread doit toujours laisser aux autres threads une chance de s'exécuter.

En Java, le programme principal est lui-même un thread mais il est possible de faire démarrer d'autres threads sans l'interrompre. Dans la pratique, le codage de thread peut se faire de deux façon :

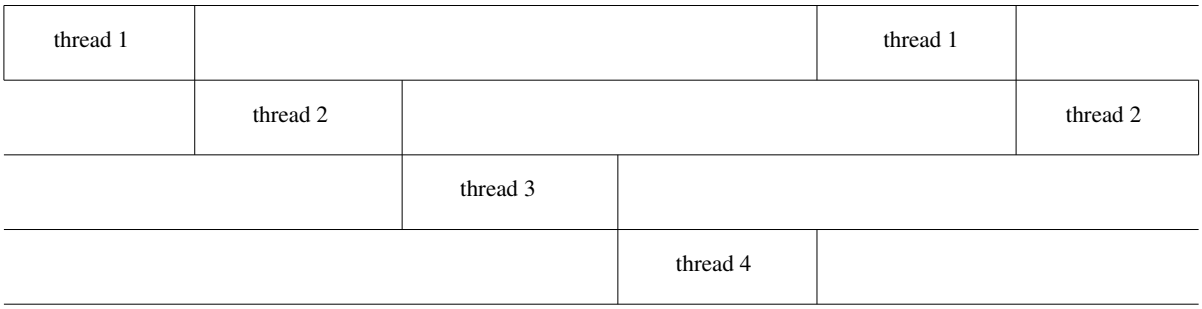

FIGURE 1 – Allocation du temps pour plusieurs threads.

— une classe peut hériter de la classe `Thread`
— une classe peut implémenter l'interface `Runnable`

Classe Thread

Pour créer un Thread, il suffit que la classe hérite de la classe `Thread`. Reportez vous à la documentation pour connaître l'ensemble de ses huit constructeurs. On retiendra :

— `Thread()` qui instancie un objet Thread dont le nom est "Thread-1", "Thread-2" ...
— `Thread(String name)` qui instancie un objet Thread dont le nom est passé en paramètre.
— `Thread(Runnable target)` instancie un Thread en utilisant un objet d'une classe qui implémente l'interface Runnable.

La classe `Thread` possède aussi des méthodes pour la gestion des processus :

— La méthode `run()`. Il est nécessaire de redéfinir cette méthode car par défaut, elle ne fait rien. C'est dans cette méthode qu'il faut placer le code exécuté par le thread.
— La méthode `setPriority(int priorite)` qui permet de définir le niveau de priorité du thread. La priorité peut prendre trois valeurs (qui sont des constantes de la classe `Thread` :
  — `Thread.MIN_PRIORITY`,
  — `Thread.NORM_PRIORITY`
  — `Thread.MAX_PRIORITY`.

— Les méthodes `setName(String nom)` et `getName()` permettent respectivement de choisir ou de lire le nom d'un thread.
— la méthode `start()` permet de démarrer un thread, dans la pratique, elle appelle la méthode `run()`.
— la méthode `yield()` permet de laisser un peu de temps pour l'exécution des autres threads du même niveau de priorité.
— la méthode `join()` permet d'attendre la fin de l'exécution du thread.
— La méthode `sleep(long millis)` permet de mettre le thread en sommeil (en pause) pendant une période exprimée en millisecondes.

Attention : les méthodes `sleep` et `join` peuvent provoquer une exception de la classe `InterruptedException`. Ce qui implique de prévoir le code pour la traiter. Pour cela, il faut :

1. placer le code qui peut provoquer une exception dans un bloc `try`
2. placer le code exécuté en cas d'exception dans un bloc `catch`.

Exemple :

```
try{
  sleep(100);
}catch(Exception e){System.out.println(e);}
```

En résumé :
Déclaration du Thread :

```
public class MonThread extends Thread{

  public void run(){
    // code de ce que fait le thread
  }
}
```

Lancement du Thread :

```
MonThread t = new MonThread();
t.start();
```

Interface Runnable

Elle ne comporte qu'une seule méthode : la méthode `run()`. Lorsqu'une classe implémente cette interface, il faut donc surcharger cette méthode. On y écrit le code exécuté par le thread.

