

Indépendance et atomicité

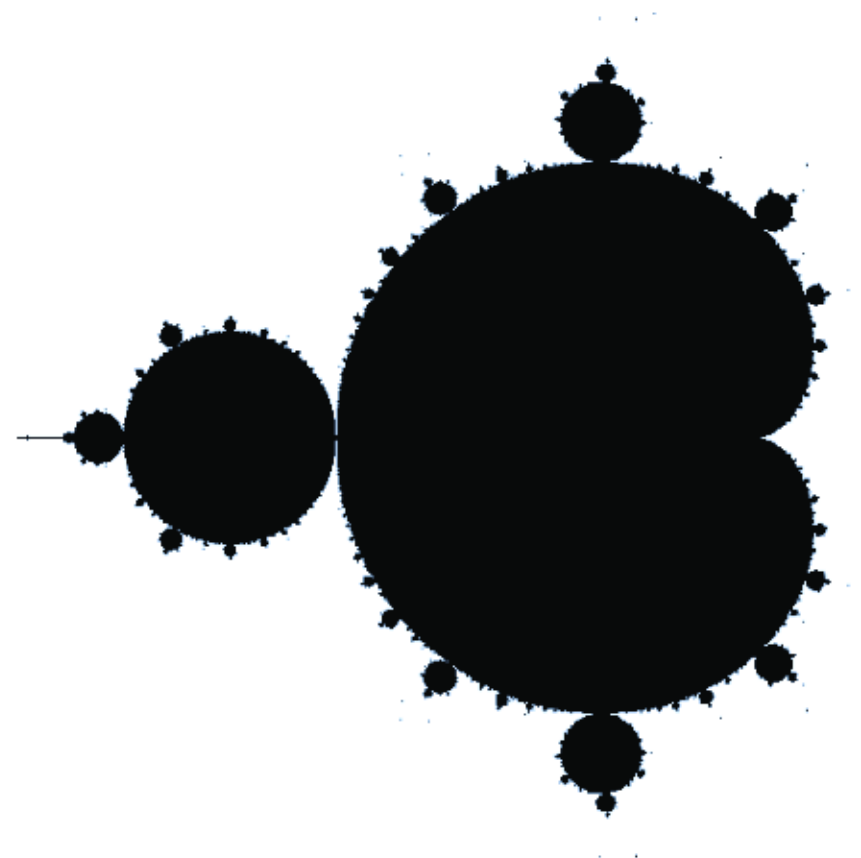
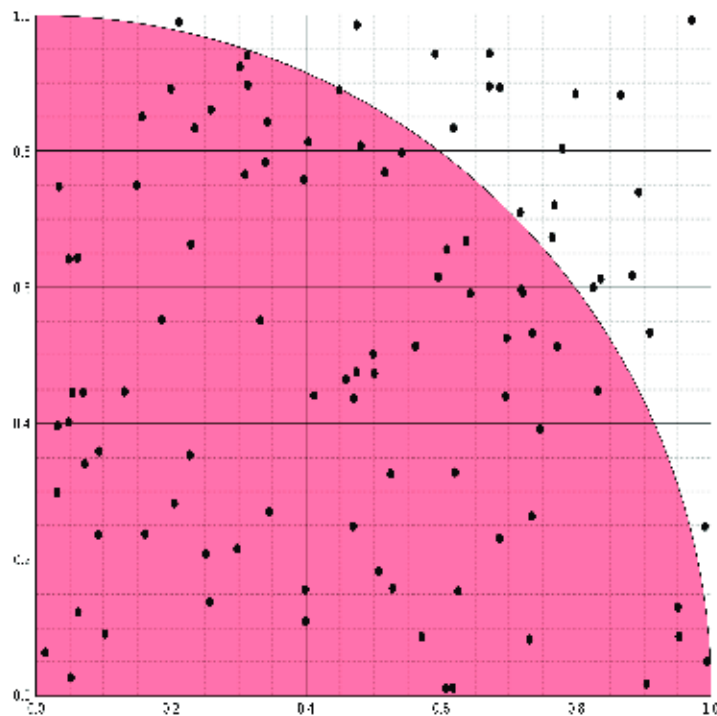
Master Informatique — Semestre 2 — UE obligatoire de 3 crédits



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 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

