Indépendance et atomicité

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

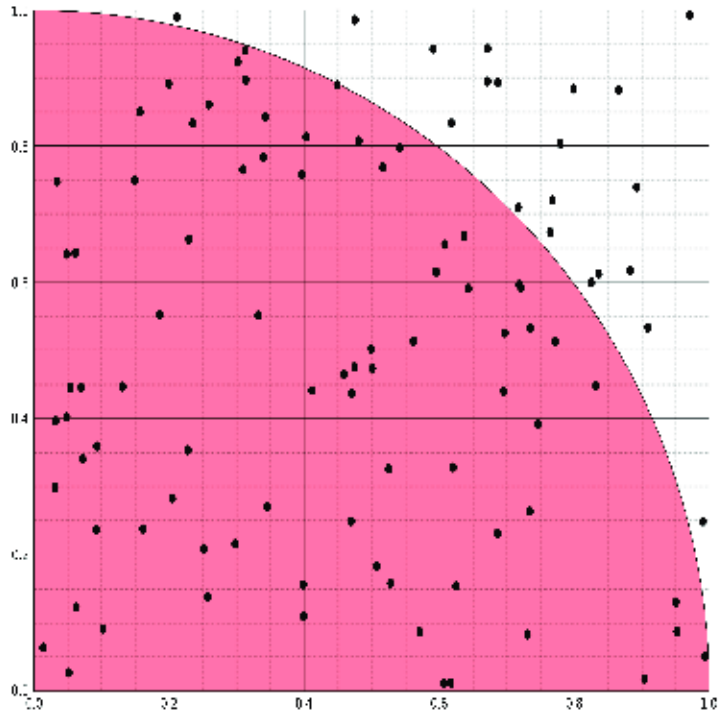
Année 2018-2019

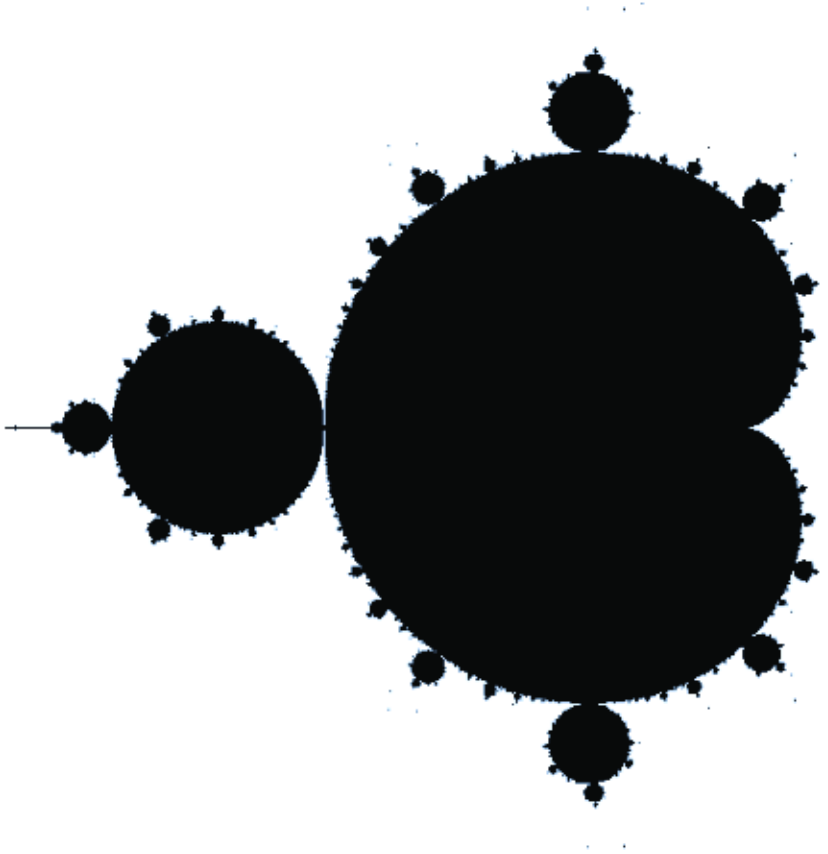
Version du 8 septembre 2020

Indépendance et parallélisation

Parallélisation

Parfois, comme vous l'avez vu en TD et en TP, écrire un programme parallèle consiste à paralléliser un programme séquentiel.



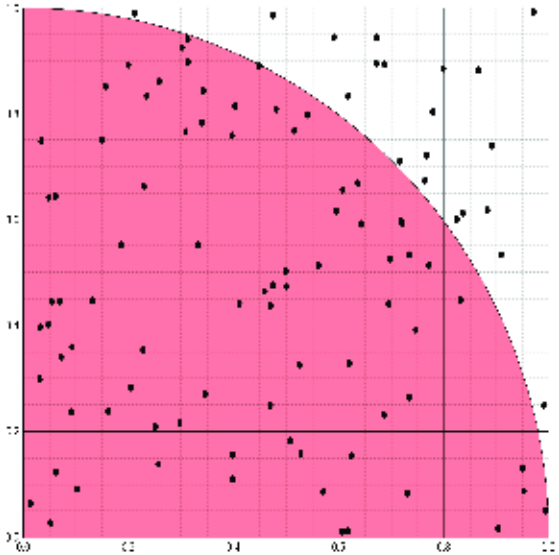


Pour qu'un programme puisse être parallélisé facilement, il doit contenir des parties *indépendantes* les unes des autres.

Indépendance entre parties de programme

La notion de « partie » doit être comprise au sens de partie d'exécution du programme et non simplement comme un morceau de code ; par exemple, on souhaite souvent partager les différentes *itérations* d'une boucle **`for`** sur plusieurs threads : il s'agit alors d'analyser l'indépendance des diverses occurrences du code itéré.

```
Random alea = new Random();
for (int i = 0; i < nbTirages; i++) {
  double x = alea.nextDouble() ;
  double y = alea.nextDouble() ;
  if (x*x+y*y <= 1) tiragesDansLeDisque++ ;
}
```



Les différentes itérations de cette boucle ne sont pas indépendantes, car la variable **`tiragesDansLeDisque`** est potentiellement incrémentée par chacune d'elles.

Il y a un risque d'*interférence* entre ces incrémentations : il faudra donc prendre des précautions à propos de cette variable lors de la parallélisation.

Indépendance entre parties de programmes

L'ensemble de lecture R_P (le « read set ») d'une partie P d'un programme est l'ensemble des variables lues par cette partie.

L'ensemble d'écriture W_P (le « write set ») d'une partie P d'un programme est l'ensemble des variables modifiées par cette partie.

Deux parties P_1 et P_2 d'un programme sont dites **indépendantes** si

- $W_{P_1} \cap W_{P_2} = \emptyset$

  $\rightsquigarrow$ P_1 et P_2 n'écrivent dans aucune variable commune

- $R_{P_1} \cap W_{P_2} = \emptyset$

  $\rightsquigarrow$ les variables lues par P_1 ne sont pas modifiées par P_2

- $R_{P_2} \cap W_{P_1} = \emptyset$

  $\rightsquigarrow$ les variables lues par P_2 ne sont pas modifiées par P_1

Ce sont les conditions de Bernstein (1966) :

A. J. Bernstein, *Program Analysis for Parallel Processing*, IEEE Trans. on Electronic Computers, EC-15, Oct. 66, 757-762.

Indépendance entre parties de programmes

L'ensemble de lecture R_P (le « read set ») d'une partie P d'un programme est l'ensemble des variables lues par cette partie.

L'ensemble d'écriture W_P (le « write set ») d'une partie P d'un programme est l'ensemble des variables modifiées par cette partie.

