

Programmation Objet Concurrente

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

La loi de Moore

L'industrie informatique s'est développée depuis 50 ans en s'appuyant sur la loi de Moore, selon laquelle les performances des microprocesseurs doublent tous les 18 mois.

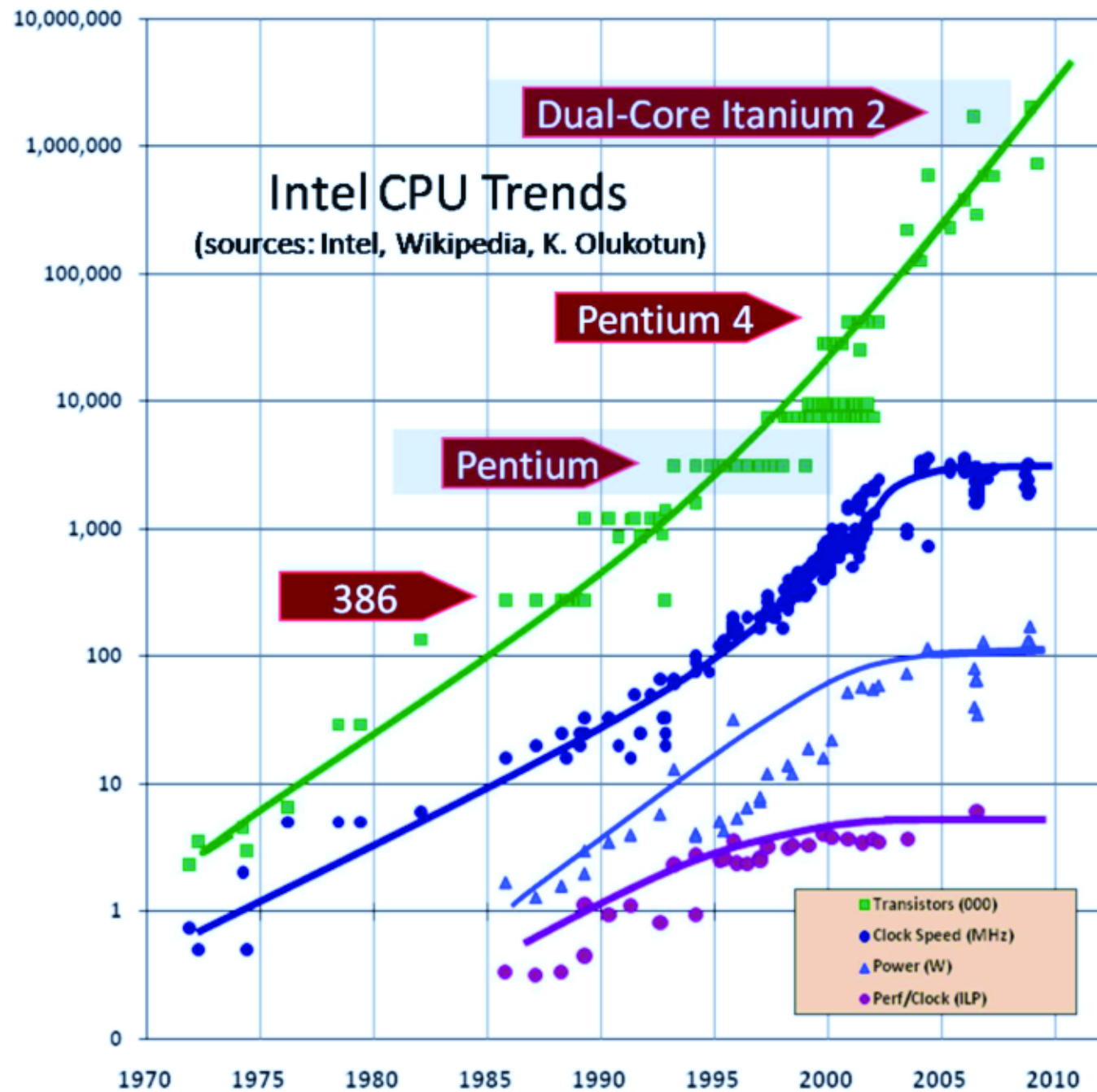
En 2004 la perspective de poursuivre la progression des performances selon le modèle classique du monoprocesseur est abandonnée par Intel.

« Intel said on Friday that it was scrapping its development of two microprocessors, a move that is a shift in the company's business strategy... »

SAN FRANCISCO, May 7. 2004, New York Times

Depuis lors, la fréquence d'horloge des microprocesseurs s'est stabilisée et la loi de Moore n'est préservée que par la multiplication du nombre de coeurs.

Fréquence d'horloge des microprocesseurs



Motivations

Si l'écriture de programmes corrects est un exercice difficile, l'écriture de programmes parallèles corrects l'est encore plus. En effet, par rapport à un programme séquentiel, **beaucoup plus de choses peuvent mal tourner** dans un programme parallèle.

Pourquoi s'intéresser alors à la concurrence ?

- ① Les threads sont le moyen le plus direct d'**exploiter la puissance** des ordinateurs actuels. À mesure que le nombre de coeurs augmente dans les machines, l'exploitation de la concurrence devient incontournable.
- ② En outre, les threads sont une fonctionnalité incontournable du langage Java ; ils permettent aussi de **simplifier** le développement d'applications en transformant du code compliqué en un code plus court et plus structuré, en le divisant en tâches distinctes.

Plan du cours

① Threads en Java

~> *Les instructions de base*

② Problème d'atomicité et techniques classique de synchronisation

~> *Les concepts fondamentaux*

③ Outils : collections concurrentes, réservoirs de threads, etc.

~> *Pourquoi réinventer la roue chaque jour ?*

④ Programmation optimiste et programmation sans verrou

~> *Un souci de performance !*

⑤ Le modèle mémoire Java

~> *Je sais ce qu'est un programme « bien synchronisé »*

Modalités de contrôle des connaissances (alias les MCC)

Il s'agit d'une UE obligatoire de 3 crédits avec 10h. de cours, 8h. de TD et 8h. de TP.
L'examen terminal dure 2h. et se déroule **sans document**.

La note finale NF se compose de deux notes

- une note d'examen terminal : ET
- une note de projet : P

$$NF = 0,25 \times P + 0,75 \times ET$$

La note de projet P influe donc sur la note finale ; elle est conservée en seconde session (mais elle peut ne pas être prise en compte).

En seconde session, il y a un nouvel examen terminal ET' .

$$NF = \max(0,25 \times P + 0,75 \times ET', ET').$$

Organisation du travail demandé

Pour compenser la réduction du nombre d'heures allouées à cette UE, certains exercices sont à préparer *avant* chaque TD/TP.

Pour éviter la surcharge de travail en fin de semestre, le projet consiste uniquement en rendus de TP.

Quelques sigles du polycopié d'exercices



Un exercice à faire *avant* le TD/TP.



Un exercice à faire absolument.



Un exercice à finir chez soi.



Un exercice à rendre.

Programmation multithreadée en JAVA

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Thread objet en Java : implémentation

Qu'est-ce qu'un thread ?

C'est un objet qui correspond à un processus qui exécute du code.

Il faut d'abord créer un objet thread puis le lancer afin qu'il s'exécute.

N.B. C'est une différence intéressante avec le **fork()** de C, qui crée et lance l'exécution d'un seul coup.

En Java, il y a deux moyens de créer des classes dont les objets correspondent à des threads.

(1) En créant une classe qui **hérite** de la classe **Thread**.

Cette classe doit nécessairement implémenter une méthode **run()** qui décrit le code exécuté. Cette méthode est appelée de manière un peu particulière car il faut appeler **start()** sur le thread pour lancer son exécution.

Sinon ?

Si un thread fait appel à `t.run()` au lieu de `t.start()` :

- ① Le thread `t` ne démarre pas.
- ② Le thread courant (c'est-à-dire celui qui a appelé `t.run()`) exécute lui-même la méthode `run()` du thread `t`.

C'est normal !

```
public class Exemple {  
    public static void main(String[] args) {  
        Thread t = new monThread();  
        t.start();  
    }  
}  
  
public class monThread extends Thread {  
    public void run() {  
        for (int i = 1; i <= 1000; i++) System.out.println(i);  
    }  
}
```

(2) En créant une classe qui **implémente** l'interface **Runnable**.

L'héritage multiple étant interdit en Java, cela permet à une classe de threads d'hériter d'une autre classe que la classe Thread.

```
public class Exemple {  
    public static void main(String[] args) {  
        Thread t = new Thread( new monRunnable() );  
        t.start();  
    }  
}  
  
public class monRunnable implements Runnable {  
    public void run() {  
        for (int i = 1; i <= 1000; i++) System.out.println(i);  
    }  
}
```

Tout en un

```
public class monThread extends Thread {  
    public static void main(String[] args) {  
        new monThread().start();  
        new monThread().start();  
    }  
    public void run() {  
        for (int i = 1; i <= 1000; i++) System.out.println(i);  
    }  
}
```

- Une classe de threads ou de runnables peut contenir le **main()**.
- Il est possible de lancer un thread en le créant.
- Il est possible de lancer plusieurs threads qui fonctionneront « en même temps ».

Nous verrons un peu plus tard aujourd'hui qu'il est possible aussi de lancer un thread sans utiliser une classe dédiée, mais le code est alors assez peu lisible...

getName() et setName()

Par défaut, le nom d'un thread est « **Thread-** » suivi d'un numéro.

Le nom d'un thread peut être obtenu par la méthode `getName()`.

Le nom d'un thread peut être modifié par la méthode `setName(String)`.

```
public class monThread extends Thread {  
    public static void main(String[] args) {  
        new monThread().start();  
        new monThread().start();  
    }  
    public void run() {  
        for (int i = 1; i <= 1000; i++)  
            System.out.println(i + "_" + currentThread().getName());  
    }  
}
```

Que verra-t-on en sortie ?

Exemple d'exécution

...

102 Thread-0

103 Thread-0

91 Thread-1

104 Thread-0

92 Thread-1

93 Thread-1

94 Thread-1

105 Thread-0

106 Thread-0

...

Gestion des priorités

La priorité d'un thread est un entier compris entre **MAX_PRIORITY** et **MIN_PRIORITY** et vaut **NORM_PRIORITY** par défaut.

Les méthodes **getPriority()** et **setPriority(int)** permettent d'y accéder.

```
public class monThread extends Thread {  
    public static void main(String[] args) {  
        monThread p1 = new monThread();  
        monThread p2 = new monThread();  
        p1.setPriority(MAX_PRIORITY);  
        p2.setPriority(MIN_PRIORITY);  
        p1.start(); p2.start();  
    }  
    public void run() {  
        for (int i = 1; i <= 1000; i++)  
            System.out.println(i + "_" + currentThread().getName());  
    }  
}
```

Exemple d'exécution

...

102 Thread-0

103 Thread-0

1 Thread-1

104 Thread-0

2 Thread-1

105 Thread-0

3 Thread-1

106 Thread-0

4 Thread-1

...

