UFR IM²AG

UNIVERSITÉ Grenoble Alpes

DU ISN Seconde année, Système et réseaux

Threads en Java [a]

a. Cet énoncé réutilise des éléments de sujets produits par V. Danjean, R. Lachaize et G.Huard

1 Les threads

Les processus système sont appelés "processus lourds" puisque leur gestion demande du temps et des ressources. Par ailleurs, les processus sont par définition isolés et ne peuvent partager de la mémoire que par des mécanismes système dédiés [1]. Les *threads*, appelés "processus légers", sont des fils d'exécution au sein d'un processus. Ils se partagent ainsi l'espace mémoire d'un processsu, travaillent avec la même copie de code et partagent des données. Ils sont largement utilisés afin de paralléliser les logiciels d'aujourd'hui mais viennent avec une complexité accrue de programmation.

Voici un programme Java avec des threads (fourni dans ExempleThread1.java).

```
public class ExempleThread1 implements Runnable {

    private String toSay;

    public ExempleThread1(String toSay) { this.toSay=toSay;}

    public void run() {
        for (int i =0; i< 10; i++) System.out.println(i + "␣" + toSay );
    }

    public static void  main(String args[]) {

        Thread thread1, thread2, thread3;

        thread1=new Thread(new ExempleThread1("Hello␣"));
        thread2=new Thread(new ExempleThread1("World␣"));
        thread3=new Thread(new ExempleThread1("and␣Everybody␣"));

        thread1.start();
        thread2.start();
        thread3.start();

        try {
            thread1.join();
            thread2.join();
            thread3.join();
        } catch (InterruptedException e) {
            System.err.println("Unexpected␣situation.");
            e.printStackTrace(System.err);
            System.exit(-1);
        }
        System.exit(0);
    }
}
```

1. que nous ne considérons pas dans ce cours

Dans cet exemple, dans la fonction `main` (exécutée par un premier thread), trois autres threads sont créés (ligne ..., `thread1`, `thread2`, `thread3`). Le lancement des threads se fait par l'appel de la méthode `start` qui se charge de lancer des fils d'exécution indépendants (concurrents, y compris avec l'exécution du main). Les threads exécuteront le code défini dans leur méthode `run`.

Dans l'exemple, le `thread1` exécutera la boucle de `run` et affichera donc 10 fois "Hello ". `thread2` affichera 10 fois "World " et `thread3` 10 fois "and Everybody ". Le thread main continuera son exécution à la ligne... et appelera la méthode `join`, conséqutivement sur les trois threads, afin d'attendre leur terminaison.

2 Environnement de travail

Java utilise de fortes conventions de nommage. Ainsi, une classe A **doit** être définie dans un fichier dont le nom est `A.java`. A l'issue de la compilation, le *bytecode* (code machine pour la machine virtuelle Java) se trouvera dans un fichier `A.class`.

Pour compiler un fichier A.java : `javac A.java`

Pour exécuter une classe compilée, si elle est dans le répertoire courant : `java -cp . A`

L'option `cp` indique à l'interpreteur `java` où chercher les classes dont il a besoin : c'est le classpath [2].
Il est possible de positionner de manière globale la variable d'environnement `CLASSPATH`. Dans la cas de bash utiliser (en adaptant la liste de répertoires bien évidemment)
`export CLASSPATH=/home/users/monrepertoire/classes:unAutreRepertoire:.`
Une fois cette variable positionnée, il ne sera plus nécessaire d'utiliser l'option -cp.

Lors de la compilation, il est possible également de demander la génération des classes compilées dans un autre répertoire que le répertoire courant. Pour ce faire, utiliser l'option `-d` :
`java -cp $CLASSPATH -d <chemin vers répertoire destination> A`

3 Observation de threads

L'objectif ici 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.

3.1 Exécution concurrente dans ExempleThread1.java

Exercice 1 Compiler et exécuter plusieurs fois le programme `ExempleThread1.java`.
Que pouvez-vous dire à propos de l'ordre d'affichage ? Expliquer.

Exercice 2 Reprendre l'exemple précédant, augmenter le nombre d'itérations dans `run` et enlever les appels à `join` à la fin. Exécuter plusieurs fois et observer. Que pouvez vous dire du nombre des affichages ?

