Race . . .
Dead-locks
"Synchroniseurs"

Programmation réseau avec Java

2/7 – Les threads

Olivier Ricou

26 avril 2010

L'écriture d'un programme multitâche est toujours une opération délicate mais indispensable pour écrire un serveur et plus généralement pour écrire des programmes réseau.

Cette partie sur les threads aborde les points suivants :

1. Gestion des threads
2. Les race conditions et la synchronisation
3. Les dead-locks
4. Sémaphores et moniteurs

1. Les threads

1.1. Comment détacher un processus léger (*thread*)

Il existe trois façons de permettre à une classe de créer des objets détachables dans des processus légers :

- la faire hériter de la classe `Thread`,
- lui implanter l'interface `Runnable` et sa méthode `run()`
- lui implanter l'interface `Callable<E>` et sa méthode `call()` (*Java 1.5*)

Hériter de `Thread` est mal car cela :

- prend l'héritage (unique en Java),
- interdit plusieurs threads sur le même objet.

Le thread est lancé

– en appelant la méthode `start()` de `Thread`,

```
Toto extends Thread          Titi implements Runnable

toto.start();                Thread t = new Thread(titi);
                             t.start();
```

– en appelant la méthode `execute()` d'un `Executor` (*Java 1.5*)

```
executor.execute(titi);  // le titi Runnable ci-dessus
executor.execute(new Titi()); // un autre Titi
```

L'`Executor` permet un meilleur contrôle des threads. (cf ci-dessous)

DamnFork.java

```
1  class DamnFork implements Runnable // extends Thread
2  {                                  // dans le cours 1
3    private int id;
4
5    public DamnFork(int i) { // will never finish
6        id = i;
7        new Thread(this).start();
8        DamnFork next = new DamnFork(++i);
9    }
10
11   public void run() {
12     while(true) System.out.println(id);
13   }
14
15   static public void main(String args[]) {
16       DamnFork d = new DamnFork(0);
17   }
18 }
```

1.2. Les `Callable<V>`

`Callable<V>` permet de lancer un thread parallèle comme `Runnable` mais il retourne un résultat :

```
package java.util.concurrent;

public interface Callable<V> {
  /**
   * Computes a result, or throws an exception if
   * unable to do so.
   *
   * @return computed result
   * @throws Exception if unable to compute a result
   */
  V call() throws Exception;
}
```

Un `Callable` est lancé dans un thread par la méthode `submit()` d'un `ExecutorService`.

1.2.1. Le `Future<V>` est asynchrone

Un `Callable` retourne un résultat, mais quand ?

Lorsqu'on lance un thread avec un objet `Callable<V>`, il s'execute en parallèle, aussi cela n'a pas de sens d'écrire

```
String result = pool.submit(my_handler);
```

en attendant le résultat car alors on bloquerait le thread qui lance le thread parallèle.

Aussi, la méthode `submit()` retourne immédiatement un objet de type `Future<V>` où `V` est la même classe que celle de `Callable<V>`.

Les méthodes d'un objet de type `Furture<v>` sont :

- `isDone()` pour savoir si l'execution du thread lancé est terminée
- `cancel()` pour interrompre l'execution du thread,
- `isCancel()` pour savoir si le thread a été interrompu,
- `get()` pour recupérer le resultat si l'execution est finie (avec une option d'attente maximale).

1.3. Fabriquer son `Executor`

En définissant la méthode `execute()` d'un `Executor`, on contrôle de prêt le fonctionnement du lancement des threads comme le montre les exemples suivants :

```
class DirectExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    }
}
```

où l'exécution est dans le thread courant.

Copie de l'ancienne méthode :

```
class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```

1.3.1. L'héritage des Executors

L'`ExecutorService` est pour les `Callable` (cf page suivante).

Le `ScheduledExecutorService` est pour lancer une execution dans un temps déterminé voire de facon répétitive (`cron`).

1.3.2. La fabrique d'executeurs

Les `Executors` de la fabrique `Executors` sont

- pour eviter de reconstruire de nombreux petits threads identiques :

newCachedThreadPool() Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.
newCachedThreadPool(ThreadFactory threadFactory) Idem + uses the provided ThreadFactory to create new threads when needed.

- pour avoir un nombre maximum de threads en même temps :

newFixedThreadPool(int nThreads) Creates a thread pool that reuses a fixed set of threads operating off a shared unbounded queue.
newFixedThreadPool(int nThreads, ThreadFactory threadFactory) Idem + using the provided ThreadFactory to create new threads when needed.

- pour gérer le temps :

newScheduledThreadPool(int corePoolSize) Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.
newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory) Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

Enfin on a Single pour les `xxxThreadPool(1)` (pas redimensionnable) :

newSingleThreadExecutor() Creates an Executor that uses a single worker thread operating off an unbounded queue.
newSingleThreadExecutor(ThreadFactory threadFactory)
newSingleThreadScheduledExecutor() Creates a single-threaded executor that can schedule commands to run after a given delay, or to execute periodically.
newSingleThreadScheduledExecutor(ThreadFactory threadFactory)

FixedPool.java

```
1  // tester Crtl-Alt- durant l'execution
2  import java.util.concurrent.ExecutorService;
3  import java.util.concurrent.Executors;
4
5  public class FixedPool
6  {
7    public static void main(String[] args)
8    {
9      ExecutorService executor
10           = Executors.newFixedThreadPool( 4 );
11
12     for( int i = 0; i < 6; i++ )
13       executor.execute( new CountDown() );
14     System.out.println("On ferme l'executeur");
15     executor.shutdown(); // ferme l'ajout de thread
16   }                      // et permet de finir le main()
17 }
```