La Javadoc dit seulement : « Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. »

Exemple d'exécution sur un monoprocesseur (machine ens1 à Luminy)

```
...@ens1:...$ java monThread
```

```
1 Thread-0
```

```
2 Thread-0
```

```
3 Thread-0
```

```
...
```

```
23 Thread-0
```

```
1 Thread-1
```

```
2 Thread-1
```

```
3 Thread-1
```

```
...
```

```
14 Thread-1
```

```
24 Thread-0
```

```
25 Thread-0
```

```
26 Thread-0
```

```
...
```

- ✓ *Construction d'un thread*
- ✓ *Propriétés d'un thread*
- ☞ *État d'un thread Java*

Différents états d'un thread

Depuis Java 5, il est possible de connaître l'*état d'un thread* via la méthode `getState()` ; cet état est un élément du type énuméré `Thread.State` :

NEW : le thread n'a pas encore démarré ;

RUNNABLE : il exécute la méthode `run()` de son code ou il attend une *ressource système*, par exemple l'accès à un processeur ;

BLOCKED : il est bloqué en attente d'un *privilège logique*, par exemple de l'acquisition d'un *verrou* ;

WAITING : il est en attente (d'une durée indéfinie) d'un évènement provoqué par un autre thread, par exemple de l'envoi d'un *signal* ;

TIMED_WAITING : il attend qu'une durée s'écoule ou, éventuellement, qu'un évènement provoqué par un autre thread survienne ;

TERMINATED : il a fini d'exécuter son code.

Pour attendre un temps déterminé, c'est-à-dire *faire une pause* : `sleep()`

`sleep(long millis)` interrompt le déroulement du thread pendant une durée spécifiée en millisecondes.

```
public class monThread extends Thread {
    public static void main(String[] args) {
        monThread p1 = new monThread();
        monThread p2 = new monThread();
        p1.start();  p2.start();
    }
    public void run() {
        for (int i = 1; i <= 10; i++){
            System.out.println(i + " " + currentThread().getName());
            sleep(1000);
        }
    }
}
```

Exemple d'exécution (Lire la JavaDoc sur `sleep(d)`)

...

15 Thread-1

15 Thread-0

16 Thread-1

16 Thread-0

17 Thread-1

17 Thread-0

18 Thread-1

18 Thread-0

19 Thread-0

19 Thread-1

20 Thread-1

20 Thread-0

21 Thread-0

...

Utilité de `sleep()`

En pratique, nous utiliserons la méthode `sleep()` pour ralentir le déroulement d'un programme à l'écran.

La méthode `sleep()` est souvent utilisée d'ailleurs pour programmer le rendu d'applications graphiques, notamment pour les jeux, sous la forme suivante :

```
while ( ! terminated ) {  
    update() ;    // Mise à jour du modèle  
    paint() ;     // Rendu du modèle sur la fenêtre graphique  
    sleep(40) ;   // Fréquence d'affichage souhaitée  
}
```


Pour attendre qu'un thread ait terminé

`t.join()` provoque l'attente de la fin de l'exécution du thread `t` (« c'est bloquant »).

De plus, les variantes

— `join(long millis)` et

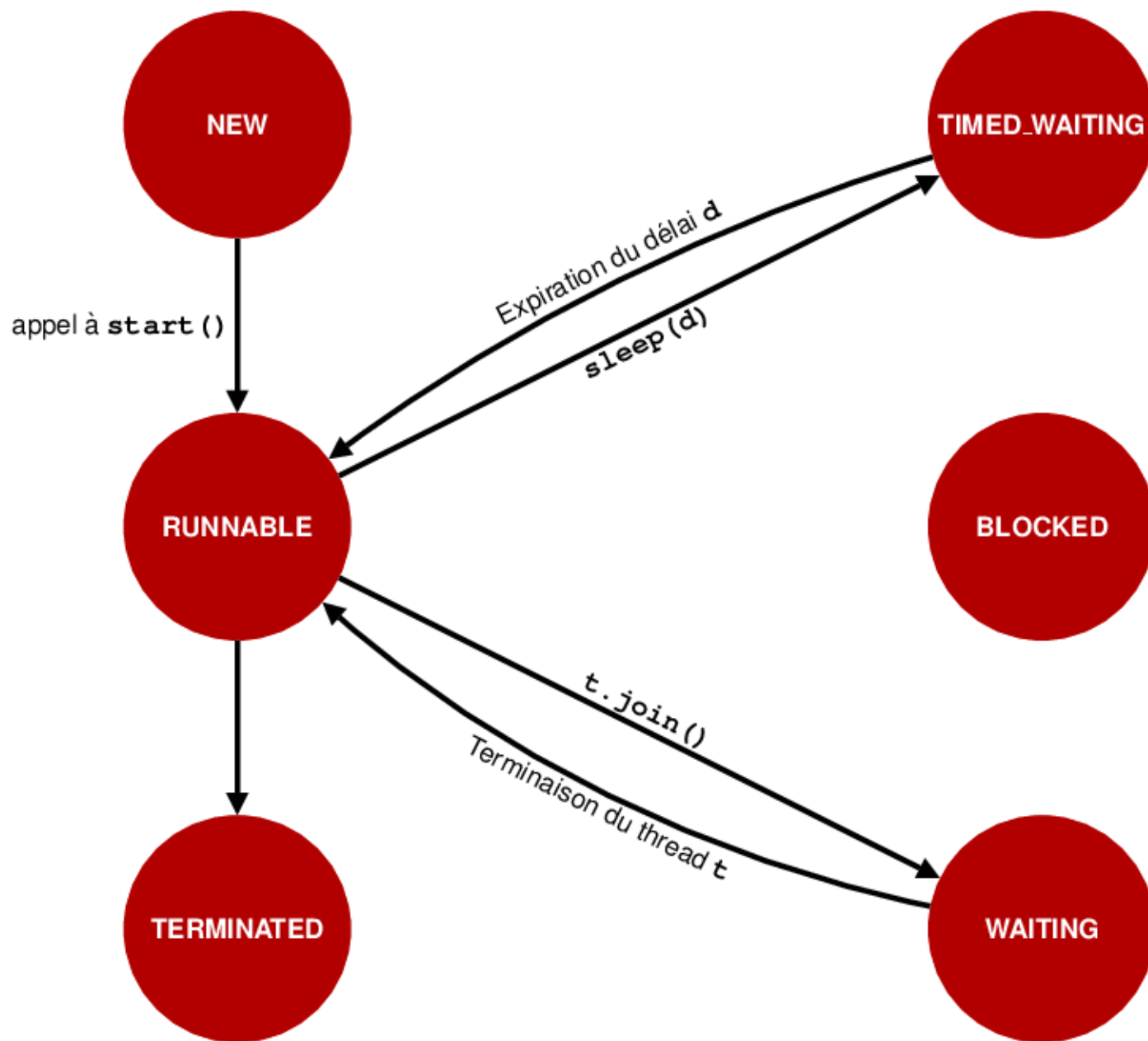
— `join(long millis, long nanos)`

limitent l'attente à une durée spécifiée.

Ces trois méthodes lancent une **InterruptedException** si le thread est *interrompu* lors de l'appel de ces méthodes ou pendant leur exécution.

Rmq. Les méthodes `sleep(long m)` et `sleep(long m, long n)` lanceront également une **InterruptedException** si le thread placé en pause est interrompu.

Les six états d'un thread



Ordonnancement des threads

yield() interrompt éventuellement le déroulement du thread afin de laisser du temps pour l'exécution des autres threads.

La Javadoc dit : « **static void yield()** : A hint to the scheduler that the current thread is willing to yield its current use of a processor. »

La méthode **yield()** laissera le thread dans l'état **RUNNABLE**.

Je déconseille pour commencer d'utiliser yield() !

car **yield()** n'offre aucune garantie permettant de résoudre les problèmes de synchronisation que nous rencontrerons.

Néanmoins, nous verrons aujourd'hui que **yield()** permet dans certains cas d'améliorer les performances.

L'accès d'un thread au(x) processeur(s)

Pour exécuter son code, un thread doit accéder à un processeur. Plusieurs cas sont possibles :

- ① il exécute son code (il a donc accès à l'un des processeurs) ;
- ② il attend l'accès à un processeur (mais il pourrait s'exécuter) ;
- ③ il attend un événement particulier l'autorisant à poursuivre son code.

Un thread peut libérer le processeur qu'il occupe

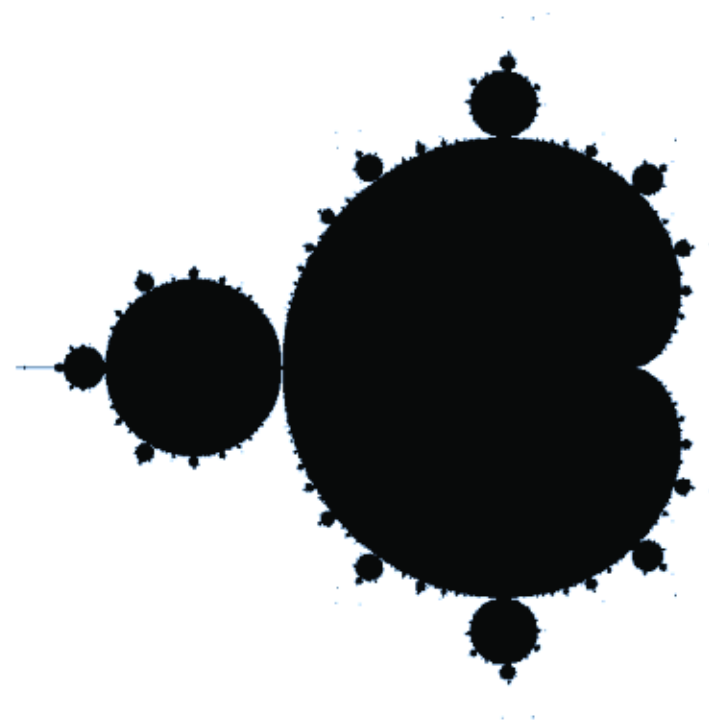
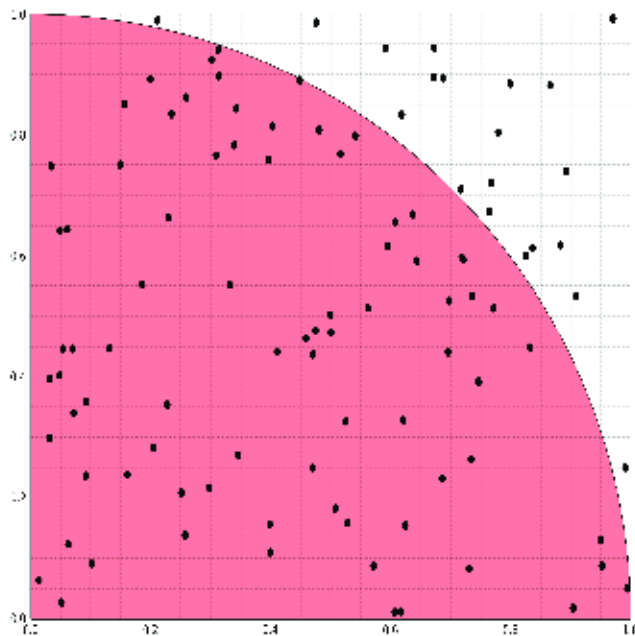
- ① si la tranche de temps qui lui a été allouée est terminée ;
- ② s'il exécute une instruction **yield()** ;
- ③ s'il exécute une méthode bloquante (**sleep()**, **join()**, **wait()**...)

mais ce n'est pas obligatoire...

En fait, c'est l'ordonnanceur de la JVM et le système d'exploitation qui répartissent l'accès et le retrait des threads aux processeurs, en prenant en compte l'état des threads, les priorités spécifiées et les instructions **yield()**.

