Programmation réseau en Java : les threads

Michel Chilowicz

Master 2 TTT
Université Paris-Est Marne-la-Vallée

Version du 28/02/2013

Plan

Intérêt de la programmation concurrente

Pour permettre le déroulement de plusieurs fils d'exécution en parallèle.
Principales applications :

- Gestion de multiples flux d'E/S
  $\longrightarrow$ permet par exemple pour un serveur de gérer plusieurs clients
- Écriture d'algorithmes exploitant des architectures parallèles.

L'ordonnanceur du système est chargé de planifier l'exécution des fils (threads) sur chacun des processeurs disponibles.

Gestion de multiples flux d'entrée/sortie

Première approche inefficace : utiliser un fil avec des méthodes d'E/S bloquantes avec *timeout* faible : traitement séquentiel des flux nécessaire.
Deux grandes familles d'approches praticables :

1. Le *multithreading* avec méthodes bloquantes : chaque fil s'occupe d'un seul flux. Lorsqu'un fil patiente sur une opération bloquante, l'ordonnanceur redonne la main à un autre fil $\longrightarrow$ optimisation de l'utilisation du temps processeur.
2. L'utilisation d'un seul fil avec des méthodes non-bloquantes. Le programmeur gère lui-même l'ordonnancement et peut utiliser un mécanisme de sélecteur afin d'être informé des flux d'entrée avec données en attente et des flux de sortie pouvant accueillir de nouvelles données.

Processus (process) et fil (thread)

1. Chaque processus dispose de son propre espace de mémoire (tas) non accessible par les autres processus.
2. Un processus peut contenir plusieurs fils, chacun associé à une pile d'appel contenant des variables locales. Un fil peut accéder aux données du tas du processus
   $\longrightarrow$ chaque structure de données doit rester à tout instant cohérente, d'où la nécessité de contrôler les accès concurrents.

Problématiques de la programmation concurrente

1. Verrouillage de ressources pour accès exclusif le plus bref possible mais garantissant la cohérence des données
2. Éviter les cas d'interblocage. Par exemple (t_1, t_2: fils, r_1, r_2: ressources à accès exclusif) :
   1. t_1 verrouille r_1
   2. t_2 verrouille r_2
   3. t_1 est en attente d'acquisition de r_2
   4. t_2 est en attente d'acquision de $r_1 \longrightarrow$ interblocage !
3. Éviter la famine (fils monopolisant des ressources au détriment d'autres)
4. Éviter l'inversion de priorité : un fil de priorité forte t_1 est bloqué par un fil de priorité faible t_2 utilisant une ressource d'accès exclusif à laquelle t_1 veut accéder. Problématique importante pour l'exécution temps réel.

Construction d'une instance de `Thread`

La classe `Thread` représente un fil que l'on peut créer de deux manières :

1. En créant une classe dérivant de `Thread` implantant la méthode abstraite `void run()` contenant le code exécuté par le fil. Le fil est créé en instantiant la classe.
2. En créant une classe implantant l'interface `Runnable` avec une méthode `void run()`, cette classe étant passée au constructeur de `Thread`.

Une fois la classe (ou un descendant de) `Thread` instantiée, le fil commence son exécution par un appel à `void start()`.

Exemple : threads d'incrémentation de compteur

Classe dérivée de Thread

```
public class IncrementingThread extends Thread
{
        public int counter = 0;

        public IncrementingThread()
        {
                super();
        }

        public void run()
        {
                while (! Thread.currentThread().isInterrupted())
                        counter++;
                System.out.println("Counter value: " + counter);
        }
}
...
Thread t = new IncrementingThread();
t.start();
Thread.sleep(WAIT_TIME);
t.interrupt();
...
```

Classe anonyme implantant Runnable

```
final int[] counter = new int[]{0};
Runnable r = new Runnable()
{
        public void run()
        {
                while (! Thread.currentThread().isInterrupted())
                        counter[0]++;
                System.out.println("Counter value: " + counter[0]);
        }
}
Thread t = new Thread(r);
t.start();
Thread.sleep(WAIT_TIME);
t.interrupt();
...
```

Interruption d'une thread

- `interrupt()` est appelé sur une thread pour l'informer que l'on souhaite qu'elle s'arrête (cette méthode met à `true` le drapeau d'interruption ou peut interrompre des opérations bloquantes — comme `Thread.sleep()` ou `Object.wait()` — en lançant `InterruptedException`).
- Il faut régulièrement tester dans le `run()` l'état d'interruption de la thread courante avec `Thread.currentThread().isInterrupted()` et agir en conséquence.
- `isInterrupted()` retourne `false` après la levée d'une `InterruptedException`
- La méthode statique `Thread.interrupted()` permet d'obtenir l'état d'interruption et le réinitialise à `false`.

