Cours 8 : Programmation concurrente

Cours 9 : Programmation concurrente

Concurrence

- Whatizit ?

Concurrence

- Whatizit ?
- Exécution de plusieurs “codes” en “même temps” sur la **même machine**.
  - Plusieurs fois le même (serveur web...).
  - Du code différent (Jeu/affichage/IA...)

Concurrence

- Whatizit ?
- Exécution de plusieurs “codes” en “même temps” sur la **même machine**.
  - Plusieurs fois le même (serveur web...).
  - Du code différent (Jeu/affichage/IA...)
- Différent du distribué (plusieurs machines et communications entre machines).

Pourquoi ?

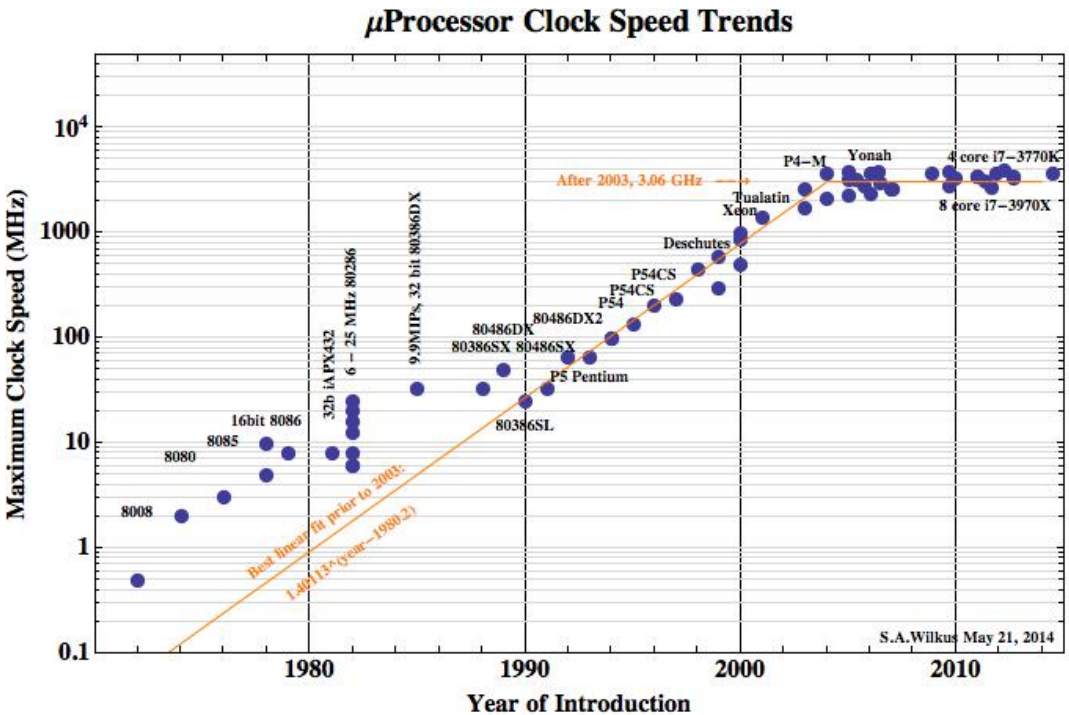
- Intérêt ?

Pourquoi ?

- Intérêt ?
- Si programme avec des **entrées-sorties bloquantes**.
  - On traite une autre requête pendant l'attente de l'entrée-sortie.
- Si gros calculs et **multi-coeurs**...

Pourquoi ?

- "Free lunch is over" (Herb Sutter).
- Jusqu'au début des années 2000, les **processeurs étaient de plus en plus rapide**.
  - Un programme était accéléré automatiquement avec le temps !
- Lois de la physique : pas de processeurs 10GHz maintenant...
- Mais plus de cœurs...
  - **Si on veut accélerer un programme, il faut utiliser les différents cœurs**...

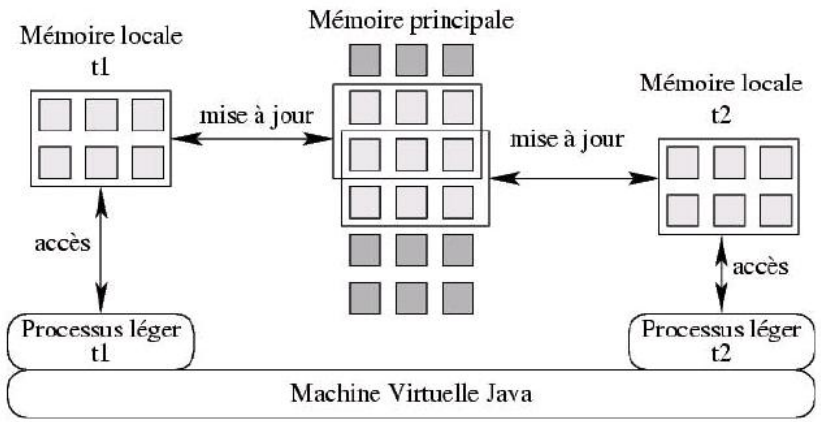


Comment ?

