Les Threads

Mohammed OUANAN

m.ouanan@umi.ac.ma

Plan

Introduction

La notion de `thread`

- C'est un processus léger hébergé par un processus lourd (la machine virtuelle)
- Le programme principal en Java (le main) est aussi un thread
- Les threads Java partagent un ensemble d'instructions, données (objets, attributs...) et ressources

Introduction

Avantages

- Rapidité de lancement et d'exécution
- Possibilité de lancer plusieurs exécutions parallèles du même code
- Exploitation de la nouvelle structure multiprocesseurs....

Inconvénients

- Problèmes de synchronisation entre threads
- Difficulté de gestion d'applications multi-thread

Création

Deux solution possibles, soit

- En créant une sous-classe Thread
- En créant une instance de la classe Thread avec un objet `Runnable`

Explication

- Il existe une classe Java appelée `Thread`.
- La classe `Thread` implémente une interface `Runnable`
- L'interface `Runnable` a une méthode abstraite `void run()`
- Pour exécuter un thread, il faut appeler la méthode `start()` pas le `run()`

Création

Exemple avec la première solution

```
public class MonThread extends Thread {
  // juste pour vous montrer que la classe a un
     attribut name
  public MonThread(String name) {
    super(name);
  }
  public void run() {
    for (int i=0; i<5; i++) {
      System.out.print(this.getName() + "␣");
      }
  }
}
```

Création

Le main

```
public class Test {
  public static void main(String[] args) {
    MonThread A = new MonThread("A");
    MonThread B = new MonThread("B");
    MonThread C = new MonThread("C");
    A.start();
    B.start();
    C.start();
    System.out.println(Thread.currentThread().
      getName() + "␣:␣j'ai␣fini");
        }
}
```

Création

Dans une exécution séquentielle classique, l'affichage sera

A A A A A B B B B B C C C C C
main : j'ai fini

Création

Dans une exécution séquentielle classique, l'affichage sera

A A A A A B B B B B C C C C C
main : j'ai fini

Ici, avec les threads

- Tout est possible
- voici une exécution possible
- main : j'ai fini
  A B B B B B C C C C C A A A A

Création

Exemple avec la deuxième solution

```
public class TonThread implements Runnable{
  private String nom;
  public TonThread(String nom) {
    this.nom = nom;
  }
  public String getNom() {
    return nom;
  }
  @Override
  public void run() {
    for (int i=0; i<5; i++) {
      System.out.print(this.getNom() + "␣");
    }
  }
}
```

Création

Le main

```
public class Test {
  public static void main(String[] args) {
    Thread A = new Thread(new TonThread("A"));
    Thread B = new Thread(new TonThread("B"));
    Thread C = new Thread(new TonThread("C"));
    A.start();
    B.start();
    C.start();
    System.out.println(Thread.currentThread().
      getName() + "␣:␣j'ai␣fini");
  }
}
```

Création

Le main

```
public class Test {
  public static void main(String[] args) {
    Thread A = new Thread(new TonThread("A"));
    Thread B = new Thread(new TonThread("B"));
    Thread C = new Thread(new TonThread("C"));
    A.start();
    B.start();
    C.start();
    System.out.println(Thread.currentThread().
      getName() + "␣:␣j'ai␣fini");
  }
}
```

A A A A A B B B B B C C C C C
main : j'ai fini (une exécution possible)

Méthodes principales

Méthodes principales

- `start()` : pour démarrer le thread
- `sleep(long dur´ee)` pour arrêter un thread pendant une durée précisée
- `getState()` pour récupérer l'état d'un thread
- `getPriority()` et `setPriority()` pour récupérer et modifier la priorité d'un thread
- `isAlive()` : pour tester si un thread est en vie. Elle retourne `false` si le thread est dans l'état `NEW` ou `TERMINATED`, `true` sinon.
- ...

Méthodes principales

Les valeurs de `getState()`

- `NEW` : s'il vient d'être crée
- `RUNNABLE` : s'il est entrain d'exécuter le `run`
- `TERMINATED` : s'il a fini d'exécuter le `run` ou qu'une exception a été levée
- `BLOCKED` : s'il est en attente d'une ressource détenue par un autre thread
- `WAITING` : s'il est en attente d'une action d'un autre thread pour une une durée indéterminée (`WAITING` et `BLOCKED` sont similaires)
- `TIMED_WAITING` : s'il est en attente pour une durée précise. Ensuite il repassera à l'tat `RUNNABLE`

Méthodes principales

Les valeurs de `setPrioriy()`

- Les valeurs de priorité varient de 1 à 10
- On peut donc faire `A.setPriority(8);`
- ou bien on peut aussi utiliser les constantes prédéfinies
  - `Thread.MIN_PRIORITY` : équivalent à 0
  - `Thread.MAX_PRIORITY` : équivalent à 10
  - `Thread.NORM_PRIORITY` : équivalent à 5 (priorité par défaut)

Problème de synchronisation

Exemple

```
public class MonCompteur {
    private int compteur = 5;

    public int getCompteur() {
        return compteur;
        }

    public void decrementerCompteur() {
        compteur--;
    }
}
```

Problème de synchronisation

```
public class TestThread implements Runnable{
  MonCompteur TC;
  private String name;
  // ajouter aussi un constructeur ave 2 paremeters
  public void run(){
    try{
      for(int i=0; i<3;i++){
        if(TC.getCompteur() >0){
          Thread.sleep(500);
          TC.decrementerCompteur();
          System.out.println("operation reussie " + TC.getCompteur() +
            " demandee par " + this.name);
        }
        else
          System.out.println("Echec " + TC.getCompteur() + " demandee
            par " + this.name);
      }
    }
    catch(InterruptedException e){
      System.out.println(e.getMessage());
    }
  }
}
```