Mise en sommeil d'une thread

- Mise en sommeil pendant un temps déterminé : `static void Thread.sleep(long millis) throws InterruptedException` `InterruptedException` est lancée si une interruption est demandée sur la thread
- Attente passive jusqu'à la terminaison d'une autre thread : `void Thread.join([long millis]) throws InterruptedException`

Propriétés des fils Java

- Les fils peuvent être organisés en arbre de groupes de fils (`ThreadGroup`). Chaque fil peut rejoindre un groupe à son initialisation (premier argument du constructeur).
- Chaque fil peut être nommé par un `String` (`{get,set}Name()`).
- Chaque fil possède une priorité (`int getPriority()`) modifiable (`void setPriority(int p)`) dans l'intervalle [Thread.MIN_PRIORITY..Thread.MAX_PRIORITY].
- Un fil peut être en mode démon (`boolean isDaemon()`).
  La JVM se termine uniquement lorsqu'il ne reste plus que des fils démons en cours d'exécution (par exemple les fils de gestion du ramasse-miette de la JVM sont en mode démon).

État d'un fil : `Thread.State`

Cinq états possibles pour un fil Java :

1. NEW : l'instance de `Thread` a été créé mais le fil n'a pas encore démarré l'exécution de `run()`.
2. RUNNABLE : le fil est en cours d'exécution par la JVM (mais le fil n'a pas nécessairement la main).
3. BLOCKED : le fil attend pour l'acquisition d'un moniteur (ressource en exclusion mutuelle).
4. WAITING : le fil attend qu'une autre fil réalise une action (après un appel à `wait()` sur un objet) ou à une méthode bloquante d'E/S. Le fil peut être sorti de cet état par un appel à `notify()` sur l'objet, un appel à `interrupt()` sur le fil (provoque une `InterruptedException`) ou par une fermeture du flux d'E/S (lance une exception).
5. TIMED_WAITING : le fil attend la réalisation d'une action avec un délai (timeout).
6. TERMINATED : fin de l'exécution de la méthode `run()`. L'instance de `Thread` continue d'exister tant qu'une référence subsiste.

Obtention des fils en cours d'exécutions

- Ctrl-\ permet d'afficher sur la sortie standard tous les fils et leur état (utile en cas de blocage).
- La méthode statique `Thread.getCurrentThread()` permet d'obtenir le fil courant (appelant la méthode).
- `Thread.getCurrentThread().getThreadGroup()` permet l'obtention du groupe d'appartenance du fil courant.
- La méthode d'instance `ThreadGroup.getParent()` permet de connaître le groupe parent d'un groupe : en répétant l'opération, on obtient le groupe racine (dont le parent est `null`).
- `int ThreadGroup.enumerate(Thread[] t)` place tous les fils actifs du groupe (et de ses sous-groupes) dans le tableau en argument et retourne le nombre de fils actifs.