```
19  class CountDown implements Runnable
20  {
21    private static int taskCount = 0;
22    private final int id = taskCount++;
23    protected int count = 10, a;
24
25    public void run()
26    {
27      while ( count-- > 0 )
28      {
29        System.out.println("Task"+ id +
30                           ", count = "+count);
31        for (int i=0;i<1E6;i++) a += Math.log(i*i);
32      }
33    }
34  }
```

On constate à l'exécution que le 5e thread ne commence qu'après qu'un des premiers soit fini.

1.4. Opérations de bases de gestion d'un thread en cours

Un thread `Thread.currentThread()` peut :

▷ être endormi pendant N millisecondes	`sleep(N)`
▷ céder la main si quelqu'un la veut, sinon il continue	`yield()`
▷ attendre que le thread `t` se finisse ▷ attendre N millisecondes le thread `t`	`t.join()` `t.join(N)`
▷ attendre qu'un autre thread lui cède la main (`notify`) ▷ attendre au plus N millisecondes qu'un autre thread lui cède la main (`notify`)	`wait()` `wait(N)`
▷ notifier à un autre thread que la voie est libre ▷ notifier à tous les threads que la voie est libre	`notify()` `notifyAll()`

Note : tout programme s'exécute dans un thread.

`wait` et `notify` sont des méthodes de la classe `Object` (donc faire `wait()` plutôt que `Thread.currentThread().wait()`).

1.5. Exceptions et interruptions

On peut gérer l'activité des threads avec `wait` et `notify`.

On peut aussi utiliser les `InterruptedException` :

– soit en utilisant `sleep` et `interrupt()` sur le thread qui dort pour que `sleep` retourne une `InterruptedException`,

– soit en regardant de temps en temps si le thread à reçu une interruption avec `Thread.interrupted()`,

– soit en passant par un `ExecutorService` à qui ont demande d'arrêter les threads qu'il contrôle via `shutdownNow()` (cela aura comme effet d'envoyer des `Thread.interrupted()`).

Interruptable.java

```
1  // manage interruptions:
2  class Interruptable implements Runnable
3  {
4    public void run() {
5      try {
6        while(true) {
7          System.out.println("I'm running");
8          Thread.currentThread().sleep(500);
9            // sleep() catches interrupt()
10       }         // & send InterruptedException
11     }
12     catch (InterruptedException e)
13     { System.out.println("I stop"); }
14   }
15
16 // A la place du sleep, on peut avoir
17 //     if (Thread.interrupted())
18 //               throw new InterruptedException();
```


```
20    static public void main(String args[])
21                throws java.lang.InterruptedException
22    {
23      Interruptable toto = new Interruptable();
24      Thread t = new Thread(toto);
25
26      t.start();
27      Thread.currentThread().sleep(1000);
28      t.interrupt();
29      t.join(); // attend la fin du thread t
30      System.out.println("The end.");
31    }
32  }
```

Donne :

```
I'm running
I'm running
I stop
The end.
```

1.6. La priorité

On peut associer à chaque thread une priorité comprise entre 0 et 10 avec 10 la plus forte priorité et 5 la priorité par défaut.

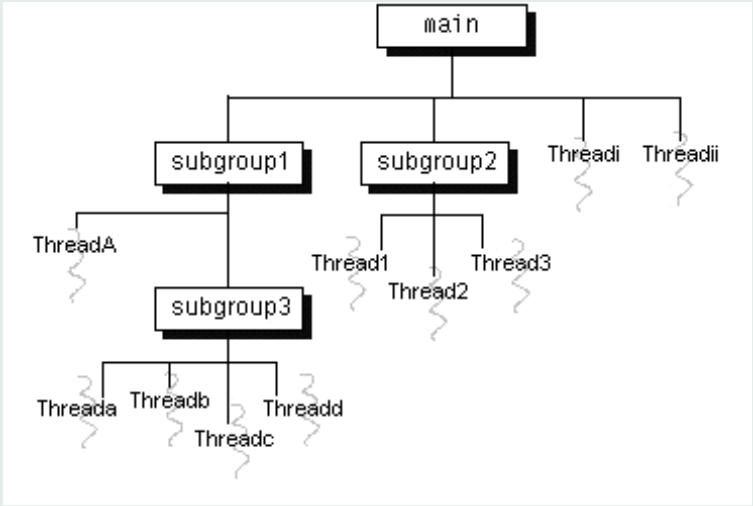
PrintPriority.java

```
1  public class PrintPriority extends Thread
2  {
3    public static void main(String[] args) {
4      PrintPriority t = new PrintPriority();
5      t.setPriority(Thread.MAX_PRIORITY);
6      t.start();
7      for (; ; ) { System.out.print("M"); }
8    }
9    public void run() {
10     for (; ; ) { System.out.print("T"); }
11   }
12 }
```

Il est difficile de crontrôler quoi que ce soit avec les priorités.

1.7. Les groupes de thread

Les `ThreadGroup` permettent d'assembler des threads et des sous-groupes de threads. Il existe un groupe père, le `ThreadGroup` nommé "system".



Les commandes principales des `ThreadGroup` sont :

- `setMaxPriority` pour fixer la priorité maximale des threads du groupe
- `enumerate` pour récupérer les threads ou les sous-groupes du groupe,
- `getParent` pour avoir le groupe père.

PrintGroup.java