Exercice 3 Que se passe-t-il si vous reprenez votre version modifiée et vous enlevez l'appel à System.exit à la fin du `main` ?

2. C'est la même logique que la variable PATH qui indique au shell où chercher les exécutables des commandes tapées.

3.2 Manipulation de variable partagée dans ExempleThread3.java

Dans ce programme [3], nous manipulons une variable partagée de type `Tableau`. Les threads exécutant le comportement défini dans `ThreadDeposer` incrémentent les valeurs du tableau, alors que les threads `ThreadRetirer` décrémentent ces valeurs.

Exercice 4 Etudier le code. Etes-vous d'accord que, logiquement, à la sortie du programme, les valeurs du tableau doivent être égales à 0 ?

Exercice 5 Compiler et exécuter plusieurs fois le programme. Qu'observez-vous ? Expliquer.

3.3 Correction de ExempleThread3.java

Exercice 6 Pour corriger le problème dans l'exemple précédent, définir les deux méthodes `incTab` et `decTab` en tant que `synchronized`. Compiler et exécuter le programme. S'assurer que le comportement est correct. Observez-vous une différence au niveau du temps d'exécution ? Expliquer.

Remarques :

— Pour mesurer le temps, on pourra utiliser la commande Unix `time`.

— Pour les temps mesurés avec `time`, est-il possible d'avoir `usr` + `sys` > `real` ? Si oui, comment peut-on l'expliquer ?

4 Ecrire un programme multi-threadé

Exercice 6 Ecrire un programme Java multi-threadée de recherche d'un élément dans un tableau d'entiers. Nous vous fournissons le programme principal (`Principal.java`, à trous) et le squelette d'un thread de recherche (`ThreadRecherche.java`).

Le programme principal :

— prend comme arguments un entier à rechercher et le nombre de threads à utiliser

— recherche l'entier au sein d'un tableau qui devra être au préalable initialisé avec des valeurs. Pour que l'expérience soit reproductible, le programme permet de générer des valeurs aléatoires pour le tableau et de les enregistrer dans un fichier pour un usage ultérieur. Ce fichier peut être passé en ligne de commande lors d'une exécution ultérieure pour une recherche sur le même tableau.

— Si le programme est lancé en séquentiel, l'élément est recherché en parcourant les éléments du tableau. Si le programme est lancé pour être exécuté en parallèle, il devra créer le nombre de threads indiqué, et les faire rechercher la valeur dans des sous-tableaux. Les threads devraient s'arrêter dès que l'un d'entre eux a trouvé la valeur.

5 Performances des programmes multi-threadés

Vous avez réussi votre recherche parallèle ? FELICITATIONS ! Est-ce plus performant en parallèle qu'en séquentiel ?

Exercice 7 Faire une étude de l'accélération du programme. Faites varier le nombre de threads pour une taille de tableau fixe, ainsi que la taille du tableau pour un nombre de threads fixe. Quelle est votre analyse ?

Le temps d'exécution d'une portion de code peut être mesuré en utilisant

```
long debut = System.currentTimeMillis();
// ... PORTION DE CODE ...
long fin = System.currentTimeMillis();
long tempsPasse = fin - debut;
```

3. Nous travaillons ici avec les fichiers `ExempleThread3.java`, `ThreadDeposer.java`, `ThreadRetirer.java`, `Tableau.java`

6 BONUS : Lien avec le système d'exploitation

6.1 Niveau d'implémentation des threads

Le programme `ExempleThread4.java` crée `NBTHREADS` 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" utilisé par la machine virtuelle Java. Pour cela, utiliser la commande Unix `ps -eLf` (voir `man ps` pour les détails). Regarder notamment le contenu des colonnes `PID`, `PPID`, `LWP` et `NLWP`.

Pour compter le nombre de threads correspondant à votre programme Java, vous pouvez utiliser la commande suivante :

```
> ps -eLf | grep ExempleThread4 | wc -l
```

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

6.2 Paramètres liés à la mémoire

Écrire un programme qui contient un boucle de création de threads (le nombre de threads a créer est passé en argument). Chaque itération de la boucle crée un nouveau thread T_i et le démarre. Le comportement de chaque thread T_i consiste simplement en une boucle infinie d'appels à `sleep`. Lorsque la machine virtuelle java ne peut plus créer de thread supplémentaire (épuisement des ressources), elle lève une erreur (`OutOfMemoryError`). Le programme doit rattraper cette erreur, afficher le nombre de threads qui ont été lancés avec succès puis se terminer.