- Plusieurs “threads” (fils d’exécutions ?).
- Chaque thread à sa propre pile (stack) avec stockage de valeurs.
- Une mémoire partagée entre les threads (tas ou heap) pour échange d’information entre threads.
- Un ordonnanceur pour **partager le temps de calcul** sur le processeur.

Comment ?

- Les variables locales sont stockées sur la pile du thread.
- Les champs des objets alloués par new sont stockés sur le tas.



Problèmes ?

- Plus compliqué..
- Le programmeur n'a pas la main sur l'archi du CPU, sur le cache, sur l'OS etc...
- Cohérence des variables partagées entre threads ?
- Préemption du scheduleur qui peut arrêter un thread quand et où il veut.
- Ordre entre les threads...

Concurrence

Classe Thread

- On cherche à exécuter un certain code :

```
1  public class Code implements Runnable {
2    public void run() {
3      System.out.println("Do the maths !") ;
4    }
5  }
```

- On crée un objet `Thread` basé sur ce code et on le démarre.

```
1  public static void main(String[] args) {
2    Runnable code = new Code() ;
3    Thread thread = new Thread(code) ;
4    thread.start() ;
5  }
```

- Lors du start :
  - Création d'une nouvelle pile.
  - Création d'un thread système.
  - Exécution du run du runnable.
  - Une fois fini, la thread meurt puis GC.

Runnable ou Thread ?

- Deux manières de créer un thread :

```
1  public class MyThread extends Thread{
2
3   @Override public void run() {
4    System.out.println("run") ;
5   }
6   public static void main(String[] a){
7    Thread t = new MyThread() ;
8    t.start() ;
9   }
10 }
```

Etendre Thread

```
1  public class MyThread2 implements
       Runnable{
2
3    @Override public void run() {
4    System.out.println("Run") ;
5   }
6   public static void main(String[] a) {
7    Thread t=new Thread(new MyThread2()) ;
8    t.start() ;
9   }
10 }
```

Implémenter Runnable

- Que préférer ?

Runnable ou Thread ?

- Deux manières de créer un thread :

```
1  public class MyThread extends Thread{
2
3   @Override public void run() {
4    System.out.println("run") ;
5   }
6   public static void main(String[] a){
7    Thread t = new MyThread() ;
8    t.start() ;
9   }
10 }
```

Etendre Thread

```
1  public class MyThread2 implements
       Runnable{
2
3    @Override public void run() {
4    System.out.println("Run") ;
5   }
6   public static void main(String[] a) {
7    Thread t=new Thread(new MyThread2()) ;
8    t.start() ;
9   }
10 }
```

Implémenter Runnable

- Que préférer ?
- `MyThread` ne peut plus étendre une autre classe ensuite !!
- Un Runnable peut être passé à autre chose qu'un Thread...

Runnable ou Thread ?

- Affiche ?

```
1  public class Mystere {
2    String name="Denis" ;
3    void setName(String name) {
4      this.name=name ;
5    }
6    void setNameThread() throws InterruptedException{
7      Thread t=new Thread() {
8        @Override public void run() { setName("Cornaz") ; }
9      } ;
10     t.start() ;
11     t.join() ; // attend la mort de t
12     System.out.println(name) ;
13   }
14   public static void main(String[] args) throws InterruptedException {
15     new Mystere().setNameThread() ;
16   }
17 }
```

Runnable ou Thread ?

- Affiche ?

```
1  public class Mystere {
2    String name="Denis" ;
3    void setName(String name) {
4      this.name=name ;
5    }
6    void setNameThread() throws InterruptedException{
7      Thread t=new Thread() {
8        @Override public void run() { setName("Cornaz") ; }
9      } ;
10     t.start() ;
11     t.join() ; // attend la mort de t
12     System.out.println(name) ;
13   }
14   public static void main(String[] args) throws InterruptedException {
15     new Mystere().setNameThread() ;
16   }
17 }
```

- Denis...
- setName est une méthode de `Thread` : redéfinition !
- `syso(this)` dans le run :Thread[Cornaz,5,main]

Ordonnancement

- L'ordonnanceur fait comme il veut...
  - On ne peut **pas contrôler l'ordre** d'exécution entre les threads.
  - Cet **ordre change** d'une exécution à une autre... AIE AIE AIE.
  - Est équitable : quand le temps va vers l'infini, chaque thread aura eu autant de temps de calcul.