```
1  public class PrintGroup implements Runnable
2  {
3    public void run() {}
4
5    public static void main(String[] args)
6    {
7      ThreadGroup tg = new ThreadGroup("Alpha-Bravo");
8      tg.setMaxPriority(Thread.NORM_PRIORITY);
9      Thread t1 = new Thread(tg,new PrintGroup(),"A");
10     Thread t2 = new Thread(tg,new PrintGroup(),"B");
11     ThreadGroup parent =
12             Thread.currentThread().getThreadGroup();
13     while (parent.getParent() != null) {
14       parent = parent.getParent();
15     }
16     parent.list(); // affiche l'arbre des threads
17   }                // et des sous-groupes
18 }
```

Mais le plus simple pour gérer des groupes de threads est encore d'utiliser

- un `ThreadPoolExecutor` pour les `Runnable`
- un `ScheduledThreadPoolExecutor` pour les `Callable`

On pourra ainsi gérer entre autres

- le nombre de threads concurrents en régime de croisière et maximum,
- la taille et le type de la file d'attente (cf les `Queue`)
- les caractéristiques des threads (sachant que par défaut ils appartiendront tous au groupe du pool).

On pourra aussi gérer plus finenent la création de thread en définissant un `ThreadFactory`.

Attention : ces `xxxPoolExecutor` doivent être arrêtés manuellement à l'aide de `shutdown` si on veut que le thread principal finisse.

1.8. L'ordonnancement des tâches

L'ordre dans lequel sont exécutés les threads peut influencer l'exécution du programme :

```
                bool continue = true;
                                  |
                                  |
while(continue) ;                 |continue = false;
```

Si le thread de droite n'a jamais la main, le programme ne se finira jamais.

L'ordonnancement est

- équitable si tout thread non bloqué peut être éxécuté,
- faiblement équitable s'il est équitable et que les threads bloqués par une condition ont une chance que leur condition soit évaluée et donc de repartir,
- fortement équitable s'il est équitable et si un thread bloqué par une condition repart si la condition est régulièrement vraie.

Il existe différentes méthode d'ordonnancement des threads :

- en préemptif, dans ce cas le système *essaye* de donner la main aux threads de priorité plus élevé en attente,
- en coopératif, on attend qu'un thread rende la main pour la passer au suivant,
- en round-robin, tourniquet, où chaque thread passe l'un après l'autre après un temps déterminé.

L'exemple suivant permet de voir que round-robin est faiblement équitable mais pas fortement :

```
boolean continue = true, try = false;
```

```
while (continue)
{ try = true;
  try = false; }
```

```
while (!try) ;
continue = false;
```

Une méthode d'ordonnancement garantissant que l'on sortira de la boucle du 2nd thread serait d'alterner après chaque instruction le thread actif.

La méthode d'ordonnancement de Java n'est pas précisément définie.

1.9. Passer la main

Un thread s'arrête et passe la main dans les cas suivants :

– sur une E/S, le temps de l'E/S,
– sur un objet synchronisé, le temps que l'objet soit libre,
– pour céder la main volontairement, `yield()`,
– pour dormir, `sleep(200)`,
– pour attendre un autre thread, `t.join()`,
– lorsqu'il se met en attente d'une notification, `wait()`,
– lorsqu'il meure, return de `run()`.

On verra qu'un thread peut poser des vérrous, le fait de s'arrêter et passer la main ne libère pas les vérrous pour autant *sauf* après un `wait()`.

1.10. Les démons

Les `daemon` sont des threads détachés en arrière plan qui servent le plus souvent à servir un thread *normal*.

Pour mettre un thread en mode démon, il faut utiliser `setDaemon(true)` avant de le démarrer.

Lorsqu'il n'y a plus de thread normaux actifs, tous les threads démon sont tués.

A l'inverse, si le thread `main()` finit mais qu'il a lancé d'autres threads normaux qui ne sont pas finis (une interface graphique par exemple) alors ces autres threads continuent à tourner.

2. Race conditions et synchronisation

2.1. Partage des variables

Chaque thread a une copie locale des variables de son objet.

Lorsqu'on a plusieurs threads liés à plusieurs objets d'une même classe,

- chaque thread gère les variables de son objet,
- les variables de classe sont communes à l'ensemble des threads.

Si plusieurs threads sont construits à partir d'un même objet *runnable*, alors les variables de l'objet sont communes aux threads.

Compte.java

```
1  public class Compte
2  {
3    private int valeur;
4
5    Compte(int val) { valeur = val; }
6
7    public int solde() { return valeur; }
8
9    public boolean retirer(int somme) throws Interrupt
10     if (somme > 0 && somme <= valeur) {
11       Thread.currentThread().sleep(50);
12       valeur -= somme;
13       Thread.currentThread().sleep(50);
14       return true;
15     }
16     else return false;
17   }
18 }
```

Banque.java

```
1  public class Banque implements Runnable
2  {
3    Compte nom;
4
5    Banque(Compte n) { nom = n; }
6
7    public void Donne(int montant) {
8     System.out.println(Thread.currentThread().getName
9             +": Voici vos " + montant + " francs.");
10   }
11
12   public void ImprimeRecu() {
13     System.out.print(Thread.currentThread().getName(
14     if (nom.solde() > 0)
15       System.out.println(": Il reste " + nom.solde()
16     else
17       System.out.println(": Vous etes fauches!");
18   }
```


