

Introduction à la programmation parallèle

Master Informatique — Semestre 2 — UE obligatoire de 3 crédits

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 au parallélisme ?

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

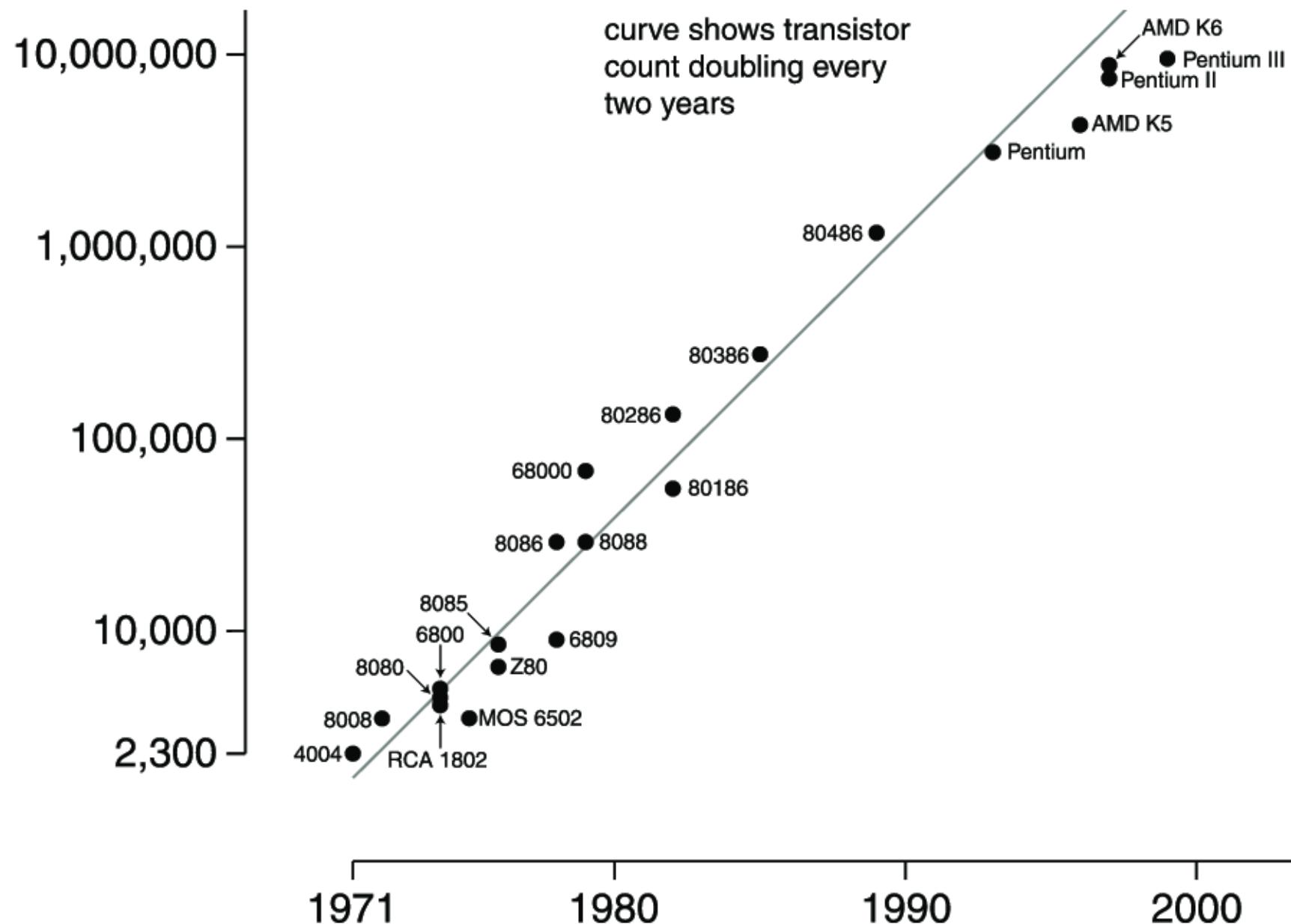
La loi de Moore

La **loi de Moore** a été formulée en 1965 par Gordon Moore (un des trois fondateurs d'Intel) dans la revue « Electronics Magazine ». Constatant que la complexité des semiconducteurs proposés en entrée de gamme *doublait tous les ans* à coût constant depuis 1959, date de leur invention, elle postulait la poursuite de cette croissance. En 1965, le circuit le plus performant comportait 64 transistors...