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

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

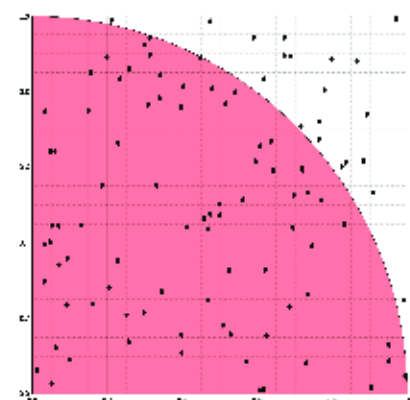
- $W_{P_2} \cap R_{P_1} = \emptyset$

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

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.

```
for (int i = 0; i < nombreDeTirages; i++) {  
    x = Math.random() ;  
    y = Math.random() ;  
    if (x*x+y*y <= 1) tiragesDansLeDisque++ ;  
}
```

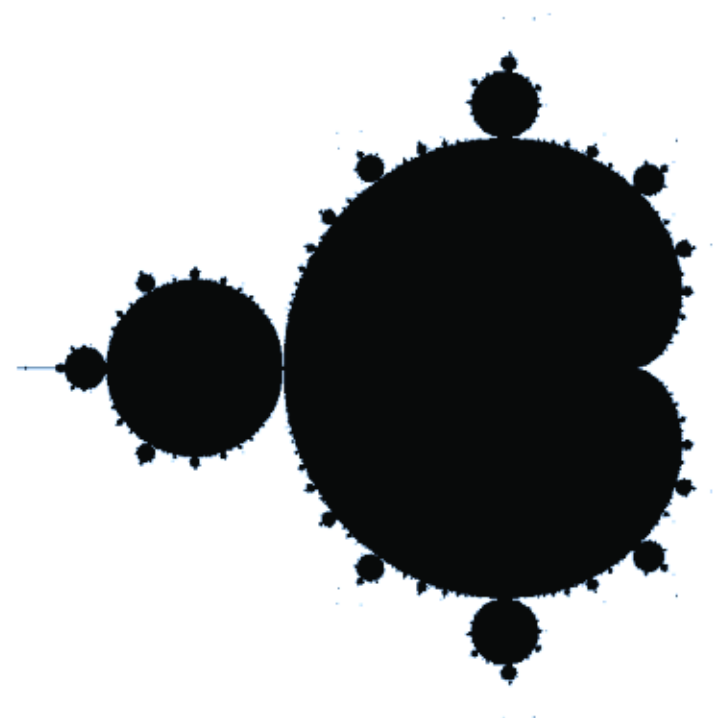


Les différentes itérations de cette boucle ne sont pas indépendantes, car la variable **tiragesDansLeDisque** est potentiellement modifiée par chacune d'elles.

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.

Exemple du calcul de la fractale de Mandelbrot

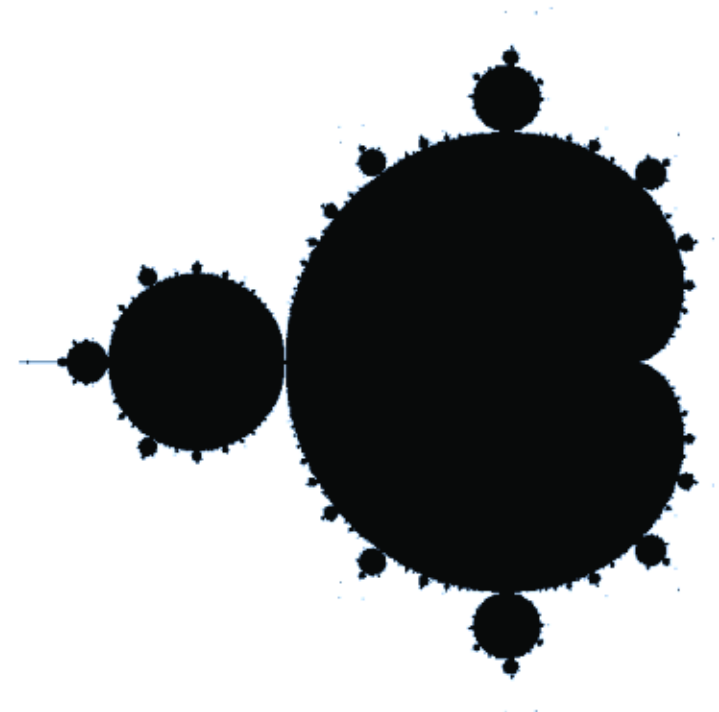
```
for (int i = 0; i < taille; i++) {  
    for (int j = 0; j < taille; j++) {  
        colorierPixel(i, j);  
    }  
}
```



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++` font partie des itérations, en particulier, si l'attribution des lignes à calculer est dynamique.

