

# Indépendance et atomicité

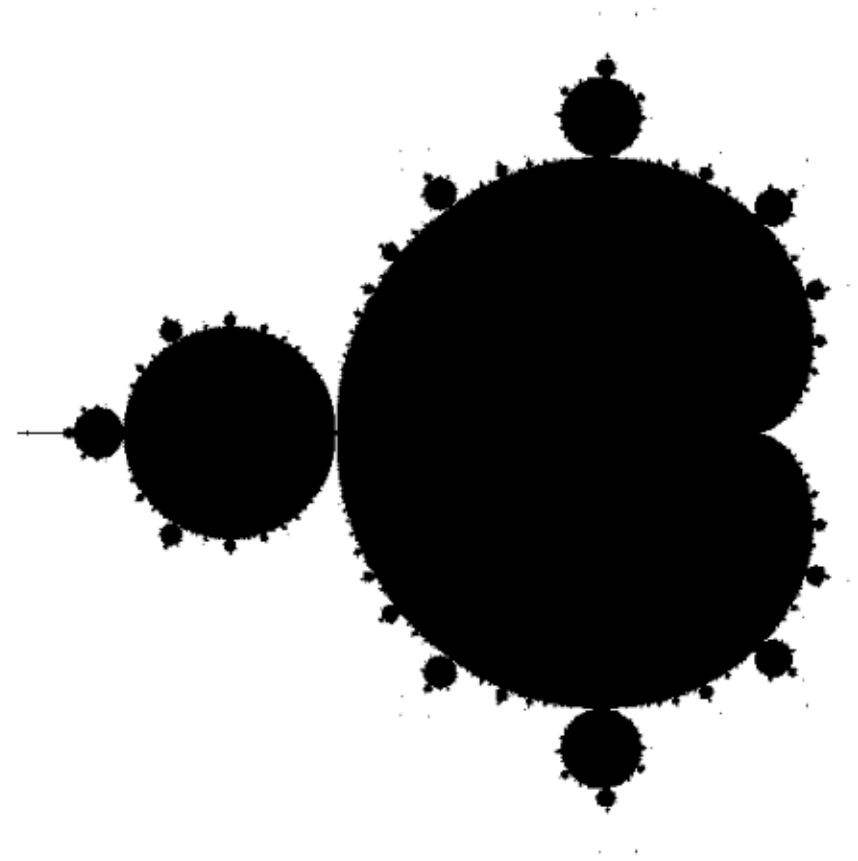
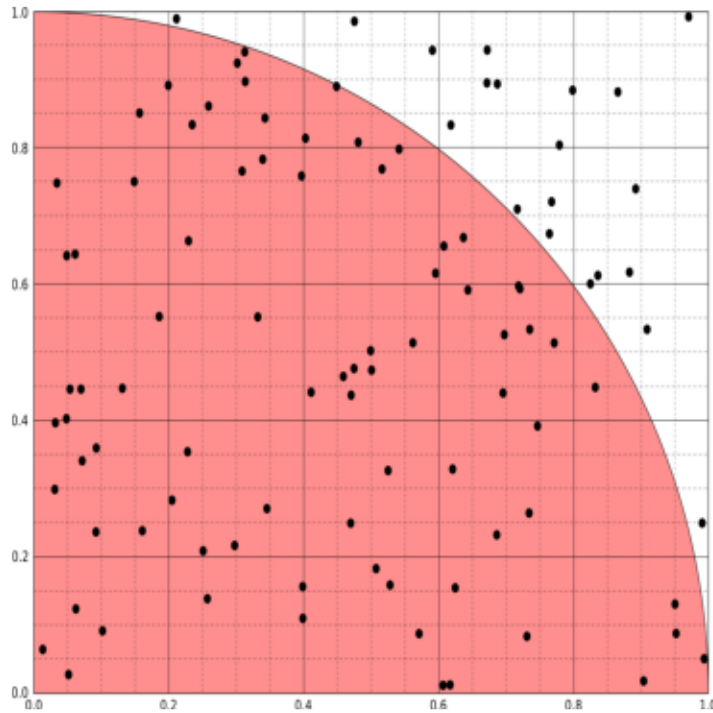
Master Informatique — Semestre 1 — UE obligatoire de 3 crédits



*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.

