Les Threads

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com

Plan

Java

Un `thread`

- Un processus léger hébergé par un processus lourd (JVM)
- Le programme principal en Java (le main) est aussi un thread
- Les threads peuvent partager un ensemble d'instructions, données (objets, attributs...) et ressources

Java

Un `thread`

- Un processus léger hébergé par un processus lourd (JVM)
- Le programme principal en Java (le main) est aussi un thread
- Les threads peuvent partager un ensemble d'instructions, données (objets, attributs...) et ressources

Exemple

- Correcteur grammatical dans un éditeur de texte
- Traitement simultané de plusieurs connexions client/serveur
- ...

Java

Avantages

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

Java

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

Java

En Java, il existe une interface `Runnable`

```
@FunctionalInterface
public interface Runnable {
  void run();
}
```

Java

En Java, il existe une interface `Runnable`

```
@FunctionalInterface
public interface Runnable {
  void run();
}
```

En Java, il existe aussi une classe `Thread`

```
public class Thread implements Runnable {

  private String name;
  private int priority;
  ...
}
```

Java

Explication

- Il existe une classe Java appelée `Thread`
- La classe `Thread` implémente une interface `Runnable`
- L'interface `Runnable` a une méthode abstraite `void run()`
- Exécuter un thread ⇒ appeler la méthode `start()` qui exécute le `run()`
- Le code de la méthode `run` s'exécute en parallèle du code qui a lancé le thread

Java

Deux solution possibles pour créer un thread, soit

- en créant un objet d'une sous-classe de Thread qui implémente la méthode `run()`
- en créant une instance de la classe thread et en lui passant comme paramètre l'instance d'une classe implémentant l'interface `Runnable`

Java

Exemple avec la première solution

```
public class MonThread extends Thread {
  // rappelez vous  que la classe Thread a un
     attribut name
  public MonThread(String name) {
    super(name);
  }

  @Override
  public void run() {
    for (int i = 0; i < 5; i++) {
      System.out.print(this.getName() + " ");
    }
  }
}
```

Java

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");
  }
}
```

Java

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

Java

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

Java

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() + " ");
    }
  }
}
```

Java

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");
  }
}
```

Java

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");
  }
}
```

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

Java

On peut aussi utiliser un objet d'une classe anonyme pour créer un thread

```
Thread thread = new Thread("A") {
  @Override
  public void run() {
  // TODO Auto-generated method stub
    for (int i = 0; i < 5; i++) {
      System.out.print(getName() + " ");
    }
  }
};
thread.start();
```

Java

On peut aussi utiliser un objet d'une classe anonyme pour créer un thread

```
Thread thread = new Thread("A") {
  @Override
  public void run() {
  // TODO Auto-generated method stub
    for (int i = 0; i < 5; i++) {
      System.out.print(getName() + " ");
    }
  }
};
thread.start();
```

Ou avec les expression lambda

```
Runnable runnable = () -> {
  for (int i = 0; i < 5; i++) {
    System.out.print("A ");
  }
};
Thread thread = new Thread(runnable);
thread.start();
```

Java

Remarque

- L'écriture du code est (légèrement) plus simple en utilisant la première méthode
- La deuxième méthode permet de partager de données entre les threads

Java

Méthodes principales

- `start()` : pour démarrer le thread
- `sleep(long durée)` : permet d'arrêter un thread pour une durée donnée en millisecondes afin de permettre l'exécution d'un autre thread
- `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.
- `join()` : bloque le thread courant jusqu'à la mort du thread appelant
- ...