Faire ensuite la série d'expériences suivante. Lancer plusieurs exécutions successives du programme en choisissant un nombre très élevé de threads. Faire ensuite varier la quantité de mémoire allouée pour la pile de chaque thread (voir l'option `-Xss` de la machine virtuelle Java, pour voir toutes les options `java -X`[4]) et observer l'impact sur le nombre de threads créés. Puis faire varier la quantité de mémoire réservée pour le tas de la machine virtuelle Java (voir les options `-Xms` et `-Xmx`). Expliquer les résultats observés.

4. Commande utile pour voir les valeurs par défaut `java -XX:+PrintFlagsFinal | grep -iE 'HeapSize|ThreadStackSize'`

7 ANNEXE : Exemple de création et de manipulation de base d'objets Java

```
public class ExempleJava {

    /**
     * ATTRIBUTS
     * chaque nouvel objet a sa propre copie de ces attributs
     * private = les attributs ne peuvent être manipulés qu'à l'intérieur
     * d'un objet
     */
    private String nom;
    private double age;
    private int num;

    /**
     * CONSTRUCTEUR
     * méthode spéciale exécutée lors de la création d'un nouvel objet
     * sert typiquement à initialiser les attributs de cet objet
     */
    public ExempleJava(String nom, double age, int num) {
        //this reference l'objet courant
        //le nom de l'objet qui est en train d'être créé prend pour valeur
        //le nom en paramètre
        this.nom=nom;
        this.age = age;
        this.num = num;
    }

    public void changerNom(String nom) {
        this.nom = nom;
    }

    public String getNom() {
        return nom;
    }

    /*
     Si on veut écrire le contenu d'un objet.
     Sans cette m"thode, Java prend l'identifiant de l'objet.
     Essayer les affichages dans le main avec et sans cette méthode (la mettre en commentaire)
     */
    public String toString() {
        return new String(this.nom + ":" + this.age + ":" + this.num);
    }

    /**
     Méthode prédéfinie qui doit être réimplémentée si on veut comparer des objet à structure complexe.
     Vous pouvez essayer les tests d'égalité à la fin du main avec et sans cette méthode.
     */
    public boolean equals(Object o) {
```

```
    if (!(o instanceof ExempleJava)) return false;

    ExempleJava autre = (ExempleJava)o;
    return (autre.nom == nom) && (autre.age == age) && (autre.num == num);
}

/** La méthode main
 *  éxécutée au lancement du programme
 */
public static void  main(String args[]) {

    int i;
    // lire les arguments passés en ligne de commande
    for (i = 0; i< args.length; i++) {
        System.out.println("arg " + i + " : " + args[i]);
    }

    //créer un objet
    System.out.println("Création d'un objet...");
    ExempleJava o1 = new ExempleJava("Vania", 10.0, 1004);

    //afficher
    System.out.println("Le contenu de l'objet est...");
    System.out.println(o1.toString());

    System.out.println("Le nom de l'objet est ...");
    //utiliser les methodes
    System.out.println(o1.getNom());

    System.out.println("Manipulation d'attributs ...");
    o1.age = 40;

    System.out.println("L'age de l'objet est ...");
    //utiliser les methodes
    System.out.println(o1.age);

    System.out.println("Création d'un autre objet...");
    ExempleJava o2 = new ExempleJava("Vincent", 5.5, 1111);

    System.out.println("Est-ce le même objet?");
    System.out.println(o1==o2);

    System.out.println("Les deux objets ont ils les mêmes attributs?");
    System.out.println(o1.equals(o2));

    System.out.println("Création d'un 3eme objet avec les mêmes attributs que o1...");
    o1.age = 10.0;
    ExempleJava o3 = new ExempleJava("Vania", 10.0, 1004);

    System.out.println("Les deux objets o1 et o3 ont ils les mêmes attributs?");
    System.out.println(o1.equals(o3));
```

```
            System.exit(0);
        }

    }
```