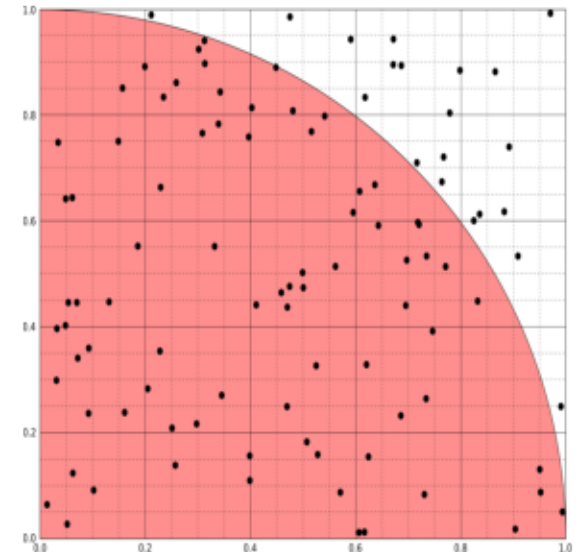


Pour qu'un programme puisse être parallélisé facilement, il doit contenir des parties *indépendantes* les unes des autres.

# Indépendance entre parties d'exécution 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 aléa = new Random();  
for (int i = 0; i < nbTirages; i++) {  
    double x = aléa.nextDouble() ;  
    double y = aléa.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 incréments : il faudra donc prendre des précautions à propos de cette variable lors de la parallélisation.

# Indépendance entre parties d'exécution de programme

Deux parties de l'exécution d'un programme sont *indépendantes*, et peuvent s'exécuter en parallèle, si elles accèdent à des données distinctes.

Cependant, la lecture en parallèle est autorisée.

## Pour formaliser la notion d'indépendance

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.

# Formalisation équivalente

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

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

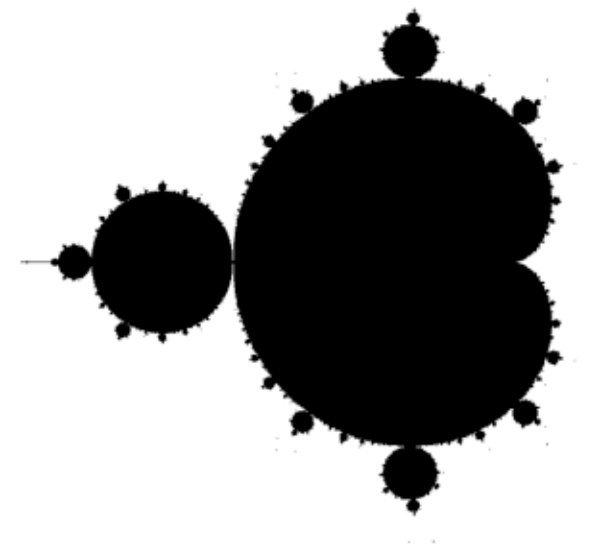
$\rightsquigarrow$  les variables lues ou modifiées par  $P_1$  ne sont pas modifiées par  $P_2$

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

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

# 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);  
    }  
}
```



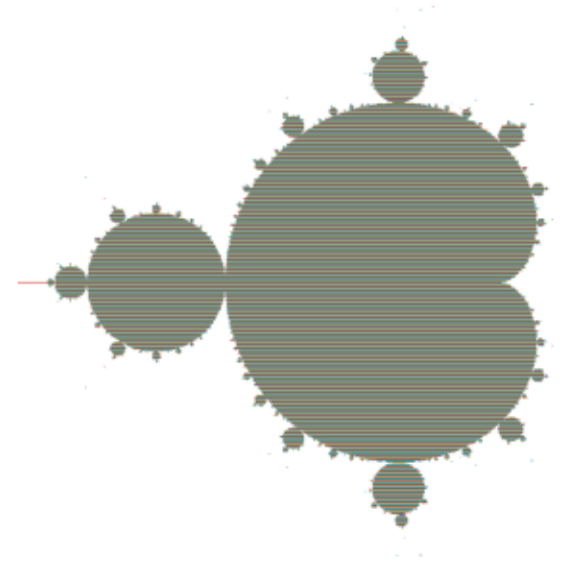
Les différentes itérations de cette double-boucle écrivent sur des données (pixel de l'image) distinctes, et ces données ne sont pas lues : chaque itération est donc indépendante de toutes les autres.

Ça vaut aussi si la partie de programme considérée n'est plus le coloriage d'un pixel mais celui d'une ligne entière.



# 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);  
    }  
}
```