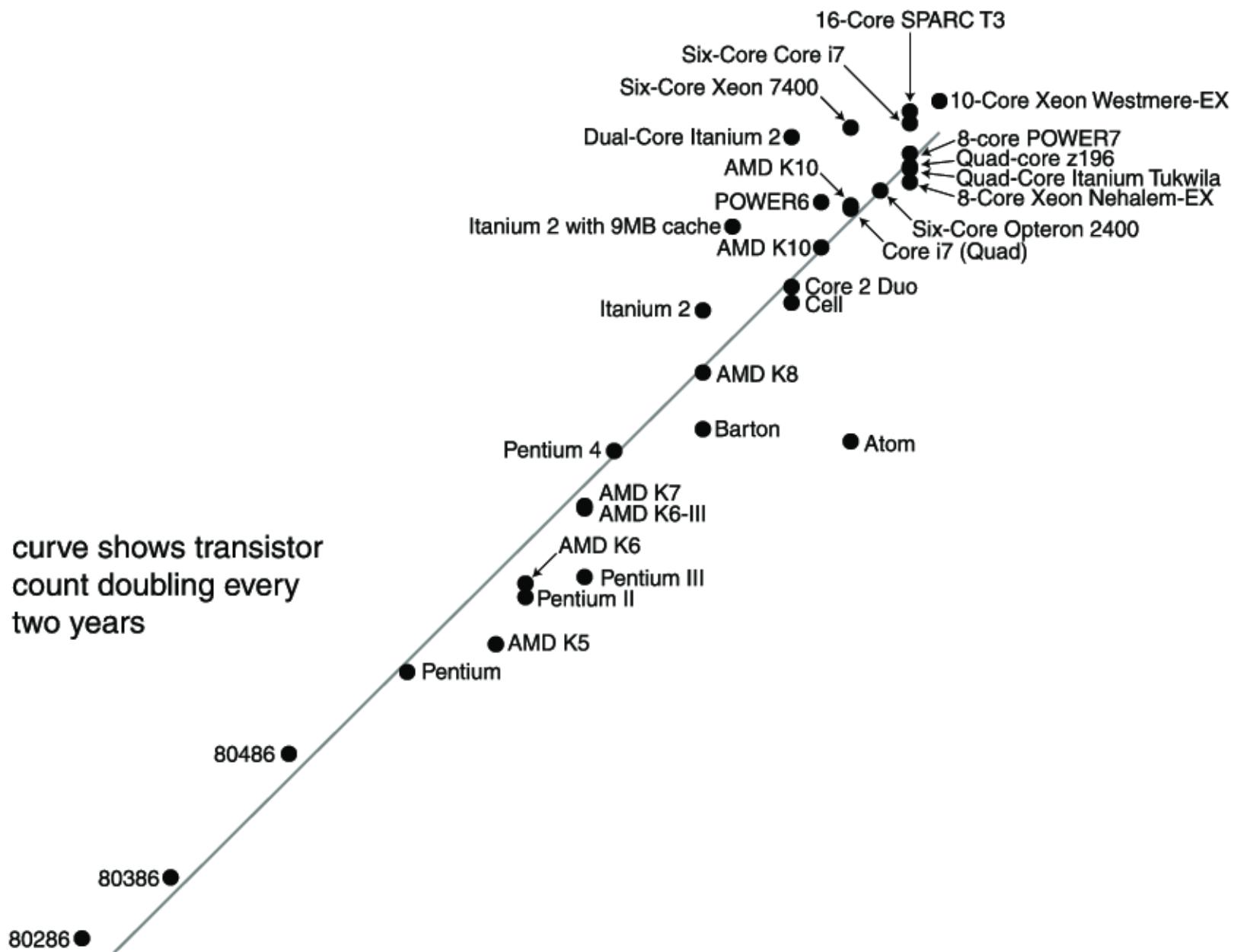
En 1975, Moore réévalua sa prédiction en posant que *le nombre de transistors* sur une puce de silicium *double tous les deux ans*.

Mais c'est David House, un collègue de Moore chez Intel, qui réduit l'intervalle en estimant que la *performance* des microprocesseurs devrait *doubler tous les 18 mois*, du fait du nombre de transistors et de la fréquence d'horloge croissants.

La loi de Moore



La loi de Moore



La loi de Moore

L'industrie informatique s'est développée depuis 40 ans en s'appuyant sur la loi de Moore, qui incite à changer d'ordinateur tous les deux ans.