Pour afficher le dernier message après la fin d'exécution des trois threads, on utilise la méthode `join()`

```
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();
    A.join();
    B.join();
    C.join();
    System.out.println(Thread.currentThread().
       getName() + " : j'ai fini");
  }
}
```

Java

Pour observer le cycle de vie d'un thread

```
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"));
    System.out.println(A.isAlive());
    // affiche false car le thread est à l'état NEW
    A.start();
    System.out.println(A.isAlive());
    // affiche true car le thread est à l'état RUNNABLE
    B.start();
    C.start();
    A.join();
    System.out.println(A.isAlive());
    // affiche false car le thread est à l'état TERMINATED
    B.join();
    C.join();
    System.out.println(Thread.currentThread().getName() + " : j'ai fini
      ");
  }
}
```

Java

En appelant la méthode `sleep()`, le `main` aura une forte probabilité de finir le dernier

```
public class Test {
  public static void main(String[] args) throws
     InterruptedException {
    // TODO Auto-generated method stub
    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();
    sleep(5000);
    System.out.println(Thread.currentThread().getName() +
        " : j'ai fini");
  }
}
```

Java

Exercice

- Écrire une classe `Compteur` qui hérite de `Thread` avec un attribut `nom`
- Un compteur a une méthode `run` qui compte de 1 à 7
- À chaque itération, le thread
  - affiche son nom + l'indice de son itération
  - puis appelle la méthode `sleep` pour une durée aléatoire (de 0 à 3 000 millisecondes).
- Quand il finit de compter, il affiche un message contenant son nom et un message du type `a fini de compter en position x` et il affiche sa position
- Créer un `main` avec 5 threads et vérifier l'exécution

Java

Correction : la classe `Compteur`

```
public class Compteur extends Thread {
private String nom;
private static int ordre = 1;
  // constructeur avec le paramètre nom
  public void run() {
    for (int i = 1; i <= 7; i++) {
      System.out.println(nom + " : " + i);
      try {
        sleep((long) (Math.random() * 3000));
      } catch (InterruptedException e) {
        System.err.println(nom + " a été interrompu.");
      }
    }
    System.out.println(nom + " a fini de compter en position "
       + ordre++);
  }
}
```

Java

Le `main`

```
public class Test {
  public static void main(String[] args) {
    Compteur[] compteurs = {
      new Compteur("a"),
      new Compteur("b"),
      new Compteur("c"),
      new Compteur("d"),
      new Compteur("e")
    };
    for (int i = 0; i < compteurs.length; i++) {
      compteurs[i].start();
    }
  }
}
```

Java

Les valeurs de `getState()`

- `NEW` : s'il vient d'être créé
- `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 (après appel de la méthode `wait()`), pour une durée indéterminée, d'une action d'un autre thread (`notify` ou `notifyAll()`)
- `TIMED_WAITING` : s'il est en attente (après appel de la méthode `sleep()`) pour une durée précise.

Java

Explorant les différents états d'un thread

```
public class Test {
  public static void main(String[] args) throws InterruptedException {
    // TODO Auto-generated method stub
    Thread A = new Thread(new TonThread("A"));
    Thread B = new Thread(new TonThread("B"));
    Thread C = new Thread(new TonThread("C"));
    System.out.println("B state : " + B.getState());
    // affiche B state : NEW
    A.start();
    B.start();
    System.out.println("B state : " + B.getState());
    // affiche B state : RUNNABLE ou dans des cas très rares TERMINATED
    C.start();
    System.out.println("B state : " + B.getState());
    // affiche B state : TERMINATED
    System.out.println(Thread.currentThread().getName() + " : j'ai fini
      ");
  }
}
```

Java

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)

Java

En affectant ces priorités, le thread `A` aura des fortes chances de finir dernier

```
public class Test {
  public static void main(String[] args) throws InterruptedException {
    // TODO Auto-generated method stub
    Thread A = new Thread(new TonThread("A"));
    Thread B = new Thread(new TonThread("B"));
    Thread C = new Thread(new TonThread("C"));
    A.setPriority(1);
    Thread.currentThread().setPriority(5);
    B.setPriority(4);
    C.setPriority(10);
    A.start();
    B.start();
    C.start();
    System.out.println(Thread.currentThread().getName() + " : j'ai fini
      ");
  }
}
```