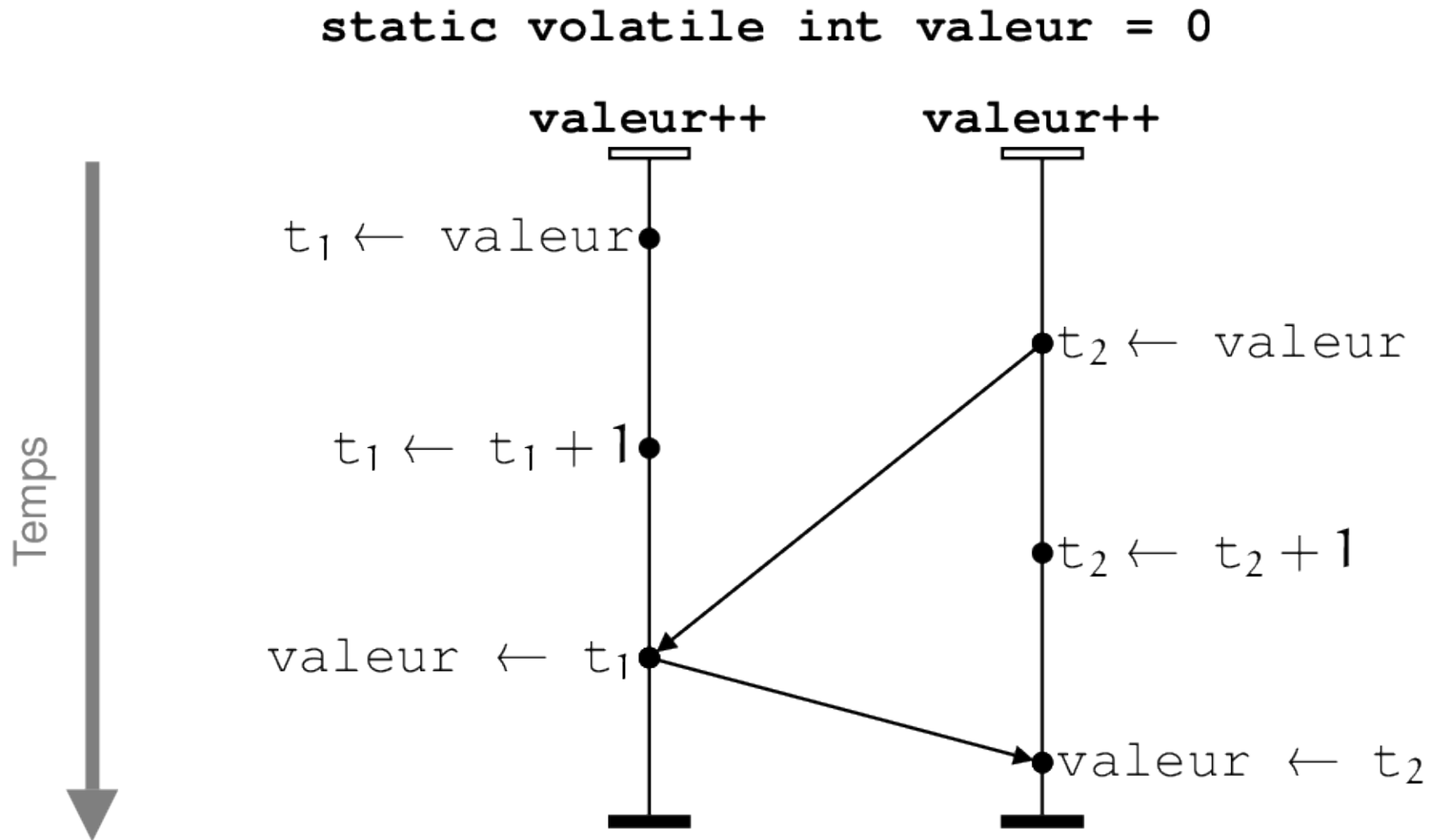


Les différentes itérations de cette double-boucle écrivent sur des données (pixel de l'image) distinctes, et ces données ne sont pas lues : chaque itération est donc indépendante de toutes les autres.

Ça vaut aussi si la partie de programme considérée n'est plus le coloriage d'un pixel mais celui d'une ligne entière.

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

## Exemple bien connu (déjà vu)



*Que vaut `valeur` à la fin ?*

✓ *Indépendance et parallélisation*

☞ *La notion cruciale d'atomicité*

## 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.* »

C'est parfois encore le cas aujourd'hui (cf. Wikipédia, 26 septembre 2022).

## 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.* »

C'est parfois encore le cas aujourd'hui (cf. Wikipédia, 26 septembre 2022).

L'atomicité est une relation entre un bout de code et le reste des opérations susceptibles d'être exécutées en même temps par d'autres processus.

# 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.* »

C'est parfois encore le cas aujourd'hui (cf. Wikipédia, 26 septembre 2022).

L'atomicité est une relation entre un bout de code et le reste des opérations susceptibles d'être exécutées en même temps par d'autres processus.

Considérons le cas simple d'une machine **monoprocasseur** chargée de plusieurs processus et qui exécute à un moment donné une suite d'instructions atomique.