Premiers exercices du TD/TP

En guise de premiers exercices, vous écrirez en TD et en TP une version parallèle d'un programme séquentiel, **en vue de l'accélérer**. Il s'agira donc en particulier de mesurer le **gain**.



Ces exercices simples illustrent quelques problèmes de synchronisation très courants.

Éléments de synchronisation

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Un problème de vue (à tester chez soi)

```
public class Exemple {  
    public static void main(String[] args) {  
        A a = new A();    // Création d'un objet "a" de la classe A  
        a.start();        // Lancement du thread "a"  
        a.fin = true;     // Fin de l'attente active du thread "a"  
        System.out.println ("Le_main_termine.");  
    }  
    static class A extends Thread {  
        public boolean fin = false;  
        public void run() {  
            while(! fin) ; // Attente active  
            System.out.println("L'objet_thread_a_terminé.");  
        }  
    }  
}
```



Que va afficher ce programme ? Pourquoi ?

Solution : le modificateur **volatile**

Nous corrigerons le programme en écrivant :

```
public volatile boolean fin = false;
```

Lorsqu'une variable est utilisée par plusieurs threads, le mot-clef **volatile** assure que chaque modification de la valeur de la variable est prise en compte immédiatement par tous les threads.

Plus précisément, le **modèle mémoire Java** (JSR-133) spécifie à quel moment les modifications effectuées par un thread seront « vues » par les autres threads.

✓ *Le mot-clef « volatile »*

☞ *Les verrous*

Vingt milliers d'incrémentations anarchiques par deux threads

```
public class Compteur extends Thread {  
    static volatile int valeur = 0;  
    public static void main(String[] args) throws Exception {  
        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++;  
    }  
}
```



Que va afficher ce programme ? Pourquoi ?

Résultats sur mon vieux MacBook (8 coeurs)

```
$ 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
```

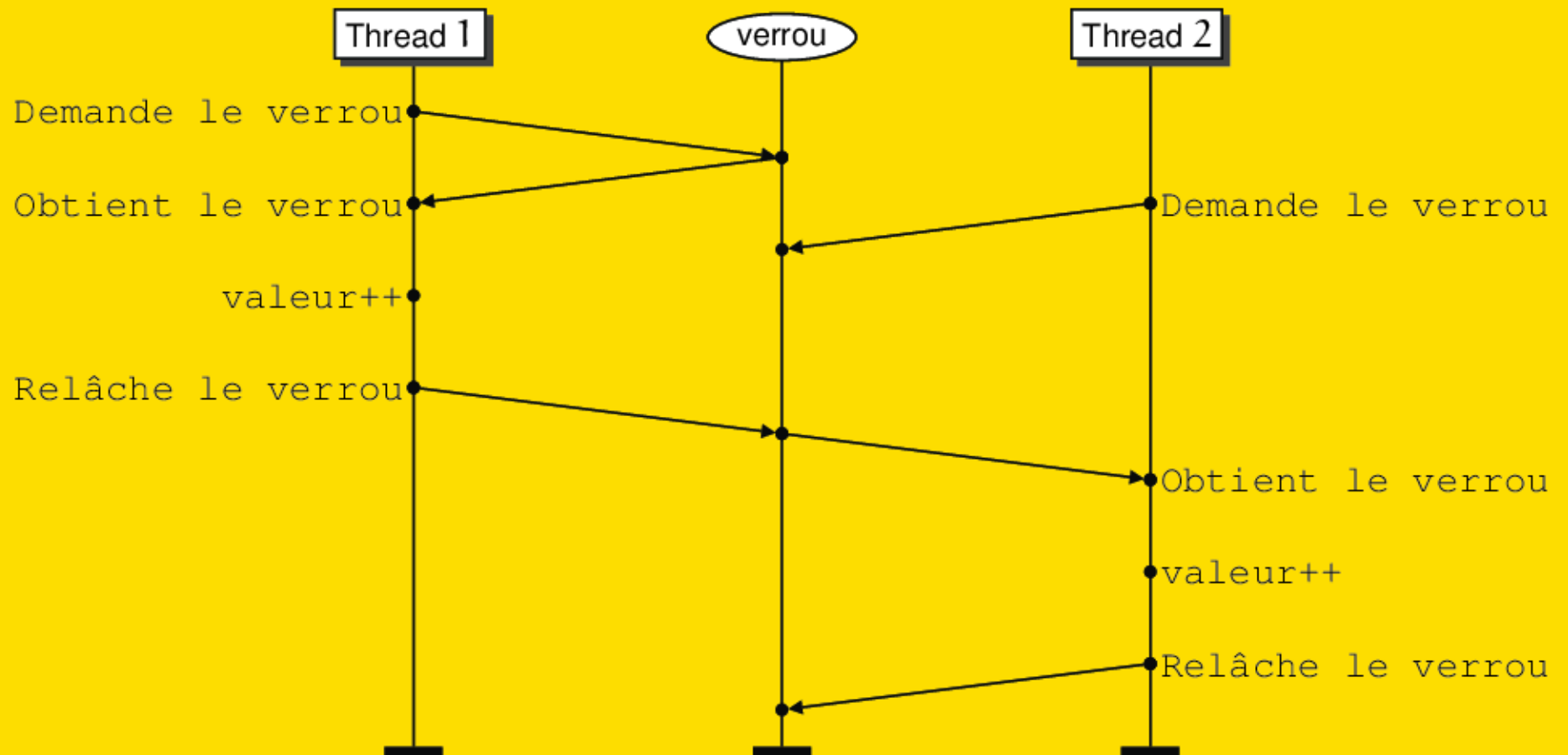
```
La valeur finale est 10000
```

```
$ java Compteur
```

```
La valeur finale est 19591
```

Premier problème de synchronisation

Lorsque deux threads peuvent accéder en même temps à un même objet (ou une variable partagée), il est *très souvent nécessaire* qu'ils ne le fassent qu'*un seul à la fois*. Les accès à l'objet (ou à la variable partagée) doivent alors être *totalement ordonnés*. L'outil le plus simple est alors le verrou.



Qu'est-ce qu'un verrou ?

Un **verrou** est une variable booléenne qui possède une **liste d'attente** et qui est manipulée uniquement par deux opérations « atomiques » : **Verrouiller(v)** et **Déverrouiller(v)**.

```
void Verrouiller(verrou v){ // Acquisition du verrou
                           // En anglais: "acquire" ou "lock"

    if (v)
        v = false ;           // Le verrou n'est plus libre
    else
        suspendre le processus et
        le placer dans la file d'attente associée à v;
}

void Déverrouiller(verrou v){ // Relâchement du verrou
                              // En anglais: "release" ou "unlock"

    if (la file associée à v != vide)
        débloquer un processus en attente dans la file associée à v
    else
        v=true ;              // Le verrou redevient libre
}
```

Les verrous intrinsèques et le mot-clef « **synchronized** »

En Java, chaque objet comporte en lui-même un verrou, appelé *verrou intrinsèque* (en anglais : « intrinsic lock »).

Il y a des verrous partout !

L'emploi de ces verrous est codé par un bloc **synchronized** :

```
synchronized (unObjet) { valeur++; }
```

L'exécution du bloc commence quand le *verrou intrinsèque* de **unObjet** est acquis.

Ce verrou est relâché lorsque l'exécution du bloc est terminée.