Ordre...

```
1  public class MyThread2 implements Runnable{
2    @Override
3    public void run() {
4      //affiche le nom du thread courrant
5      System.out.println(Thread.currentThread()) ;
6    }
7    public static void main(String[] args) {
8      Thread t = new Thread(new MyThread2()) ;
9      t.start() ;
10
11     Thread t2 = new Thread(new MyThread2()) ;
12     t2.start() ;
13   }
14 }
```

- Affiche ?

Ordre...

```
1  public class MyThread2 implements Runnable{
2    @Override
3    public void run() {
4      //affiche le nom du thread courrant
5      System.out.println(Thread.currentThread()) ;
6    }
7    public static void main(String[] args) {
8      Thread t = new Thread(new MyThread2()) ;
9      t.start() ;
10
11     Thread t2 = new Thread(new MyThread2()) ;
12     t2.start() ;
13   }
14 }
```

- Affiche ?
-

```
1  $ java MyThread2
2  Thread[Thread-0,5,main]
3  Thread[Thread-1,5,main]
4  $ java MyThread2
5  Thread[Thread-1,5,main]
6  Thread[Thread-0,5,main]
7  $ java MyThread2
8  Thread[Thread-1,5,main]
9  Thread[Thread-0,5,main]
```

Ca va trancher

- Attention, le scheduleur **peut couper le thread** courant :
  - Au milieu d'un test.
  - Au milieu d'un affichage.
  - Au milieu d'un changement de valeurs...

Attendre

- On peut attendre la fin de l'exécution d'un Thread : `t1.join() ;`
- (bloquant)
- Après cet appel, on est certain que le code du run de t1 est terminé.

Attendre

```
1    Thread t1 = new Thread(new Runnable() {
2      @Override public void run() {
3        System.out.println("Denis") ;
4      }
5    }) ;
6    t1.start() ;
7    System.out.println("Cornaz") ;
8    t1.join() ;
9    System.out.println("Bye") ;
```

- Affiche ?

Attendre

```
1    Thread t1 = new Thread(new Runnable() {
2      @Override public void run() {
3        System.out.println("Denis") ;
4      }
5    }) ;
6    t1.start() ;
7    System.out.println("Cornaz") ;
8    t1.join() ;
9    System.out.println("Bye") ;
```

- Affiche ?
- Bye APRES le reste.
- Ordre sur les 2 premiers aléatoire.

Terminer

- Un thread est terminé lorsque le code de son run() est terminé.
- **Pas possible de tuer un Thread**.
  - destroy() sur Thread est non implémenté, volontairement (pas de relachement de verrous...)

Terminer

- De manière **coopérative** via des FLAGS.
  - `t.interrupt()` : pose un flag.
  - Vérifiable dans le run via `Thread.interrupted()` ou `t.isInterrupted()`
  - Attention, interrupted remet à FAUX le FLAG après coup ! (nom trompeur donc).

Terminer

```
1   Thread t = new Thread(new Runnable() {
2     @Override public void run() {
3       int i=0 ;
4       while( !Thread.interrupted()) {
5         //attente active, chauffe  !
6         i++ ;
7       }
8       System.out.println(i) ;
9     }
10  }) ;
11  t.start() ;
12  t.interrupt() ;
```

Terminer

```
1   Thread t = new Thread(new Runnable() {
2     @Override public void run() {
3       int i=0 ;
4       while( !Thread.interrupted()) {
5         //attente active, chauffe  !
6         i++ ;
7       }
8       System.out.println(i) ;
9     }
10  }) ;
11  t.start() ;
12  t.interrupt() ;
```

- InterruptedException levée si `interrupt()` est demandée sur une thread qui était bloquée sur un `sleep`, un `join` ou un appel entrée/sortie.

Cours 9 : Programmation concurrente

Mystère

```
1  public class Counter {
2    private int value ;
3    public void add10000() {
4      for(int i = 0 ; i < 10000 ; i++) {
5        value++ ;
6      }
7    }
8    public static void main(String[] args) throws InterruptedException {
9      Counter counter = new Counter() ;
10     Runnable runnable = new Runnable() {
11       @Override public void run() {
12         counter.add10000() ;
13       }
14     } ;
15     Thread t1 = new Thread(runnable) ;
16     Thread t2 = new Thread(runnable) ;
17     t1.start() ; t2.start() ;
18     t1.join() ; t2.join() ;
19     System.out.println(counter.value) ;
20   }
21 }
```

- Affiche ?

Mystère

- Affiche ?
  - 14213 (??)
  - 20000 (ah !)
  - 12157 (???)

Mystère

- Affiche ?
  - 14213 (??)
  - 20000 (ah !)
  - 12157 (???)
- Pourquoi ?

Mystère

- Affiche ?
  - 14213 (??)
  - 20000 (ah !)
  - 12157 (???)
- Pourquoi ?
  - Une thread peut être deschedulée entre lecture de la valeur et incrémentation.
  - `i++` n'est pas une opération **atomique**, mais une lecture, une incrémentation et une écriture.