Alors le reste du programme ne peut pas accéder aux variables lues ou modifiées par cette suite d'instructions puisque l'ensemble du reste du programme est à l'arrêt !

En conséquence, les instructions du bout de code atomique sont *indépendantes* des opérations simultanément exécutées par le reste du programme.

# 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*.
- ② Les variables **modifiées** par cette instruction ne peuvent pas être **lues** par le reste du programme *au cours de son exécution*

Contrairement à la notion d'indépendance, qui est relative à deux parties de l'exécution du programme bien déterminées, l'atomicité d'une suite d'opérations se réfère à l'ensemble des autres instructions du programme qui peuvent s'exécuter de manière simultanée ; ces dernières doivent être *indépendantes* de chacune des opérations constituant l'instruction atomique.

## Ce que ça veut dire en pratique

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

En effet, 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 *les modifications de l'état global du programme par l'instruction atomique* ne dépendent pas des actions des autres processus s'exécutant en même temps.

- ② « *Les variables modifiées par cette instruction ne peuvent pas être lues par le reste du programme au cours de son exécution.* »

Donc *les modifications de l'état global du programme par les processus concurrents pendant l'instruction atomique* ne dépendent pas des opérations effectuées par l'instruction atomique ; elles auraient ainsi pu avoir lieu avant l'exécution de l'opération atomique, sans effet observable sur le résultat obtenu.



## Pourquoi c'est important

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

Une erreur lors de l'exécution d'une suite d'instructions atomique ne peut provenir que du code de cette suite d'instructions.

*Inutile d'aller chercher ailleurs !*

## Atomicité absolue ou relative

L'atomicité d'un bout de code peut être

- *relative*, si elle dépend du code effectif du reste du programme ;
- *absolue*, si elle ne dépend pas du reste du programme.

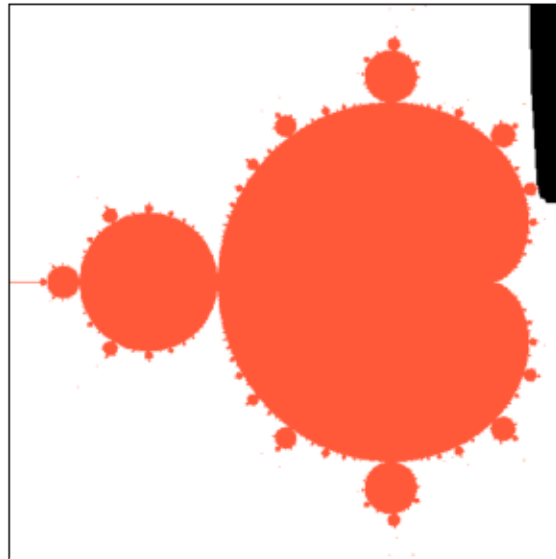
Autant que possible, les méthodes des objets partagés que nous construirons seront atomiques de manière absolue : il sera donc inutile d'explorer l'ensemble du programme pour s'assurer du bon fonctionnement des classes conçues.

Le plus simple sera d'encapsuler les données dans un **objet**, de munir cet objet d'attributs **privés** et d'utiliser un **verrou** pour synchroniser l'ensemble des méthodes permettant l'accès à ces données.

- ✓ *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 de la suite de pixels dans le fichier.

## L'instruction `print()` est atomique en pratique

```
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.

# Jargon des granularités

On distingue habituellement 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 mêmes 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 inquiétant

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

```
i = 2 ; j = i
```

```
i = 9223372036854775807
```

Temps

i = 2

i = 9223372036854775807

j = i

*Que vaut j à la fin ?*

## Exemple avec volatile

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