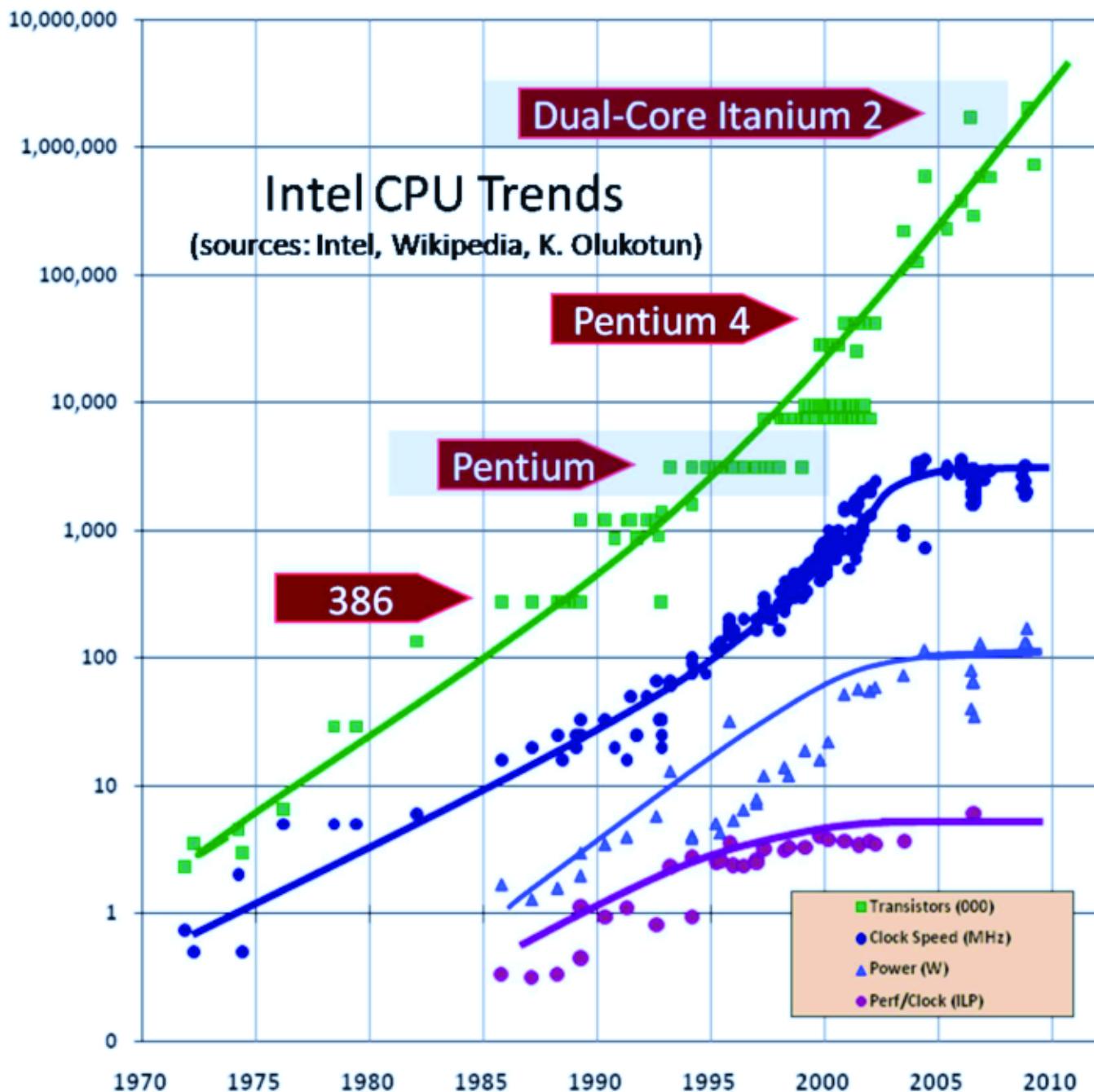
En 2004 la perspective de poursuivre la progression des performances selon le modèle classique du monoprocesseur est abandonnée par Intel.

« 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 maintenue que par la multiplication du nombre de coeurs.

Fréquence d'horloge des microprocesseurs



L'ordinateur le plus puissant au monde (Juin 2016)

Le supercalculateur chinois Sunway TaihuLight affiche une puissance de calcul record de 93 pétaFLOPS, ce qui représente 93×10^{15} (93 millions de milliards) opérations par seconde.

Il rassemble 10,65 millions de cœurs fonctionnant en parallèle...