Correction du programme Compteur.java

```
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

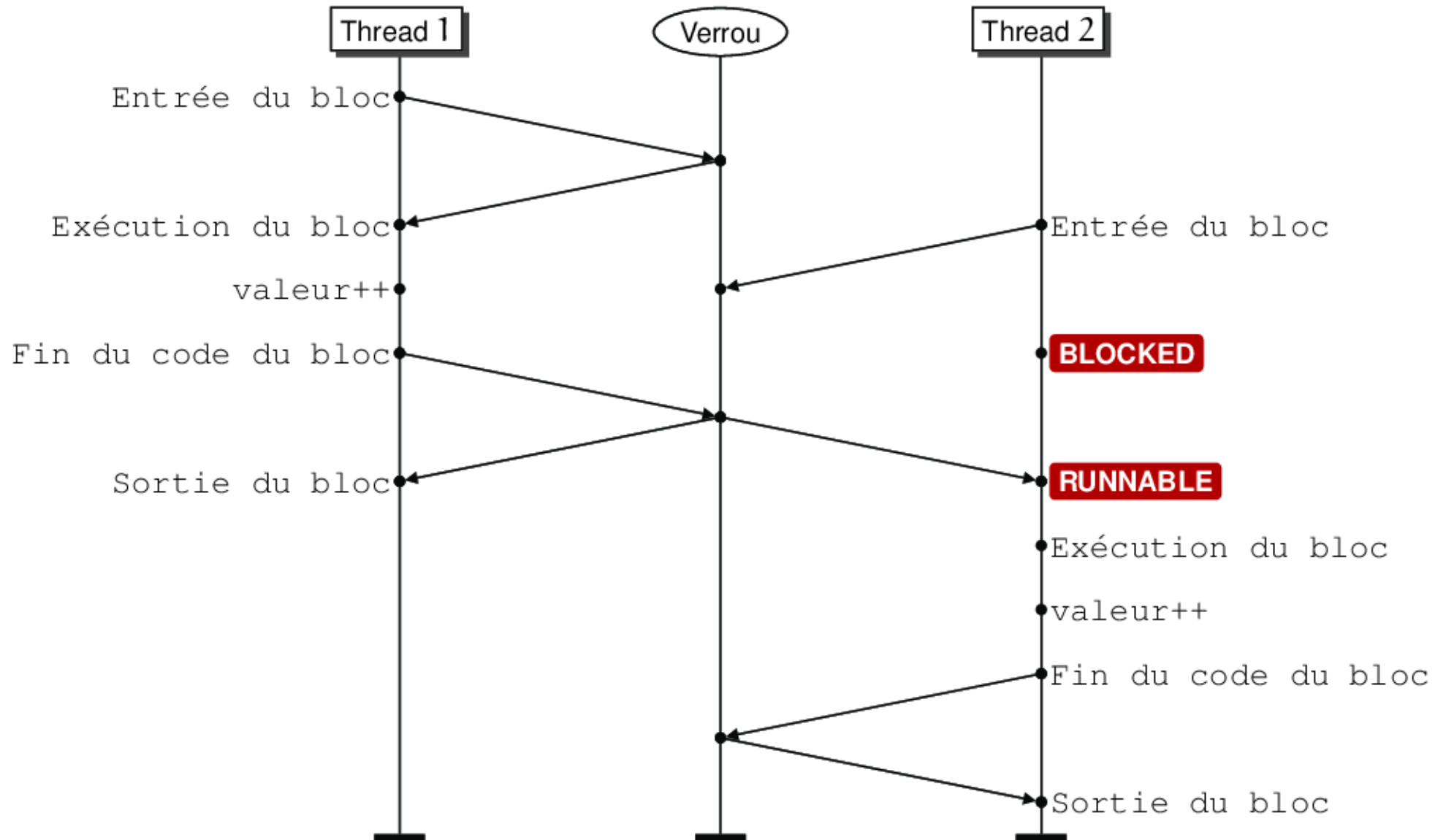
\$ java Compteur

La valeur finale est 20000

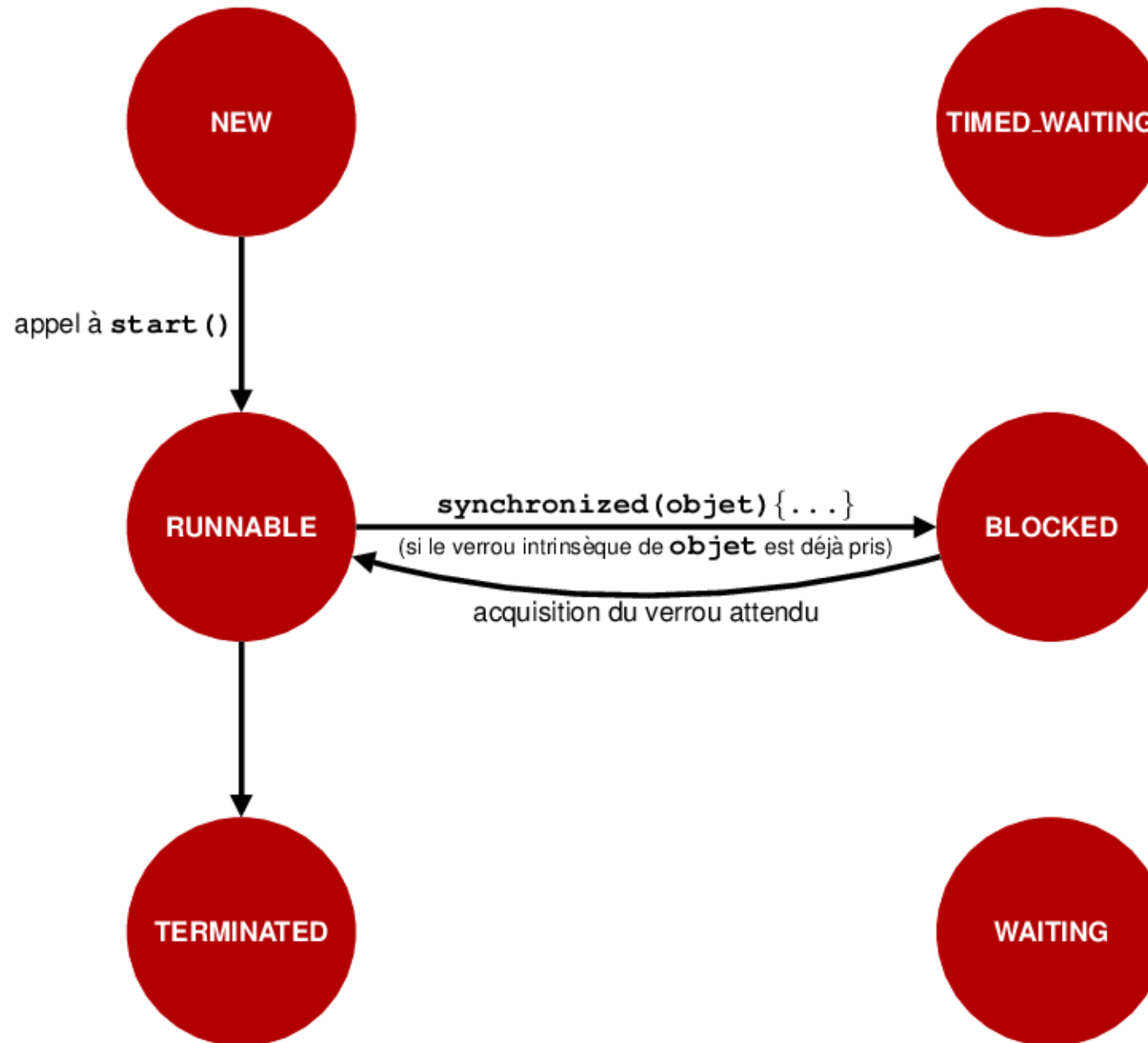
...

Il y a désormais « *exclusion mutuelle* » des accès à la variable **valeur** : les incrémentations sont dites « *synchronisées* ». Un seul thread peut incrémenter cette variable à chaque instant : celui qui possède le verrou.

Fonctionnement du bloc « synchronized »



Les six états d'un thread (suite)



Verrou intrinsèque d'un objet en Java

① Un seul thread peut posséder un verrou donné à un moment donné.

~> C'est le principe du verrou !

~> Les autres threads qui demandent le même verrou seront placés dans l'état d'attente **BLOCKED**.

② Il est impossible de seulement **tenter** d'acquérir un verrou intrinsèque.

~> C'est le principe du verrou : le thread prend le risque d'être bloqué.

③ Si un thread possède le verrou, il peut l'acquérir **à nouveau** !

Il s'agit donc d'un verrou « réentrant ! »

~> **Ça évite de se bloquer bêtement soi-même.**

~> Alors le verrou sera libéré (ou transmis à un autre thread) lorsqu'il aura été relâché **autant de fois** qu'il a été pris.

Modificateur **synchronized** d'une méthode

<pre>synchronized void inc() { valeur++; }</pre>		<pre>void inc() { synchronized(this) { valeur++; } }</pre>
---	--	---

Ces deux méthodes sont strictement équivalentes.

Le modificateur **synchronized** d'une méthode revient à exécuter le code de la méthode dans un bloc **synchronized(this) { ... }**.

Il y a donc :

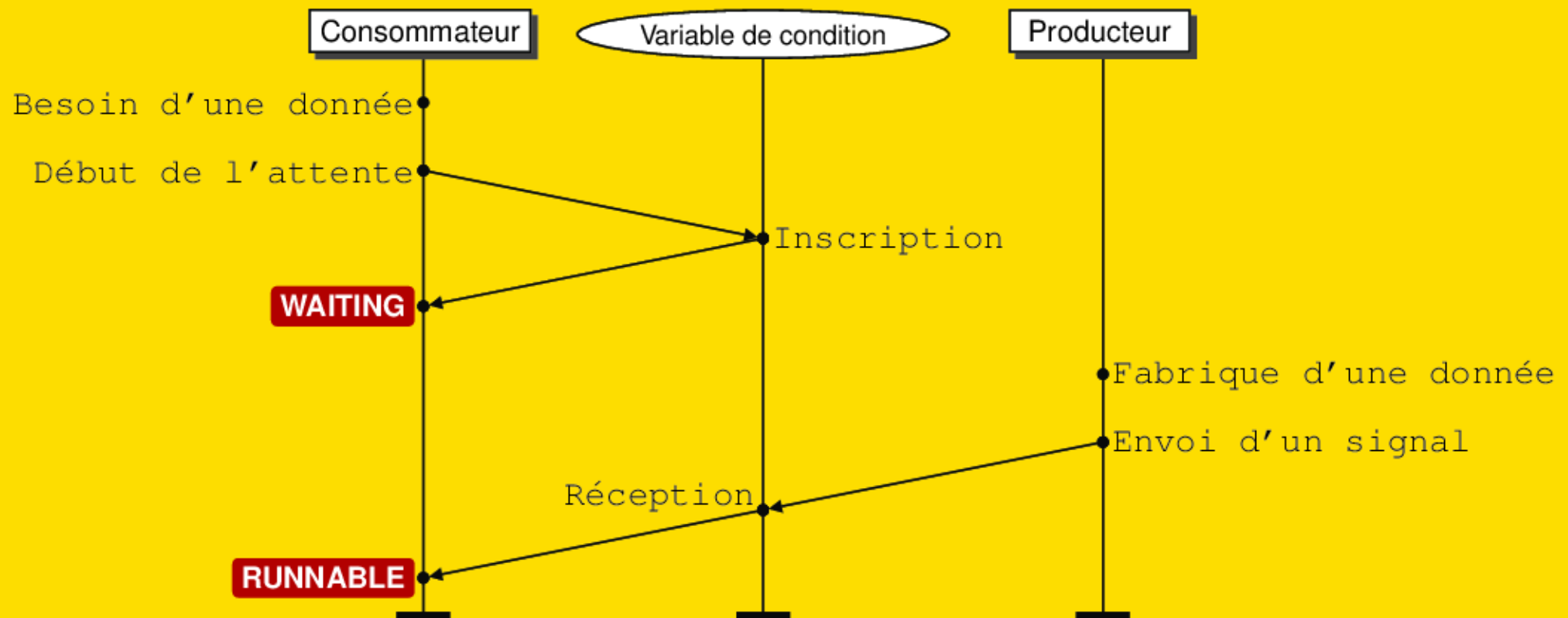
- acquisition du verrou de **this** au début de la méthode.
- relâchement du verrou de **this** à la fin de la méthode.

Pour être exact, une méthode déclarée **synchronized** requiert le verrou de **this**, c'est-à-dire de l'objet sur lequel est appliquée la méthode. En revanche une méthode *static* déclarée **synchronized** requiert elle le verrou de la classe correspondante (qui est aussi un objet).

- ✓ *Le mot-clef « volatile »*
- ✓ *Les verrous*
- ☞ *Synchronisation par signaux*

Second problème de synchronisation

Les threads ont souvent besoin de se coordonner, en particulier quand le résultat de l'un est utilisé par un autre. Ce dernier se place en *attente* de la réception d'un *signal*, à l'aide d'une « variable de condition » qu'il faut voir comme un lieu d'attente de signaux.



La notion de « variable de condition »

Une **variable de condition** est un moyen qui permet de *suspendre* un processus jusqu'à l'arrivée ultérieure d'un « signal. »

La valeur d'une variable de condition **cv** est constituée par l'ensemble (ou la file) des processus qui *attendent* un signal.

Deux opérations « atomiques » permettent d'accéder à cette valeur.

- ① La mise en attente d'un processus, c'est-à-dire l'ajout à la fin de la file est obtenue via l'opération **wait (cv)**.
- ② La réactivation d'un processus en attente est effectuée par l'opération **signal (cv)**.
Cette opération n'a aucun effet si la file est vide.

Attente et envoi d'un signal en Java

Chaque objet en Java comporte en lui-même une variable de condition. Il est possible d'appliquer les méthodes ci-dessous sur n'importe quel objet !

a.wait() *Suspend* le thread qui appelle cette méthode jusqu'à un appel à **a.notify()** ou **a.notifyAll()** par un *autre* thread.


Le thread attend donc sur un objet précis !

a.notify() Redémarre le thread qui a appelé **wait()** sur l'objet **a**. S'il y en a plusieurs, ce ne sera pas obligatoirement le premier ! S'il n'y en a aucun, **a.notify()** ne fait rien...

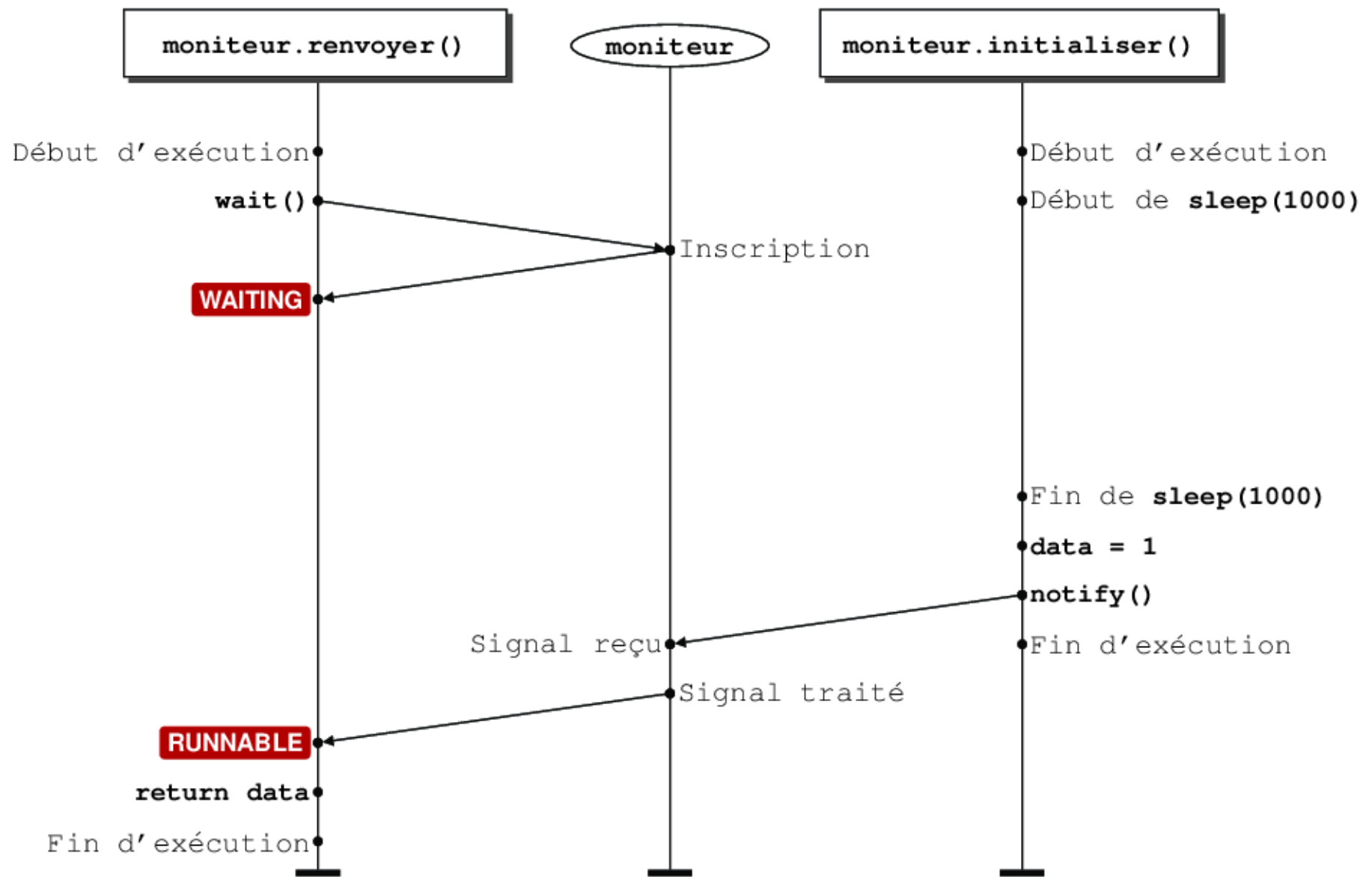
a.notifyAll() Redémarre tous les threads qui ont appelé **wait()** sur l'objet **a**.

Un exemple de moniteur qui protège une donnée