Java

Les interruptions

- Il est possible de demander à un thread d'interrompre son exécution avec la méthode `interrupt()`
- Cette méthode n'interrompt pas brutalement le thread mais elle signale à ce dernier la demande
- Le thread peut vérifier s'il a été interrompu en appelant la méthode `static boolean isInterrupted()`
- Une fois cette méthode est appelée, sa valeur est remise à `false`

Java

Modifions la classe `MonThread`

```
public class MonThread extends Thread {

  public MonThread(String name) {
    super(name);
  }

  @Override
  public void run() {
    for (int i = 0; i < 5; i++) {
      if (!isInterrupted()) {
        System.out.print(this.getName() + " ");
      }
    }
  }
}
```

Java

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();
    A.interrupt();
    System.out.println(Thread.currentThread().
       getName() + " : j'ai fini");
  }
}
```

Java

Considérons l'exemple suivant

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

    public int getCompteur() {
        return compteur;
        }

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

Java

```
public class TestThread implements Runnable{
  MonCompteur TC;
  private String name;
  // ajouter aussi un constructeur à 2 paramètres
  public void run(){
    try{
      for(int i = 0; i < 3; i++){
        if(TC.getCompteur() > 0){
          TC.decrementerCompteur();
          System.out.println("Opération réussie " + TC.getCompteur() +
             " demandée par " + name);
          Thread.sleep(500);
        }
        else
          System.out.println("Échec " + TC.getCompteur() + " demandée
             par " + name);
      }
    }
    catch(InterruptedException e){
      System.out.println(e.getMessage());
    }
  }
}
```

Java

```
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();
  }
}
```

Java

```
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();
  }
}
```

Tous les threads utilisent le même compteur

Java

L'affichage peut être ainsi :

```
opération réussie 2 demandée par          t3
opération réussie 2 demandée par  t1
opération réussie 2 demandée par     t2
opération réussie 1 demandée par     t2
opération réussie -1 demandée par         t3
opération réussie 0 demandée par  t1
Échec  -1  demandée par        t3
Échec  -1  demandée par     t2
Échec  -1  demandée par  t1
```

Java

L'affichage peut être ainsi :

```
opération réussie 2 demandée par          t3
opération réussie 2 demandée par  t1
opération réussie 2 demandée par     t2
opération réussie 1 demandée par     t2
opération réussie -1 demandée par         t3
opération réussie 0 demandée par  t1
Échec  -1  demandée par          t3
Échec  -1  demandée par     t2
Échec  -1  demandée 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

Java

L'affichage peut être ainsi :

```
opération réussie 2 demandée par          t3
opération réussie 2 demandée par  t1
opération réussie 2 demandée par     t2
opération réussie 1 demandée par     t2
opération réussie -1 demandée par         t3
opération réussie 0 demandée par  t1
Échec  -1  demandée par          t3
Échec  -1  demandée par     t2
Échec  -1  demandée 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

Java

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())

Java

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){
            TC.decrementerCompteur();
            System.out.println("Opération réussie " + TC.getCompteur()
               + " demandée par " + name);
            Thread.sleep(500);
          } else
            System.out.println("Échec " + TC.getCompteur() + " demandée
                par " + name);
        }
      }
    } catch(InterruptedException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Java

Problème de synchronisation : solution avec Sémaphore

- C'est une classe Java
- Elle permet d'autoriser plusieurs threads à accéder à une 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;
  // définir un constructeur avec trois paramètres
  public void run(){
    try{
      for(int i = 0; i < 3; i++){
        sem.acquire();
        if(TC.getCompteur() > 0){
          TC.decrementerCompteur();
          System.out.println("Opération réussie " + TC.getCompteur() +
            " demandée par " + name);
          Thread.sleep(500);
        } else
          System.out.println("Échec " + TC.getCompteur() + " demandée
            par " + name);
        sem.release();
      }
    } catch(InterruptedException e){
      System.out.println(e.getMessage());
    }
  }
}
```

Java

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();
  }
}
```