✓ *Un peu de motivation*

👉 *Un peu d'algorithmique parallèle*

Calcul du maximum d'un tableau (**pseudo-code**)

```
FONCTION max( s: tableau{integer}; i, j: nat ): integer
DEBUT
    max <- s[i]
    POUR k <- i+1 A j FAIRE
        SI s[k] > max ALORS
            max <- s[k]
        FIN
    RETOURNER (max)
FIN
```

Que calcule cette fonction ?

Quel sera le temps d'exécution pour un tableau s de longueur $n = j - i + 1$?

Plus précisément, quelle est la complexité asymptotique de cet algorithme ?

Question : Est-il possible de faire mieux, c'est-à-dire d'obtenir un temps d'exécution qui soit inférieur à n pour une séquence de longueur n ?

Solution alternative par dichotomie

```
FONCTION max( s: tableau{integer}; i, j: nat ): integer
DEBUT
    SI i = j ALORS RETOURNER( s[i] )
    SINON
        m <- (i+j) / 2
        max1 <- max( s[i..m] )
        max2 <- max( s[m+1..j] )
        SI max1 > max2 ALORS
            RETOURNER( max1 )
        SINON
            RETOURNER( max2 )
        FIN
    FIN
FIN
```

Question : Quel sera le temps d'exécution pour un tableau s de longueur n ?

Et si les appels de fonction se déroulent en parallèle ?

```
FONCTION max( s: tableau{integer}; i, j: nat ): integer
DEBUT
    SI i = j ALORS
        RETOURNER( s[i] )
    SINON
        m <- (i+j) / 2
        EN PARALLELE
            max1 <- max(s[i..m]) || max2 <- max(s[m+1..j])
        SI max1 > max2 ALORS
            RETOURNER( max1 )
        SINON
            RETOURNER( max2 )
        FIN
    FIN
FIN
```

Question : Quel sera alors le temps d'exécution en fonction de n ?

Autre problème : trouver la fin d'une liste chaînée

Considérons une liste chaînée de longueur n .



Supposons que chaque nœud de la liste soit sur un processeur différent.

Supposons que chaque nœud puisse aussi avoir un pointeur auxiliaire « voisin » en plus du pointeur vers le nœud suivant.

Question : Quelle sera la complexité asymptotique du temps d'exécution pour trouver le dernier élément d'une liste de longueur n et faire en sorte que le champ « voisin » du premier nœud pointe vers le dernier nœud ?

Considérons l'algorithme suivant

```
SUR CHACUN des processeurs FAIRE
    voisin <- suivant
    TANTQUE (voisin!=null) & (voisin->voisin!=null) FAIRE
        voisin <- voisin->voisin
    FIN
FIN
```

Question : Quelle est alors la complexité asymptotique du temps d'exécution de cet algorithme pour une liste de longueur n ?

Peut-on faire mieux ?

Dans ce cours

- Nous n'aborderons quasiment plus les questions de complexité relatives aux machines parallèles.
- Nous aborderons assez peu les questions d'algorithme parallèle, même si nous présenterons ici et là des algorithmes qui répondent à des problèmes spécifiques de programmation parallèle.

Il s'agit d'un cours de **programmation** qui vise à enseigner comment écrire des programmes parallèles avant tout **corrects** mais aussi **performants**.

- ✓ *Un peu de motivation*
- ✓ *Un peu d'algorithmique parallèle*
- 👉 *Un peu de terminologie*

Qu'est-ce que la programmation concurrente ?

Un **programme séquentiel** est défini par une séquence d'actions.

Les instructions sont exécutées les unes après les autres (apparemment).

Il correspond à un seul processus, une seule tâche, un seul thread (fil d'exécution)

Un **programme concurrent** est un programme qui contient deux ou plusieurs processus qui communiquent entre eux.

On distingue :

1. Le parallélisme **explicite** : Instructions spéciales de création de threads, de synchronisation et communication, etc.
2. Le parallélisme **implicite** : c'est le compilateur qui se charge d'identifier et d'exploiter le parallélisme.

Variables partagées vs. Échanges de messages

Il faut aussi distinguer deux façons de communiquer :

1. Par l'intermédiaire de **variables partagées**

- Les processus partagent un état global modifiable
- L'information est communiquée entre processus en modifiant les variables partagées

2. Par l'échange de **messages** (ou de signaux)

- Chaque processus n'a accès qu'à ses propres variables locales
- Les processus communiquent en s'envoyant des messages de manière
 - **synchrone** : Les deux processus s'attendent l'un l'autre pour communiquer (téléphone sans boîte vocale), ou
 - **asynchrone** : Un processus peut envoyer un message sans attendre que l'autre soit prêt (téléphone avec boîte vocale)

Différents types d'applications concurrentes (1/3)

On peut distinguer **trois** types de programmation concurrente.