Problème de synchronisation

```
public class Test {
  public static void main(String[] args) {
    MonCompteur TC = new MonCompteur();
    Thread t1 = new Thread(new TestThread(TC,"␣t1␣")
       );
    Thread t2 = new Thread(new TestThread(TC,"␣␣␣␣t2
       ␣"));
    Thread t3 = new Thread(new TestThread(TC,"␣␣␣␣␣␣
       ␣t3␣␣␣"));
    t1.start();
    t2.start();
    t3.start();
  }
}
```

Problème de synchronisation

L'affichage peut être ainsi :

```
 operation reussie 2 demandee par          t3
 operation reussie 2 demandee par  t1
 operation reussie 2 demandee par     t2
 operation reussie 1 demandee par     t2
 operation reussie -1 demandee par         t3
 operation reussie 0 demandee par  t1
Echec  -1  demandee par        t3
Echec  -1  demandee par     t2
Echec  -1  demandee par  t1
```

Problème de synchronisation

L'affichage peut être ainsi :

```
 operation reussie 2 demandee par          t3
 operation reussie 2 demandee par  t1
 operation reussie 2 demandee par     t2
 operation reussie 1 demandee par     t2
 operation reussie -1 demandee par         t3
 operation reussie 0 demandee par  t1
Echec  -1  demandee par        t3
Echec  -1  demandee par     t2
Echec  -1  demandee par  t1
```

6 opérations réussies alors qu'on ne devait avoir que 5 car compteur = 5 et on teste chaque fois s'il est supérieur à 0

Problème de synchronisation

L'affichage peut être ainsi :

```
 operation reussie 2 demandee par          t3
 operation reussie 2 demandee par  t1
 operation reussie 2 demandee par     t2
 operation reussie 1 demandee par     t2
 operation reussie -1 demandee par         t3
 operation reussie 0 demandee par  t1
Echec  -1  demandee par        t3
Echec  -1  demandee par     t2
Echec  -1  demandee par  t1
```

6 opérations réussies alors qu'on ne devait avoir que 5 car compteur = 5 et on teste chaque fois s'il est supérieur à 0

Le multi-thread utilisant une même ressource peut violer les contraintes de notre système

Problème de synchronisation

Terminologie

- Une **ressource critique** est une ressource qui ne doit être accédée que par un seul thread à la fois.
- Un **moniteur** est un verrou qui ne laisse qu'un seul thread à la fois accéder à la ressource.

Règles

- Un thread n'accède à la section critique que si le moniteur est disponible
- Un thread qui entre en section critique bloque l'accès au moniteur
- Un thread qui sort de section critique libère l'accès au moniteur
- `sleep()` ne fait pas perdre le moniteur (contrairement à wait())

Problème de synchronisation

Plusieurs solutions possibles

- Utilisation de `synchronized()`, `wait()`, `notify()` et `notifyAll()`
- Utilisation de Sémaphore
- Utilisation de Lock

Problème de synchronisation : solution avec Synchronised

```
public class TestThread implements Runnable{
  MonCompteur TC;
  private String name;
  public void run(){
    try{
      for(int i=0; i<3;i++){
        synchronized(TC){
          if(TC.getCompteur() >0){
            Thread.sleep(500);
            TC.decrementerCompteur();
            System.out.println("␣operation␣reussie" + TC.getCompteur()
              + "␣demandee␣par␣" + this.name);
          } else
            System.out.println("Echec␣␣" + TC.getCompteur() + "  ␣␣
              demandee␣par␣" + this.name);
        }
      }
    } catch(InterruptedException e){
      System.out.println(e.getMessage());
    }
  }
}
```

Problème de synchronisation : solution avec Sémaphore

Sémaphore

- C'est une classe Java
- Elle permet d'autoriser plusieurs threads àaccéder àune partie du code
- Elle utilise le concept de permis : en fonction de nombre de permis, elle autorise l'accès au code.
  - `Semaphore sem = new Semaphore(int i);` permet de créer un objet de type Sémaphore avec *i* permis
  - `sem.acquire();` : permet de requérir un permis et bloque le thread demandeur jusqu'à ce qu'un permis soit disponible
  - `sem.release();` : augmente le nombre de permis disponibles de 1

Problème de synchronisation : solution avec Sémaphore

```
public class TestThread implements Runnable{
  private Semaphore sem; //on ajoute la semaphore comme attribut
  MonCompteur TC;
  private String name;
  public void run(){
    try{
      for(int i=0; i<3;i++){
        sem.acquire();
        if(TC.getCompteur() >0){
          Thread.sleep(500);
          TC.decrementerCompteur();
          System.out.println("␣operation␣reussie␣" + TC.getCompteur() +
              "␣demandee␣par␣" + this.name);
        } else
          System.out.println("Echec␣" + TC.getCompteur() + "␣demandee␣
            par␣" + this.name);
        sem.release();
      }
    } catch(InterruptedException e){
      System.out.println(e.getMessage());
    }
  }
}
```

Problème de synchronisation : solution avec Sémaphore

```
public class Test {
  public static void main(String[] args) {
    Semaphore sem= new Semaphore(1);
    MonCompteur TC = new MonCompteur();
    Thread t1 = new Thread(new TestThread(TC,"␣t1␣",
       sem));
    Thread t2 = new Thread(new TestThread(TC,"␣␣␣␣t2
       ␣",sem));
    Thread t3 = new Thread(new TestThread(TC,"␣␣␣␣␣␣
       ␣t3␣␣␣",sem));
    t1.start();
    t2.start();
    t3.start();
  }
```