Mystère

- Affiche ?
  - 14213 (??)
  - 20000 (ah !)
  - 12157 (???)
- Pourquoi ?
  - Une thread peut être deschedulée entre lecture de la valeur et incrémentation.
  - `i++` n'est pas une opération **atomique**, mais une lecture, une incrémentation et une écriture.
- Solution ?

Mystère

- Affiche ?
  - 14213 (??)
  - 20000 (ah !)
  - 12157 (???)
- Pourquoi ?
  - Une thread peut être deschedulée entre lecture de la valeur et incrémentation.
  - `i++` n'est pas une opération **atomique**, mais une lecture, une incrémentation et une écriture.
- Solution ?
  - Rendre `i++` atomique : pas possible.
  - Empêcher la lecture tant qu'écriture non terminée : **section critique**.

Section critique

- Indiquer qu'une seule thread à la fois peut rentrer dans la "section critique".

```
1  public class Counter {
2    private int value ;
3    public void add10000() {
4      for(int i = 0 ; i < 10000 ; i++) {
5        //debut section critique
6        value++ ;
7        //fin section critique
8      }
9    }
10   public static void main(String[] args) throws InterruptedException {
11     Counter counter = new Counter() ;
12     Runnable runnable = new Runnable() {
13       @Override public void run() {
14         counter.add10000() ;
15       }
16     } ;
17     Thread t1 = new Thread(runnable) ;
18     Thread t2 = new Thread(runnable) ;
19     t1.start() ; t2.start() ;
20     t1.join() ; t2.join() ;
21     System.out.println(counter.value) ;
22   }
```

Section(s) critique(s)

- Mais là ?

```
1  public class Counter {
2    private int value ;
3    public void add10000() {
4      for(int i = 0 ; i < 10000 ; i++) {
5        //debut section critique
6        value++ ;
7        //fin section critique
8      }
9    }
10   public void add1() {
11       //debut section critique
12       value++ ;
13       //fin section critique
14   }
15   public static void main(String[] args) throws InterruptedException {
16     Counter counter = new Counter() ;
17     Runnable runnable = new Runnable() {
18       @Override public void run() {
19         counter.add10000() ;
20       }
21     } ;
22     Thread t1 = new Thread(runnable) ;
23     Thread t2 = new Thread(runnable) ;
24     t1.start() ; t2.start() ;
```

Section(s) critique(s)

- Utilisation d'un **moniteur**.

```
1  public class Counter {
2    private int value ;
3    private final Object monitor = new Object() ;
4
5    public void add10000() {
6      for(int i = 0 ; i < 10000 ; i++) {
7        synchronized (monitor) {
8          value++ ;
9        }
10     }
11   }
12   public void add1() {
13     synchronized (monitor) {
14       value++ ;
15     }
16   }
17 }
```

- "Jeton" associé à l'objet pris par la thread au début du bloc (redonné à la fin).
- Un seul thread peut avoir le jeton.

Section(s) critique(s) et mémoire

- A l'entrée du bloc :
  - Force la **relecture** des variables depuis la RAM.
- A la sortie du bloc :
  - **Écriture** des variables modifiées vers la RAM.

Section(s) critique(s) - réentrance

- Un thread peut reprendre un moniteur qu'il a déjà.

```
1  public class Foo {
2    private int value ;
3    private final Object monitor = new Object() ;
4
5    public void setValue(int value) {
6      synchronized(monitor) {
7        this.value = value ;
8      }
9    }
10   public void reset() {
11     synchronized(monitor) {
12       setValue(0) ;
13     }
14   }
15 }
```