- ✓ *Indépendance et parallélisation*
- ☞ *La notion cruciale d'atomicité*

Exemple

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

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

```
i = 9223372036854775807
```

Temps

i = 2

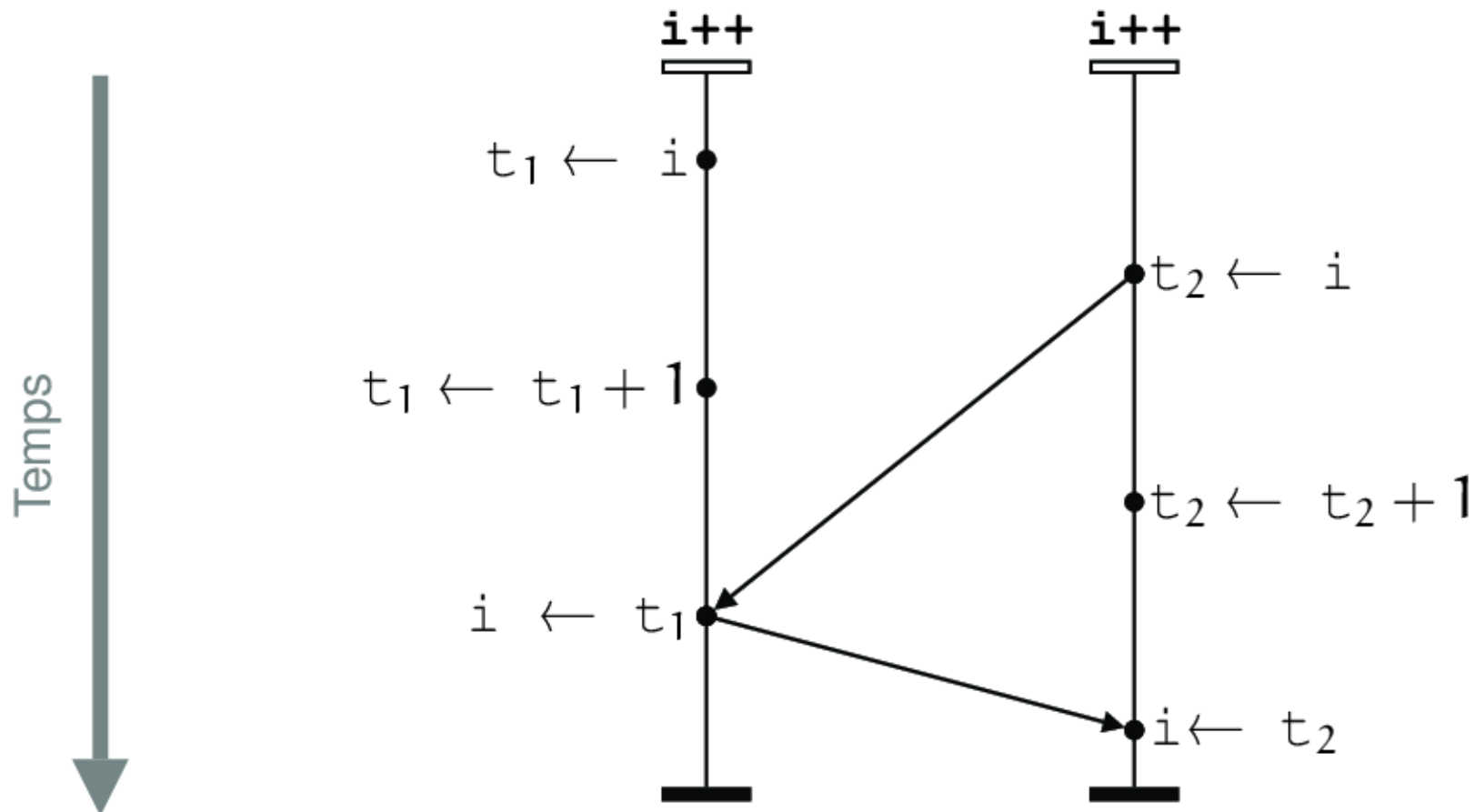
i = 9223372036854775807

j = i

Que vaut j à la fin ?

Autre exemple

```
static volatile int i = 0
```



Que vaut i à la fin ?

La notion d'instruction atomique, intuitivement

Intuitivement, une instruction ou une suite d'instructions d'un programme est **atomique** si elle s'exécute toujours « *comme si* » le programme n'exécute aucune autre action entre le début et la fin de l'instruction.

- **Sur une machine monoprocesseur**, ceci correspond intuitivement à la garantie d'une *absence d'interruption* entre le début et la fin de cette suite d'instructions.
- Pour les machines multi-coeurs, de nos jours, cette notion correspond au *blocage virtuel* de tous les autres threads du programme.

Un peu plus formellement, une suite d'actions forme une instruction atomique si elle s'exécute toujours de façon indépendante des actions concurrentes du programme.