1. Les applications multi-contextes (multi-threaded) contiennent plusieurs threads.

Utilisations : Pour mieux organiser/structurer une application (plus grande modularité)

- Système d'exploitation multi-tâches
- Interfaces homme-machine
- ...

Ce cours s'appuie essentiellement sur Java

Différents types d'applications concurrentes (2/3)

2. Application parallèle

Chaque processus s'exécute sur son propre **processeur**

Utilisations : Pour résoudre plus rapidement un problème donné ou pour résoudre un type de problème sur des données plus grandes

- Prévisions météorologiques
- Prospection minière
- Physique moderne
- Bio-informatique (génomique)

OpenMP est le standard industriel pour le calcul parallèle.

Différents types d'applications concurrentes (3/3)

3. Application distribuée

Les processus communiquent entre eux par l'intermédiaire d'un réseau (délais plus longs)

Utilisations :

- Serveurs de fichiers
- Accès à distance à des banques de données
- Jeux en réseau
- ...

MPI est le standard industriel pour le calcul distribué.

Pour simplifier le tout

Au niveau du matériel, il existe différentes classes de machines

- Machine uniprocesseur : boîte contenant un seul processeur
- Multi-processeurs : boîte contenant plusieurs processeurs
 - ~~> Communication via un **bus**
- Multi-ordinateurs : plusieurs boîtes inter-connectées
 - ~~> Communication via un **réseau**

Mais aussi des grappes de multi-processeurs !

Programmation parallèle : plan du cours

- ① Threads en Java
 - ~~> *Les instructions de base*
- ② Atomicité, verrous, variables de condition et moniteurs
 - ~~> *Les concepts fondamentaux*
- ③ Classes « concurrentes » et autres outils en Java
 - ~~> *Pourquoi réinventer la roue ?*
- ④ Construction de verrous et programmation sans verrou
 - ~~> *Question de performance !*
- ⑤ Le modèle mémoire Java
 - ~~> *Qu'est-ce qu'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, 10h. de TD et 10h. 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

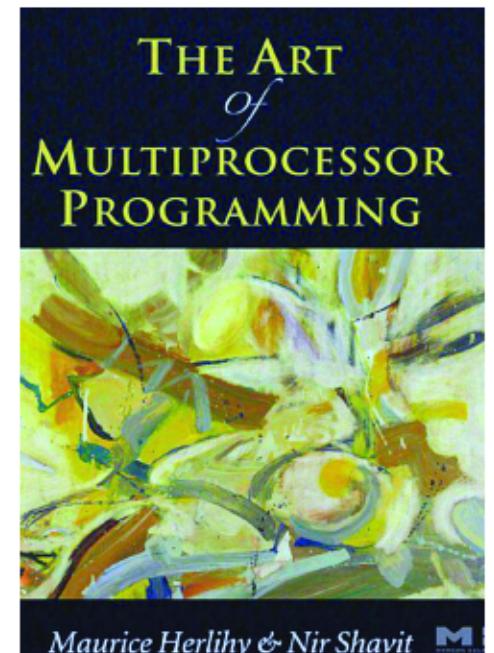
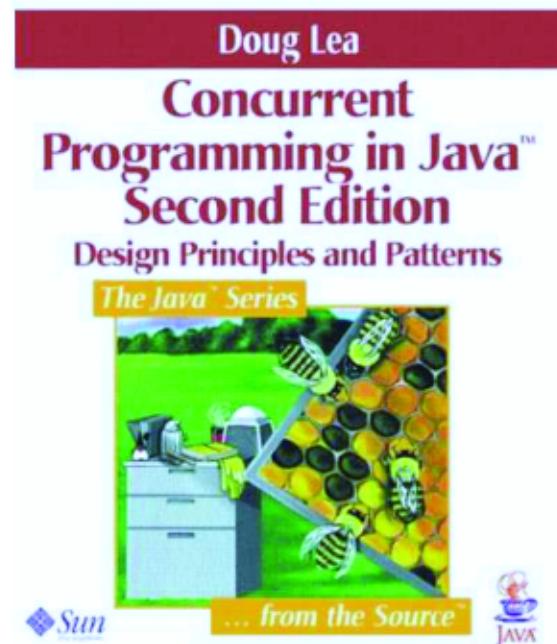
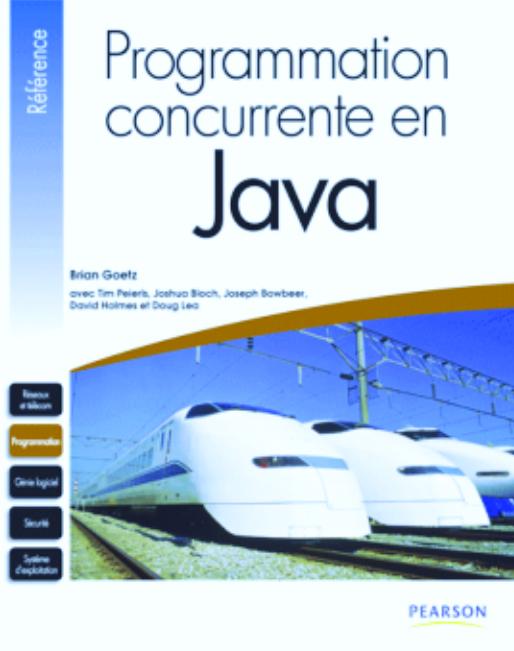
$$NF = 0,4 \times P + 0,6 \times ET$$