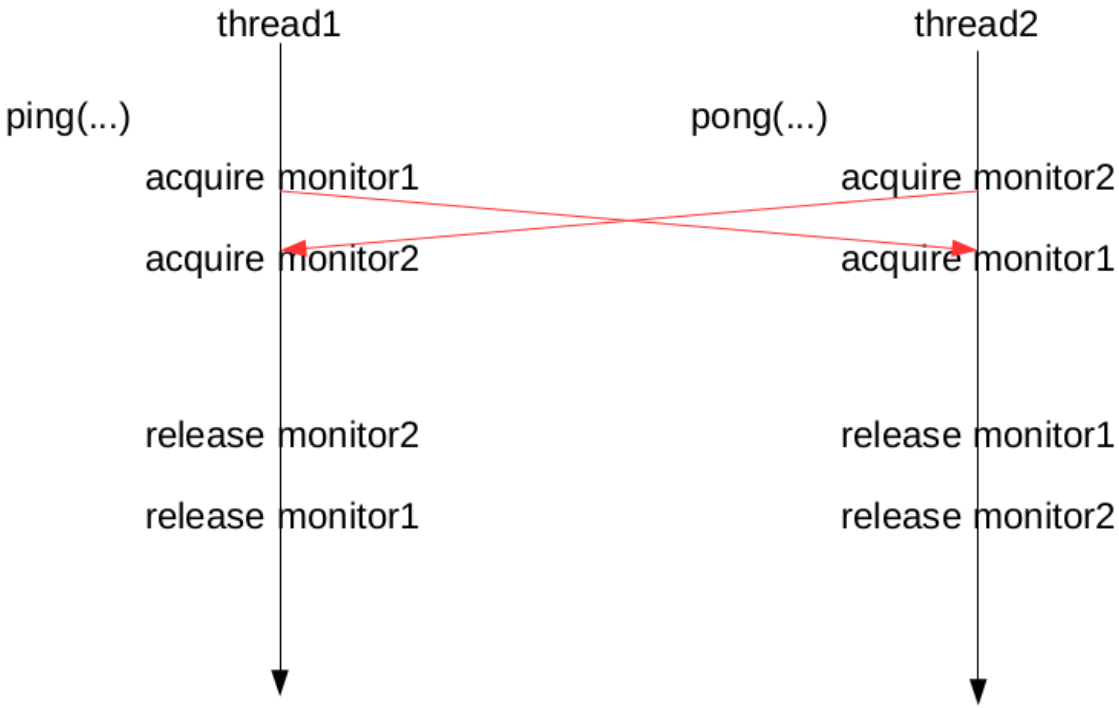
Section(s) critique(s) - Salade et fromage

```
1  public class PingPong {
2    private final Object monitor1 = new Object() ;
3    private final Object monitor2 = new Object() ;
4
5    public void ping() {
6      synchronized (monitor1) {
7        synchronized (monitor2) {
8          //
9        }
10     }
11   }
12   public void pong() {
13     synchronized (monitor2) {
14       synchronized (monitor1) {
15         //
16       }
17     }
18   }
19 }
```

- Si T1 lance ping et T2 lance pong ?

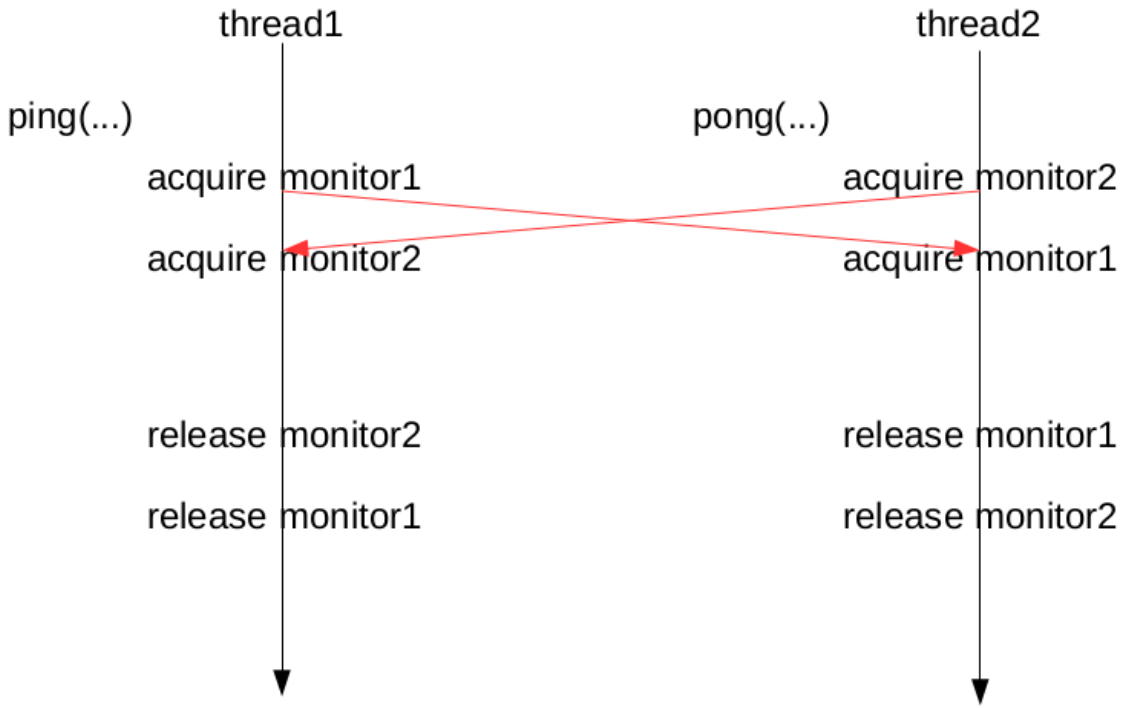
Section(s) critique(s) - Salade et fromage

- "Deadlock" potentiel.



Section(s) critique(s) - Salade et fromage

- "Deadlock" potentiel.



- Machin à les tomates et les concombres dans ses mains.
- Machine à le fromage et le gateau dans ses mains.
  - Machin veut le fromage et le gateau, Machine veut les tomates et les concombres, mais ne peuvent pas échanger...