- **Quelles sont les actions concurrentes ?**

Ce sont *toutes les actions du reste du programme* qui peuvent avoir lieu entre le début et la fin de l'instruction atomique.

- **Que signifie « de façon indépendante ? »**

Grosso-modo, les actions potentiellement concurrentes sont indépendantes de chacune des actions constituant l'instruction atomique.

La notion d'instruction atomique, précisément

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

- ① Les variables **modifiées** par cette instruction ne peuvent pas être **modifiées** par le reste du programme *au cours de son exécution*.
- ② Les variables **lues** par cette instruction ne peuvent pas être **modifiées** par le reste du programme au cours de son exécution.
- ③ Les modifications effectuées par cette instruction ne sont **visibles** par le reste du programme qu'*à la fin* de son exécution.

Contrairement à la notion statique et locale d'indépendance, l'atomicité se réfère *aux exécutions potentielles du programme dans sa globalité*.

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* est le même que s'ils étaient suspendus jusqu'à la fin de l'instruction atomique.

Ainsi, comme annoncé, 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*

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 mêmes données, le plus souvent **à l'aide d'un verrou**. On parle alors d'atomicité **de grande granularité**.

```
for (int i = 0; i < nombreDeTirages; i++) {  
    x = Math.random() ;  
    y = Math.random() ;  
    if (x * x + y * y <= 1) {  
        synchronized(Tireur.class) { tiragesDansLeDisque++; }  
    }  
}
```

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 2 — 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

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.

N.B. Ce deadlock n'arrivera pas forcément à chaque fois.

Correction vs. performances

Solution générale : « **ordonner pour régner** »

Il faut choisir, dès la conception, un *ordre total* sur l'ensemble des verrous utilisés. Puis **toujours prendre les verrous dans cet ordre.**

Aucun interblocage ne pourra alors avoir lieu.

- ~> S'il y a un seul verrou, il n'y aura pas d'interblocage ; il pourra y avoir cependant d'*autres types de blocage*, notamment lors d'un appel à **wait()**.
- ~> Si un verrou protège des sections critiques qui ne font appel à aucun code tiers et ne requièrent aucun autre verrou, alors ce verrou ne conduira jamais à un interblocage.

Le plus souvent, *par souci d'efficacité*, il faut éviter d'utiliser un seul verrou (même si c'est une solution radicale contre l'interblocage). 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.

✓ *B-A BA pour éviter les interblocages*

☞ *Sémaphores (Edsger Dijkstra, 1963)*

Qu'est-ce qu'un sémaphore ?

Un peu comme un verrou, un **sémaphore** est une variable spéciale contenant un nombre entier positif et manipulée **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 une 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.

Les sémaphores ne sont pas conseillés !

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)*

☞ *Moniteurs*

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

Un **moniteur** est une forme de module qui assure l'**exclusion mutuelle** et la **synchronisation conditionnelle** à l'aide de deux mécanismes indépendants.

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

- les processus qui agissent ;
- les moniteurs qui subissent.

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

Un moniteur est un module qui regroupe et encapsule une ressource partagée (les données) ainsi que les procédures qui permettent de manipuler cette ressource.

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

Exclusion mutuelle des accès au moniteur

Un moniteur est une sorte de type abstrait qui sépare de façon explicite l'exclusion mutuelle de la synchronisation conditionnelle.

Le plus souvent, un moniteur assure qu'une procédure (ou méthode) qu'il exporte sera exécutée de façon « **exclusive** », c'est-à-dire qu'il y aura au plus un appel d'une procédure du moniteur actif à un instant donné.

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

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

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

Moniteur de Blanche-Neige (1/2)



```
class BlancheNeige {  
    private volatile boolean libre = true;  
        // Initialement, Blanche-Neige est libre  
  
    public synchronized void requerir() {  
        System.out.println(Thread.currentThread().getName() +  
            + "_veut_la_ressource");  
    }  
}
```

Les méthodes d'un moniteur sont toutes synchronisées !

Moniteur de Blanche-Neige (2/2)

```
public synchronized void acceder() {
    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 relacher() {
    System.out.println(Thread.currentThread().getName()
        + "_relâche_la_ressource.");
    libre = true;
    notifyAll();
}
}
```

Ces méthodes sont elles atomiques ?

Ces méthodes sont elles atomiques ?

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

relacher() 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 **relacher()** est en cours d'exécution, aucun autre thread ne peut lire ou modifier la variable **libre**.

La méthode **relacher()** est donc également atomique.