Deux parties P_1 et P_2 d'un programme sont dites **indépendantes** si

- $(R_{P_1} \cup W_{P_1}) \cap W_{P_2} = \emptyset$

  $\rightsquigarrow$ les variables lues ou modifiées par P_1 ne sont pas modifiées par P_2

- $R_{P_2} \cap W_{P_1} = \emptyset$

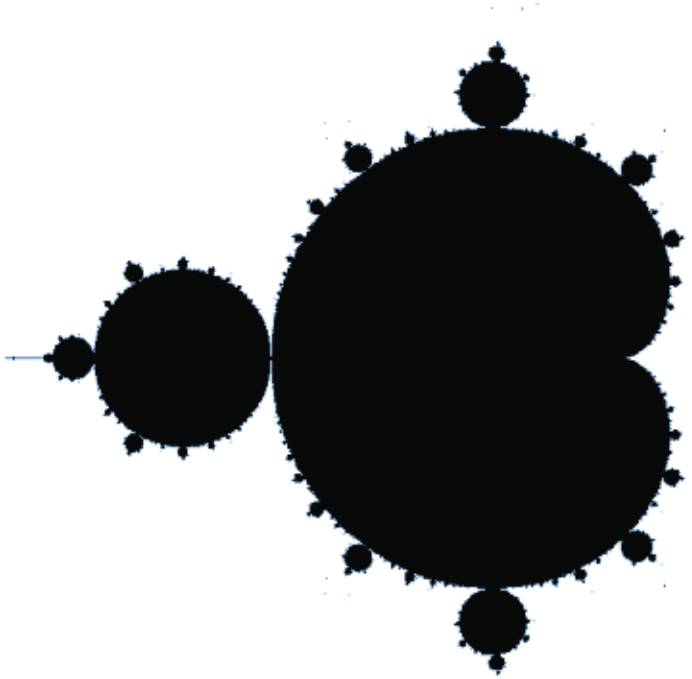
  $\rightsquigarrow$ les variables modifiées par P_1 ne sont pas lues par P_2

Ce sont les conditions de Bernstein (1966) :

A. J. Bernstein, *Program Analysis for Parallel Processing*, IEEE Trans. on Electronic Computers, EC-15, Oct. 66, 757-762.

Exemple du calcul de la fractale de Mandelbrot

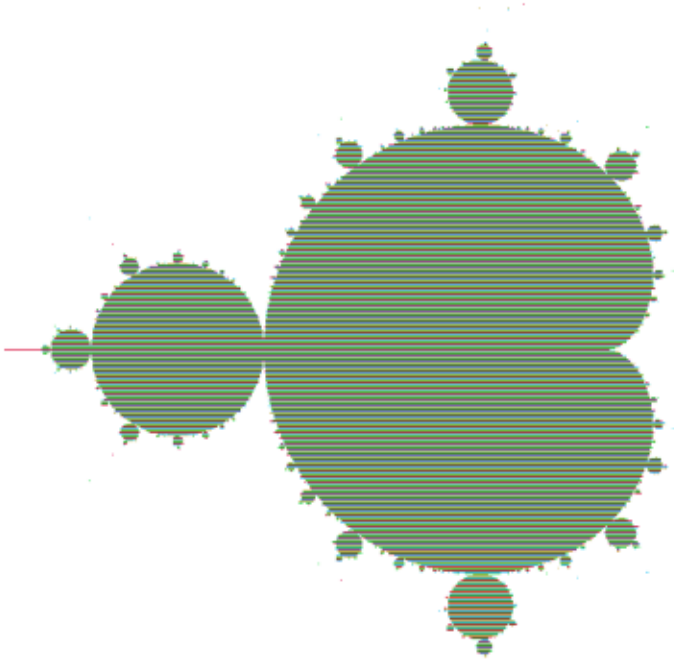
```
for (int i = 0; i < taille; i++) {
  for (int j = 0; j < taille; j++) {
    colorierPixel(i, j);
  }
}
```



Les différentes itérations de cette double-boucle écrivent sur des données (pixel de l'image) distinctes, et ces données ne sont pas lues : chaque itération est donc indépendante de toutes les autres.

Exemple du calcul de la fractale de Mandelbrot

```
for (int i = 0; i < taille; i++) {
  for (int j = 0; j < taille; j++) {
    colorierPixel(i, j);
  }
}
```



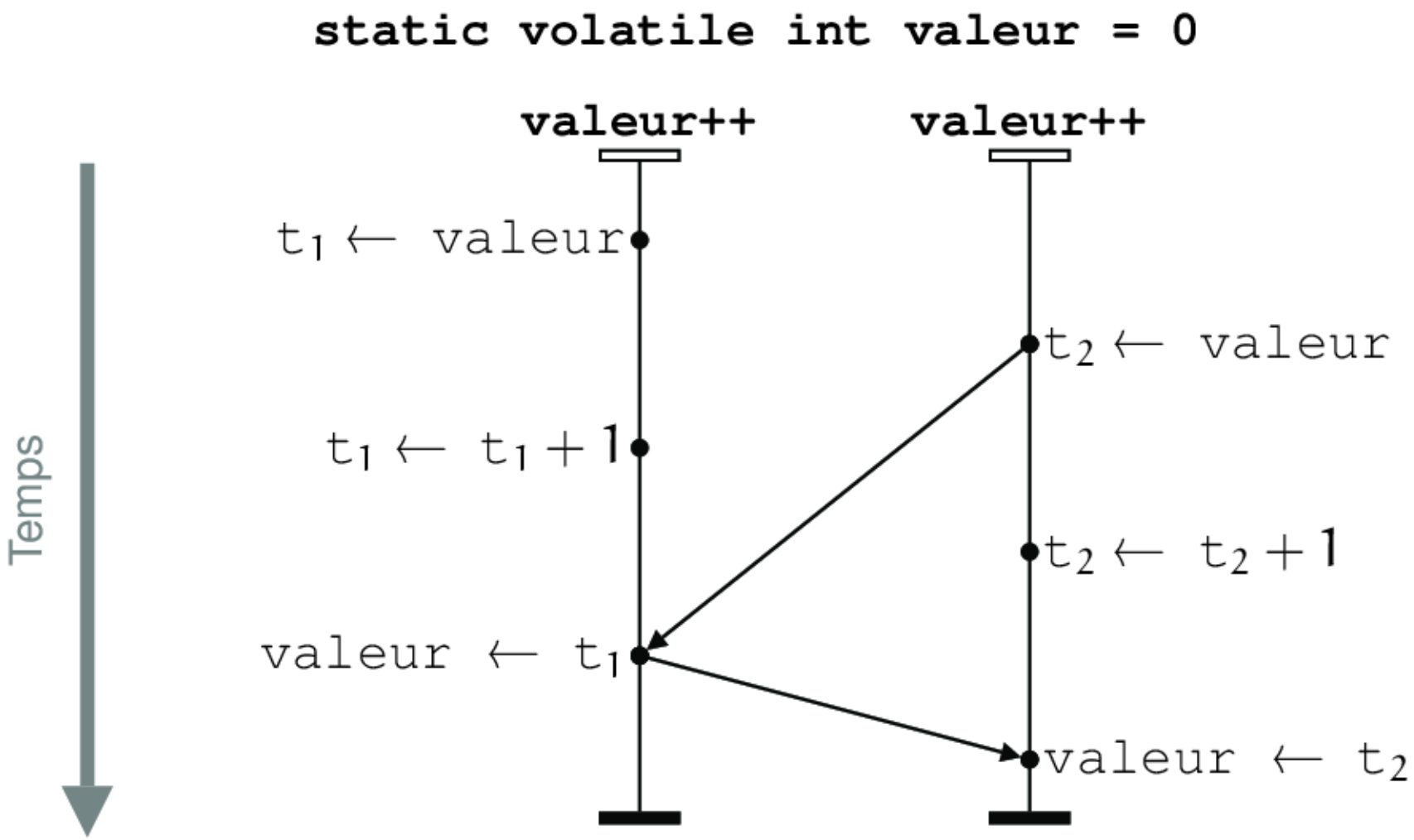
Les différentes itérations de cette double-boucle écrivent sur des données (pixel de l'image) distinctes, et ces données ne sont pas lues : chaque itération est donc indépendante de toutes les autres.

Sauf si l'on considère que **`i++`** (ou **`j++`**) fait partie de l'itération, en particulier, si l'attribution des lignes à calculer est dynamique.