Section(s) critique(s) - Deadlocks

- Risques si plusieurs threads prennent plusieurs moniteurs dans un ordre différent.
  - Utiliser un seul moniteur si possible.
  - Les prendre toujours dans le même ordre...

Section(s) critique(s) - Deadlocks

- Pour éviter les deadlock, un moniteur doit être encapsulé.
- Visible par le strict nécessaire.
  - `private` !
  - Pas de getter.

Section(s) critique(s) - moniteur

- Ne peut pas être un type primitif.
- Pas null.
- Éviter les objets où le JDK optimise pour ne pas réallouer la mémoire (Integer, Class.forName()...).
- On utilisera un champ **privé final**.
  - Peut utiliser un champ déjà utilisé dans la classe (type liste...).
  - Sinon on le déclare.
- `final` important : sinon risque de NPE (si deschedule avant initialisation...)

Threadsafe

- Une classe est **Threadsafe** si utilisable par plusieurs threads sans incohérences possibles.
- Si la classe est **non mutable**, elle est threadsafe !!!
- Par défaut une classe n'est pas threadsafe sauf si spécifié dans la doc.
- Exemple :
  - `String` : threadsafe car non mutable.
  - `HashMap` : mutable, pas threadsafe.
  - `Random` : mutable, mais threadsafe car spécifié dans la javadoc.
- Tout `java.util.concurrent` est threadsafe.

Méthode synchronized

- On peut déclarer une méthode synchronized (dans sa signature).
- Equivalent à faire `synchronized(this)` sur toute la méthode (ou `synchronized(Class)` pour une méthode statique).
- Mauvaise idée !!
  - `this` est visible de l'extérieur de la méthode : deadlock possible.
  - La classe aussi pour du statique.

Méthode synchronized

```
1 public class Bidule {
2   public synchronized void f() {
3
4   }
5 }
```

```
1 public class Chouette {
2   Bidule b = new Bidule() ;
3   public void g() {
4     synchronized (b) {
5
6     }
7   }
8 }
```

- Les deux blocs synchronized se font sur le même moniteur !

Taille

- Il faut tenter de raccourcir au strict nécessaire la taille des sections critiques (sinon intérêt des threads diminue !).

```
1  public class AttributeStore {
2    private final HashMap<String, String> attr = new HashMap<>() ;
3
4    public boolean userLocationMatches(String name, String regex) {
5      synchronized (attr) {
6        String key = "users." + name + ".location" ;
7        String location = attr.get(key) ;
8        if(location==null) return false ;
9        else return Pattern.matches(regex, location) ;
10     }
11   }
12 }
```

Pas bien !

Taille

- Il faut tenter de raccourcir au strict nécessaire la taille des sections critiques (sinon intérêt des threads diminue !).

```
1  public class AttributeStore {
2    private final HashMap<String, String> attr = new HashMap<>() ;
3
4    public boolean userLocationMatches(String name, String regex) {
5      String key = "users." + name + ".location" ;
6      String location ;
7      synchronized (attr) {
8        location = attr.get(key) ;
9      }
10     if(location==null) return false ;
11     else return Pattern.matches(regex, location) ;
12   }
13 }
```

Mieux !

Locks

- Autre mécanisme pour sections critiques (depuis Java 5).
- Interface : `Lock` (dans `java.util.concurrent.locks`).
- Une implémentation : `ReentrantLock`.
- Permet d'avoir des opérations supplémentaires.

ReentrantLock

- Idée similaire au bloc synchronized :
  - On prend un **verrou** avec `lock()`.
  - On lâche le verrou avec `unlock()`.
- **TOUJOURS** faire un unlock apres le lock (section critique entre les deux).

unlock

- Attention, si exception dans le bloc critique...

```
1  private final Lock lock = new ReentrantLock() ;
2
3  public void erk() throws InterruptedException {
4    lock.lock() ;
5    Thread.sleep(10) ;
6    lock.unlock() ;
7  }
```

- unlock jamais fait !!

unlock

- Toujours mettre la section critique dans un try et le unlock dans un **finally** !

```
1  public void erk() throws InterruptedException {
2    lock.lock() ;
3    try {
4      Thread.sleep(10) ;
5    }
6    finally {
7      lock.unlock() ;
8    }
9  }
```

unlock

- Toujours mettre la section critique dans un try et le unlock dans un **finally** !

```
1  public void erk() throws InterruptedException {
2    lock.lock() ;
3    try {
4      Thread.sleep(10) ;
5    }
6    finally {
7      lock.unlock() ;
8    }
9  }
```

- Lock n'est **pas** `Autoclosableable` : pas de try with ressources possible.