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

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

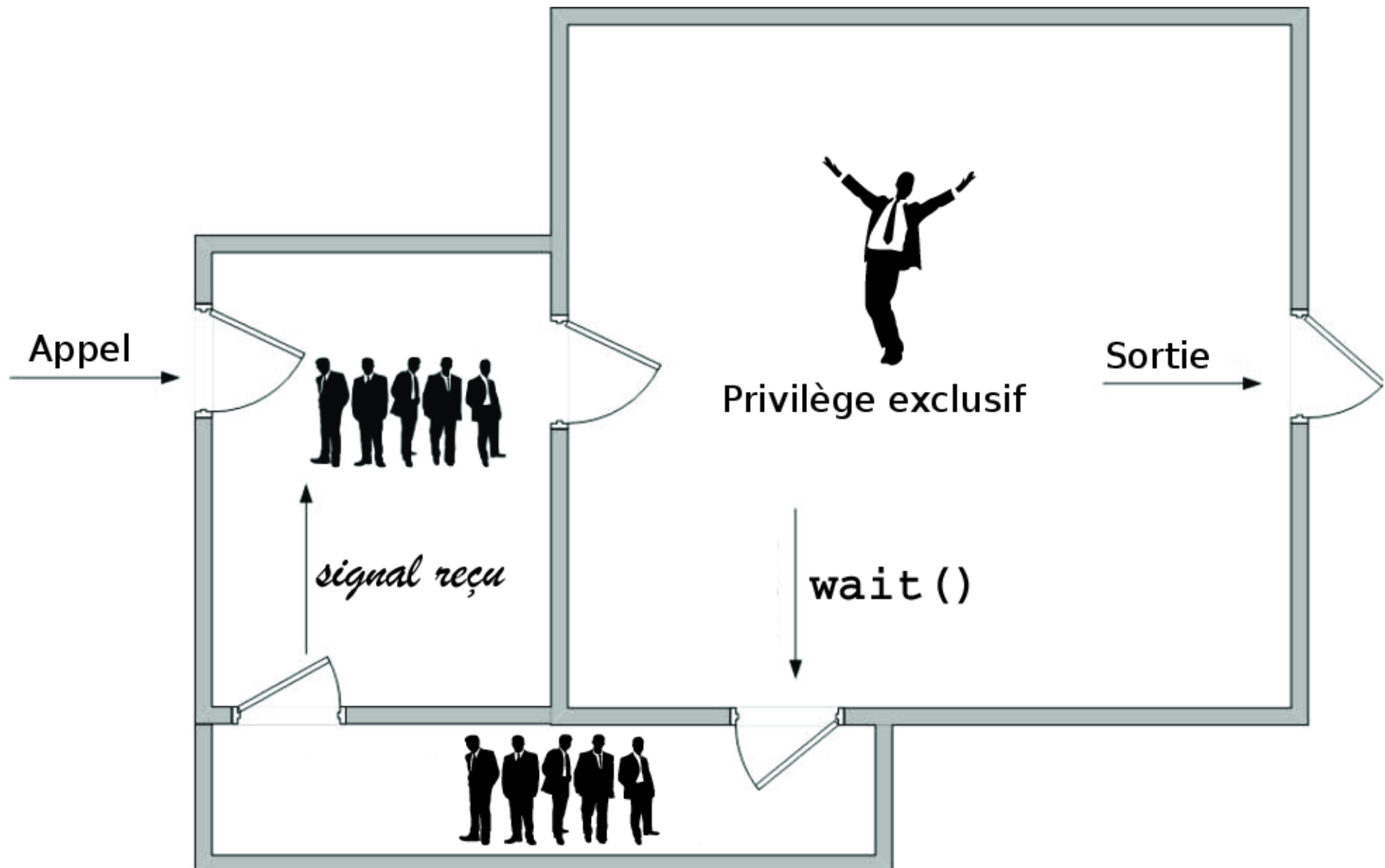
La méthode **accéder()** agit de manière atomique

Si on met de côté la phase potentielle d'attente de conditions favorable, au début de la méthode **accéder()**, alors le code résiduel est lui atomique, pour les mêmes raisons que **relacher()**.

Puisque que la phase d'attente préalable ne modifie pas les données de l'objet, chaque appel à **accéder()** fonctionne « comme si » cette méthode était atomique après une phase d'attente : seul compte ce qui suit l'attente préliminaire de conditions favorables.