Ensuite, on peut instancier un Thread avec une instance de cette classe. La méthode `run()` sera exécutée lorsque la méthode `start()` du thread sera appelée.

En résumé :

Déclaration du Thread :

```
public class MonThread implements Runnable{

  public void run(){
    // code de ce que fait le thread
  }
}
```

Lancement du Thread :

```
Thread t = new Thread(new MonThread());
t.start();
```

Attention : appeler la méthode `run()` d'un thread ne lance pas le thread, il faut dans tous les cas appeler la méthode `start()`

Travail à réaliser

Exercice 1

Question 1

Ecrivez une classe `MonThread` qui hérite de `Thread`. Elle comprend un attribut `compteur` de type `int`.

Lorsqu'il sera exécuté, le thread exécutera 10 fois la séquence suivante :

— affichage du nom du thread courant et la valeur du compteur ;

— effectuer une pause d'une durée aléatoire comprise entre 0 et 1 seconde.
A la fin de son exécution, le thread indique qu'il s'arrête en affichant une chaîne de caractères (qui comprend son nom).

Question 2

Testez votre classe dans une nouvelle classe `TestMonThread` qui est exécutable. Cette classe crée autant de threads que demandé sur la ligne de commande et les lance simultanément.

Question 3

On veut maintenant afficher le message « Fin de tous les threads » lorsque tous les threads sont terminés. Utilisez la méthode `join` pour attendre l'arrêt des threads.

Exercice 2

L'exercice représente plusieurs files d'impression d'une imprimante.

Écrivez une classe `Impression` qui implémente l'interface `Runnable`. La classe possède un attribut nom qui contient une chaîne de caractères (le nom du document) et un autre attribut qui indique le nombre de pages. Écrivez le constructeur de cette classe, le construteur prend les deux paramètres (nom et nombre de pages).

La classe redéfinit la méthode `run` et affiche n pages (une page sera simplement représentée par son numéro et le nom du document).

Dans une classe exécutable `TestImpression`, vous devez instancier deux Threads avec des objets `Impression`. Lancez les threads et observez le résultat.

Explication : puisque plusieurs threads peuvent imprimer en même temps, on n'a pas de garantie de l'ordre d'impression des pages des différents documents. Ce qui peut poser problème ...

Solution : imposer que certaines parties du codes ne soient exécutables que par un seul thread à la fois. Pour cela, on utilise le mot clé `synchronized`. Le mot clé `synchronized` prend un paramètre : le nom d'un objet. Un objet java possède un mécanisme de verrouillage. Lorsque la méthode est synchonisée sur un objet aucune autre partie de code synchronisée sur le même objet ne

peut être exécutée. Dans le cas présent, vous pouvez par exemple verrouiller l'utilisation de la sortie standard :

```
synchronized(System.out){
  ... // code protégé
}
```

Exercice 3

Écrivez une classe `Compteur`. Dans cette classe qui sera un `Thread`, vous déclarez un attribut `Integer` qui sera un attribut de classe (`static`). De cette façon, toutes les instances de la classe auront la même valeur pour cet attribut.

Chaque instance du thread pourra incrémenter la variable par l'intermédiaire d'une méthode `incremente()`. Voici le corps de la méthode, elle contient une pause pour simuler un traitement long. L'attribut `compteur` est un objet de la classe `Integer` et `MAX` est une constante de type `int`.

```
public void incremente(){

  int compteurInt = compteur.intValue();
    try{
      sleep(100);
    }catch(Exception e){System.err.println(e);}
    if (compteurInt < MAX) compteurInt++;
      compteur = new Integer(compteurInt);
}
```

On souhaite écrire la méthode run() qui indique la valeur du compteur ainsi que le nom du thread qui incrémente. Voici une implémentation possible de cette méthode.

```
public void run(){

     while (isAlive){

        incremente();
```

```
        // afficher
        System.out.println(getName()+" valeur = "+(compteur));

        // sortir du thread
        if (compteur.intValue() >= MAX) isAlive = false;

    }
    System.out.println("*** Fin de "+getName()+" ***");
}
```

Ecrivez une classe main qui permette de lancer autant de threads qu'indiqué sur la ligne de commande. Chaque thread doit incrémenter le compteur (le fonctionnement de la méthode main est identique à celui de l'exercice 1).