```
i = 2 ; j = i
```

```
i = 9223372036854775807
```

Temps

i = 2

i = 9223372036854775807

j = i

*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 des deux premiers types **ne sont pas atomiques** : 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

# 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 !*



*B-A BA pour éviter les interblocages*

# Qu'est qu'un programme correct ?

Une **propriété** d'un programme concurrent est une assertion qui est vraie pour toutes les exécutions possibles de ce programme.

On distingue souvent deux catégories de propriétés :

1. **Propriété de sécurité** = « rien de mal ne peut arriver »
2. **Propriété de vivacité** = « quelque chose de bon va arriver »

## Exemples de propriétés

- Terminaison : le programme va toujours terminer.
- Exclusion mutuelle des processus pour l'accès à une section critique.
- Un processus qui le souhaite pourra toujours réussir à entrer dans une section critique.
- Absence de blocage et d'interblocage.

# 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 un 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 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()**.

✓ *B-A BA pour éviter les interblocages*

☞ *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.

- ✓ *B-A BA pour éviter les interblocages*
- ✓ *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 de la programmation par moniteurs** : le programme est composé

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

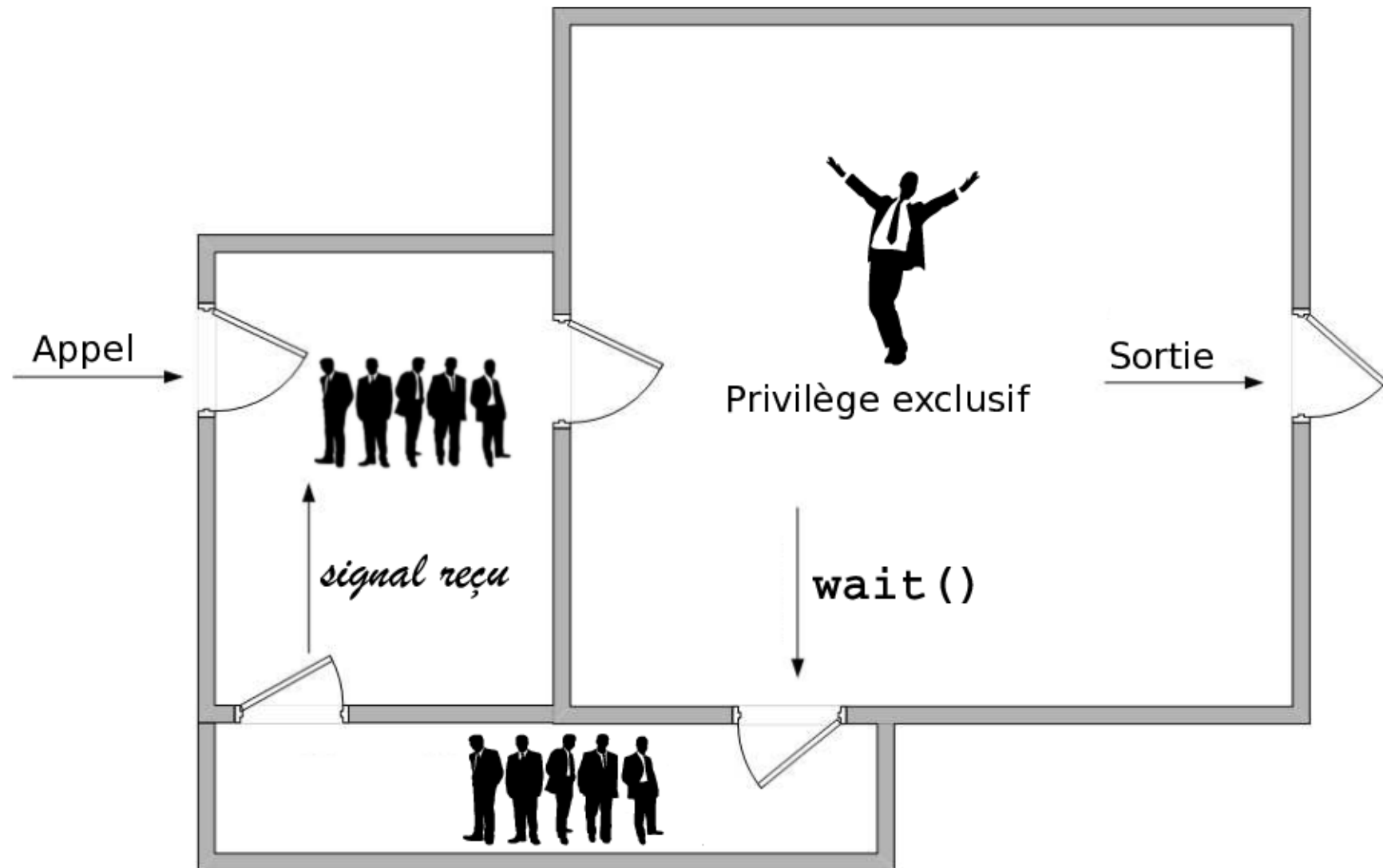
Les interactions et synchronisations entre processus (actifs) se font alors par l'interface des moniteurs (passifs) qui encapsulent les données.

Un moniteur est un module qui regroupe et encapsule des données partagées ainsi que les procédures qui permettent d'ordonner les opérations effectuées sur ces données.