✔ *Indépendance et parallélisation*

☛ *La notion cruciale d'atomicité*

Exemple bien connu (déjà vu)



Que vaut **`valeur`** *à la fin ?*

La notion d'instruction atomique, autrefois

Le terme **atomique** était employé autrefois pour désigner « *une opération ou un ensemble d'opérations d'un programme qui s'exécutent entièrement sans pouvoir être interrompues avant la fin de leur déroulement* » *(Wikipédia).*

C'est curieusement parfois encore le cas aujourd'hui.

La notion d'instruction atomique, autrefois

Le terme **atomique** était employé autrefois pour ≪ *désigner une opération ou un ensemble d'opérations d'un programme qui s'exécutent entièrement sans pouvoir être interrompues avant la fin de leur déroulement* ≫ *(Wikipédia).*

C'est curieusement parfois encore le cas aujourd'hui.

Considérons **une machine monoprocesseur** chargée de plusieurs processus (ou plusieurs threads) et qui exécute une suite d'instructions atomique. Alors

① les variables lues ou modifiées par cette suite d'instructions ne peuvent pas être modifiées *au cours de son exécution* par le reste du programme ;

② les modifications de la mémoire effectuées par cette instruction ne seront visibles par le reste du programme qu'*à la fin* de son exécution.

puisque l'ensemble du reste du programme est à l'arrêt.

La notion d'instruction atomique adoptée dans ce cours

Une **instruction atomique** est une suite d'opérations telle que

① Les variables **lues** ou **modifiées** par cette instruction ne peuvent pas être **modifiées** par le reste du programme *au cours de son exécution*.

C'est une forme d'exclusion mutuelle !

② Les valeurs de variables modifiées par cette instruction ne sont **visibles** par le reste du programme qu'*à la fin* de son exécution.

C'est une forme de synchronisation !

Contrairement à la notion statique et locale d'indépendance, l'atomicité se réfère à l'ensemble des autres instructions du programme qui peuvent s'exécuter de manière simultanée ; celles-ci doivent être grosso-modo *indépendantes* de chacune des opérations constituant l'instruction atomique.

Ce que ça veut dire en pratique

Une **instruction atomique** est une suite d'actions telle que

① « *Les variables lues ou modifiées par cette instruction ne peuvent pas être modifiées par le reste du programme au cours de son exécution.* »

Donc *l'effet de l'instruction atomique sur l'état global du programme* est le même que les autres processus poursuivent leur exécution ou non.

② « *Les modifications effectuées par cette instruction ne sont visibles par le reste du programme qu'à la fin de son exécution.* »

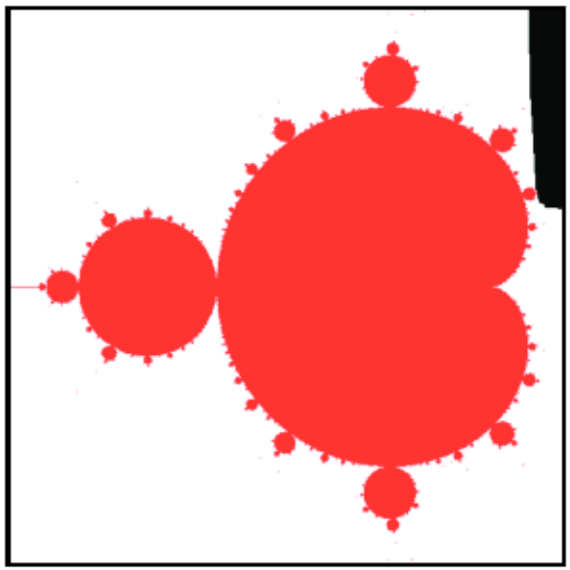
Donc *l'effet des processus concurrents à l'instruction atomique sur l'état global du programme* est le même que s'ils étaient suspendus du début à la fin de l'exécution de l'instruction atomique.

Ainsi, tout se passe « comme si » les autres processus sont suspendus ! c'est-à-dire que l'instruction semble s'exécuter de façon exclusive.

- ✔ *Indépendance et parallélisation*
- ✔ *La notion cruciale d'atomicité*
- ☛ *Instructions atomiques*

Aperçu d'une méthode qui n'est pas atomique

```
public void run() {
   NumberFormat formateur = new DecimalFormat("000");
   for(int i=0; i<1000&&!Thread.currentThread().interrupted(); i++)
   {
      image.save("PIC/pic" + formateur.format(i) + ".png");
   }
}
```



L'instruction **`save()`** n'est pas atomique : l'image est complétée entre le début et la fin de l'enregistrement du fichier.

L'instruction `print()` est atomique

```
Thread t1 = new Thread (new Runnable() {
  public void run() {
    System.out.print("A"+"B");
  }});
Thread t2 = new Thread (new Runnable() {
  public void run() {
    System.out.print("C");
  }});
t1.start();
t2.start();
```

Ce programme n'affichera jamais **"ACB"**, car de manière sous-jacente l'instruction **`print()`** utilise un verrou associé à la sortie standard pour conserver le privilège d'écriture lors de l'affichage.

Granularités

Il faut distinguer deux types d'atomicité :

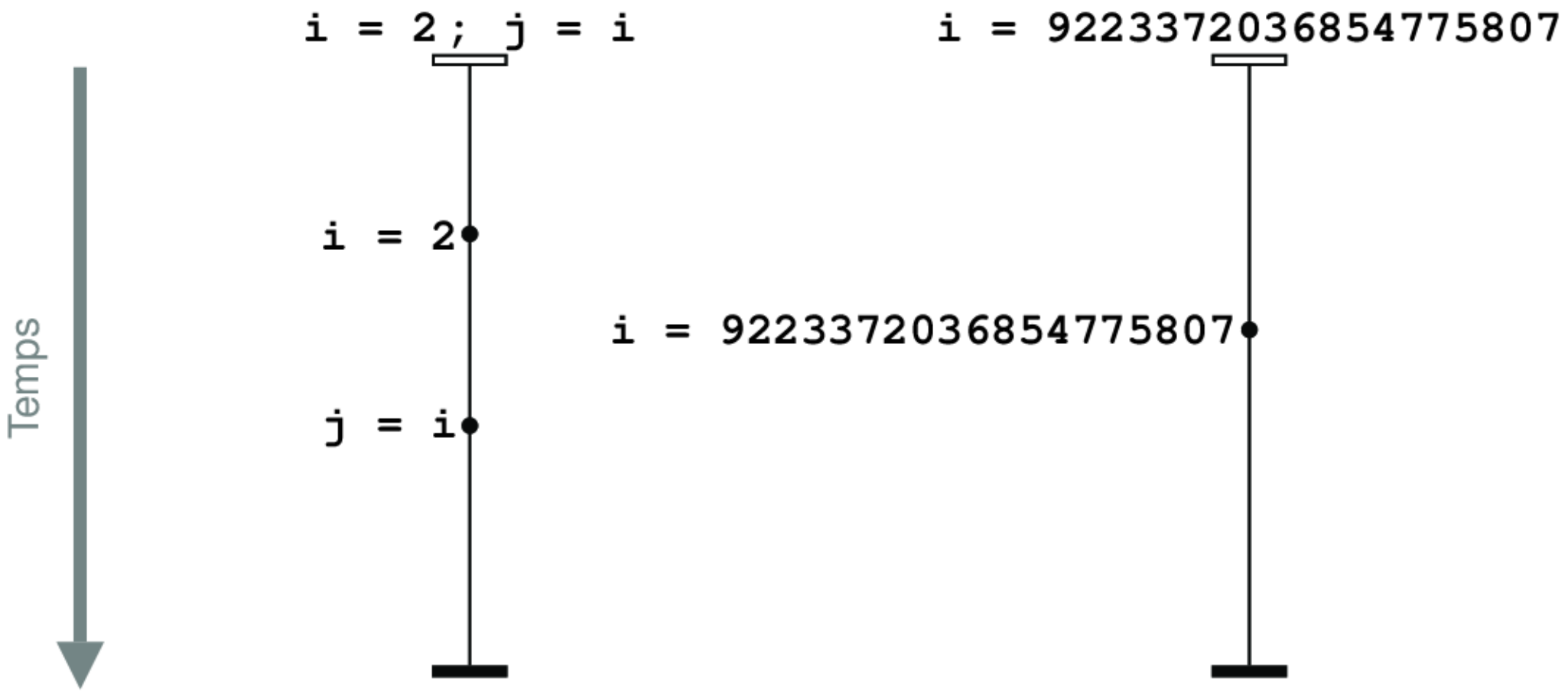
① Une instruction atomique **de granularité fine** est réalisée directement *au niveau de la machine* (ou du langage) : il n'y a rien à programmer.