Exercice 4 : arrêter des Threads

Principes généraux

- règle de base : le thread est terminé lorsqu'il sort de sa méthode run...
- Il ne faut pas utiliser la méthode stop() qui est *deprecated* (risque de deadlock)
- Utiliser la méthode interrupt() qui permet de lever l'exception InterruptedException (lorsque le thread est placé en état d'attente).
- Le thread n'est pourtant pas interrompu, il faut prévoir dans la clause catch de postionner le *flag* d'interruption avec Thread.currentThread().interrupt() ;
- ensuite prévoir une boucle (dans la méthode run) sur ! isInterrupted().

Exercice à réaliser

Ecrire une classe qui hérite de Thread. La méthode run sera un simple compteur qui se met en sommeil pendant 1 seconde (en boucle).

Ecrire ensuite un programme qui lance ce thread puis attend une saisie de l'utilisateur. Suite à cette saisie, le thread est interrompu. On affichera le temps passé dans chaque boucle d'attente.

Exercice 5

Vous allez maintenant utiliser des threads pour remplir un tableau de valeurs aléatoires. Ce qui vous donnera l'occasion de comparer le résultat en terme de temps d'exécution. Vous pourrez ainsi comparer l'exécution sur un ou plusieurs threads.

Initialisation du tableau

On donne la classe suivante qui permet de lancer un thread. Ce thread initialise une partie d'un tableau (la partie du tableau traitée par le thread est définie par le constructeur) avec des valeurs aléatoires (un peu spécifiques).

```
public class CalculAlea extends Thread {

// tableau à traiter
    private double[] t;
    // attribut qui indique sur quelle partie du tableau on travaille
    private int debut, fin;

// constructeur
    public CalculAlea(double[] t, int debut, int fin){
       this.t = t; this.debut = debut; this.fin = fin;
    }

    // méthode run
    public void run(){
      for (int i=debut; i<fin; i++){
         t[i] = Math.sqrt(Math.pow(Math.random(),2.0)+Math.pow(Math.random(),2.0))
      }
    }
}
```

Code à écrire

Créez une classe `Tableau` et écrivez une méthode de classe (méthode `static`) qui va permettre d'initialiser un tableau passé en paramètre. La signature de cette méthode est :

```
public static void alea2Thread(double[] tab)
```

Cette méthode utilise la classe CalculAlea pour initialiser toutes les cases du tableau avec deux threads.

Ensuite écrivez la méthode `alea` qui permet de faire le même travail avec un nombre de thread choisi lors de l'appel de la méthode. La signature de la méthode est :

```
public static void alea(double[] tab, int nbThread)
```

Vous écrirez enfin une classe `TestTableau` afin de comparer l'impact du nombre de threads. Afin de mesurer le temps d'exécution, vous pourrez par exemple utiliser `System.nanoTime()` qui retourne un `long`. Cette valeur représente un nombre de nanosecondes écoulées depuis une date de référence. La soustraction entre deux valeurs donne le nombre de nanosecondes écoulées entre les deux instants.

Vous indiquerez dans votre compte rendu les caractéristiques de la machine sur laquelle vous avez fait les tests (en particulier quel CPU est utilisé).

Question subsidaire

Comptez le nombre de fois où la valeur dans le tableau est inférieur à 1, divisez cette valeur par le nombre de cases (du tableau) et multipliez par 4.

Comment expliquer le résultat ?