```
class Moniteur {  
    private volatile int data = 0;  
  
    synchronized int renvoyer() throws InterruptedException {  
        wait();  
        return data;  
    }  
  
    synchronized void initialiser() throws InterruptedException {  
        Thread.sleep(1000);  
        data = 1;  
        notify();  
    }  
}
```

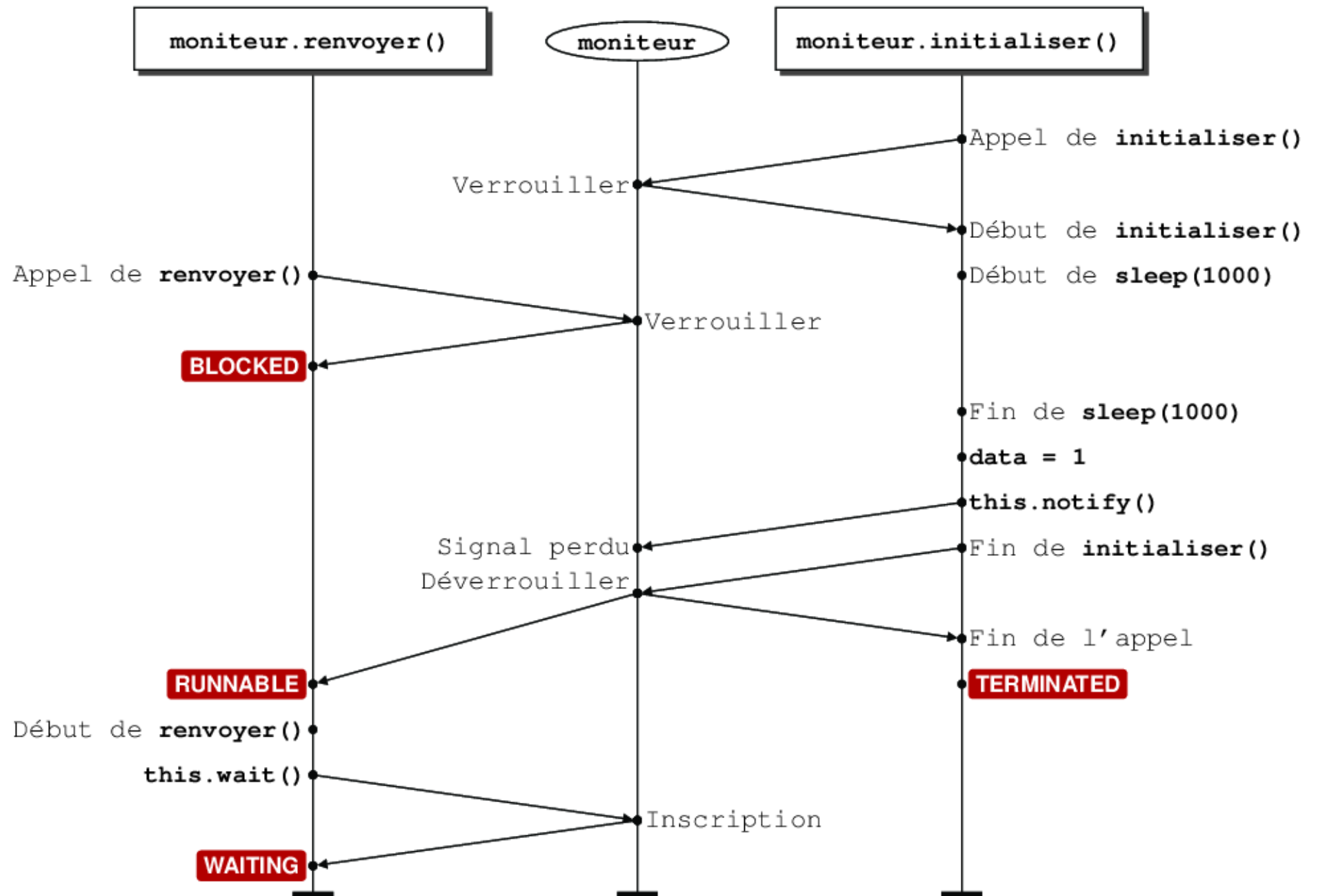


Principe approximatif de l'exécution

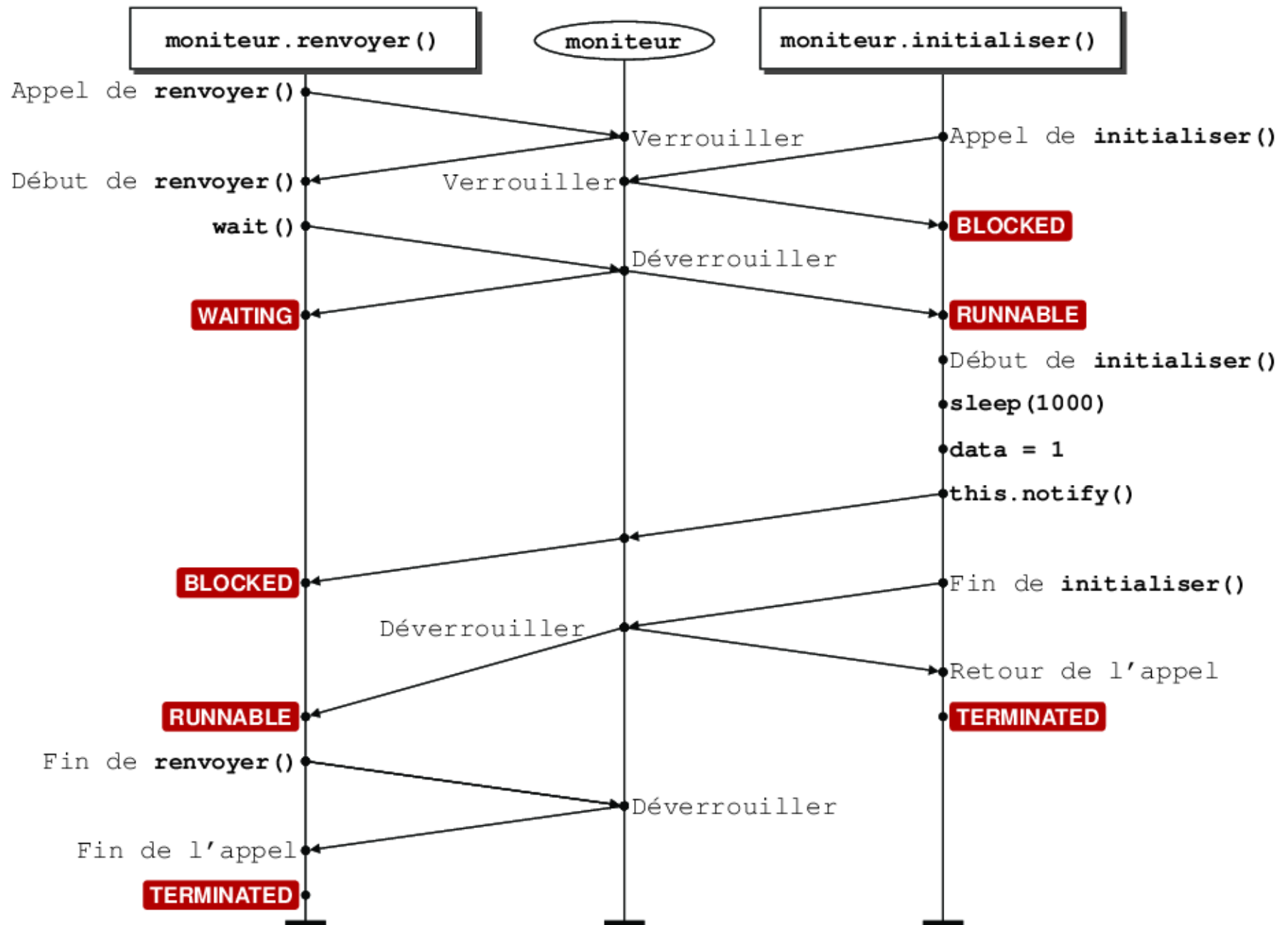


Ce scenario est-il vraiment envisageable ?

Ce système peut bloquer !



Ce système termine parfois ! Comment ?



Les méthodes `wait()` et `notify()` requièrent `synchronized()`

Les méthodes `wait()` et `notify()` doivent obligatoirement être appelées à l'intérieur d'un bloc (ou d'une méthode) `synchronized` appliqué au même objet `a`.

`a.wait()`

- ↪ inscrit le thread courant sur la liste d'attente de la variable de condition de `a` ;
- ↪ suspend le thread courant, qui entre alors dans l'état **WAITING** ;
- ↪ *relâche le verrou intrinsèque* de `a`.

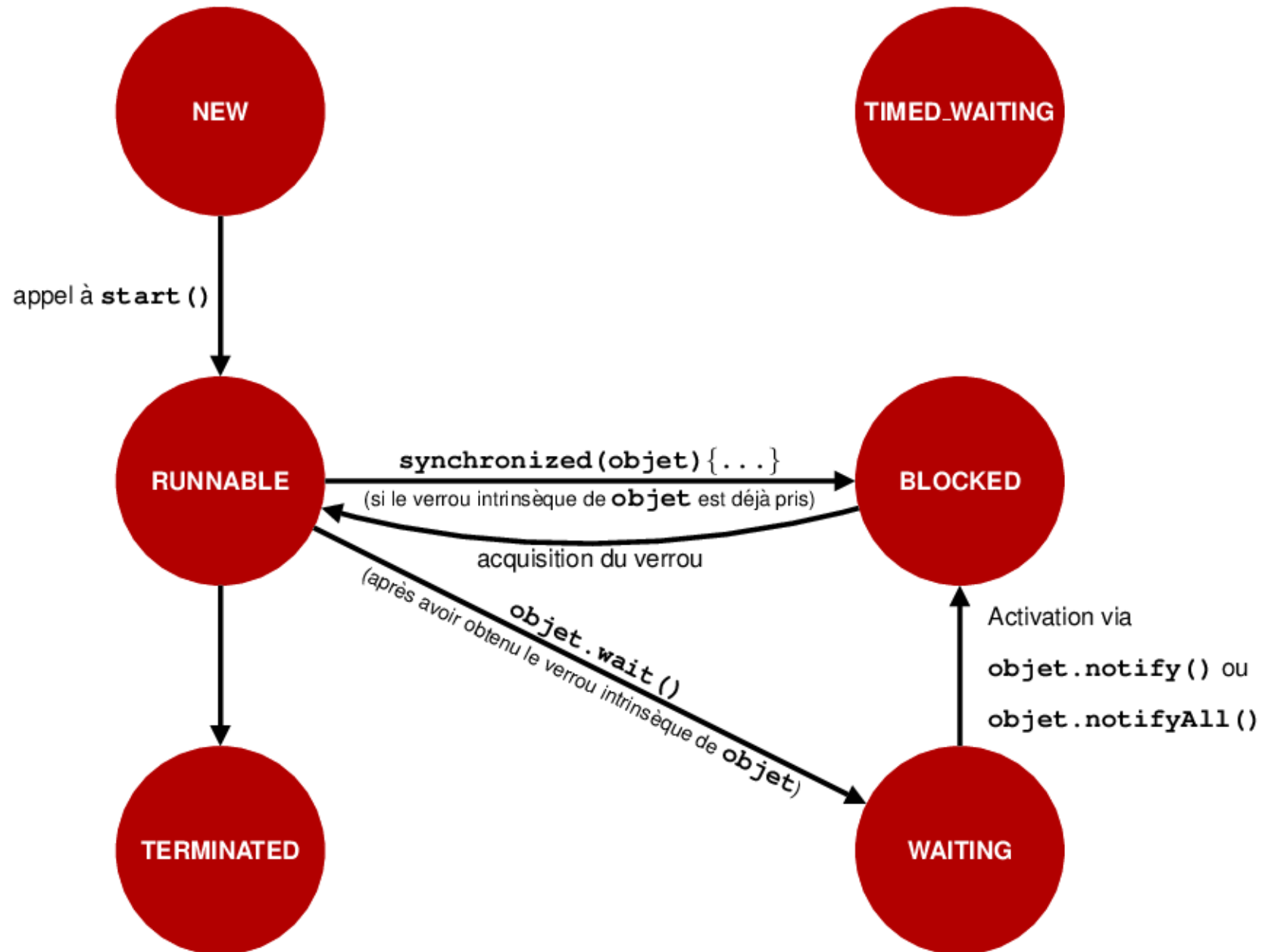
`a.notify()`

- ↪ *garde pour lui le verrou intrinsèque* de `a` ;
- ↪ relance un thread placé sur la liste d'attente de `a` s'il y en a un, en le retirant de la liste d'attente. Ce thread passe de l'état **WAITING** à l'état **BLOCKED** et doit récupérer le verrou de l'objet `a` avant de retourner dans l'état **RUNNABLE**.

Il n'y a aucun contrôle sur le thread libéré !

`a.notifyAll()` relance tous les threads de la liste d'attente.

Les six états d'un thread (fin)



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

Causes current thread to wait until either another thread invokes the **`notify()`** method or the **`notifyAll()`** method *for this object*, or a specified amount of time has elapsed.

The current thread must own this object's monitor.

This method causes the current thread (call it `T`) to place itself in the wait set for this object and then to **relinquish any and all synchronization claims** on this object. Thread `T` becomes disabled for thread scheduling purposes and lies dormant until one of four things happens :

- ① Some other thread invokes the `notify()` method for this object and thread `T` happens to be arbitrarily chosen as the thread to be awakened.
- ② Some other thread invokes the `notifyAll()` method for this object.
- ③ Some other thread *interrupts* thread `T`.