Exemple d'affichage de fils actifs sur une JVM

```
Full thread dump OpenJDK 64-Bit Server VM (14.0-b08 mixed mode):

"Low Memory Detector" daemon prio=10 tid=0x00000000016be000 nid=0x5751 runnable [0x0000000000000000..0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"CompilerThread1" daemon prio=10 tid=0x00000000016bb000 nid=0x5750 waiting on condition [0x0000000000000000..0x00007fa24963f510]
   java.lang.Thread.State: RUNNABLE

"CompilerThread0" daemon prio=10 tid=0x00000000016b7000 nid=0x574f waiting on condition [0x0000000000000000..0x00007fa249740480]
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" daemon prio=10 tid=0x00000000016b4800 nid=0x574e waiting on condition [0x0000000000000000..0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Finalizer" daemon prio=10 tid=0x0000000001694000 nid=0x574d in Object.wait() [0x00007fa249982000..0x00007fa249982be0]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x00007fa27df41210> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:133)
        - locked <0x00007fa27df41210> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:149)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:177)

"Reference Handler" daemon prio=10 tid=0x000000000168c800 nid=0x574c in Object.wait() [0x00007fa249a83000..0x00007fa249a83b60]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x00007fa27df41078> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:502)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:133)
        - locked <0x00007fa27df41078> (a java.lang.ref.Reference$Lock)

"main" prio=10 tid=0x0000000001623000 nid=0x5748 runnable [0x00007fa298610000..0x00007fa298610e60]
   java.lang.Thread.State: RUNNABLE
        at Test.main(Test.java:5)

"VM Thread" prio=10 tid=0x0000000001687800 nid=0x574b runnable

"GC task thread#0 (ParallelGC)" prio=10 tid=0x000000000162d800 nid=0x5749 runnable

"GC task thread#1 (ParallelGC)" prio=10 tid=0x000000000162f800 nid=0x574a runnable

"VM Periodic Task Thread" prio=10 tid=0x0000000001735800 nid=0x5752 waiting on condition

JNI global references: 601

Heap
 PSYoungGen      total 18432K, used 1581K [0x00007fa27df40000, 0x00007fa27f3d0000, 0x00007fa292940000)
  eden space 15808K, 10% used [0x00007fa27df40000,0x00007fa27e0cb560,0x00007fa27eeb0000)
  from space 2624K, 0% used [0x00007fa27f140000,0x00007fa27f140000,0x00007fa27f3d0000)
  to   space 2624K, 0% used [0x00007fa27eeb0000,0x00007fa27eeb0000,0x00007fa27f140000)
 PSOldGen        total 42240K, used 0K [0x00007fa254b40000, 0x00007fa257480000, 0x00007fa27df40000)
  object space 42240K, 0% used [0x00007fa254b40000,0x00007fa254b40000,0x00007fa257480000)
 PSPermGen       total 21248K, used 2753K [0x00007fa24a340000, 0x00007fa24b800000, 0x00007fa254b40000)
  object space 21248K, 12% used [0x00007fa24a340000,0x00007fa24a5f06e0,0x00007fa24b800000)
```

Serveur TCP en multithread

1. Accepter les connexions entrantes de client dans un fil principal.
2. Création d'au moins un fil par socket connectée à un client. Ce fil lit et écrit sur la socket. Si les opérations de lecture et écriture sont simultanées et non séquentielles, nécessité d'un fil pour l'`InputStream` et d'un autre fil pour l'`OutputStream`.

Squelette de code de serveur TCP concurrent

```
public class ConcurrentTCPServer
{
        public class ClientCommunicator implements Runnable
        {
                private final Socket s;

                public ClientCommunicator(Socket s)
                {
                        this.s = s;
                }

                @Override
                public void run()
                {
                        try {
                                InputStream is = s.getInputStream();
                                OutputStream os = s.getOutputStream();
                                ...
                        } catch (IOException e)
                        {
                                e.printStackTrace();
                        }
                }
        }

        private final int port;

        public ConcurrentTPCServer(int port)
        {
                this.port = port;
        }

        public void launch() throws IOException
        {
                ServerSocket ss = new ServerSocket(port);
                while (! Thread.currentThread().isInterrupted())
                {
                        Socket s = ss.accept(); // Fetch the socket of the next client
                        Runnable r = new ClientCommunicator(s);
                        Thread t = new Thread(r);
                        t.start();
                }
        }
}
```

Crochet d'arrêt pour gérer la fermeture de l'application

Lorsqu'une exception est remontée par la méthode main ou qu'un signal de terminaison est intercepté par la JVM $\longrightarrow$ arrêt brutal du programme.
Problème : certains fichiers ou communications socket peuvent rester dans un état inconsistant.
Un fichier ou une communication peut être terminée par du code spécifié par un crochet d'arrêt.

Gestion des crochets d'arrêt

- Ajout d'un crochet par spécification d'un fil à exécuter lors de la fermeture : `Runtime.getRuntime().addShutdownHook(Thread hook)`.
- Suppression d'un crochet : `Runtime.getRuntime().removeShutdownHook(Thread hook)`.
- Les fils des crochets sont démarrés dans un ordre quelconque et sont en concurrence lors de la fermeture demandée de l'application.
- Le code d'un fil mis en crochet doit être exécuté rapidement sinon la terminaison forcée par le système de la JVM peut être nécessaire (signal Unix SIGKILL).

Exclusion mutuelle

Certains objets doivent pouvoir être accessibles que par un unique fil lors de l'atteinte de certaines instructions.
Sinon risque d'état incohérent.

Instructions atomiques en Java

Certaines instructions sont garanties atomiques (lecture ou écriture d'un *int*, d'une référence d'objet) mais d'autres pas (lecture ou écriture de *long*). Dans tous les cas, une séquence d'instructions ne l'est jamais.
Pour l'exclusion mutuelle, moniteurs (ou verrous) indispensables.