```
20    public void Liquide (int montant)
21                          throws InterruptedException
22    {
23      if (nom.retirer(montant)) {
24        Thread.currentThread().sleep(50);
25        Donne(montant);
26        Thread.currentThread().sleep(50);
27      }
28      ImprimeRecu();
29      Thread.currentThread().sleep(50);
30    }
31
32    public void run() {
33      try {
34        for (int i=1;i<10;i++) {
35          Liquide(100*i);
36          Thread.currentThread().sleep(100+10*i);
37        }
38      } catch (InterruptedException e) { return; }
39    }
```


```
41    public static void main(String[] args)
42    {
43      Compte commun = new Compte(1000);
44      Runnable mari = new Banque(commun);
45      Runnable femme = new Banque(commun);
46      Thread tMari = new Thread(mari);
47      Thread tFemme = new Thread(femme);
48      tMari.setName("Conseiller Mari");
49      tFemme.setName("Conseiller Femme");
50      tMari.start();
51      tFemme.start();
52    }
53  }
```

Ce qui donne à l'exécution :

```
(hermes)../java2/cours2>java Banque
Conseiller Mari: Voici vos 100 francs.
Conseiller Femme: Voici vos 100 francs.
Conseiller Mari: Il reste 800 francs.
Conseiller Femme: Il reste 800 francs.
Conseiller Mari: Voici vos 200 francs.
Conseiller Femme: Voici vos 200 francs.
Conseiller Mari: Il reste 400 francs.
Conseiller Femme: Il reste 400 francs.
Conseiller Mari: Voici vos 300 francs.
Conseiller Femme: Voici vos 300 francs.
Conseiller Mari: Vous etes fauches!
Conseiller Femme: Vous etes fauches!
```

La banque a donné 2 fois 300 francs lorsqu'il ne restait que 400 francs sur le compte... Cela est du à la vitesse d'exécution de chaque thread.

On se trouve face à une condition de course, *a Race Condition*.

2.2. `synchronized`

Pour éviter que la condition de course produise ce genre de résultat, il faut bloquer l'écriture sur les variables risquées durant les opérations délicates. Cela peut se faire à différents niveau à l'aide du mot clé `synchronized`.

```
Object ob = new Object;

synchronized(ob) {
...
}
```

Le premier thread entrant dans ce bloc protégé par `synchronized(ob)`, va poser un verrou sur l'objet, verrou qu'il ne retirera qu'à la fin du bloc. Ce verrou empêche un autre thread de modifier `ob` dans un bloc synchronisé sur le même objet.

`synchronized` sur une méthode vérouille l'objet `this` le temps de l'exécution de la méthode.

Dans notre cas, on peut synchroniser la méthode `retirer` du `Compte` afin qu'il n'y ait pas 2 retraits en même temps.

```
public synchronized boolean retirer(int somme)
                                throws InterruptedException
```

ainsi lorsqu'un thread entre dans cette méthode, un autre thread sur le même objet ne peut plus y entrer tant que le premier n'en est pas sorti. On a ainsi :