② En revanche, si une séquence d'actions ou une méthode doit s'exécuter de manière atomique, il faudra garantir que les autres threads sont contraints d'attendre que la suite d'instructions soit terminée, s'ils doivent modifier ou lire les même données, le plus souvent **à l'aide d'un verrou**. On parle alors d'atomicité **de grande granularité**.

```
public void run() {
  for (int i = 1 ; i <= 10_000 ; i++) {
    synchronized ( this.getClass() ){ valeur++ ; }
  }
}
```

Exemple

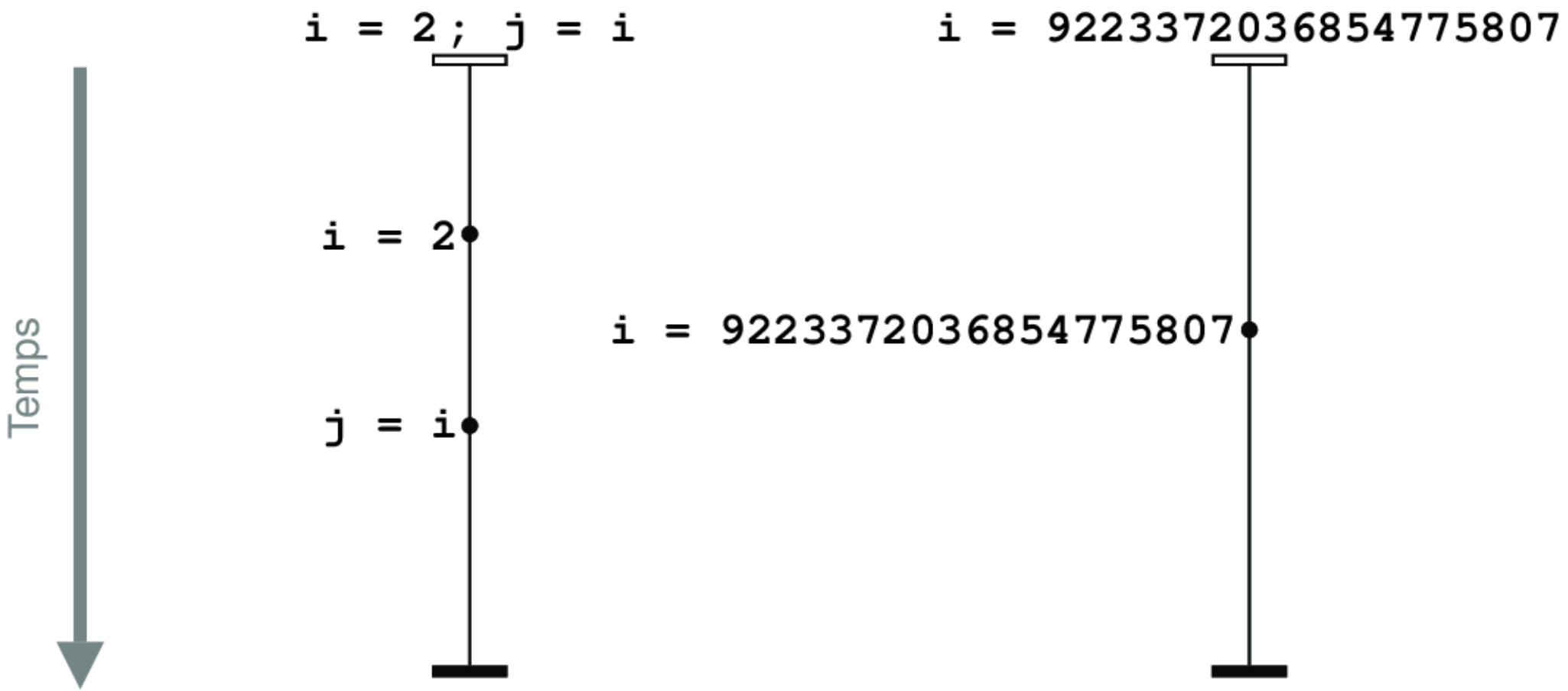
```
static long i, j;
```



Que vaut `j` à la fin ?

Exemple avec volatile

```
static volatile long i, j;
```



Que vaut `j` à la fin ?

Cas des champs sur 64 bits

Il y a trois types de champs sur 64 bits dans Java :

- les entiers de type **`long`** ;
- les flottants de type **`double`** ;
- et éventuellement, selon la machine, les **références vers des objets**.

Les écritures sur les champs de deux premiers types **ne sont pas atomiques** a priori : écrire un **`long`** peut être décomposé en deux temps. Il est donc possible, dans un contexte fortement concurrent, qu'un premier thread écrive sur les 32 bits de poids faibles alors qu'un autre thread écrive sur les 32 bits de poids forts.

En revanche, si ces champs sont déclarés **`volatile`**, alors chaque accès mémoire (lecture ou écriture) est atomique.

Toute référence 64 bits d'un objet est garantie d'avoir des accès mémoire atomiques.

La spécification du langage Java indique : ≪ Writes to and reads of references are always atomic, regardless of whether they are implemented as 32 or 64 bit values. ≫

Techniques fondamentales

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2018-2019

Version du 8 septembre 2020

Élimination d'un scenario d'exécution

Le rôle des opérations de **synchronisation** dans un programme est d'exclure certains scenarios d'exécution indésirables. On distingue généralement deux types d'opérations de synchronisation :

① **Exclusion mutuelle** : il s'agit former des séquences d'actions, appelées *sections critiques*, de sorte qu'à chaque instant **au plus un** processus exécute le code d'une section critique.
Les verrous sont les outils les plus simples pour assurer cela.

② **Synchronisation conditionnelle** : il s'agit de *retarder* une action d'un processus jusqu'à ce que l'état du programme satisfasse une certaine *condition*.
Les variables de condition sont là pour ça.

Ces deux problématiques sont liées !

Sémaphores (Edsger Dijkstra, 1963)

Qu'est-ce qu'un sémaphore ?

Un peu à l'image d'un verrou, un **sémaphore** est formé par un nombre entier positif et une liste d'attente ; il est manipulé uniquement par deux opérations **atomiques** : `P` et `V`.

P « Passeren » En français : Prendre, « Puis-je ? ».

Cette opération décrémente la variable à moins qu'elle ne soit déjà 0 ; dans ce cas, le processus est placé dans la file d'attente et suspendu.

V « Vrijgeven » En français : Relâcher, « Vas-y ! »

Cette opération incrémente la variable (de manière atomique) sauf si des processus sont en attente dans la file, auquel cas elle réactive l'un d'entre eux.

L'usage de sémaphores en Java n'est pas recommandé !

Les sémaphores peuvent être utilisés pour résoudre à peu près n'importe quel problème d'exclusion mutuelle ou de synchronisation... mais, ils possèdent certains inconvénients en programmation de haut niveau :

① Le rôle d'une opération P ou V (exclusion mutuelle ? synchronisation conditionnelle ?) dépend du type de sémaphore, de la façon dont il est initialisé et manipulé par les divers processus : **ce n'est pas explicite**.

② **Mécanisme de bas niveau** qui demande une **discipline sévère** dans la façon dont ils sont utilisés, sous peine d'erreurs : que se passe-t-il si on oublie d'indiquer un appel à V ? Ou si on effectue une action P en trop ?