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' .

$$NF = \max(0,4 \times P + 0,6 \times ET', ET').$$

L'essentiel de ce cours s'inspire de



dont je recommande la lecture...

Programmation multithreadée en JAVA

Master Informatique — Semestre 2 — 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 indépendant.

Il faut d'abord créer un thread objet puis le lancer pour 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 alors implémenter une méthode **run()** qui permet de lancer l'objet.

Cette méthode est appelée de manière un peu particulière car il faut faire un appel à **start()** pour la lancer !

```
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 peut contenir le **main ()** du programme.
- Il est possible de lancer un thread en le créant.
- Il est possible de lancer plusieurs threads qui fonctionnent « en même temps ».

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

Tous les threads progressent « en même temps »

Ce que « en même temps » veut dire dépend du matériel utilisé. Dans un système possédant plusieurs processeurs il est possible que divers threads s'exécutent chacun sur un processeur distinct.

Mais, le plus souvent, les systèmes ne bénéficient que d'un, deux ou quatre processeurs : les divers threads disposent du processeur à tour de rôle selon **l'ordonnancement**.

Un thread cède l'activité du processeur à un autre thread

- si la tranche de temps allouée au thread actif est écoulée.
- s'il est bloqué par une commande (comme `sleep()`, ou `wait()`) ou une opération d'entrée-sortie (comme `read()`) ;
- si un thread ayant une priorité supérieure à celle du thread actif est débloqué ;

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 + " " + getName());  
    }  
}
```

Que voit-on en sortie ?

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 monThreadBis extends Thread {  
    public static void main(String[] args) {  
        monThreadBis p1 = new monThreadBis();  
        monThreadBis p2 = new monThreadBis();  
        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 + " " + getName());  
    }  
}
```

✓ *Construction d'un thread*

✓ *Propriétés d'un thread*

👉 *État d'un thread Java*

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 milisecondes.