- ④ The specified amount of real time has elapsed, more or less. If timeout is zero, however, then real time is not taken into consideration and the thread simply waits until notified.

`wait(0)`, c'est simplement `wait()`

The thread `T` is then removed from the wait set for this object and re-enabled for thread scheduling.

It then competes in the usual manner with other threads for the right to synchronize on the object ; once it has gained control of the object, all its synchronization claims on the object are restored to the status quo ante - that is, to the situation as of the time that the `wait` method was invoked.

Thread `T` then returns from the invocation of the `wait()` method.

Thus, on return from the `wait()` method, the synchronization state of the object and of thread `T` is exactly as it was when the `wait()` method was invoked.


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

...

Note that the `wait()` method, as it places the current thread into the wait set for this object, unlocks only this object ; **any other objects on which the current thread may be synchronized remain locked** while the thread waits.

`wait()` relâche seulement le verrou intrinsèque de l'objet sur lequel le thread patiente.

Ce qu'il faut retenir

Les priorités des threads et la méthode **yield()** ne servent a priori à rien, pour commencer.

Les variables susceptibles d'être accédées par plusieurs threads doivent *a priori* être déclarées **volatile** par précaution.

Les *verrous* associés aux objets en Java sont un outil fondamental pour écrire un programme correct en Java. La syntaxe de **synchronized** assure que chaque verrou pris sera relâché (à la fin du bloc).

sleep() permet de faire une pause *un temps déterminé*.

wait() permet à un thread *d'attendre sur un objet* jusqu'à ce qu'un autre thread lui lance un *signal*, via un appel à **notify()** ou **notifyAll()** sur cet objet.

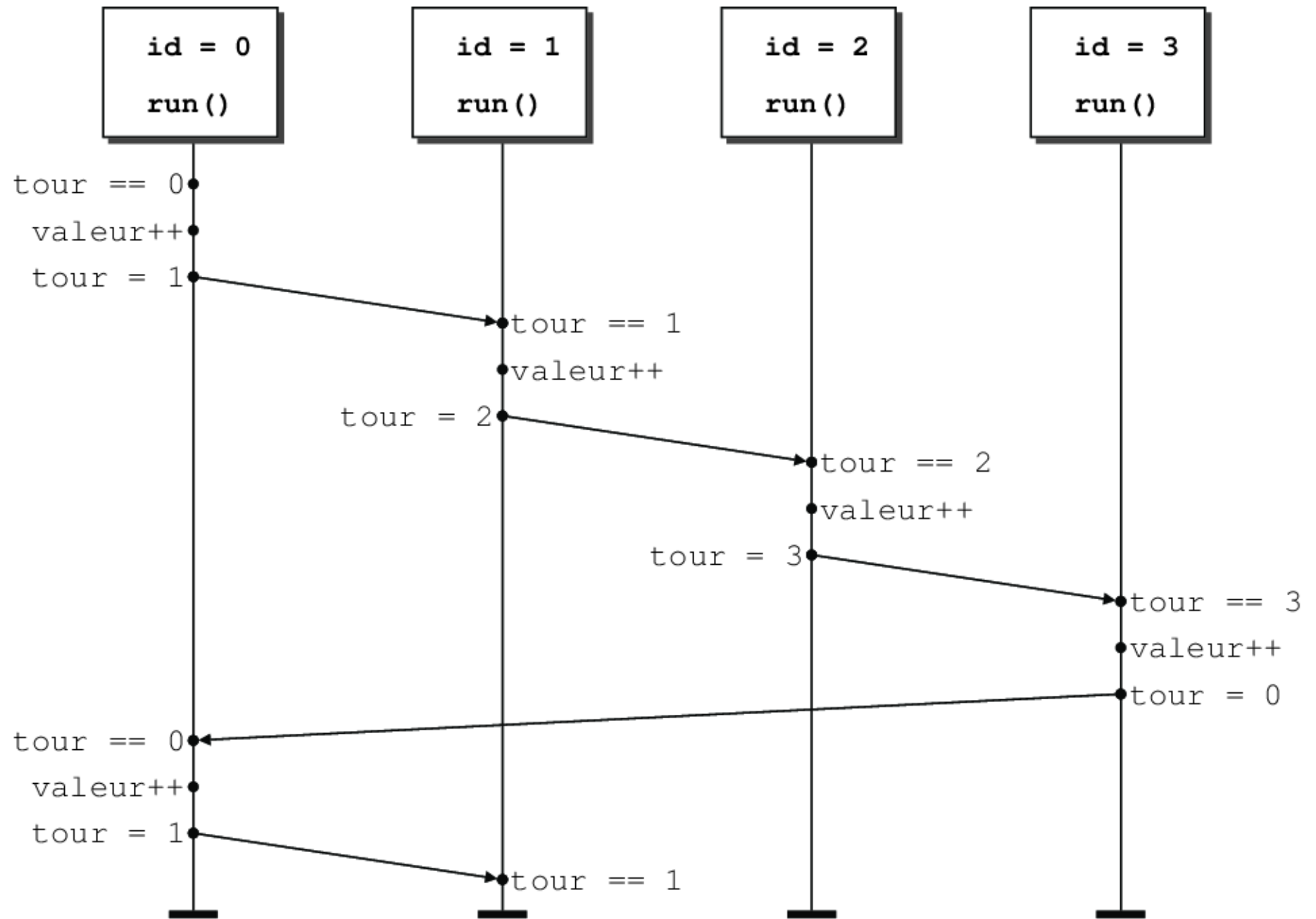
La méthode **wait()** nécessite d'acquérir au préalable le verrou intrinsèque de l'objet sur lequel elle est appliquée, à l'aide de **synchronized**. *Mais ce verrou est alors relâché !*

Les méthodes **notify()** et **notifyAll()** nécessitent également au préalable **synchronized**, mais *ces méthodes ne relâchent pas le verrou !*

Attente active vs attente passive

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Le benchmark adopté : les compteurs en rond



Le benchmark adopté : attente active