③ **Mécanisme sans localité** : un sémaphore doit être connu et accessible par tous les processus qui pourraient devoir l'utiliser. Ce doit être une **variable globale**. Donc tout le programme doit être examiné pour voir où et comment le sémaphore est utilisé.

✔ *Sémaphores (Edsger Dijkstra, 1963)*

☛ *Le patron de conception par moniteur*

La notion de moniteur (P. Brinch Hansen, 1973 & C. A. R. Hoare 1974)

Caractéristique principale d'un programme avec moniteurs : il est composé de deux sortes d'objets :

- les processus qui agissent ; Par exemple, les 7 nains.
- les moniteurs qui subissent. Par exemple, Blanche-Neige.

Les interactions et synchronisations entre processus (actifs) se font alors par l'interface des moniteurs (passifs).

Un moniteur est un module qui regroupe et encapsule des données partagées ainsi que les procédures qui permettent de synchroniser les opérations sur ces données.

En Java, ce sera simplement un objet d'une classe particulière.

Exclusion mutuelle des accès au moniteur

Un moniteur permet de séparer de façon explicite l'*exclusion mutuelle* et la *synchronisation conditionnelle*.

En général un moniteur assure qu'une procédure (ou méthode) qu'il exporte sera toujours exécutée de façon ≪ **exclusive** ≫, c'est-à-dire qu'il y aura, à chaque instant, au plus une procédure du moniteur en cours d'exécution.

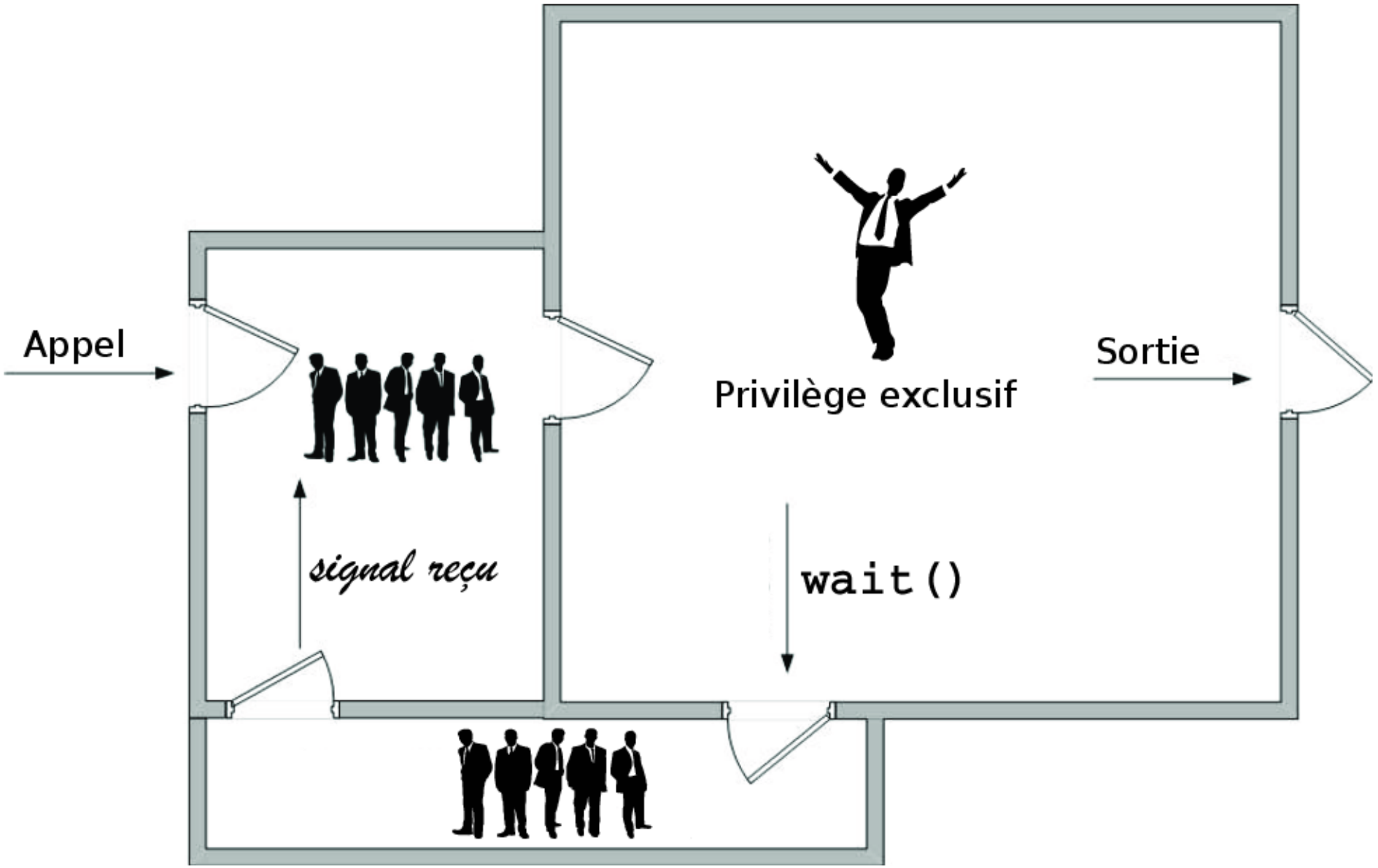
En Java, il suffira de déclarer **`synchronized`** *toutes les méthodes .*

D'autre part, les synchronisations conditionnellees sont programmées le plus souvent à l'aide de **variables de condition**.

En Java, nous utiliserons donc **`wait()`** *et* **`notifyAll()`** *.*

Pour certains auteurs, la notion de moniteur se limite à associer une variable de condition à un verrou, sans qu'il y est d'autres données en jeu. Ainsi, chaque objet en Java possède un moniteur. La document Java elle-même évoque le moniteur d'un objet.

Fonctionnement d'un moniteur



Blanche-Neige est un moniteur (1/2)

```
class BlancheNeige {
  private volatile boolean libre = true;
                          // Initialement, Blanche-Neige est libre

  public synchronized void requérir() {
    System.out.println(Thread.currentThread().getName()
                       + "␣veut␣la␣ressource");
  }
```

Les méthodes d'un moniteur sont toutes synchronisées !

Blanche-Neige est un moniteur (2/2)

```
  public synchronized void accéder() throws InterruptedException {
    while( ! libre ) {
      wait();                          // Le nain patiente sur l'objet bn
    }
    libre = false;
    System.out.println(Thread.currentThread().getName()
                       + "␣accède␣à␣la␣ressource.");
  }
  public synchronized void relâcher() {
    System.out.println(Thread.currentThread().getName()
                       + "␣relâche␣la␣ressource.");
    libre = true;
    notifyAll();
  }
}
```

Ces méthodes sont elles atomiques ?

Ces méthodes sont elles atomiques ?

`requérir()` ne lit aucune variable, et n'en modifie aucune : elle est donc atomique.

`relâcher()` modifie une seule variable, **`libre`**, qui est un attribut **`privé`** : elle ne peut être modifiée que par l'application d'une méthode de l'objet.

Toutes les méthodes de l'objet étant « synchronized », lorsque la méthode **`relâcher()`** est en cours d'exécution, aucun autre thread ne peut lire ou modifier la variable **`libre`**.

La méthode **`relâcher()`** est donc également atomique.

`accéder()` lit et modifie une seule variable : **`libre`**. Cependant, si un nain exécute l'instruction **`wait()`**, il relâche le verrou. Un autre nain devra donc modifier **`libre`**, en appliquant **`relâcher()`**, pour lui permettre de poursuivre son code : la méthode **`accéder()`** n'est donc pas atomique.

Pourtant, **`accéder()`** agit de manière atomique !

