[AM]412 :Programmation répartie 1/6 Concurrence en Java: les threads

DIPLÔME UNIVERSITAIRE DE TECHNOLOGIE – INFORMATIQUE PROGRAMME PÉDAGOGIQUE NATIONAL

SEMESTRE 4 / UE 41 : COMPLÉMENTS D'INFORMATIQUE - MODULE M4102C AT

CHAMPS DISCIPLINAIRES : ARCHITECTURE MATÉRIELLE – SYSTÈMES D’EXPLOITATION - R

ET ANALYSE, CONCEPTION ET DÉVELOPPEMENT D'APPLICATIONS

Rappels du module [AM]311

Processus "lourds": souvent une application en cours d'exécution

- Communication par IPC (zones de mémoire partagée dans le système, files de message, pipes, sockets, ...)

Processus "légers": plusieurs fils d'exécution au sein d'un même processus lourd

- Mémoire partagée
- Problèmes de synchronisation / Exclusion mutuelle

Ordonnancement / temps partagée

Importance accrue du modèle de concurrence lié à l'extension des architectures multi cœurs

La concurrence est un paradigme fondamental même sur une seule unité de traitement

Concurrence en Java

- En général une JVM = un Processus
- Une application Java peut créer des processus lourds à la manière de l'appel système Unix **`exec`** avec **`Runtime.exec()`** ou la nouvelle classe **`java.lang.ProcessBuilder`** (depuis 1.5)
- Une application Java est constitué d'au moins un thread (du point de vue du programmeur): **`main()`**
- En fait il y en a souvent d'autres : garbage collector, GUI Swing, Servlets, RMI, ...
- On peut en créer d'autres ...
- Le développement, le test et le débogage des programmes multithreads peut se révéler si difficile que la priorité doit être donnée à la conception de politiques et de patrons de conception "sûrs" (*thread-safe*).

Problèmes

- Atomicité : ++compteur
- Race condition

Solutions

- Verrous
- Variables volatiles
- Champs de type final
- méthodes et collections **`synchronized`**
- sémaphores
- barrières ...

Gestion des threads

- contrôle direct des objets de type Thread
- contrôle délégué à des **`executors`** (prochaine séance)
- Dans la suite, on illustre le cours avec des exemples issus du tutoriel Oracle :

  http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

  Vous êtes fortement invités à lire ce tutoriel et poser toute question à son sujet au cours des TP.

Création de Threads en Java

`java.lang.Thread`

Méthode 1: dériver la classe **`Thread`**

- la classe ne peut donc plus hériter d'une autre classe

- on crée obligatoirement un objet pour chaque thread

- bon principe OO : si vous ne souhaitez pas modifier le comportement d'une classe, alors il n'est pas utile de la dériver

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {

        (new HelloThread()).start();
    }

}
```

Création de Thread en Java

Méthode 2: fournir une implémentation de **`Runnable`** (méthode plus générale et utilisable avec les **`Executors`**)

java.lang.Runnable permet d'encapsuler un traitement sous la forme d'un composant autonome et réutilisable :

du coup on peut hériter d'une autre classe que **`Thread`** et lancer plusieurs threads sur le même objet

```
public class HelloRunnable implements Runnable
{
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new
HelloRunnable())).start();
    }
}
```

Démarrer un thread

Quelle que soit la méthode de création choisie, le code à exécuter est défini dans sa méthode **run**(),

La méthode **start**() permet de le démarrer de manière asynchrone. Attention, on n'invoque pas **run()** directement comme pour une invocation synchrone habituelle.

À tester avec (dans run()):

System.out.println("Je suis le thread : " + Thread.currentThread().getName());

Illustration des deux méthodes et passage de paramètres + return

- Pour passer des paramètres à un thread, on crée un constructeur : pas d'arguments dans la méthode `run()`. De plus, il n'y a pas de notion de mémoire globale comme avec les *pthreads* en C. Pour partager la mémoire entre threads, on va donc passer les références des objets partagés dans le constructeur.
- Pour récupérer le résultat, on utilise un accesseur sur un champ de la classe choisi pour écrire ce résultat.