```
final int id; // Identité de chaque thread
static volatile int valeur = 0; // La variable à incrémenter
static volatile int tour = 0; // Tour de rôle circulant

static final int valeur_finale = 840; // Nombre d'incrémentations
static int nombre_de_compteurs; // Nombre de threads utilisés
static final int part = valeur_finale / nombre_de_compteurs ;
...
public void run() {
    for (int i = 1; i <= part; i++) {
        while(tour != id); // Attente active du tour
        valeur++;
        tour = (tour+1) % nombre_de_compteurs;
    }
}
```

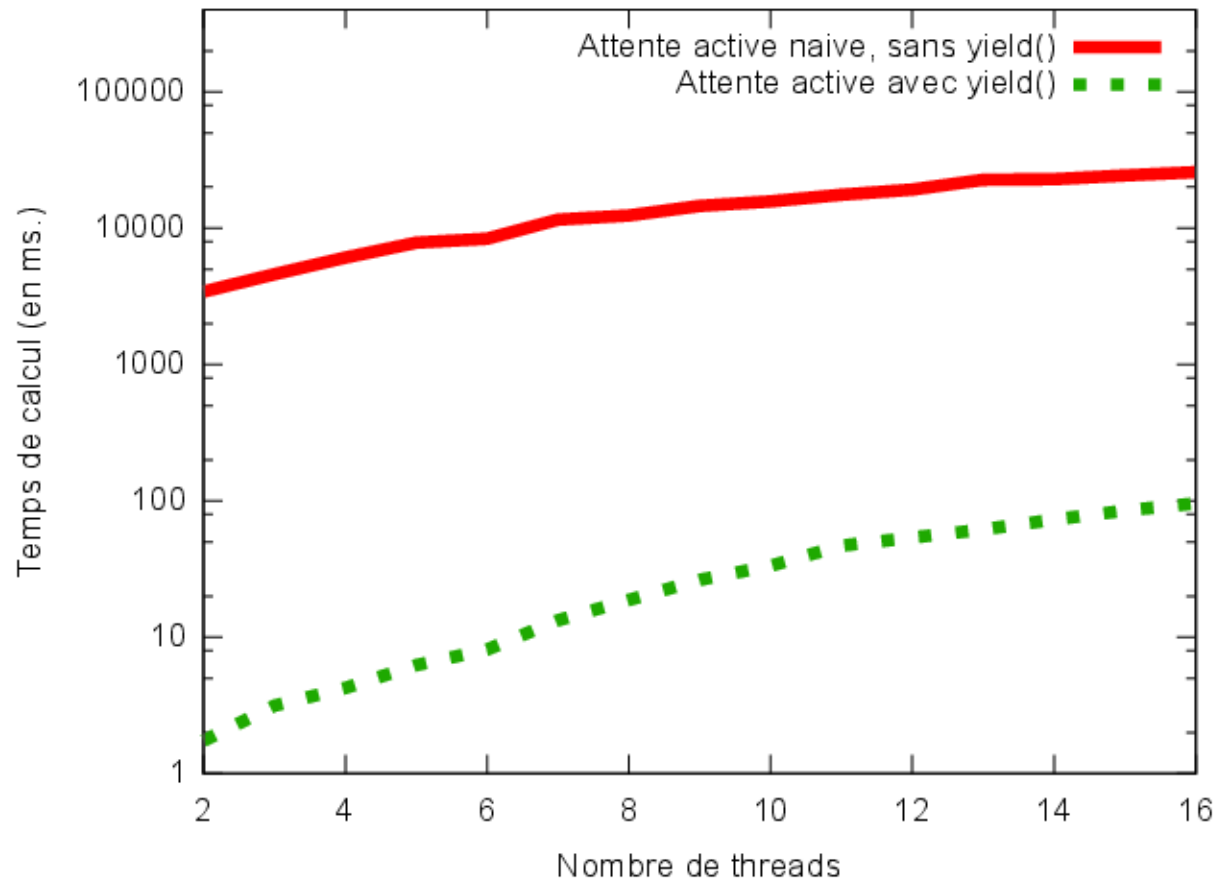
Intérêt de l'instruction `yield()` sur un monoprocesseur

Si l'on choisit d'ajouter l'instruction **`yield()`**, c'est-à-dire d'attendre son tour via l'instruction

```
while(tour!=id) yield();
```

alors le thread en attente active peut relâcher le processeur afin de ne pas gâcher la tranche de temps qui lui est allouée alors que ce n'est pas son tour de travailler.

Intérêt de `yield()` sur un monoprocesseur



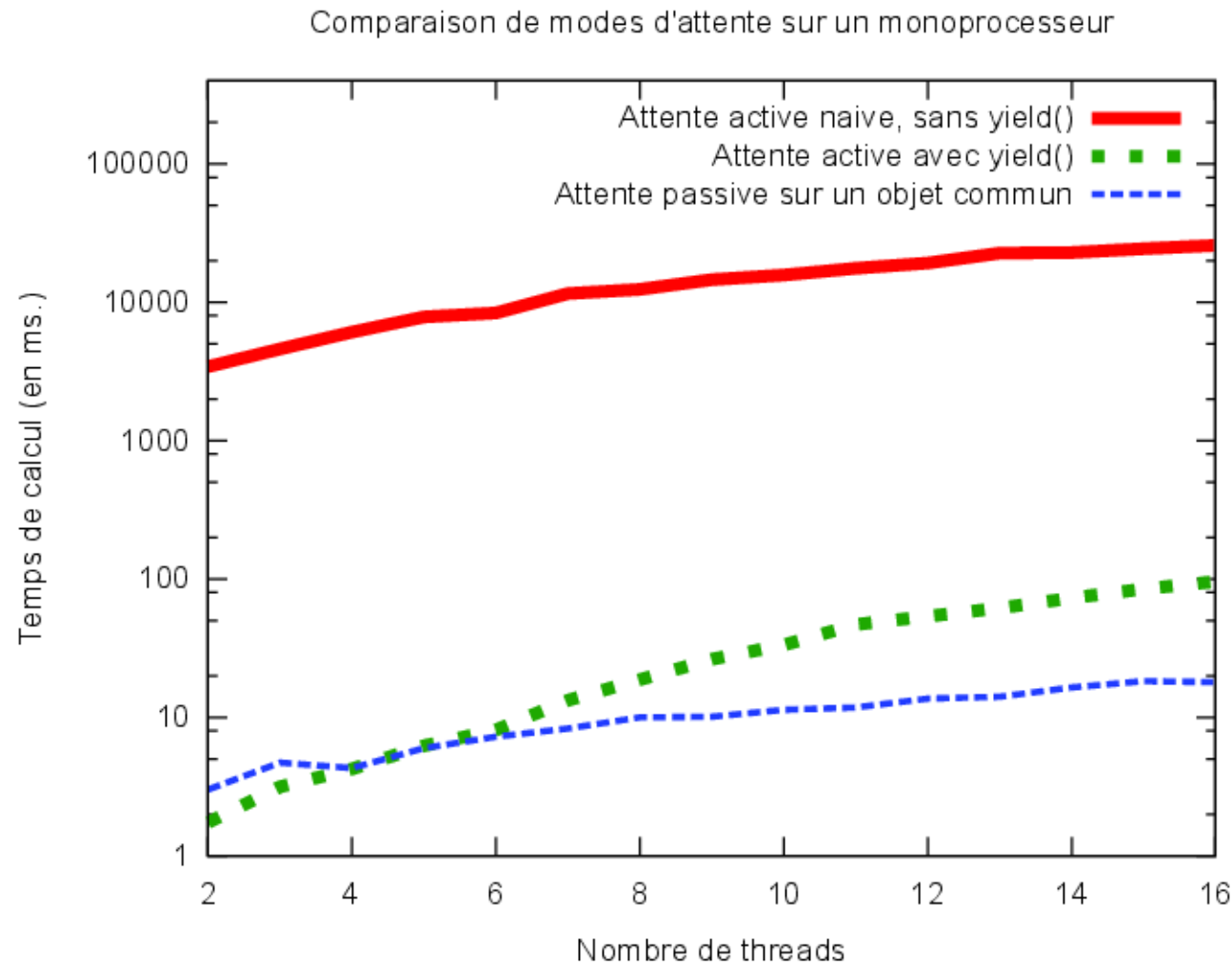
Alternative à l'attente active : l'attente passive

La méthode `run()` alternative place les threads en *attente passive (sur une variable de condition)* via un appel à `wait()` sur un objet `rendezvous` partagé.