```
Conseiller Mari: Voici vos 100 francs.
Conseiller Mari: Il reste 800 francs.<-|le retrait de la
Conseiller Femme: Voici vos 100 francs.|Femme a été fait
Conseiller Femme: Il reste 800 francs.
Conseiller Mari: Voici vos 200 francs.
Conseiller Mari: Il reste 400 francs.
Conseiller Femme: Voici vos 200 francs.
Conseiller Femme: Il reste 400 francs.
Conseiller Femme: Il reste 100 francs.<-| la Femme a
Conseiller Mari: Voici vos 300 francs.  | doublé le Mari
Conseiller Mari: Il reste 100 francs.
```

2.3. Mélange de synchronisé et non synchronisés

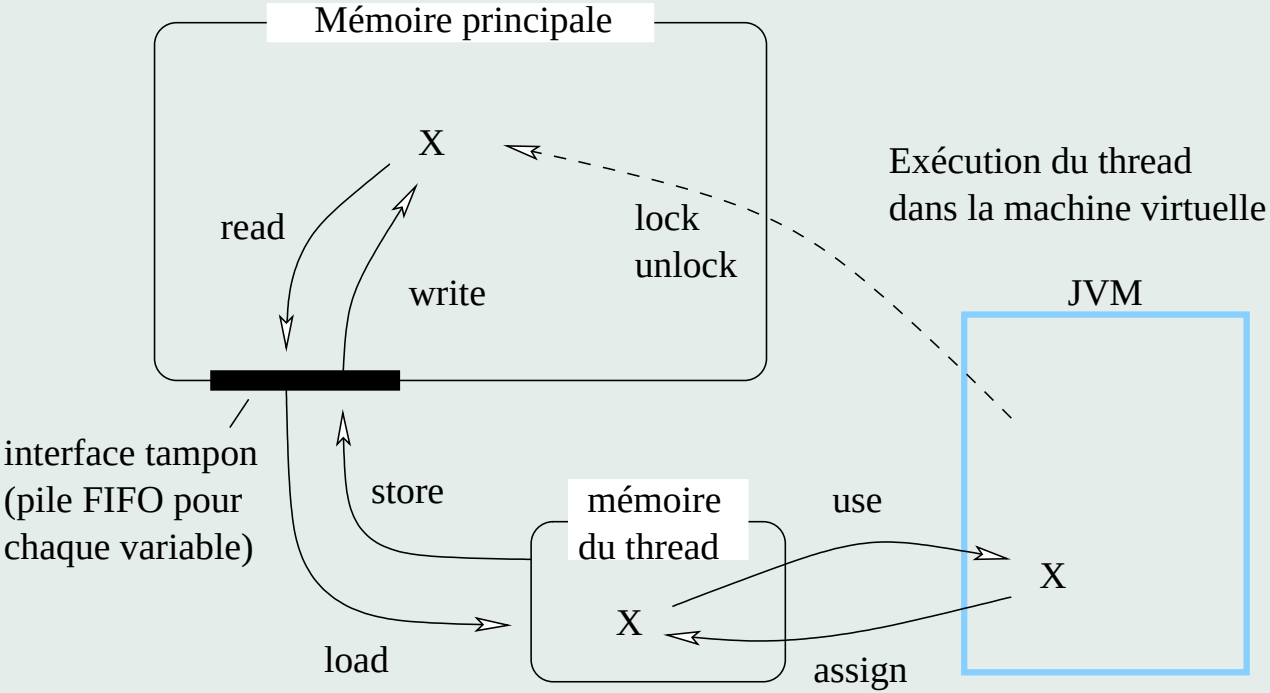
La synchronisation ne bloque que les autres threads synchronisés.

Cela implique en particulier qu'une méthode non synchronisée peut lire ou écrire sur des variables vérouillées par une méthode synchronisée.

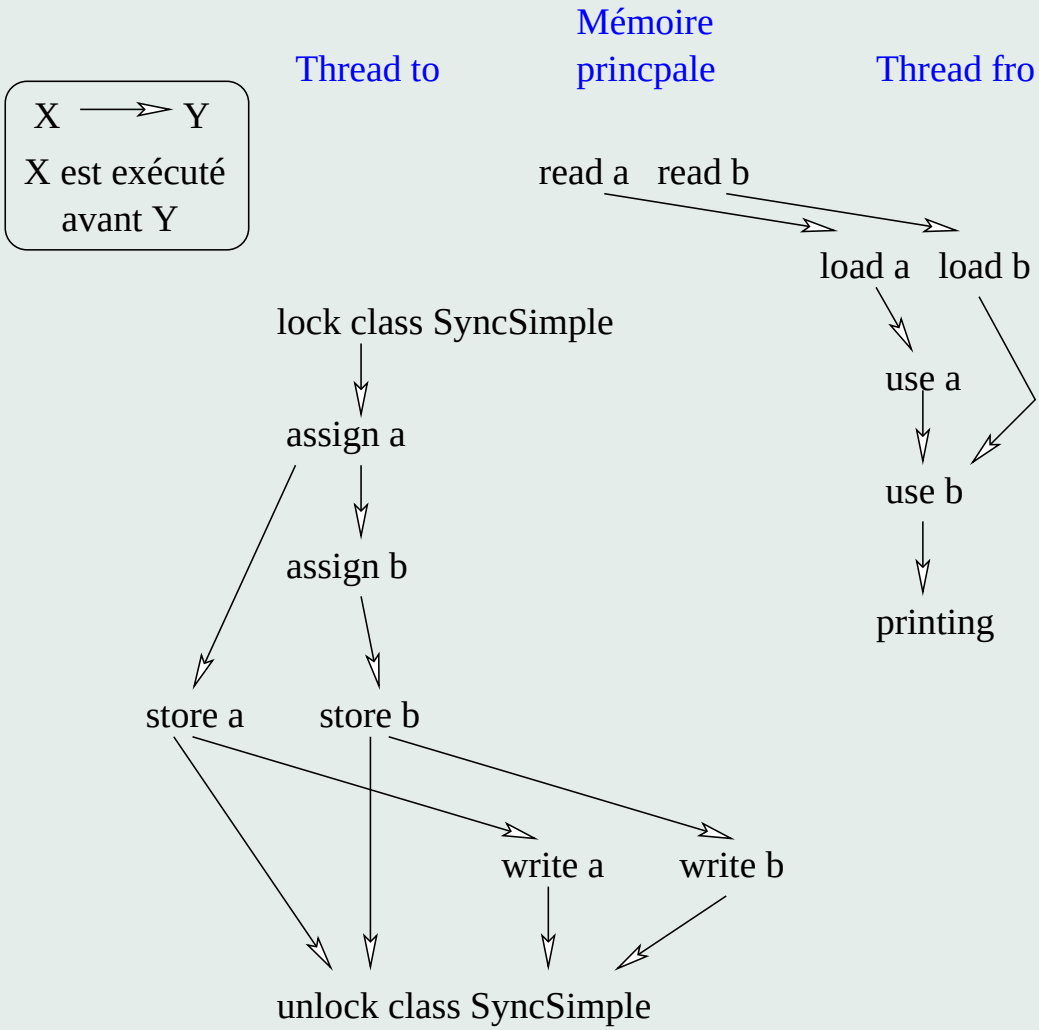
```
1  class SynchSimple{
2    int a = 1, b = 2;
3
4    synchronized void to() {
5      a = 3;
6      b = 4;
7    }
8    void fro() {
9      System.out.println("a="+a+", b="+b);
10   }
11 }
```

Avec un thread lançant `to()` et un autre lançant `fro()`, il sera affiché 1 ou 3 pour `a` et 2 ou 4 pour `b`.

Fonctionnement de la mémoire



La mémoire principale effectue un `read` avant chaque `load` d'un thread et un `write` après chaque `store`.


X → Y
X est exécuté avant Y
Thread to
Mémoire princpale
Thread fro
read a
read b
load a
load b
use a
use b
printing
lock class SyncSimple
assign a
assign b
store a
store b
write a
write b
unlock class SyncSimple

2.4. Exemple du double-check locking

L'idée du double-check locking, DCL, est de ne lancer le constructeur, que l'on imagine long, que si l'on a bien besoin de l'objet :