Un moniteur permet de séparer explicitement l'*exclusion mutuelle*, assurée par un verrou, et la *synchronisation conditionnelle*, obtenue à l'aide d'une variable de condition.

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

# Fonctionnement d'un moniteur : verrou et variable de condition



Si l'instruction **wait ()** est rencontrée au cours de l'exécution d'une méthode, le privilège est perdu. Le thread doit recevoir un signal puis réacquérir le verrou pour poursuivre l'exécution de son code.

# Exclusion mutuelle des accès au moniteur

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 **private** l'ensemble des attributs et **synchronized** chacune des méthodes.*

D'autre part, les synchronisations conditionnelles sont implémentées à l'aide de la **variable de condition** associée au moniteur.

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

Pour certains auteurs, la notion de moniteur se réduit à l'association d'une *variable de condition* et d'un *verrou*. Ainsi, chaque objet en Java dispose d'un moniteur.

La documentation Java elle-même évoque le « moniteur d'un objet. »

# 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

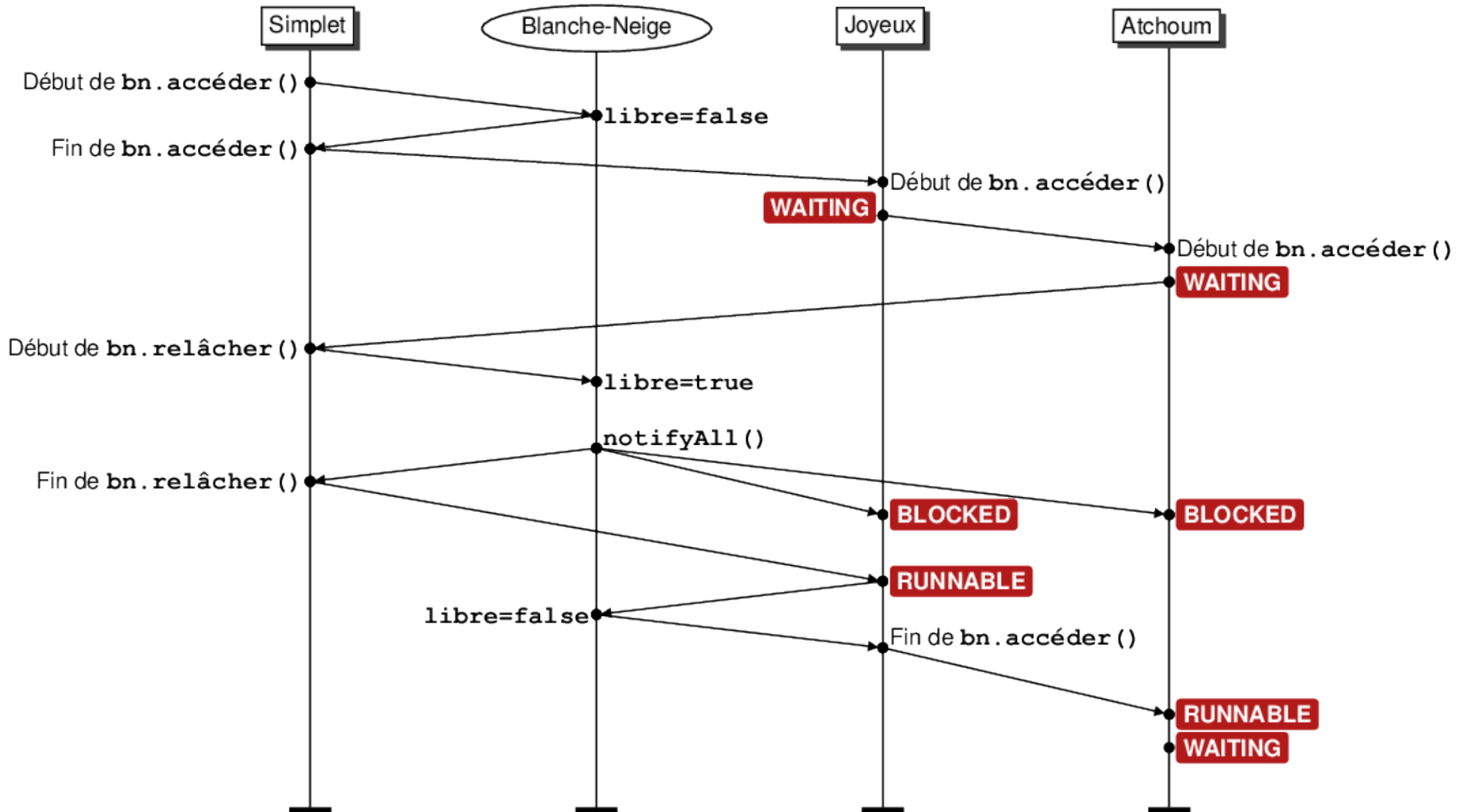
Si on met de côté la phase éventuelle d'attente de conditions favorables, matérialisée par la boucle **while()** au début de la méthode **accéder()**, alors le code résiduel est lui atomique, pour les mêmes raisons que **relâcher()**, puisque le nain qui exécute ce code possèdera alors le verrou intrinsèque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

Chaque appel à **accéder()** fonctionne « comme si » cette méthode était atomique, une fois passée la phase d'attente de conditions favorables à son exécution.

Pour rester cohérents, nous dirons que **accéder()** « *agit de manière atomique.* »

# Fonctionnement du moniteur Blanche-Neige

Initialement libre = true



# 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.
```