```
static private Object rendezvous = new Object();  
...  
public void run() {  
    for (int i = 1; i <= part; i++) {  
        synchronized(rendezvous) {  
            while(tour!=id) rendezvous.wait();    // Attente passive  
            valeur++;  
            tour = (tour+1) % nombre_de_compteurs;  
            rendezvous.notifyAll();  
        }  
    }  
}
```

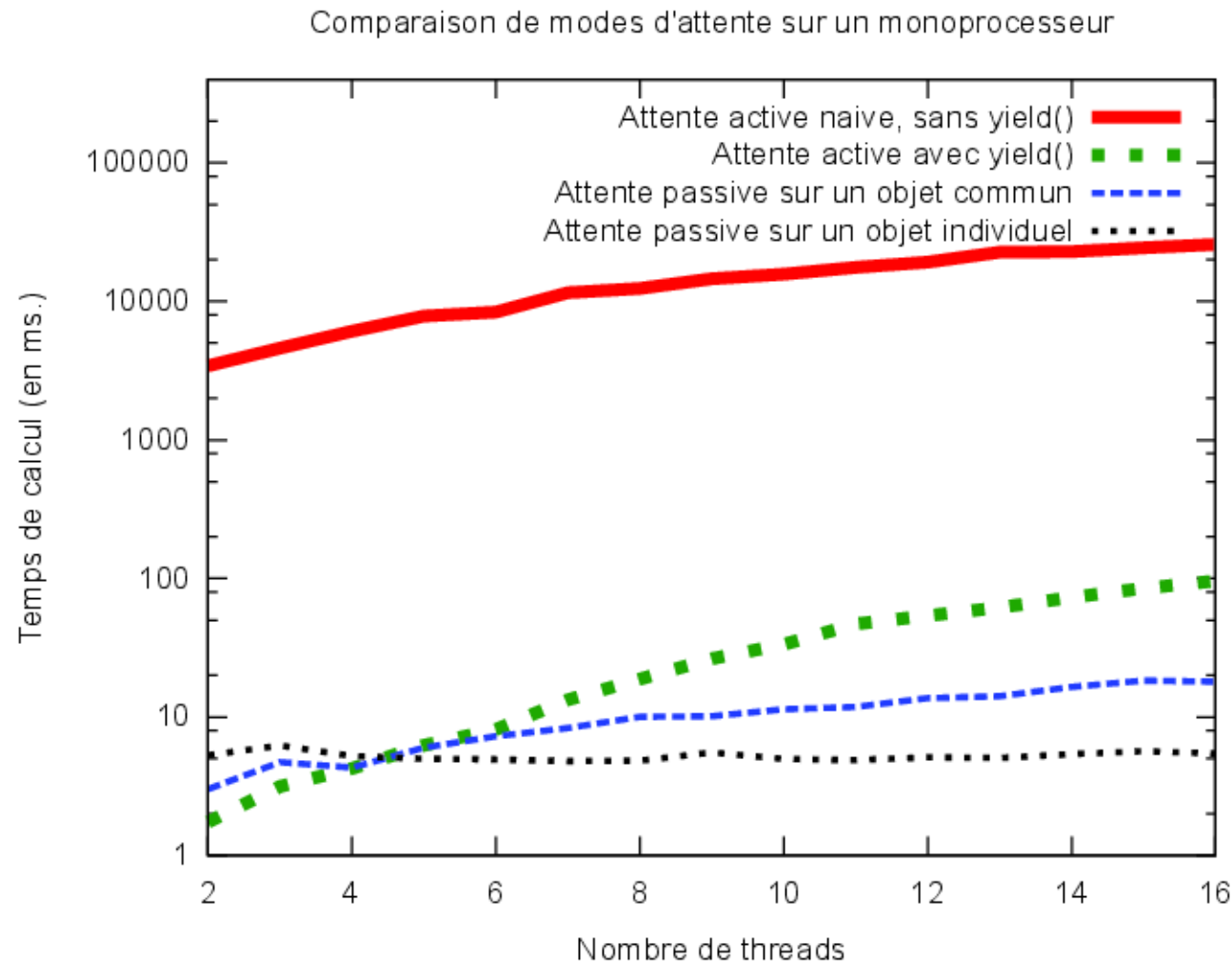
Pourquoi faut-il réveiller tout le monde ?

Intérêt de l'attente passive sur un monoprocesseur



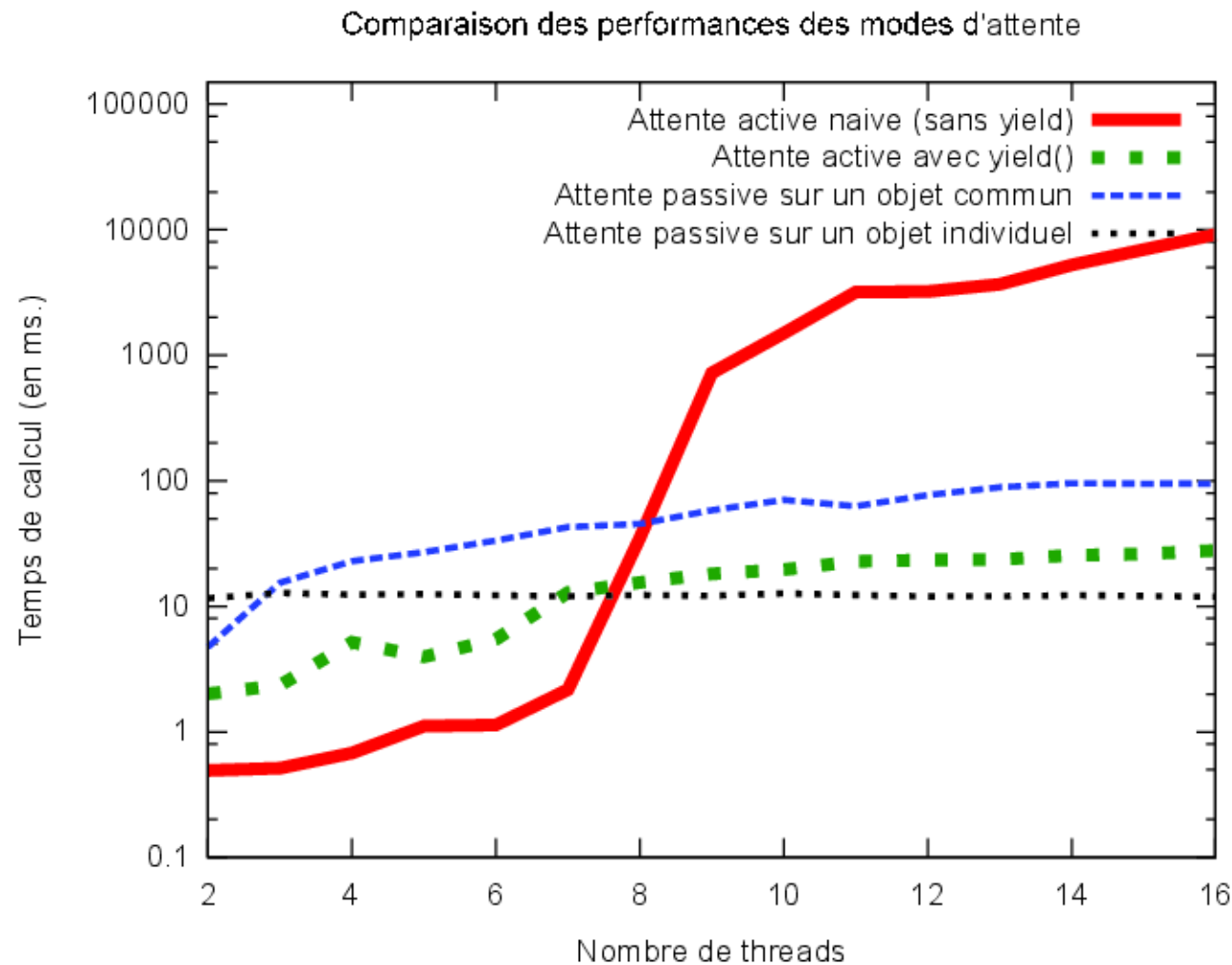
S'il y a encombrement de threads sur un processeur, l'attente passive apparaît plus efficace que l'attente active, même avec **yield()**.

Réduction de nombre de signaux et de réveils



Pour réduire le nombre de signaux envoyés à chaque phase, et ne réveiller que le thread dont c'est le tour, il faudrait appliquer **wait()** et **notify()** sur un objet propre à chaque thread. On observe alors sans surprise un gain de temps et, signe de robustesse, une courbe plane.

Résultats sur une machine à 8 coeurs (avec plus d'incrémentations)



Tant que le nombre de threads est inférieur au nombre de coeurs, l'attente active naïve (c'est-à-dire sans **yield()**) apparaît la plus performante. Au-delà, l'envoi d'un seul signal sur un objet individuel demeure l'implémentation la plus efficace et la plus robuste.

Conclusions

Dans le cas d'un monoprocesseur, l'attente active est proscrite :

- Le processus attend qu'une condition soit satisfaite ;
 - Le seul processus actif ne fait rien : il attend ;
 - Aucune modification n'est effectuée sur les données.
- ~> La tranche de temps allouée est donc purement gaspillée !
- ~> Il faut, au minimum, susciter le relâchement par **yield()**.

C'était une règle générale il y a quelques années !

En revanche, dans un environnement multiprocesseur, l'attente active peut être efficace

- si le temps d'attente est moindre qu'un changement de contexte ;
- ou s'il n'y a pas d'autres threads actifs sur le processeur.

Cependant, au-delà des performances, programmer avec attente active est en général techniquement risqué, car cela soulève des questions subtiles d'atomicité.

Interruptions et exceptions

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Arrêt d'un thread

Un thread se termine normalement lorsqu'il a terminé d'exécuter sa méthode `run ()` . Il faut donc obliger le thread à terminer proprement cette méthode.

Pour cela, les interruptions peuvent aider !

Le *statut d'interruption* est un booléen attaché à chaque thread et distinct de son *état* ; un thread dans l'état **RUNNABLE** peut voir son propre statut d'interruption levé, ou non.

La méthode `interrupt ()` appelée sur un thread `t` a pour simple effet de lever le *statut d'interruption* du thread `t`.

Si ce thread consulte ce statut périodiquement, il peut alors détecter qu'il lui est demandé de s'arrêter de lui-même, proprement. En outre, si le thread interrompu est dans un état d'attente, une exception du type `InterruptedException` sera soulevée.

Alternatives à oublier

Il existe d'autres moyens pour stopper, suspendre ou redémarrer un thread.

Néanmoins, les méthodes **stop()**, **suspend()**, **resume()** sont **dépréciées**, car elles risquent de laisser le programme dans un « sale » état !

La méthode **destroy()** n'est plus implémentée : sa spécification est trop brutale : il faut l'oublier aussi.

Exemple peu lisible (comme promis) malgré une assez bonne indentation

```
Thread[] mesThreads = new Thread[5];
for(int i=0; i<mesThreads.length; i++) {
    final int id = i;
    Thread t = new Thread( new Runnable() { public void run() {
        for(int j=0 ; !Thread.interrupted() ; j++) {
            System.out.println(j + "ième_exécution_de_" + id);
        }
        System.out.println("Fin_d'exécution_du_code_" + id);
        System.out.println(Thread.isInterrupted());
    }
    });
    mesThreads[i] = t;
    t.start();
}
Thread.sleep(10);
for(int i=0; i<mesThreads.length; i++) mesThreads[i].interrupt();
```

Exemple : exécution

53ième exécution de 3

48ième exécution de 0

Fin d'exécution_du_code_0

false

1ième_exécution_de_2

Fin_d'exécution du code 2

false

1ième exécution de 4

Fin d'exécution_du_code_4

false

0ième_exécution_de_1

Fin_d'exécution du code 1

false

54ième exécution de 3

Fin d'exécution_du_code_3

false

Consulter le statut d'interruption

Le statut d'interruption ne peut être consulté que par les méthodes `interrupted()` et `isInterrupted()`.

```
public static boolean interrupted()
```

retourne **true** si le statut du thread sur lequel est appelée la méthode a été positionné.
Si tel est le cas, *réinitialise* ce statut à **false**.

```
public boolean isInterrupted()
```

retourne **true** si le statut du thread sur lequel est appelée la méthode a été positionné ;
mais ne modifie pas la valeur du statut d'interruption.

✓ *Statut d'interruption*

☞ *Les exceptions*

Rappel sur les exceptions

Les exceptions correspondent au mécanisme de gestion des erreurs du langage Java.

Ces erreurs sont représentées par des objets manipulés par trois mots clés qui permettent de détecter et de traiter ces erreurs : **try**, **catch** et **finally**.

Il est également possible de créer ou de propager des exceptions à l'aide des mots-clefs **throw** ou **throws**.

Lors de la détection d'une erreur, un objet qui hérite de la classe **Exception** est créé : on dit alors qu'une exception est **levée**. Cet objet est propagé à travers la pile d'exécution jusqu'à ce qu'il soit traité.

Exemples d'exceptions

sleep(délai) lance une **InterruptedException** si le thread est déjà interrompu, ou s'il est interrompu par un autre thread pendant le laps de temps passé dans l'état **TIMED_WAITING**.

wait() lance également une **InterruptedException** si le thread qui l'applique est *interrompu* au moment de l'appel, ou s'il est interrompu par un autre thread pendant le laps de temps passé dans l'état **WAITING**.

join() et ses variantes lancent aussi une **InterruptedException** si le thread sur lequel elles sont appliquées est *interrompu* au moment de l'appel, ou ultérieurement.

N.B. Dans tous les cas, *le statut est réinitialisé lorsque l'exception est levée.*

Mauvaise pratique

S'il n'y a aucune instruction `interrupt()` dans le code, il n'y aura aucune `InterruptedException`.

On pourrait alors être tenté d'ignorer cette exception et écrire :

```
try {  
    Thread.sleep(1000);  
} catch (InterruptedException ignorée) {}
```



C'est là une très mauvaise pratique qu'il faut proscrire !

Vous écrirez donc par exemple :

```
try {  
    Thread.sleep(1000);  
} catch (InterruptedException e) {e.printStackTrace();}
```

pour afficher l'origine de l'erreur qui ne se produira jamais.