```
class SomeClass {
  private Resource resource = null;

  public Resource getResource() {
    if (resource == null)
      resource = new Resource();
    return resource;
  }
}
```

Le problème est qu'avec plusieurs threads travaillant sur le même objet on risque de créer 2 fois la Resource.

On peut s'en sortir en synchronisant la méthode `getResource`.

Cela marche mais synchroniser une méthode la rend nettement plus lente car il faut synchroniser la mémoire cache du thread avec la mémoire principale. Dans notre cas, on a un facteur 20 avec Java 1.3 d'IBM.

Alors on écrit l'astuce suivante pour éviter de ne passer par la synchronisation que lorsque `resource` n'est pas `null` :

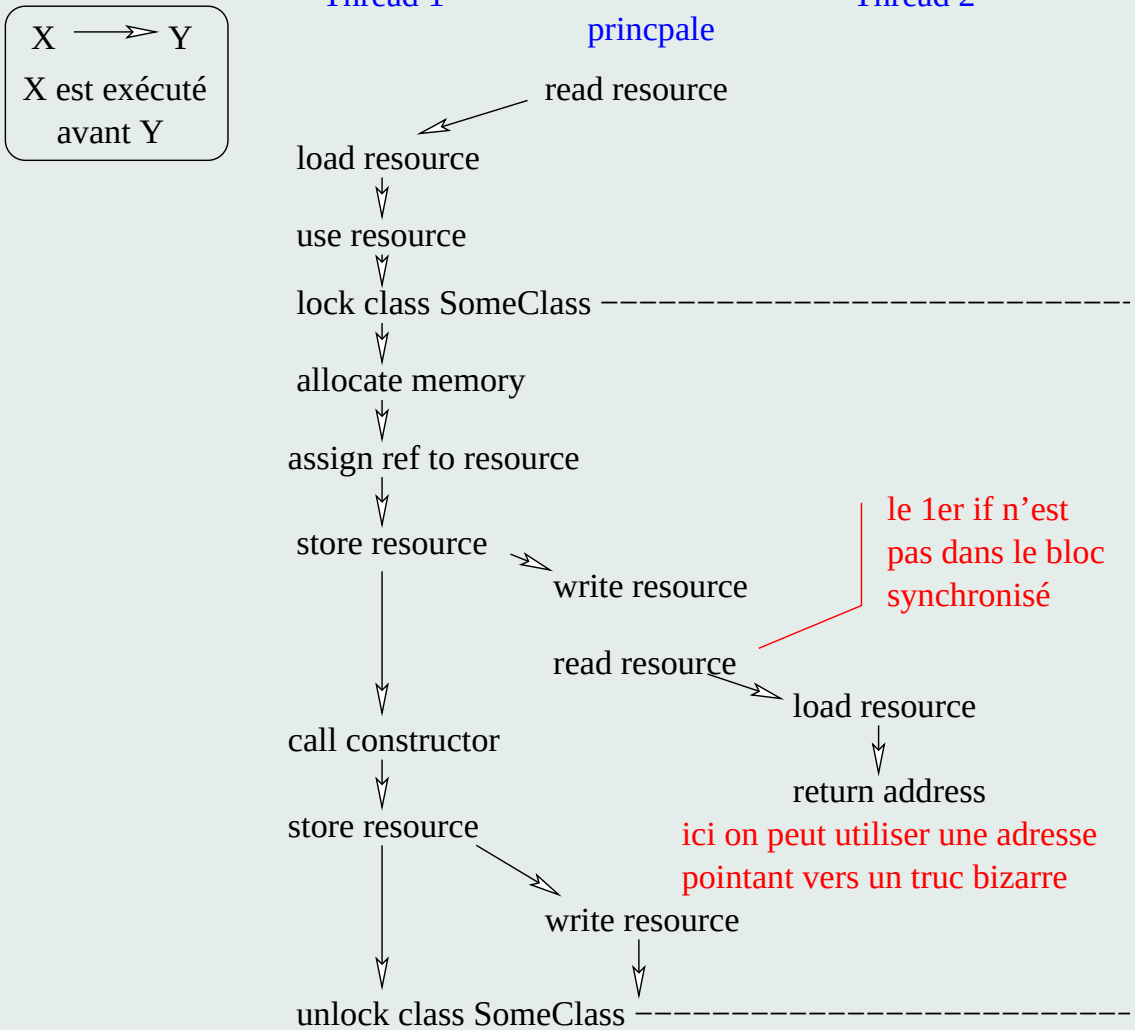
```
class SomeClass {
  private Resource resource = null;

  public Resource getResource() {
    if (resource == null) {
      synchonized {
        if (resource == null)
          resource = new Resource();
      }
    }
    return resource;
  }
}
```

Avec certains compilateurs, cela peut être parfait, malheureusement cela peut ne pas marcher avec d'autres compilateurs.

Les règles de gestions des différentes mémoires sont assez souples pour permettre au compilateur d'optimiser la vitesse d'exécution.

Voici un exemple d'exécution qui ne marche pas comme on l'espère :


X → Y
X est exécuté avant Y
Thread 1
Mémoire princpale
Thread 2
read resource
load resource
use resource
lock class SomeClass
allocate memory
assign ref to resource
store resource
write resource
le 1er if n'est pas dans le bloc synchronisé
read resource
load resource
return address
call constructor
store resource
ici on peut utiliser une adresse pointant vers un truc bizarre
write resource
unlock class SomeClass

2.5. Gros grain

Si `synchronized` permet d'éviter que 2 threads se gènent durant une méthode, on peut avoir le même type de problème au niveau supérieur :