Lock et équitée

- Deux versions de Lock, une “fair” et une non (par défaut).
- Fair : si plusieurs threads en attente sur le verrou, favorise celle qui attend depuis plus longtemps (sinon hasard).
- Unfair plus rapide.
- Indépendant des priorités données par le scheduler (peut avoir plusieurs fois le verrou mais pas le processeur !).

- `ReentrantLock() ;`
- `ReentrantLock(boolean fair) ;`

Lock - méthodes supplémentaires

- `tryLock()` : renvoie `false` (de manière atomique) au lieu de bloquer si verrou non disponible.
  - Permet d'effectuer une opération alternative.
  - Une version de tryLock permet de tester que pendant un certain temps.
- `lockInterruptibility()` : lève une InterruptedException si on tente de l'interrompre pendant qu'elle attend d'avoir le lock.

Sémaphores

- Permet de limiter le nombre d'**accès simultanés** à une ressource partagée.
- Ensemble de "jetons" virtuels (nombre initial passé au constructeur).
- `acquire()` : prendre **un** jeton s'il en reste (attendre sinon, ou interrompu).
- `release()` : rendre **un** jeton.

Sémaphores binaires

- Donc sémaphore avec nombre max de jeton = 1 est un `Lock` ?

Sémaphores binaires

- Donc sémaphore avec nombre max de jeton = 1 est un `Lock` ?
- Pas tout à fait !
  - ReentrantLock : un même thread obtient plusieurs fois le verrou s'il demande le même.
  - Ici, il faut un jeton à chaque fois...

Sémaphores - exemple 1

- Pool de connexions à une BDD :
  - Demande de connexion “en attente” si pas de thread disponible, débloquée quand non vide.
  - Avec un sémaphore, prendre un jeton avant de tenter d'obtenir la ressource, et le rendre lorsque fait.
  - `acquire()` bloquant jusqu'à obtenir un jeton.

Sémaphore : Collection synchro de taille bornée

```
1  public class BoundedHashSet<T> {
2    private final Set<T> set ;
3    private final Semaphore sem ;
4    public BoundedHashSet(int maxBound) {
5      set = Collections.synchronizedSet(new HashSet<T>()) ;
6      sem = new Semaphore(maxBound) ;
7    }
8    public boolean add(T t) throws InterruptedException {
9      sem.acquire() ;
10     boolean wasAdded = false ;
11     try {
12       wasAdded = set.add(t) ;
13       return wasAdded ;
14     }
15     finally {
16       if( !wasAdded) //can't add (already exists..)
17         sem.release() ;
18     }
19   }
20   public boolean remove(Object o) {
21     boolean wasRemoved = set.remove(o) ;
22     if(wasRemoved)
23       sem.release() ;
24     return wasRemoved ;
25   }
26 }
```

Cours 9 : Programmation concurrente

Echange d'informations

- Si deux threads veulent échanger des données :
  - Données (partagées) dans des champs.
  - Des sections critiques pour éviter les états incohérents.
- Suffisant ?

Echange d'informations

- Si deux threads veulent échanger des données :
  - Données (partagées) dans des champs.
  - Des sections critiques pour éviter les états incohérents.
- Suffisant ?
- Il faut aussi un mécanisme pour qu'un thread puisse attendre des données.

Exemple

```
1  public class Bla {
2    private int v ;
3
4    public static void main(String[] args) {
5      Bla b = new Bla() ;
6      new Thread(new Runnable() {
7        @Override
8        public void run() {
9          b.v = 42 ;
10       }
11     }).start() ;
12     System.out.println(b.v) ;
13   }
14 }
```

- Affiche ?

Exemple

```
1  public class Bla {
2    private int v ;
3
4    public static void main(String[] args) {
5      Bla b = new Bla() ;
6      new Thread(new Runnable() {
7        @Override
8        public void run() {
9          b.v = 42 ;
10       }
11     }).start() ;
12     System.out.println(b.v) ;
13   }
14 }
```

- Affiche ?
- 0 ou 42.

Exemple - corrigé ?

```
1  public class Bla {
2    private int v ;
3    private boolean done ;
4    public static void main(String[] args) {
5      Bla b = new Bla() ;
6      new Thread(new Runnable() {
7        @Override public void run() {
8          b.v = 42 ;
9          b.done = true ;
10       }
11     }).start() ;
12
13     while( !b.done)  ;
14     System.out.println(b.v) ;
15   }
16 }
```

- Fonctionne ?

Exemple - corrigé ?

```
1  public class Bla {
2    private int v ;
3    private boolean done ;
4    public static void main(String[] args) {
5      Bla b = new Bla() ;
6      new Thread(new Runnable() {
7        @Override public void run() {
8          b.v = 42 ;
9          b.done = true ;
10       }
11     }).start() ;
12
13     while( !b.done)  ;
14     System.out.println(b.v) ;
15   }
16 }
```

- Fonctionne ?
- Problèmes :
  1. Attente active du thread main (utilise les ressources) pour le while.
  2. done peut ne pas avoir été mis à jour par le thread en RAM.