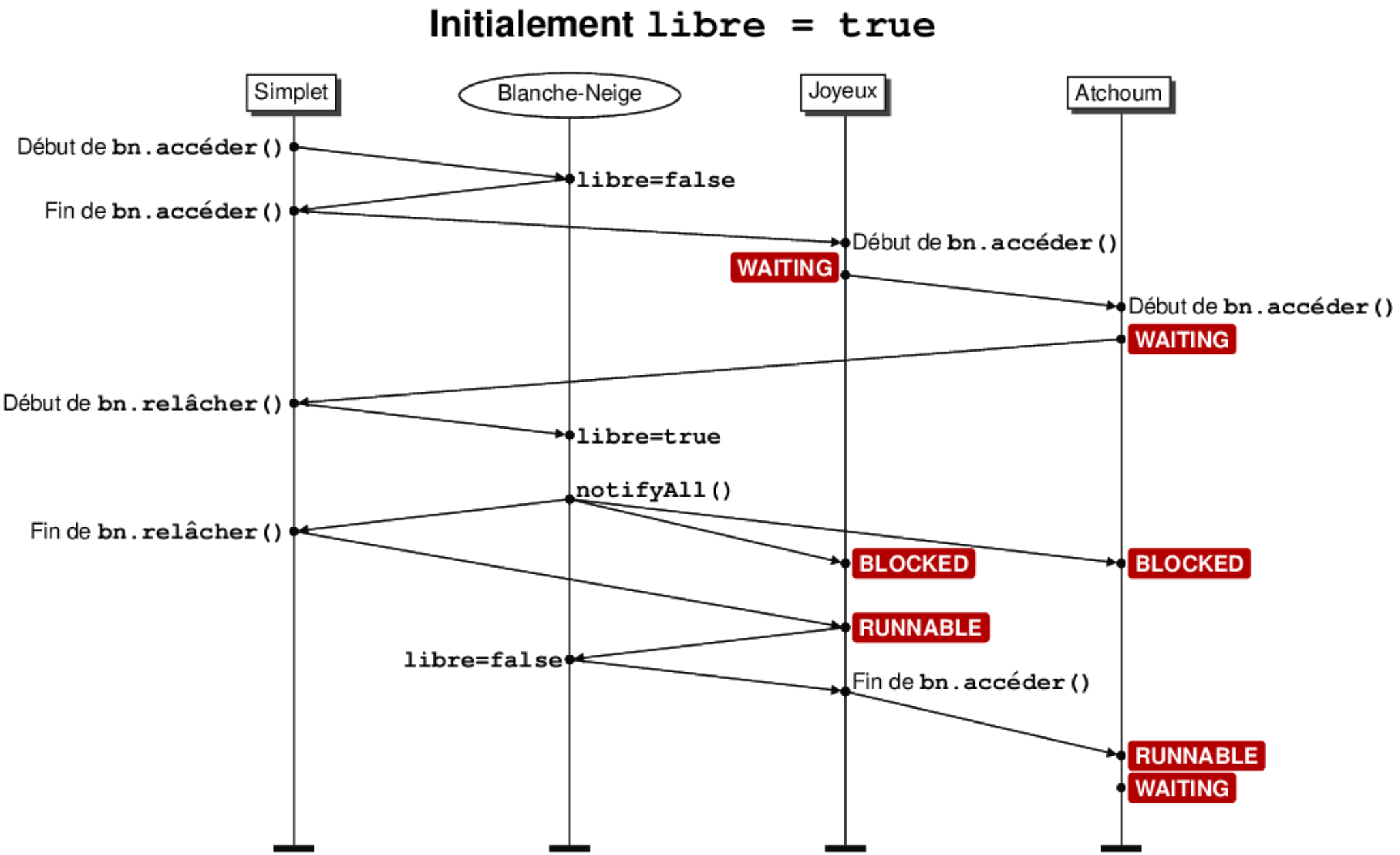
La méthode `accéder()` agit en fait de manière atomique

Si on met de côté la phase éventuelle d'attente de conditions favorables, matérialisée par la boucle **`while()`** au début de la méthode **`accéder()`**, alors le code résiduel est lui atomique, pour les mêmes raisons que **`relâcher()`**, puisque le nain qui exécute ce code possèdera alors le verrou intrinsèque de l'objet Blanche-Neige.

Puisque que la phase d'attente préalable ne modifie pas aucune donnée, chaque appel à **`accéder()`** fonctionne « comme si » cette méthode était atomique une fois la phase d'attente éventuelle passée.

La méthode **`accéder()`** agit donc en fait de manière atomique.

Fonctionnement du moniteur Blanche-Neige



Une exécution

```
$ javac SeptNains.java
$ java SeptNains
Simplet veut la ressource
        Simplet accède à la ressource.
Atchoum veut la ressource
Timide veut la ressource
Joyeux veut la ressource
Grincheux veut la ressource
Dormeur veut la ressource
Prof veut la ressource
                Simplet relâche la ressource.
Simplet veut la ressource
        Simplet accède à la ressource.
                Simplet relâche la ressource.
Simplet veut la ressource
        Simplet accède à la ressource.
                Simplet relâche la ressource.
```

- ✔ *Sémaphores (Edsger Dijkstra, 1963)*
- ✔ *Le patron de conception par moniteur*
- ☛ *Protection contre les signaux intempestifs*

Les signaux intempestifs

```
class WakeUp extends Thread {
  private final Object unObjet = new Object();

  public static void main(String [] args) {
    new WakeUp().start();        // Un seul thread est lancé
  }

  public void run() {
    synchronized (unObjet) {
      try { unObjet.wait(); }  // Le thread attend sur unObjet
      catch(InterruptedException e){ e.printStackTrace(); } ;
    }
  }
}
```

Ce programme peut-il terminer ?

Extrait de la documentation de `java.lang.Object`

```
public final void wait(long timeout)
                    throws InterruptedException
```

...

A thread can also wake up without being notified, interrupted, or timing out, a so-called **spurious wakeup**. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. In other words, waits should always occur in loops, like this one :

```
synchronized (obj) {
  while (<condition does not hold>) obj.wait(timeout);
  ... // Perform action appropriate to condition
}
```

- ✔ *Sémaphores (Edsger Dijkstra, 1963)*
- ✔ *Le patron de conception par moniteur*
- ✔ *Protection contre les signaux intempestifs*
- ☛ *B-A BA pour éviter les interblocages*

Interbloquage avec deux verrous

```
public class Deadlock {
  Object m1 = new Object();
  Object m2 = new Object();

 public void ping() {
   synchronized (m1) {
     synchronized (m2) {
   // Code synchronisé sur
   // les deux verrous
     }
   }
 }
```

```
public void pong() {
  synchronized (m2) {
    synchronized (m1) {
  // Code synchronisé sur
  // les deux verrous
    }
  }
}
```

Il se peut que les deux threads attendent chacun que l'autre libère le verrou.

Correction vs. performances

Solution générale (mais un peu délicate) : ≪ **ordonner pour régner** ≫

Il faut choisir, dès la conception du programme, un *ordre total* sur l'ensemble des verrous qui seront utilisés. Puis, si un thread doit acquérir plusieurs verrous en même temps, **toujours prendre les verrous selon l'ordre établi.**

Aucun interblocage ne pourra alors avoir lieu.

- ⇝ S'il y a un seul verrou, il n'y aura jamais d'interblocage. Néanmoins, *par souci de performance*, il faut souvent au contraire ≪ **diviser pour régner** ≫ :
  - — Il vaut mieux utiliser plusieurs verrous distincts, s'il n'y a pas de dépendance.
  - — Il vaut mieux aussi réduire la taille des ≪ sections critiques ≫ au minimum.
- ⇝ S'il y a un seul verrou, il pourra y avoir d'*autres types de blocage*, notamment lors d'un appel à `wait()` qui n'est pas suivi d'un `notify()`.

Parfois, si un verrou est déclaré et utilisé localement dans une méthode qui ne fait appel à aucun code tiers et qui ne requiert aucun autre verrou, alors ce verrou ne pourra jamais conduire à un cycle d'interblocage.

Ce qu'il faut retenir

Les problèmes d'*atomicité* sont inhérents à la programmation parallèle (notamment la programmation distribuée) et une source d'erreurs fréquente dans les applications.