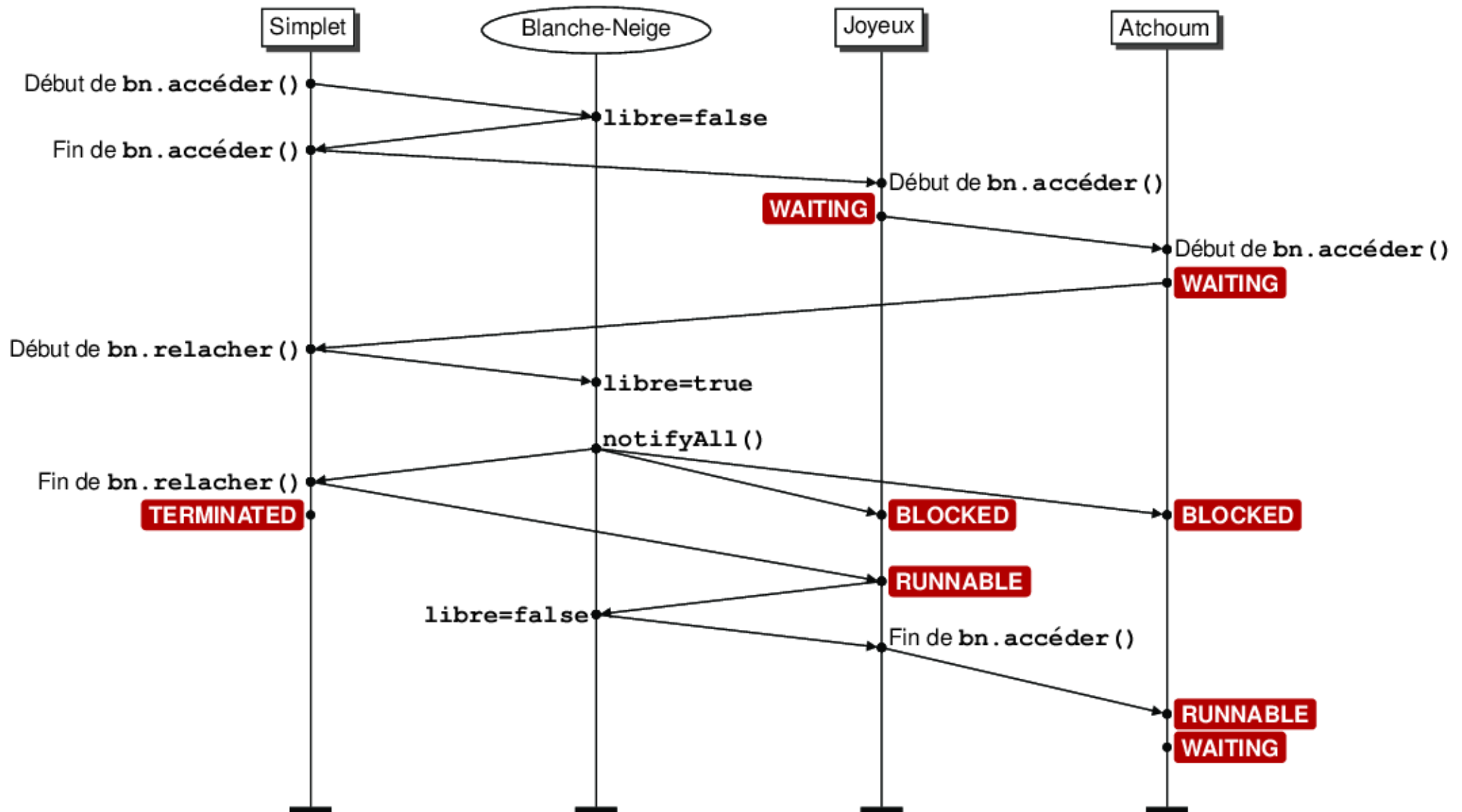
La méthode **accéder()** agit donc de manière atomique.