```
public class HelloMultiRunnable implements Runnable {
    private String name;
    public HelloMultiRunnable(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void run() {
        System.out.println("Hello from thread “ + Thread.currentThread().getName() + " aka " +
name);
    }
    public static void main(String args[]) {
        (new Thread(new HelloMultiRunnable("A"))).start();
        (new Thread(new HelloMultiRunnable("B"))).start();
        (new Thread(new HelloMultiRunnable("C"))).start();
        (new Thread(new HelloMultiRunnable("D"))).start();
        HelloMultiRunnable h = new HelloMultiRunnable("E");
        Thread t = new Thread(h);
        t.start();
        Thread t2 = new Thread(h);
        t2.start();
    }
}
```

```java
//Implement Runnable Interface...
public class ImplementsRunnable implements Runnable {

	private int counter = 0;

	public void run() {
		counter++;
		System.out.println("ImplementsRunnable : Counter : " +
counter);
	}
}

// Extend Thread class...
class ExtendsThread extends Thread {

	private int counter = 0;

	public void run() {
		counter++;
		System.out.println("ExtendsThread : Counter : " + counter);
	}
}
```

```java
//Use above classes here in main to understand the differences more clearly...
public class ThreadVsRunnable {
    public static void main(String args[]) throws Exception {
        // Multiple threads share the same object.
        ImplementsRunnable rc = new ImplementsRunnable();
        Thread t1 = new Thread(rc);
        t1.start();
        Thread.sleep(1000); // Waiting for 1 second before starting next thread
        Thread t2 = new Thread(rc);
        t2.start();
        Thread.sleep(1000); // Waiting for 1 second before starting next thread
        Thread t3 = new Thread(rc);
        t3.start();
        // Creating new instance for every thread access.
        ExtendsThread tc1 = new ExtendsThread();
        tc1.start();
        Thread.sleep(1000); // Waiting for 1 second before starting next thread
        ExtendsThread tc2 = new ExtendsThread();
        tc2.start();
        Thread.sleep(1000); // Waiting for 1 second before starting next thread
        ExtendsThread tc3 = new ExtendsThread();
        tc3.start();
    }
}
```

Contrôle des threads

`Thread.sleep()`

`interrupt ():` on peut interrompre un thread, mais uniquement sur des instructions interruptibles comme sleep ou certaines I/O : pas une boucle de calcul

`Thread.join():` bloque jusqu'à ce que le thread termine ou qu'un délai expire

Attention, en l'absence de join(), on peut terminer le thread appelant (main par exemple) avant la fin des traitements et ne pas afficher les résultats escomptés!

Tous les programmes ont au moins un thread : main()

D'habitude on invoque une méthode sur un objet de manière synchrone : on reste bloqué en attente du retour. Dans le cas du multithread, on invoque run() de manière asynchrone, il faut invoquer join() au moment où l'on souhaite récupérer un résultat ou pour être sûr que la méthode est terminée et a écrit le résultat dans le champ choisi

Illustration: exemple SimpleThreads du tutoriel Oracle

L'application **`SimpleThread`** est composée de deux threads:

main (comme toutes les applications Java)

MessageLoop (Runnable)

- main crée *MessageLoop* et attend qu'il termine, en cas de délai trop important il l'interrompt
- *MessageLoop* affiche une série de message, s'il est interrompu avant la fin il affiche un message et termine.

```
public class SimpleThreads {
    // Display a message, prefixed by the name of the current thread
    static void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s%n", threadName, message);
    }
    private static class MessageLoop implements Runnable {
        public void run() {
            String importantInfo[] = { "Mares eat oats", "Does eat oats",
                    "Little lambs eat ivy", "A kid will eat ivy too" };
            try {
                for (int i = 0; i < importantInfo.length; i++) {
                    // Pause for 4 seconds
                    Thread.sleep(4000);
                    // Print a message
                    threadMessage(importantInfo[i]);
                }
            } catch (InterruptedException e) {
                threadMessage("I wasn't done!");
            }
        }
    }
```

