Threads en Java
TP1 : Observation*

1 Threads en Java

Les threads permettent d'avoir plusieurs activités concurrentes dans un programme. Du point de vue d'un programmeur d'application, les threads Java peuvent être créés de deux façons.

Utilisation de la classe `Thread` : La première façon consiste à étendre la classe prédéfinie `Thread` :

```
// L'interface Runnable et la classe Thread
// sont predefinies par Java
interface Runnable {
    void run();
}
public class Thread extends Object
                    implements Runnable {
    void run() {...}
    void start() {...}
    ...
}

public class Compteur extends Thread {
    // la methode run definit le comportement du thread
    public void run() {...}

    public static main() {
        Compteur c = new Compteur();
        c.start();
    }
}
```

L'héritage à partir de `Thread` est contraignant car il empêche tout autre héritage (en Java, une classe ne peut hériter que d'une seule autre classe).

*Cet énoncé est basé sur un document initial de V. Danjean et V. Marangozova-Martin.

Utilisation de l'interface `Runnable`: La deuxième façon de faire est d'implémenter l'interface `Runnable`. Ceci permet l'héritage d'autres classes et l'implémentation d'autres interfaces.

```
interface Runnable { // definie par Java
    void run();
}
public class Thread extends Object
                    implements Runnable { // definie par Java
    void run() {...}
    void start() {...}
    ...
}

public class Compteur implements Runnable{
    // la methode run definit le comportement du thread
    public void run() {...}

    public static main() {
        Compteur c = new Compteur();
        // creation et demarrage d'un thread :
        Thread t = new Thread(c);
        t.start();
    }
}
```

Remarque : la façon choisie pour la création des threads (c'est-à-dire via l'héritage à partir de `Thread` ou via l'utilisation de l'interface `Runnable`) n'a pas d'impact sur la gestion des threads par le système (choix d'ordonnancement, gestion de la concurrence, etc.).

2 TP1 première partie : Observation de threads

L'objectif de ce TD/TP est de vous faire programmer des threads en Java, de vous faire observer le comportement des programmes à activités concurrentes (*multi-threads*) et de vous montrer le besoin de synchronisation entre threads.

2.1 Environnement de travail

Nous travaillerons dans un environnement Unix. L'utilisation de la ligne de commande est vivement encouragée (i.e., les commandes de compilation et de lancement seront à taper dans un terminal). L'utilisation d'environnements de développement intégrés (IDE) est envisageable mais sans support de la part des enseignants.

2.2 ExempleThread1.java

Compiler et exécuter le programme `ExempleThread1.java` (pour exécuter plusieurs fois, il est possible d'utiliser le script **exec.sh** fourni)

Le programme crée trois threads qui affichent 1000 fois respectivement "Hello" "World" et "and Everybody".

Que pouvez vous dire à propos de l'ordre d'affichage ? Expliquer. Que pouvez vous dire à propos du nombre d'affichages ? Expliquer.

Le programme utilise l'héritage de `Thread` pour obtenir des flots d'exécutions parallèles. Changer le code pour utiliser l'interface `Runnable`.

2.3 ExempleThread2.java

Compiler et exécuter plusieurs fois le programme `ExempleThread2`. Voyez vous tous les affichages ? Essayez de diminuer et d'augmenter les valeurs d'attente (les arguments passés lors de la création des threads qui sont utilisés pour les appels à sleep). Si vous mettez de trop grandes valeurs, vous risquez de ne plus voir aucun affichage. Pourquoi ?

2.4 ExempleThread3.java

Dans ce programme, nous rajoutons la fonction `join` qui force le programme principal à attendre la terminaison des threads. Compiler le programme, l'exécuter plusieurs fois et s'assurer que tous les affichages sont présents.

Étudier ensuite le code et le comportement des programmes `ExempleThread3bis` `ExempleThread3ter`. Expliquer les différences observées à l'aide de la documentation Java.

2.5 ExempleThread4.java

Dans ce programme, nous manipulons une variable partagée `Tableau`. Les threads de `ThreadDeposer` incrémentent les valeurs du tableau, alors que les threads `ThreadRetirer` décrémentent ces valeurs. Logiquement, à la fin les valeurs doivent être égales à 0 puisque les effets des threads incrémentant et décrementant s'annulent.

Compiler et exécuter plusieurs fois le programme. Y a-t-il des cas où le résultat est différent ? Expliquer.

Modifier les fichiers `ExempleThread4.java`, `ThreadDeposer.java` et

`ThreadRetirer.java` pour utiliser l'interface `Runnable`. (Noter que cette modification est purement d'ordre syntaxique. Elle n'est pas censée avoir d'impact sur le comportement observé ... sauf si elle est mal réalisée.)