Fonctionnement d'un moniteur



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)*
- ✓ *Moniteurs*
- ☞ *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 ignoree) {};  
        }  
    }  
}
```



Ce programme peut-il terminer ?

Extrait de la Javadoc `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 (ou distribuée) et une source d'erreurs fréquente dans les applications.

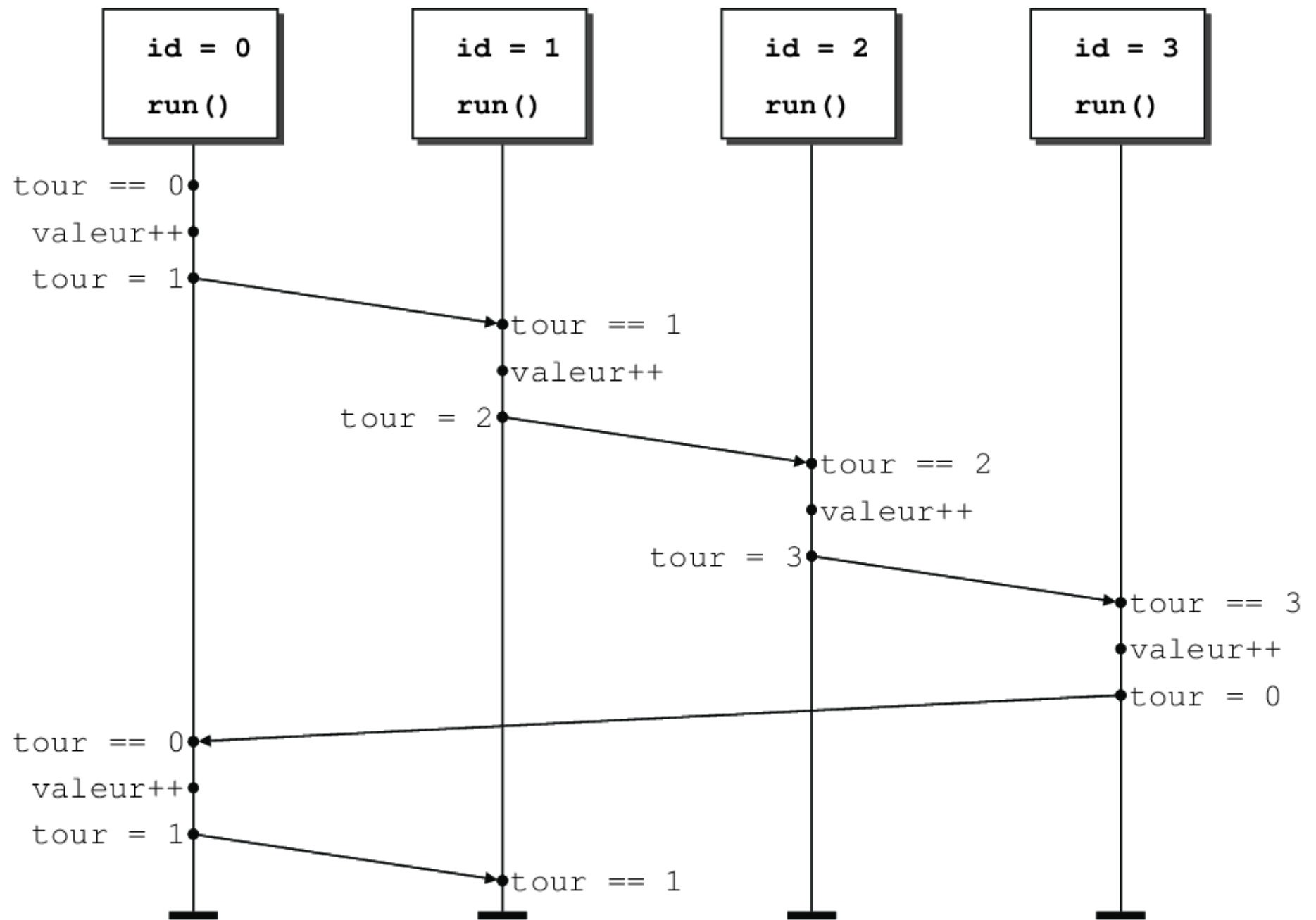
Les *verrous* sont bien pratiques pour assurer l'atomicité ; cependant, mal utilisés, ils provoquent facilement des *interblocages*. Les *sémaphores* sont aussi une technique très puissante, mais risquée, peu claire et donc non recommandée dans ce cours. En revanche, le concept de *moniteur* permet d'aborder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

Un thread peut quitter l'instruction **wait()** sans qu'il y ait de **notify()** exécuté. Ce phénomène 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 Javadoc.

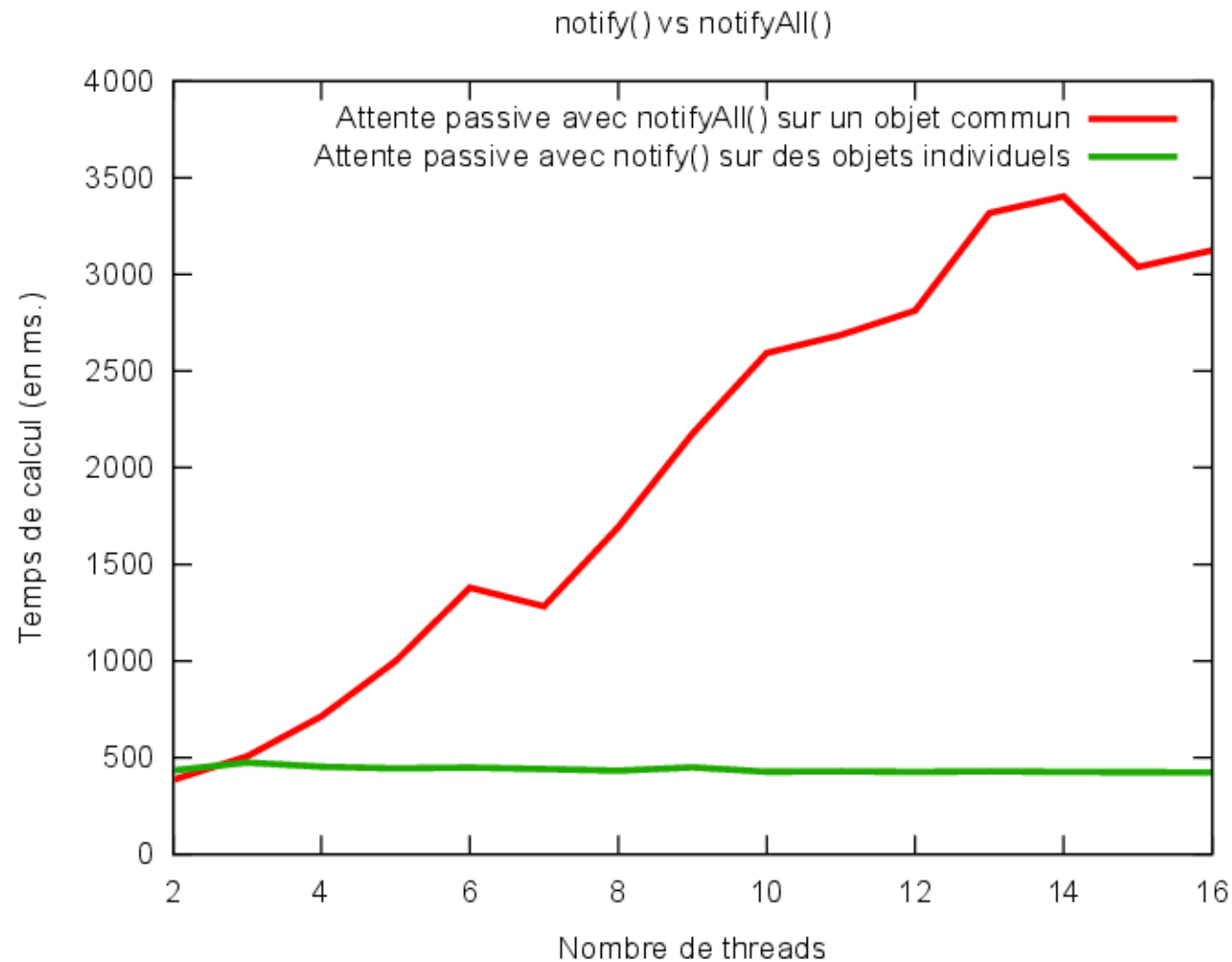
Notez aussi qu'un programme correct restera correct en remplaçant chaque **notify()** par **notifyAll()**.

Pourquoi 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 2 — UE obligatoire de 3 crédits

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

Pour chacune de ces classes, ce paquetage

- ① propose des opérations **atomiques** (ou qui agissent de manière atomique) sur ces objets : **get ()** , **set ()** , **getAndSet ()** , etc. **sans requérir à un verrou** ;
- ② garantit la **visibilité** des opérations réalisées sur l'objet comme s'il était déclaré **volatile**.