```
private int foo;
public synchronized int getFoo() { return foo; }
public synchronized void setFoo(int f) { foo = f; }
```

protège la variable `foo` parfaitement, mais si 2 threads exécutent

```
setFoo(getFoo() + 1);     |    setFoo(getFoo() + 1);
```

`foo` sera augmenté de 1 ou de 2 suivant l'ordre des appels aux méthodes...

2.6. Synchronisation des méthodes statiques

On peut synchroniser une méthode statique, l'objet bloqué sera celui représentant la classe de l'objet, `Fred.class` ici :

```
class Fred {
  public static synchronized void sayHi() {
    System.out.println("Hi!");
  }
}
```

est équivalent à

```
class Fred {
  public static void sayHi(){
   synchronized (Fred.class){System.out.println("Hi!");}
  }
}
```

Il n'y a qu'un objet qui représente une classe donc plusieurs threads exécutant `sayHi` devront attendre que le vérrou sur `Fred.class` soit levé.

2.7. Synchronisation sur les variables : `volatile`

Un thread est obligé de synchroniser sa mémoire local avec la mémoire principale lorsqu'il
- entre ou sort d'une section synchronisée,
- exécute soit `wait`, `notify` ou `notifyAll`.

Que se passe-t-il si un thread fait tourner en boucle `one` et un autre `two` ?

```
1  class TestVolatile1
2  {
3      static int i = 0, j = 0;
4
5      static void one() {i++; j++; }
6
7      static void two() {
8         System.out.println("i="+i+" j="+j);
9      }
10 }
```

Le thread de `one` actualisera les valeurs de i et j en mémoire principale quand il veut et surtout pas obligatoirement ensemble. On peut tout à fait avoir $j > i$ dans la mémoire principale.

En synchronisant les 2 méthodes on a toujours $i = j$ en mémoire et à l'affichage.

En rendant `volatile` les variables i et j, on impose au thread d'actualiser ces variables à chaque accès.

```
1  class TestVolatile1
2  {
3      static volatile int i = 0, j = 0;
```

Aussi le thread de `one` synchronise toujours ses variables et donc on n'aura jamais $j > i$ en mémoire principale.

Par contre il se peut que `two` affiche $j > i$, j pouvant être lu en mémoire principale longtemps après la lecture de i.

Note : `volatile` n'est pas toujours parfaitement implanté dans les JVMs... (seulement à partir de la version 1.4 pour le Java de Sun)

2.8. L'atomicité

Une opération atomique s'effectue d'un seul bloc c.a.d. qu'elle ne peut pas être interrompue.

En Java de base ce genre d'opération est rare, il s'agit uniquement de l'affectation et de la lecture des types simples tenant sur un mot (4 octets sous Intel).

L'affectation et la lecture de variables de type `long` ou `double` ne sont pas des opérations atomiques. Il est vivement recommandé de les déclarer `volatile` !

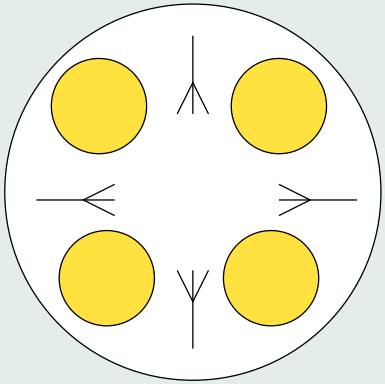
La synchronisation d'une méthode ou d'un bloc ne garanti absolument pas l'atomicité de la méthode ou du bloc (cf les exemples précédants).

`java.util.concurrent.atomic` ajoute de nouvelles variables atomiques dont :

- `AtomicLong`,
- `AtomicLongArray` (la mise à jour des variables est atomique),
- `AtomicReference<V>`.

3. Dead-locks

Le dîner des philosophes :



Chaque philosophe doit prendre les 2 fourchettes autour de son assiette pour manger ses spaghettis.

L'algorithme naïf consiste à dire à chaque philosophe de prendre sa fourchette de gauche puis sa fourchette de droite.

Si chacun prend sa fourchette de gauche, personne ne pourra prendre sa fourchette de droite et tout le monde meurt de faim.

C'est un interblocage ou *dead-lock*.

Deadlock.java

```
1  public class Deadlock extends Thread
2  {
3    static Object o1 = new Object();
4    static Object o2 = new Object();
5    boolean do01first;
6
7    public Deadlock(boolean is_first)
8    {
9      this.do01first = is_first;
10   }
11
12   public static void main(String[] args)
13   {
14     Deadlock d1 = new Deadlock(true);
15     Deadlock d2 = new Deadlock(false);
16     d1.start();
17     d2.start();
18   }
```