```
public static void main(String args[]) throws InterruptedException {
    long patience = 1000 * 10; // * 60; // Delay, in milliseconds before we interrupt MessageLoop thread
    if (args.length > 0) {// If command line argument present, gives patience in seconds.
        try {
            patience = Long.parseLong(args[0]) * 1000;
        } catch (NumberFormatException e) {
            System.err.println("Argument must be an integer.");
            System.exit(1);
        }
    }
    threadMessage("Starting MessageLoop thread");
    long startTime = System.currentTimeMillis();
    Thread t = new Thread(new MessageLoop());
    t.start();
    threadMessage("Waiting for MessageLoop thread to finish"); // loop until MessageLoop thread exits
    while (t.isAlive()) {
        threadMessage("Still waiting...");
        t.join(1000); // Wait maximum of 1 second for MessageLoop thread to finish.
        if (((System.currentTimeMillis() - startTime) > patience)  && t.isAlive()) {
            long delay = System.currentTimeMillis() - startTime;
            threadMessage("Tired of waiting! "+ delay + " ms elapsed");
            t.interrupt(); // Shouldn't be long now -- wait indefinitely
            t.join();
        }
    }
    threadMessage("Finally!");
```

Synchronisation des Threads

Nécessaire pour gérer les accès concurrents aux champs mémoire partagés : variables / champs et références des objets concernés.

Les problèmes d'équité, de famine, de vivacité ou d'interblocage ont été partiellement vus en M311 (voir par exemple le problème des philosophes et des spaghettis) et seront traités sur des exemples au fil des TP.

De manière générale, on crée un tableau de threads pour pouvoir faire une boucle join() sur les références de threads crées, et un tableau des objets crées pour pouvoir récupérer les résultats des méthodes invoquées par run()

Moniteurs Java

Chaque objet Java est associé à un moniteur (mutex + condition)

Entrer dans un moniteur (prendre le mutex) : exécuter une méthode **`synchronized`**

Sortir du moniteur : sortir de la méthode

Deux threads différents ne peuvent pas exécuter en même temps des méthodes marquées **`synchronized`** sur un même objet.

Le moniteur est **réentrant**: la méthode **`synchronized`** peut appeler une autre méthode **`synchronized`** sur le même objet sans créer d'interblocage.

Exemple

Soit la classe **`Counter`** du tutoriel:

- Écrire la classe **`TestCounter`** qui crée deux threads et met en évidence le problème d'interférence quand les deux threads utilisent simultanément les méthodes de Counter.
- Écrire une version **`SynchronizedCounter`** qui règle ce problème d'interférence.

```
class Counter {
      private int c = 0;

      public void increment() {
            c++;
      }

      public void decrement() {
            c--;
      }

      public int value() {
            return c;
      }

}
```

Counter

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

```
public class CounterThread implements Runnable
{
    private Counter count;
    private int N;

    public CounterThread(Counter count, int N) {
        this.count = count;
        this.N = N;
    }

    public void run() {
        for (int i = 0; i < N; i++) {
            count.increment();
        }

    }
```

```
    public static void main(String[] args) throws
InterruptedException {

        Counter count = new Counter();
        System.out.println("Initial value for count: " +
count.value());

        CounterThread ct1 = new CounterThread(count, 10000);
        Thread t1 = new Thread(ct1);
        CounterThread ct2 = new CounterThread(count,
        Thread t2 = new Thread(ct2);

        t1.start();
        t2.start();

        t2.join();
        t1.join();

        System.out.println("Final value for count: " +
count.value());
    }
```

```
Initial value for count: 0
Final value for count: 19136
...
Initial value for count: 0
Final value for count: 15928
```

Race condition

Le problème de comptage illustre un danger classique de la concurrence, appelé situation de compétition (*race condition*). Si un threads A invoque **`increment()`** en même temps (ou Presque) que un autre thread B qui invoque **`decrement()`** sur le compteur c (valeur initiale 0), alors on peut avoir l'entrelacement de threads suivant:

1. Thread A: Récupère la valeur de c.
2. Thread B: Récupère la valeur de c.
3. Thread A: Incrémente la valeur récupérée; le résultat est 1.
4. Thread B: Décrémente la valeur récupérée; le résultat est -1.
5. Thread A: Enregistre le résultat dans c; c vaut 1.
6. Thread B: Enregistre le résultat dans c; c vaut -1.

Conséquences parfois dramatiques : Northeast blackout of 2003

```
class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }

}
```

```
public class SynchronizedCounterThread implements Runnable {
    private SynchronizedCounter count;
    private int N;

    public SynchronizedCounterThread(SynchronizedCounter count, int
N) {
        this.count = count;
        this.N = N;
    }

    public void run() {
        for (int i = 0; i < N; i++) {
            count.increment();
        }

    }
```

```
    public static void main(String[] args) throws InterruptedException {

        SynchronizedCounter count = new SynchronizedCounter();
        System.out.println("Initial value for count: " + count.value());