2.6 ExempleThread5.java

Pour corriger le problème observé dans l'exemple précédent, on définit les deux méthodes `incTab` et `decTab` en tant que `synchronized`. Commencer par remarquer les différences subtiles entre les fichiers utilisés pour cette version et ceux de la question précédente (pour cela, la commande Unix `diff` pourra être utile). Compiler et exécuter le programme. S'assurer que le comportement est correct.

Bonus 1 : Proposer une variante (en définissant éventuellement un ou plusieurs nouvelles classes) qui permet de conserver un fonctionnement correct mais d'obtenir un meilleur temps d'exécution que celui obtenu avec le code fourni dans cet exercice.

Bonus 2 : Une autre manière de régler le problème sans utiliser le mot clé `synchronized` consiste à utiliser, pour les valeurs stockées dans les cases du tableau, des variables Java basées sur un type dit *atomique*. Écrire une nouvelle version du code pour implémenter cette approche [1].

3 TP1 deuxième partie : lien avec le système d'exploitation

3.1 Chronométrage

La commande `time` permet de chronométrer le temps d'exécution d'un processus. Elle affiche plusieurs temps différents (`usr`, `real` et `sys`). Essayer de comprendre la signification et la complémentarité de ces différentes informations (éventuellement à l'aide du manuel : `man 1 time`).

1. Remarques concernant les variables Java atomiques :
   - Les documentations suivantes peuvent vous être utiles :
     - `https://docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html`
     - `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html`
   - Il n'existe pas, en Java, de type de variable atomique correspondant à la notion de `double`. Pour contourner le problème, il est possible d'utiliser des conversions de types (voir documentation ci-dessus) mais dans le cadre de cet exercice, le plus simple consiste à remplacer le type `double` par un type entier.

Il est recommandé d'utiliser cette commande avec l'option `-p` afin d'obtenir un affichage plus lisible.

En guise d'exemple, reprendre le programme `ExempleThread4.java` et observer les informations remontées par la commande `time`. Est-il possible d'avoir `usr` + `sys` > `real` ? Si oui, comment peut-on l'expliquer ? Sinon pourquoi ?

Remarque : Il existe plusieurs commandes `time` sur un système Unix : une commande externe (généralement installée dans le dossier `/usr/bin` et une commande intégrée au shell (notamment le shell Bash). L'exercice ci-dessus fait référence à la première. Il faudra donc lancer la commande en la préfixant par son chemin absolu (`/usr/bin/time`) ; sinon, c'est la commande intégrée au shell qui prime).

Bonus : Utiliser ensuite l'option `-v` (*verbose*), qui affiche davantage de statistiques remontées par le système. Essayer de comprendre la signification de ces différentes informations (faire le lien avec les notions du cours d'introduction).

3.2 Niveau d'implémentation des threads

Écrire un programme java qui créé N threads, affiche un message après avoir créé et démarré tous les threads, puis attend la terminaison de tous ces threads. Chacun des threads effectue simplement un appel à `sleep` pour se bloquer pendant deux minutes avant de se terminer. Le nombre de threads est passé en argument via la ligne de commande.

Faire plusieurs exécutions successives en faisant varier N (par exemple avec les valeurs suivantes : 1, 2, 5, 10). À chaque fois observer, le nombre de threads "noyau" utilisés par la machine virtuelle Java. Pour cela, utiliser la commande Unix `ps -eLf` (voir `man ps` pour les détails [2]). Regarder notamment le contenu des colonnes `PID`, `PPID`, `LWP` et `NLWP` (après avoir cherché leur signification dans la documentation).

En déduire le type d'implémentation utilisé par la machine virtuelle Java pour la gestion des threads de l'application (threads "utilisateur" ou threads "noyau").

2. Ou la version en ligne de la documentation (contenu identique) : `http://man7.org/linux/man-pages/man1/ps.1.html`

4 Bonus

Voici quelques pistes facultatives pour approfondir les notions abordées dans ce TP :

1. Regarder la documentation de la machine virtuelle Java pour voir la définition des paramètres suivants : `-Xss`, `-Xss`. À votre avis, en quoi la configuration de ces paramètres peut-elle impacter l'exécution d'une application Java ?
2. Pour approfondir la compréhension du fonctionnement interne de la machine virtuelle Java, vous pouvez consulter la documentation suivante [3] : `http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html`

3. Voir notamment les sections suivantes : *Thread Management*, *Threading Model*, *Thread Creation and Destruction*, *Thread States*, et *Internat VM Threads*.