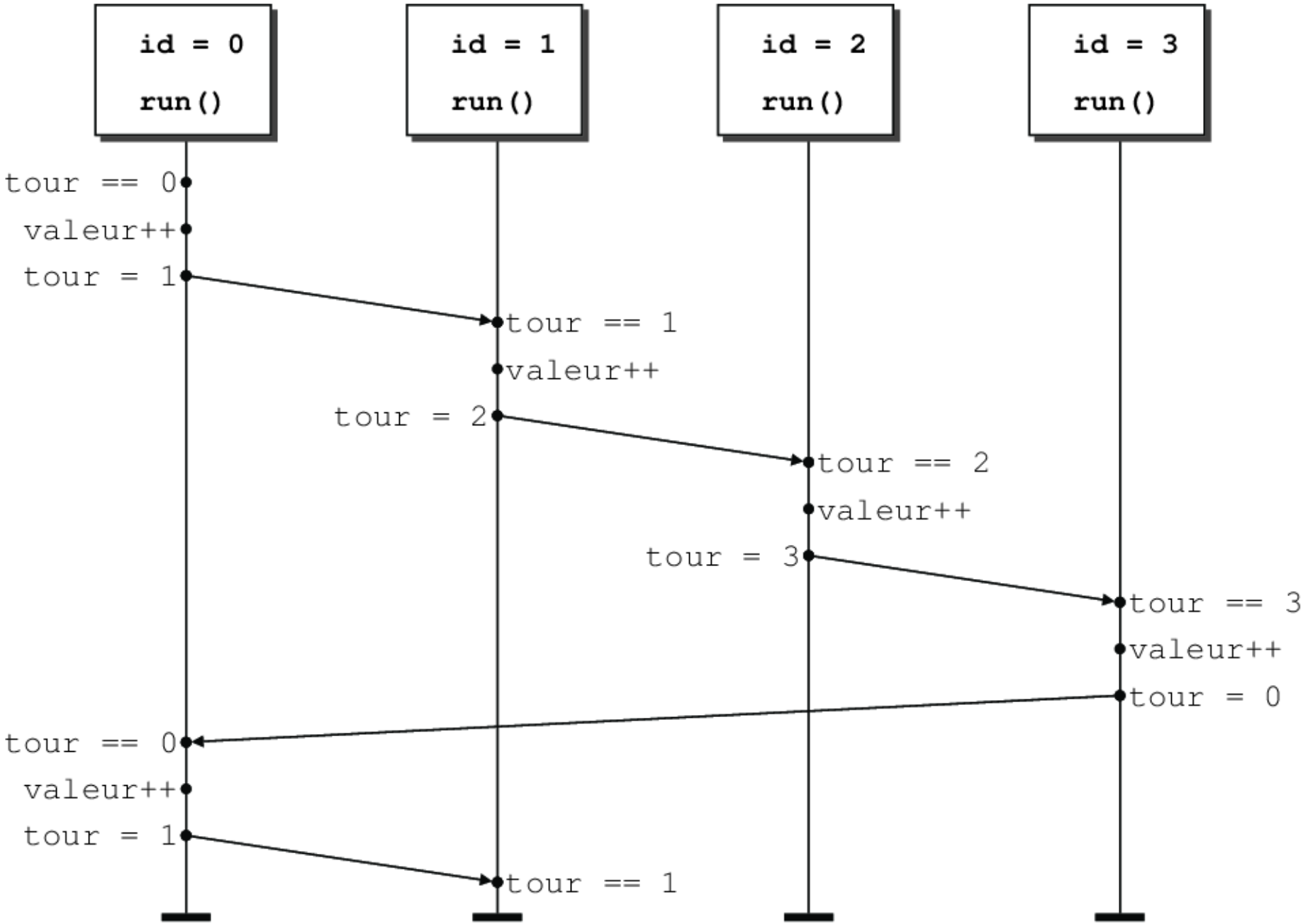
Les *verrous* sont bien pratiques pour assurer l'atomicité ; cependant, mal utilisés, ils provoquent facilement des *interblocages*. Les *sémaphores* sont aussi une technique très puissante, mais risquée, peu claire et donc non recommandée dans ce cours. En revanche, le patron de conception par *moniteur* permet d'aborder méthodiquement les problèmes de programmation concurrente. Il sera illustré également par les *collections synchronisées*.

Un thread peut quitter l'instruction **`wait()`** sans qu'il y ait eu d'instruction **`notify()`** exécutée. Ce phénomène étrange et rare se nomme « *spurious wake-up* » que je traduis en « signal intempestif » car cela n'a rien à voir avec **`sleep()`**. Il faut par conséquent protéger chaque **`wait()`** par une boucle **`while()`** comme l'indique la documentation officielle.

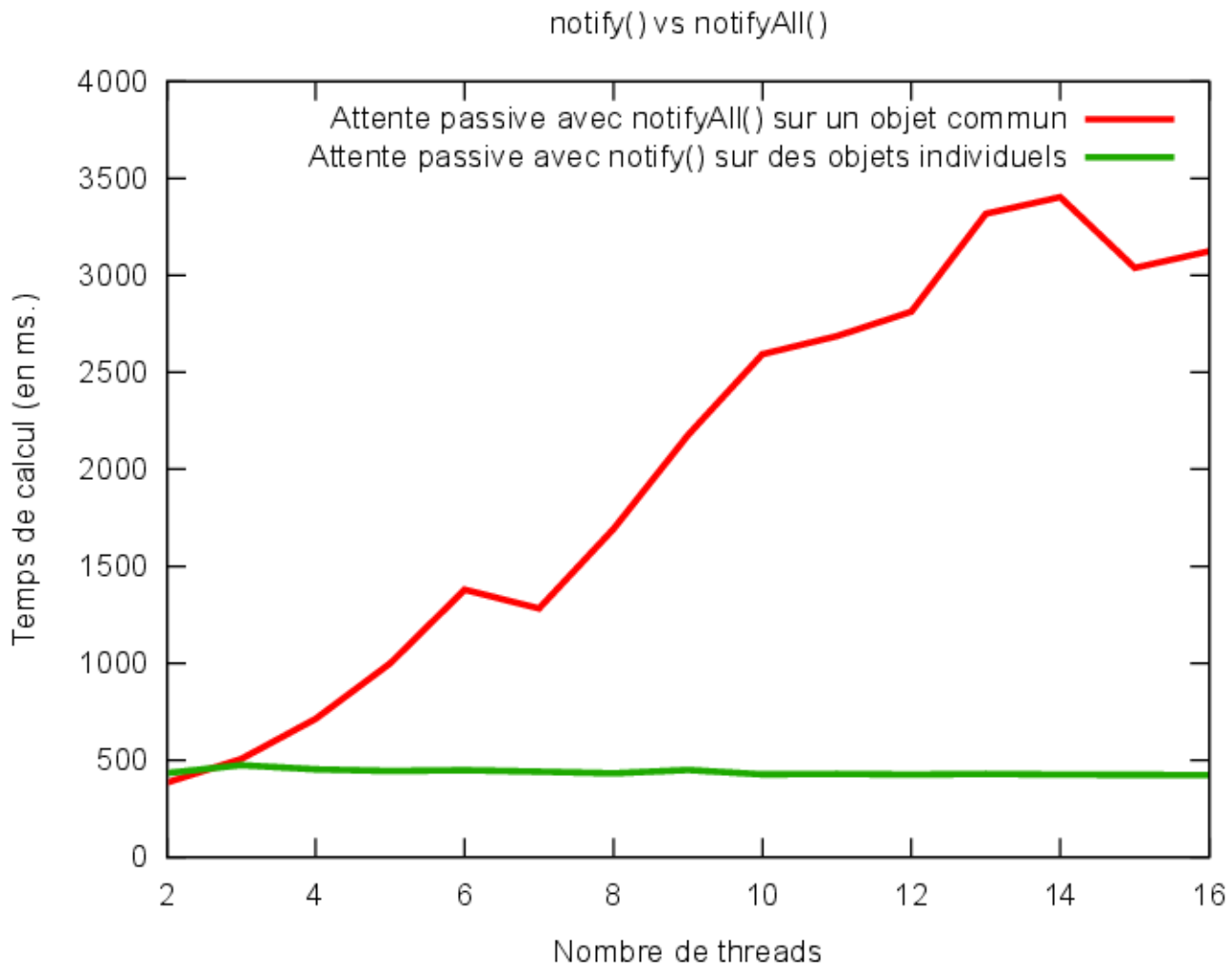
Il en résulte qu'un programme correct restera forcément correct en remplaçant chaque appel à **`notify()`** par un **`notifyAll()`**.