        SynchronizedCounterThread sct1 = new SynchronizedCounterThread(count, 10000);
        Thread t1 = new Thread(sct1);
        SynchronizedCounterThread sct2 = new SynchronizedCounterThread(count, 10000);
        Thread t2 = new Thread(sct2);

        t1.start();
        t2.start();

        t2.join();
        t1.join();

        System.out.println("Final value for count: " + count.value());
    }

}
```

Atomicité

Chaque thread gère une mémoire cache : même si une variable est déclarée **`static`**, un thread peut avoir une valeur erronée de cette variable

Les variables déclarées de type **`volatile`** garantissent que les valeurs seront lues en RAM (commune à tous les threads) et non pas en cache

On peut déclarer des types **`Atomic`**

Les read / write sont atomiques, mais les problèmes de cohérence restent possibles

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
   private AtomicInteger c = new
AtomicInteger(0);

   public void increment() {
      c.incrementAndGet();
   }

   public void decrement() {
      c.decrementAndGet();
   }

   public int value() {
      return c.get();
   }

}
```

Instruction synchronized

On peut réduire la granularité de la synchronisation de toute une méthode à un simple bloc d'instructions :

```
private Object lock = new Object();

   public void increment() {
      synchronized(lock) {
         c++;
      }
   }
```

Conditions : wait(), notify()/notifyAll()

Les moniteurs définissent une file d'attente (condition) unique.

L'ordre des threads réveillés n'est pas garanti.

Si des threads attendent sur **`wait()`**, d'autres doivent appeler **`notify()/notifyAll()`**

```
while(!test){
      wait();
}
```

On exécute la boucle uniquement en cas de notification (pas forcément celle attendue)

Voir **`ProducerConsumerExample`** (tutoriel Oracle)

Illustration des *Guarded Blocks*

Attendre une condition avant d'exécuter un bloc d'instructions

On va réaliser un exemple Producteur / Consommateur:

Le thread producteur crée des données qui seront utilisées par le thread consommateur. Les deux threads communiquent à travers un objet partagé de type *Drop*.

Le consommateur ne doit pas essayer de récupérer des données avant que le producteur n'ait fini de les produire

Le producteur ne doit pas produire de nouvelles données tant que le consommateur n'a pas fini de récupérer les anciennes.

```
public class Drop {
    private String message; // Message sent from producer to consumer
    // true if consumer should wait for producer to send message,
    // false if producer should wait for consumer to retrieve message.
    private boolean empty = true;

    public synchronized String take() {
        while (empty) { // Wait until message is available.
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        empty = true; // Toggle status.
        notifyAll();  // Notify producer that status has changed.
        return message;
    }
```

```
    public synchronized void put(String message) {
        while (!empty) { // Wait until message has been retrieved.
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        empty = false; // Toggle status.
        this.message = message; // Store message.
        notifyAll(); // Notify consumer that status has changed.
    }
}
```

```
public class Producer implements Runnable {
    private Drop drop;

    public Producer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        String importantInfo[] = { "Mares eat oats", "Does eat oats",
                "Little lambs eat ivy", "A kid will eat ivy too" };
        Random random = new Random();

        for (int i = 0; i < importantInfo.length; i++) {
            drop.put(importantInfo[i]);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {
            }
        }
        drop.put("DONE");
    }
}
```

```
public class Consumer implements Runnable {
    private Drop drop;

    public Consumer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        Random random = new Random();
        for (String message = drop.take(); !message.equals("DONE"); message = drop.take()) {
            System.out.format("MESSAGE RECEIVED: %s%n", message);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

```
public class ProducerConsumerExample {
	public static void main(String[] args) {
		Drop drop = new Drop();
		(new Thread(new
Producer(drop))).start();
		(new Thread(new
Consumer(drop))).start();
	}
}
```

```
MESSAGE RECEIVED: Mares eat oats
MESSAGE RECEIVED: Does eat oats
MESSAGE RECEIVED: Little lambs eat ivy
MESSAGE RECEIVED: A kid will eat ivy too
```

Suite

API de plus haut niveau

Références

- The Java™ Tutorials / Lesson: Concurrency http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
- Java Concurrency in Practice - *Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea* - Addison Wesley Professional - May 2006

et d'autres au fil des TPs.