- ✓ *B-A BA pour éviter les interblocages*
- ✓ *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 unObjetVide = new Object();  
  
    public static void main(String [] args) {  
        new WakeUp().start();          // Un seul thread est lancé  
    }  
  
    public void run() {  
        synchronized (unObjetVide) {  
            try {  
                unObjetVide.wait();      // Le thread attend sur unObjetVide  
            } 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  
}
```

## Ce qu'il faut retenir

Les problèmes d'*atomicité* sont inhérents à la programmation parallèle (notamment la programmation concurrente) 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*.

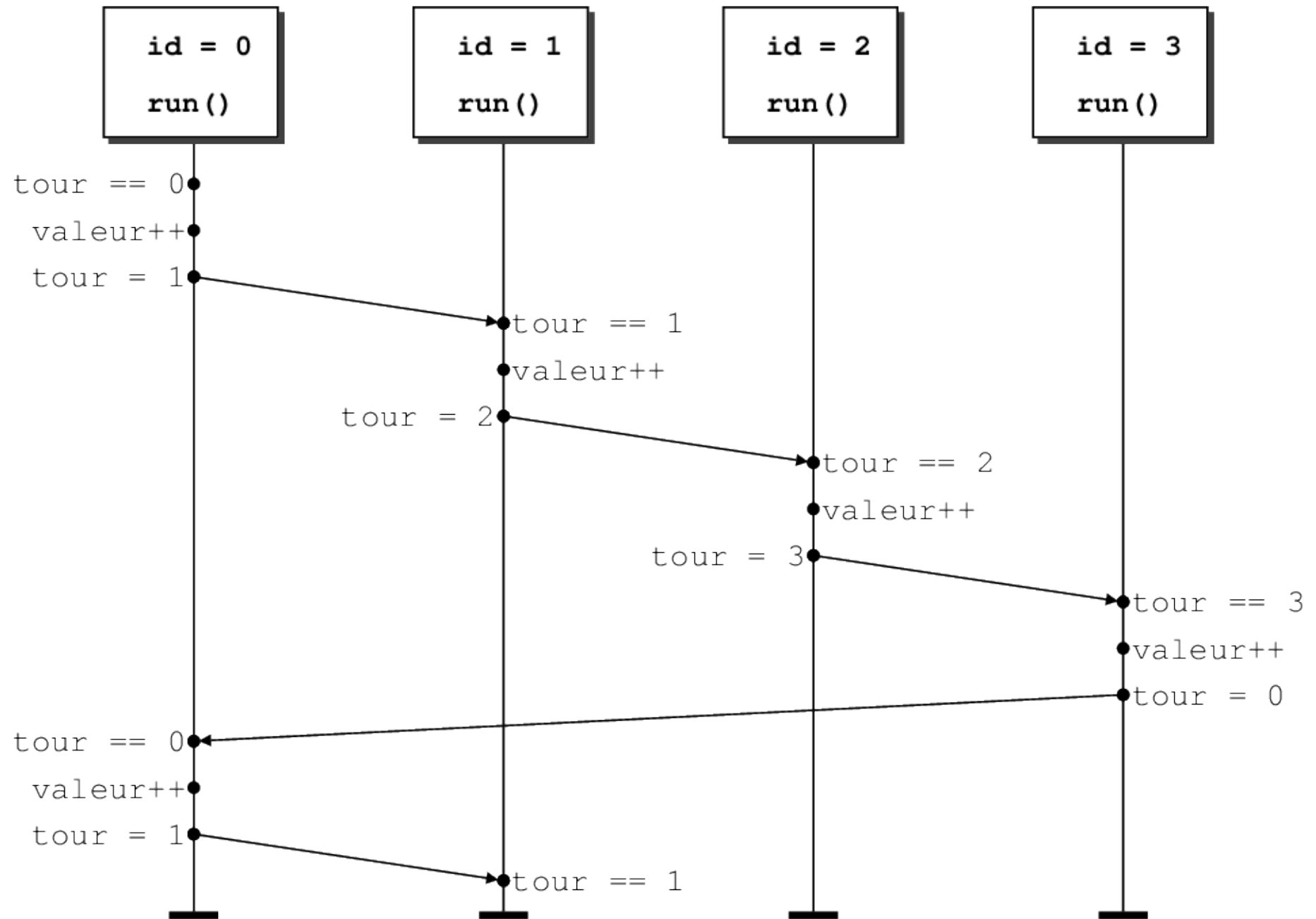
Un thread peut quitter l'instruction **wait()** sans qu'il y ait eu d'instruction **notify()** exécutée. Ce phénomène étrange et rare se nomme « *spurious wake-up* » que je traduis en « signal intempestif » car cela n'a rien à voir avec **sleep()**. Il faut par conséquent protéger chaque **wait()** par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

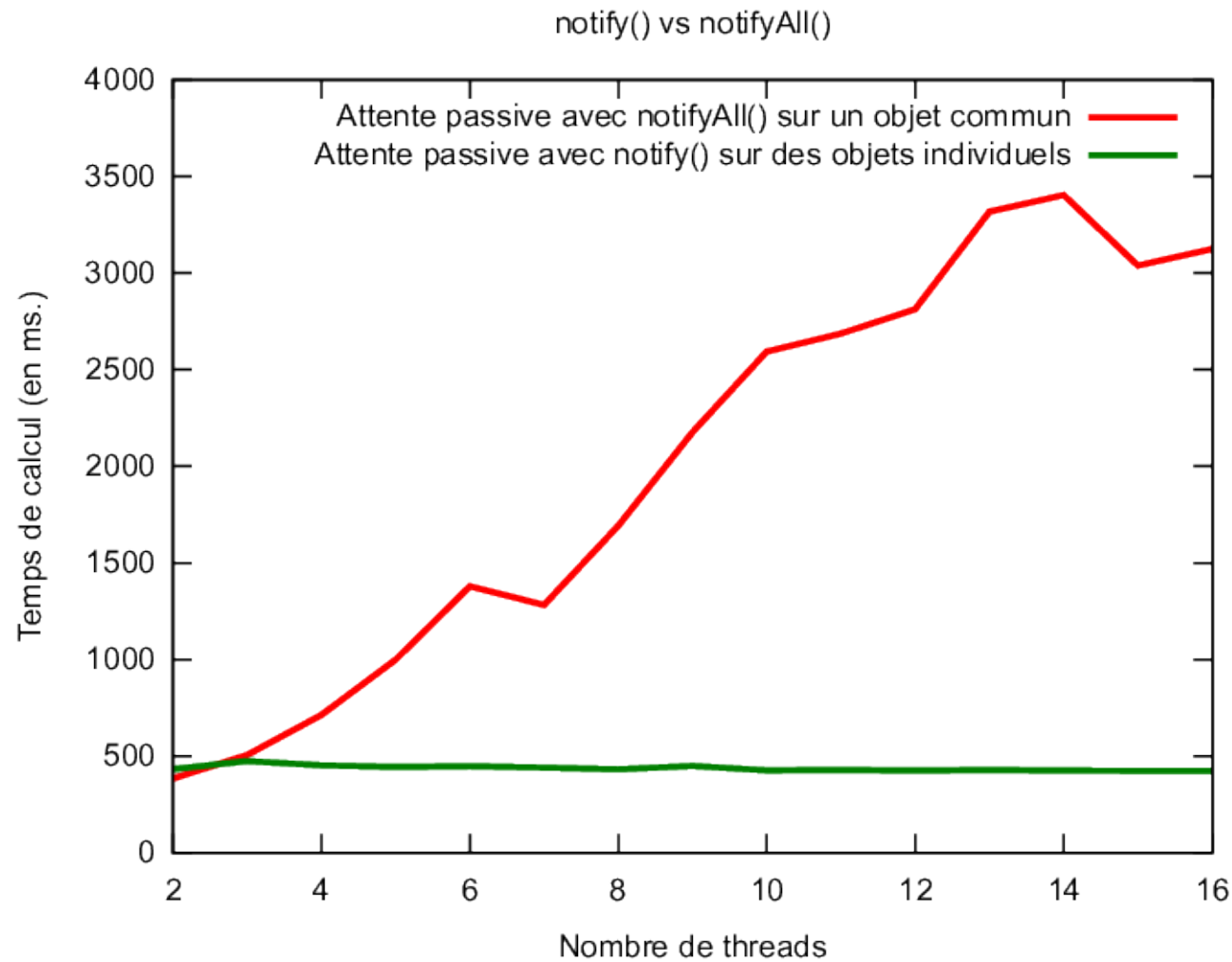
*Pourquoi continuer à hésiter ?*



# 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.