Pourquoi continuer à hésiter ?

Le benchmark adopté : les compteurs en rond



notify vs notifyAll()



Faire patienter chaque compteur sur un objet particulier permet d'améliorer les performances en utilisant **`notify()`**. Le coût d'un **`notifyAll()`** sur un objet partagé, plus simple à programmer, est proportionnel au nombre de threads utilisés.

Les variables atomiques

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2018-2019

Version du 8 septembre 2020

Deux compteurs anarchiques non synchronisés (rappel)

```
public class Compteur extends Thread {
  static volatile int valeur = 0;
  public static void main(String[] args) {
    Compteur Premier = new Compteur();
    Compteur Second = new Compteur();
    Premier.start();
    Second.start();
    Premier.join();
    Second.join();
    System.out.println("La␣valeur␣finale␣est␣" + valeur);
  }
  public void run(){
    for (int i = 1 ; i <= 10_000; i++) {
      valeur++;
    }
  }
}
```

Vingt milliers d'incrémentations : résultats sur mon ancien MacBook

```
$ java Compteur
La valeur finale est 4593
$ java Compteur
La valeur finale est 10000
$ java Compteur
La valeur finale est 19522
$ java Compteur
La valeur finale est 10000
$ java Compteur
La valeur finale est 10000
$ java Compteur
La valeur finale est 10000
$ java Compteur
La valeur finale est 10000
$ java Compteur
La valeur finale est 10000
$ java Compteur
La valeur finale est 19591
```

Compteurs anarchiques synchronisés sur un verrou commun

```
public void run() {
  for (int i = 1;i <= 10_000; i++) {
    synchronized ( this.getClass() ){ valeur++; }
  }
}
```

```
$ java Compteur
La valeur finale est 20000
$ java Compteur
La valeur finale est 20000
$ java Compteur
La valeur finale est 20000
...
```

Le verrou assure ici « *l'atomicité* » de l'incrémentation de la variable **`valeur`** : un seul thread peut manipuler cette variable à chaque instant : celui qui possède le verrou.

Un peu mieux...

```
private final static Object monVerrou = new Object();
...
public void run() {
  for (int i = 1; i <= 10_000; i++) {
    synchronized (monVerrou){ valeur++; }
  }
}
```

```
$ java Compteur
La valeur finale est 20000
$ java Compteur
La valeur finale est 20000
...
```

Utiliser un verrou *privé* (et donc inutilisable par le reste du programme) limite fortement les risques d'interblocage.

Encore mieux : un entier atomique

```
static AtomicInteger valeur = new AtomicInteger(0);
...
public void run() {
  for (int i = 1; i <= 10_000; i++) {
    valeur.incrementAndGet();
  }
}
```

```
$ java Compteur
La valeur finale est 20000
$ java Compteur
La valeur finale est 20000
...
```

Chaque appel à **`incrementAndGet()`** garantit une incrémentation de la valeur contenue dans l'objet **`valeur`** de manière atomique, c'est-à-dire « comme si » aucun n'autre thread ne lit ou ne modifie cette valeur dans le même temps. Il n'y a plus de verrou, donc plus aucun risque d'interblocage !

- ✔ *Retour aux compteurs anarchiques*
- ☛ *Les objets atomiques (depuis Java 5)*

java.util.concurrent.atomics

Le paquetage **`java.util.concurrent.atomics`** permet d'appliquer des opérations atomiques sur des objets qui correspondent à certains types de *variables* ou à des *références* : **`AtomicBoolean`**, **`AtomicInteger`**, **`AtomicLong`**, **`AtomicReference`**, etc.

Chacune de ces classes

① propose des opérations **atomiques** : **`get()`**, **`set()`**, **`getAndSet()`**, etc. **sans requerir à un verrou** ;

② garantit la **visibilité** immédiate et absolue des opérations réalisées sur l'objet, comme si sa valeur était déclarée **`volatile`**.

③ propose une méthode **`compareAndSet()`** un peu bizarre mais très pratique !

Modification de valeur

Si deux threads non coordonnés exécutent **`ai.getAndAdd(5)`** sur un objet déclaré par **`AtomicInteger ai=new AtomicInteger(10)`**, alors au final un appel à **`ai.get()`** renverra à coup sûr la valeur **20**.

Il n'y a pas de verrou : c'est plus **simple** (puisqu'il n'y pas de précaution à prendre) et plus **sûr** puisqu'il ne pourra y avoir d'interblocage sur ce code.

Les objets atomiques permettent donc de se dispenser parfois de **`synchronized`**.

C'est aussi plus léger à mettre en oeuvre et plus performant que d'utiliser un verrou.

Opération atomique d'affectation conditionnelle

Les objets atomiques en Java bénéficient d'une méthode très particulière qui réalise une affectation atomique *conditionnelle* :

```
boolean compareAndSet(valeurAttendue, valeurNouvelle)
```

- ⇝ Affecte **atomiquement** la valeur **`valeurNouvelle`** dans l'objet **si** la valeur courante de cet objet est effectivement égale à la valeur **`valeurAttendue`**.

Il faut deviner la valeur courante pour la modifier !

- ⇝ Retourne **`true`** si l'affectation a eu lieu, **`false`** si la valeur courante de l'objet est différente de **`valeurAttendue`** au moment de l'appel.

Un appel à **`compareAndSet()`** sera remplacé par la machine virtuelle Java par l'opération assembleur correspondante dans la machine : par exemple, les instructions de comparaison et échange CMPXCHG8B ou CMPXCHG16B des processeurs Intel.

Implémentation de la méthode `incrementAndGet()`

Il est facile d'implémenter **`incrementAndGet()`** à l'aide de **`compareAndSet()`**.

```
public final int incrementAndGet() {
  for (;;) {
    int copie = this.get() ;
    int suivant = copie + 1 ;
    if ( this.compareAndSet(copie, suivant)) return suivant ;
  }
}
```

La méthode **`incrementAndGet()`** ne semble pas a priori atomique : la valeur de l'objet **`this`** peut être modifiée par plusieurs threads exécutant cette méthode simultanément. Cependant **`incrementAndGet()`** garantit que, tôt ou tard, une incrémentation atomique de l'entier aura lieu, car **`compareAndSet()`** est elle-même atomique.

Cette implémentation ne fait entretemps rien d'autre que *patienter de manière active* en ne modifiant que des variables *locales*. Elle agit donc, en fait, de manière atomique.

- ✔ *Retour aux compteurs anarchiques*
- ✔ *Les objets atomiques (depuis Java 5)*
- ☛ *Construction artisanale d'un verrou*

Mon verrou artisanal (non réentrant !)

```
public class MonVerrou implements Lock {
  AtomicBoolean libre = new AtomicBoolean(true);
                                  // Initialement, le verrou est libre.
  public void lock() {
    while (! libre.compareAndSet(true,false)); // Attente active
      // Le thread tourne en boucle tant que libre vaut faux.
  }
  public void unlock() {
    libre.set(true);              // Le verrou est de nouveau libre.
  }
}
```

`libre.compareAndSet(true,false)` n'effectue une écriture dans la variable **`libre`** que si la valeur courante de cette variable vaut **`true`**, c'est-à-dire que le verrou est libre. Elle écrit alors **`false`** dans la variable **`libre`**, puis renvoie **`true`** pour indiquer que l'écriture a réussi, ce qui provoque la sortie de la boucle et de la méthode **`lock()`**.