```
20    public void run() {
21      for (; ; ) {
22        if (doO1first) {
23          synchronized (o1) {      // acquire o1
24            synchronized (o2) {    // acquire o2
25              System.out.println(Thread.currentThread(
26                                 + "Got both monitors");
27            }                      // release o2
28          }                        // release o1
29        } else {
30          synchronized (o2) {      // acquire o2
31            synchronized (o1) {    // acquire o1
32              System.out.println(Thread.currentThread(
33                                 + "Got both monitors");
34            }                      // release o1
35          }                        // release o2
36        }
37      }                            // end for(;;) loop
38    }
39  }
```

Pour voir ce qui bloque, on peut demander à Java une analyse en tapant `Ctrl-\` sous UNIX et infoCtrl-BREAK sous Windows.
On peut aussi utiliser `jstack <ID>` avec l'ID obtenue par `jps`.

Le résultat dépend de la machine virtuelle.

```
...
Java stack information for the threads listed above:
===================================================
"Thread-2":
        at Deadlock.run(Deadlock.java:32)
        - waiting to lock <0x44242558> (a java.lang.Obje
        - locked <0x44242560> (a java.lang.Object)
"Thread-1":
        at Deadlock.run(Deadlock.java:25)
        - waiting to lock <0x44242560> (a java.lang.Obje
        - locked <0x44242558> (a java.lang.Object)

Found 1 deadlock.
```

Conclusion : Evitez d'acquérir 2 objets ensemble.

Race . . .
Dead-locks
"Synchroniseurs"

4. "Synchroniseurs"

Les synchroniseurs, dont les sémaphores et moniteurs, sont des modèles d'objet bien utiles pour contrôler le déroulement des threads.

4.1. Les sémaphores

Les sémaphores sont un système de vérouillage défini par Dijkstra à la fin des années 60. On les retrouve pour la synchronisation sous UNIX.

Les sémaphores ont plusieurs utilisations :

- le démarrage retardé de threads,
- l'accès exclusif à une partie critique,
- la limitation du nombre n de thread pouvant intervenir dans une même partie.

Semaphore.java

```
1  public class Semaphore
2  {
3    int n;
4
5    public Semaphore(int max) { n = max; }
6
7    public synchronized void P() throws InterruptedExc
8      while (n == 0) wait();
9      n--;
10   }
11
12   public synchronized void V() {
13     n++;
14     notify();
15   }
16 }
```

P du hollandais “ passeren” signifie passer et V “vrijgeven” libérer.


```
static Semaphore sema = new Semaphore(n);
```

`n=0` : En bloquant dès le début le sémaphore, on retarde le lancement des threads qu'il contrôle jusqu'à ce qu'une intervention en libère un à l'aide de `sema.V()`.

```
while (true) {
  sema.P();
  // partie critique
}
```

`n=1` : Ce type de sémarphore, appelé mutex pour *mutual exclusion* permet de bloquer le thread avant la partie critique si un autre est dedans.

```
sema.P();
// partie critique
sema.V();
```

`n>1` : Il est aussi intéressant d'utiliser un sémaphore autorisant un nombre limité d'accès à la partie critique.

Ainsi un `Semaphore(3)` garanti qu'il n'y aura jamais plus de 3 threads à l'ouvrage en même temps.

Malheureusement si les sémaphores règlent le problème de l'accès concurrent à une partie critique, ils n'évitent pas les dead-locks :

1er thread

```
sema1.P();
sema2.P();
...
sema2.V();
sema1.V();
```

2nd thread :

```
sema2.P();
sema1.P();
...
sema1.V();
sema2.V();
```

Une simple faute de frappe risque aussi de générer un interblocage :

```
sema1.P()
...
sema1.P()
```

Les sémaphores sont des bêtes délicates à manier.

Dans *Java 1.5*, les sémaphores sont une classe.

L'exemple suivant créé un mutex ayant un comportement "fair" i.e. qui donne la main aux threads dans l'ordre où ils ont été bloqués (FIFO).

```
Semaphore s = new Semaphore(1, true);

s.acquire();          // s.P() en Dijkstra
value = balance +1; // section critique exclusive
s.release();          // s.S() en Dijkstra
```

4.2. Les autres "synchroniseurs" de *Java 1.5*

En plus des `Semaphores`, *Java 1.5* a introduit :

`CountDownLatch`, un loquet pour bloquer plusieurs threads jusqu'à ce qu'un certain nombre d'opérations soient effectuées :

```
CountDownLatch doneSignal = new CountDownLatch(2);
...
doneSignal.countDown();
...
doneSignal.countDown(); // les autres threads repartent
```

et pendant ce temps, les autres processus qui partagent `doneSignal` ont lancé

```
doneSignal.await();
```

`CyclicBarrier`, une barrière qui bloque les threads y arrivant jusqu'a ce que le dernier arrive,

```
barrier = new CyclicBarrier(3);
```

`...`	`...`	`...`
`barrier.await();`	`barrier.await();`	`barrier.await();`
`...`	`...`	`...`
`barrier.await();`	`barrier.await();`	`barrier.await();`

Il est possible d'associé une action à lancer lorsque le dernier arrive à la barrière (voir les constructeurs de `CyclicBarrier`).

`Exchanger<V>` pour échanger des donnés avec un autre thread.

```
Exchanger<DataBuffer> exchanger = new Exchanger();
...
currentBuffer = exchanger.exchange(currentBuffer);
// echangera le buffer avec le prochain thread qui
// lance la meme commande
```