Moniteurs implicites (synchronized)

- Tout objet peut être moniteur.
- Lorsqu'un moniteur est acquis sur un objet par un fil, aucun autre fil ne pourra utiliser le moniteur de l'objet.
- Construction syntaxique : `synchronized (obj) { code à protéger }`.
- Le moniteur est acquis lors de l'entrée dans le bloc `synchronized` et est libéré à la sortie du bloc (que ce soit par sortie classique ou par levée d'exception).
- Une méthode peut être totalement *synchronisée* sur l'instance de l'objet par l'usage du mot-clé en tête de sa déclaration (exemple : `public synchronized String toString() { ...}` équivalent à `public String toString() { synchronized(this) { ... } }`).

Moniteurs explicites : Lock

- Les classes implantant `Lock` peuvent servir de moniteurs plus évolués.
- Utilisation classique :

```
Lock l = new ReentrantLock(); // Creation du verrou
...
l.lock(); // Verrouillage
try {
        ... // Code a executer
} finally {
        l.unlock(); // Dans tous les cas, deverrouillage
}
```

Possibilités avancées de `Lock`

- `boolean tryLock()` : méthode non-bloquante essayant d'acquérir le verrou (si ce n'est pas possible retourne immédiatement faux). Peut permettre à un fil de réaliser des opérations alternatives en attendant avant de retenter le verrouillage.
- `boolean tryLock(long time, TimeUnit unit)` : variante autorisant un bloquage pendant un temps indiqué (sauf interruption).
- `Condition newCondition()` : retourne une nouvelle condition associée au verrou. Le verrou peut ainsi avoir plusieurs conditions sur lesquelles on peut attendre un changement (`Condition.await()`) ou signaler un changement (`Condition.signal()` et `Condition.signalAll()`).

Verrou lecture/écriture : `ReadWriteLock`

- Classe implantant `ReadWriteLock` : `ReentrantReadWriteLock`
- `Lock writeLock()` fournit un verrou pour l'écriture.
- `Lock readLock()` fournit un verrou pour la lecture.
- Les deux verrous sont liés :
  - writeLock déjà acquis $\longrightarrow$ ni writeLock ni readLock ne peut être acquis par un autre fil
  - readLock déjà acquis $\longrightarrow$ writeLock ne peut être acquis par un autre fil mais readLock peut être réacquis.
- Utile pour protéger une structure dont la lecture peut être réalisée par plusieurs fils.

Mécanisme d'attente et de signalisation

- Attente
  - Possibilité de libérer temporairement un moniteur ou verrou en attente de satisfaction d'un condition par `while (! condition) Object.wait();`.
  - Un `wait()` est interruptible par un appel à `interrupt()` sur la thread : lancement de `InterruptedException`.
- Signalisation
  - `Object.notify()` réveille un fil en attente sur le moniteur qui peut reprendre le moniteur.
  - `Object.notifyAll()` réveille tous les fils en attente sur le moniteur et les place en compétition pour le réacquérir.
- La classe `Condition` permet un mécanisme d'attente/signalisation avec plusieurs conditions par verrou : évite un réveil trop intempestif.

Récapitulatif des avantages des verrous (vs. moniteurs)

- Méthode non-bloquante d'acquisition de verrou échouant immédiatement si le verrou n'est pas disponible (*Lock.tryLock()*)
- Moniteur : une seule condition / verrou : plusieurs conditions possibles
  $\longrightarrow$ possibilité de cibler les notifications vers une condition
- Équitabilité possible pour l'acquisition d'un verrou : premier arrivé, premier servi
- Existence de verrous couplés lecture/écriture (`ReadWriteLock`)

Comment arrêter ou mettre en attente temporairement un fil ?

1. Une mauvaise méthode : utiliser les méthodes `stop()`, `pause()` et `resume()`. Peut laisser des données dans un état inconsistant si une zone mutex était en cours d'exécution.
2. Une bonne méthode : introduire une variable indiquant l'état futur désiré (actif, en pause ou stoppé) et la vérifier régulièrement dans la méthode `run()`. Utiliser `interrupt()` pour interrompre une opération bloquante.

Exemple : arrêt ou attente temporaire d'un fil

```
// We often check the threadState variable
public void run() { while( isNotStopped() && ...) ... }

int threadState; // 0 if paused, 1 if active, -1 if stopped

public void setThreadState(int state)
{
        synchronized(this) {
                if (threadState == 1) interrupt(); // interrupt the blocking method
                if (threadState == 0) notify(); // notify the monitor to exit from the wait(
                threadState = state;
        }
}

public boolean isNotStopped()
{
        synchronized(this) {
                while (threadState == 0) wait();
                return (threadState == 1);
        }
}
```