Exemple - corrigé ?

```
1  public class Bla {
2    private int v ;
3    private boolean done ;
4    private final Object monitor = new Object() ;
5
6    public static void main(String[] args) {
7      Bla b = new Bla() ;
8      new Thread(new Runnable() {
9        @Override
10       public void run() {
11         synchronized (b.monitor) {
12           b.v = 42 ;
13           b.done = true ;
14         }
15       }
16     }).start() ;
17
18     while(true) {
19       synchronized (b.monitor) {
20         if(b.done) break ;
21       }
22     } ;
23     synchronized (b.monitor) {
24       System.out.println(b.v) ;
25     }
26   }
```

- Sections critiques pour la cohérence.
- Reste l'attente active...

Exemple - corrigé !

```
1  public class Bla {
2    private int v ;
3    private boolean done ;
4    private final Object monitor = new Object() ;
5
6    public static void main(String[] args) throws
         InterruptedException {
7      Bla b = new Bla() ;
8      new Thread(new Runnable() {
9        @Override
10       public void run() {
11         synchronized (b.monitor) {
12           b.v = 42 ;
13           b.done = true ;
14           b.monitor.notify() ;//reveil si en attente
15         }
16       }
17     }).start() ;
18
19     synchronized (b.monitor) {
20       while( !b.done) {
21         b.monitor.wait() ;//endors
22       }
23     }
24     System.out.println(b.v) ;
25   }
```

- Endormir le thread tant que la condition n'est pas vérifiée.
- Reveiller le thread quand la condition est vraie.
- `notifyAll` réveille tous les threads en attente, coûte cher ! (re-bataille pour 1 vainqueur)

Nofify-wait

- `notify, notifyAll, wait` s'appellent uniquement sur l'objet monitor.
- `wait` libère le verrou.
- Lorsque réveillé, doit redemander le verrou.

Spurious wakeups

- Toujours faire un `wait` dans une boucle testant la condition
- Un thread peut être réveillé de manière erroné.
- Impose un nouveau test :
  - Soit c'était valide et on passe à la suite
  - C'était erroné on se rendort

```
1  synchronized (obj) {
2      while (<condition does not hold>) {
3          obj.wait() ;
4      }
5      ... // Perform action appropriate to condition
6  }
```

Cours 9 : Programmation concurrente

Collections

1. Collections pas thread safe en général.
2. Collections **synchronisées** (Java 1) : `Vector`, `Hashtable`.
   - Chaque méthode publique est synchronisée.
   - Beaucoup plus lent, préférer les collections classiques en temps normal.
3. `Collections.synchronizedXxx(col)` (Java 2) : proxy dont les appels sont synchronisés pour la collection prise en argument (decorator pattern).
   - `List<Type> syncList = Collections.synchronizedList(new ArrayList<Type>())`
4. Collections **concurrentes** (Java 5) : `ConcurrentHashMap`, `CoyOnWriteArrayList`.
   - Pas de synchronized, mais variables atomiques et locks.

Collections synchronisées

- Sont thread-safe...
- ... mais pas pour des opérations composées ! (**itérations**, addLast (size + add, peut etre interrompu entre et OutOfBounds ou pas dernier !...).
  - Synchronise des opérations unique, pas une séquence, ce qui est en général recherché !
  - Javadoc : *"It is imperative that the user manually synchronize on the returned list when iterating over it. Failure to follow this advice may result in non-deterministic behavior. "*
- Doit rajouter de la synchro côté client si au moins 2 opérations sur la collection !

```
1 Collection<Type> c = Collections.synchronizedCollection(myCollection) ;
2 synchronized(c) {
3     for (Type e  : c)
4         foo(e) ;
5 }
```

- Sans le bloc synchronized, risque de ConcurrentModificationException.

Iterateurs cachés...

```
1  public class HiddenIterator {
2    private final Set<Integer> set = new HashSet<Integer>() ;
3    public void add(Integer i) { synchronized (set) {set.add(i) ;}}
4    public void remove(Integer i) { synchronized (set) {set.remove(i) ;}  }
5    public void addTenThings() {
6      Random r = new Random() ;
7      for (int i = 0 ; i < 10 ; i++) add(r.nextInt()) ;
8      System.out.println("DEBUG : added ten elements to " + set) ;
9    }
10   public static void main(String[] args) {
11     final HiddenIterator hi = new HiddenIterator() ;
12     Runnable r = new Runnable() {
13       @Override public void run() {hi.addTenThings() ;}
14     } ;
15     new Thread(r).start() ;new Thread(r).start() ; new Thread(r).start() ;
16   }
17 }
```

- Pas thread-safe ! ConcurrentModificationEx peut etre levé : l'affichage cache un itérateur !