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** (puisque'il n'y pas de précaution à prendre) et plus **sûr** puisque'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.

- ✓ *Les objets atomiques (depuis Java 5)*
- ☞ *Exemple des compteurs anarchiques*

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 19591
```

Compteurs anarchiques synchronisés sur un verrou commun

```
public void run() {  
    for (int i = 1; i <= 10_000; i++)  
        synchronized (Compteur.class) { 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'incrément 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.

Compteurs anarchiques partageant un entier atomique

```
static AtomicInteger valeurAtomique = new AtomicInteger(0);  
...  
public void run() {  
    for (int i = 1; i <= 10_000; i++)  
        valeurAtomique.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 **valeurAtomique** 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 !

- ✓ *Les objets atomiques (depuis Java 5)*
- ✓ *Exemple des compteurs anarchiques*
- ☞ *Construction artisanale d'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.

La méthode compareAndSet, sous la forme d'un pseudo-code

```
boolean CAS(int* reg, int attendue, int nouvelle) {  
    <  
        int courante = *reg ;  
        if (courante == attendue) {  
            *reg = nouvelle ;  
            return true ;  
        }  
        return false ;  
    >  
}
```

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));  
        // Le thread tourne en boucle tant que le verrou est pris.  
    }  
    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`, ce qui provoque la sortie de la boucle et de la méthode `lock()`.