```
public class monThreadTer extends Thread {  
    public static void main(String[] args) {  
        monThreadTer p1 = new monThreadTer();  
        monThreadTer p2 = new monThreadTer();  
        p1.start(); p2.start();  
    }  
    public void run() {  
        for (int i = 1; i<=10; i++) {  
            System.out.println(i + " " + getName());  
            sleep(100);  
        }  
    }  
}
```

Ordonnancement des threads

`join()` attend la fin de l'exécution du thread sur lequel elle s'applique. De plus, les variantes

- `join(long millis)` et
- `join(long millis, long nanos)`

limitent l'attente de la terminaison du thread à une durée spécifiée.

La variante `sleep(long millis, long nanos)` permet aussi de spécifier une durée à une nanoseconde près.

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

Je déconseille pour commencer d'utiliser `yield()` !

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. »

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

Différents états possibles d'un thread

- ① exécute son code cible (il a accès à l'un des processeurs pour cela)
- ② attend l'accès à un processeur (mais pourrait s'exécuter)
- ③ attend un événement particulier (pour être autorisé à commencer, ou poursuivre, son exécution)

Un thread peut libérer le processeur qu'il occupe

1. s'il exécute une instruction `yield()` ;
2. si la tranche de temps allouée est terminée ;
3. s'il exécute une méthode bloquante (`sleep()`, `join()`, `read()`, `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 respectant éventuellement les priorités spécifiées et les instructions `yield()`.

Différents états d'un thread

Depuis Java 1.5 (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` qui comporte :

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 ;

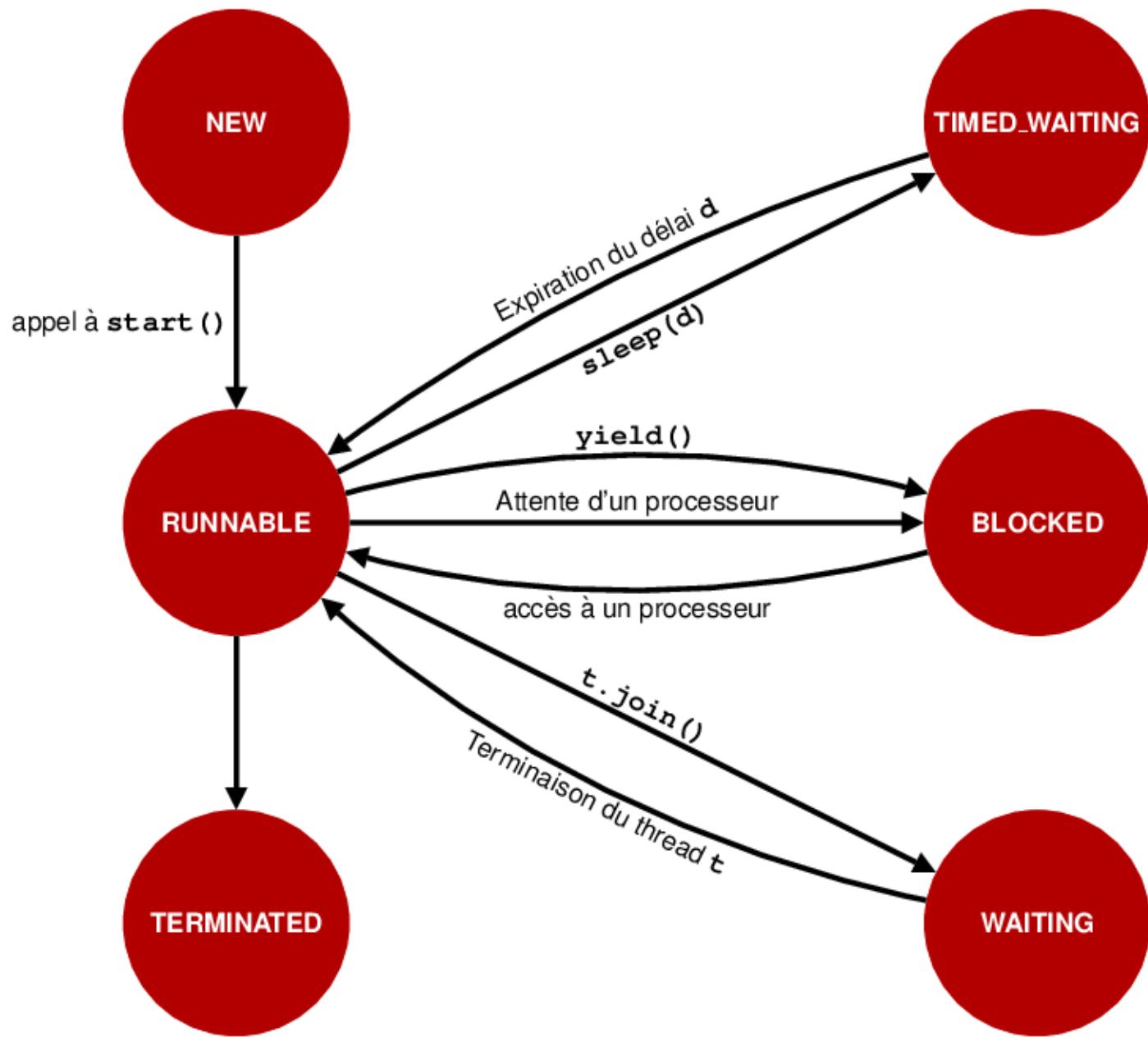
BLOCKED : il est bloqué en attente d'un *privilege logique*, par exemple de l'acquisition d'un *verrou* ;

WAITING : il est en attente (d'une durée indéfinie) d'un évènement provoqué par un autre thread ;

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

TERMINATED : il a fini d'exécuter son code.

Les six états d'un thread



- ✓ *Construction d'un thread*
- ✓ *Propriétés d'un thread*
- ✓ *État d'un thread Java*
- 👉 *Synchronisation et verrous*

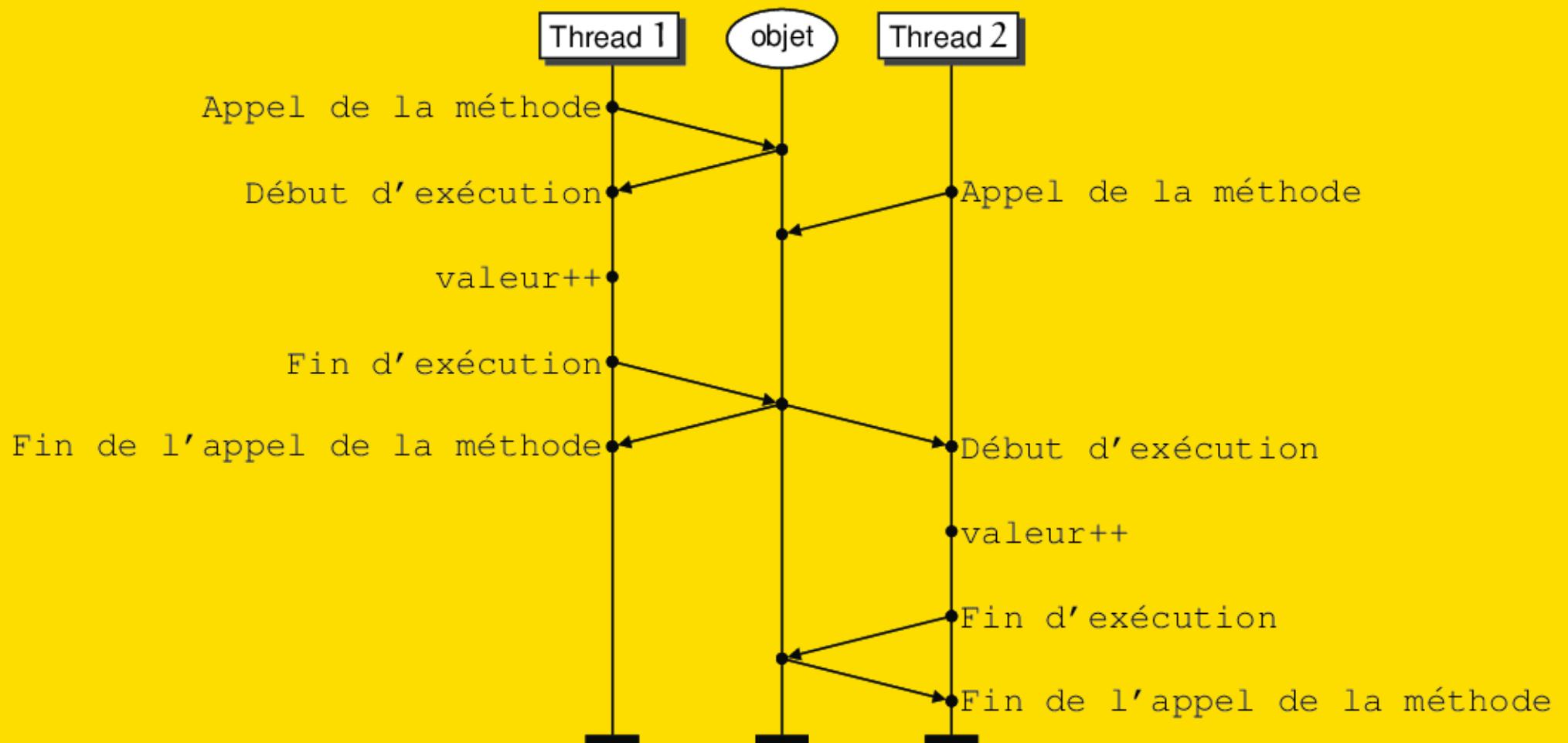
Vingt milliers d'incrémentations anarchiques par deux threads

```
public class Compteur extends Thread {  
    static volatile int valeur = 0;  
    public static void main(String[] args) {  
        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<=10000 ; i++)  
            valeur++;  
    }  
}
```



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 *totallement ordonnés*.



Correction du programme Compteur.java

```
public void run() {  
    for (int i = 1; i <= 10000; i++)  
        synchronized (Compteur.class) { valeur++; }
```

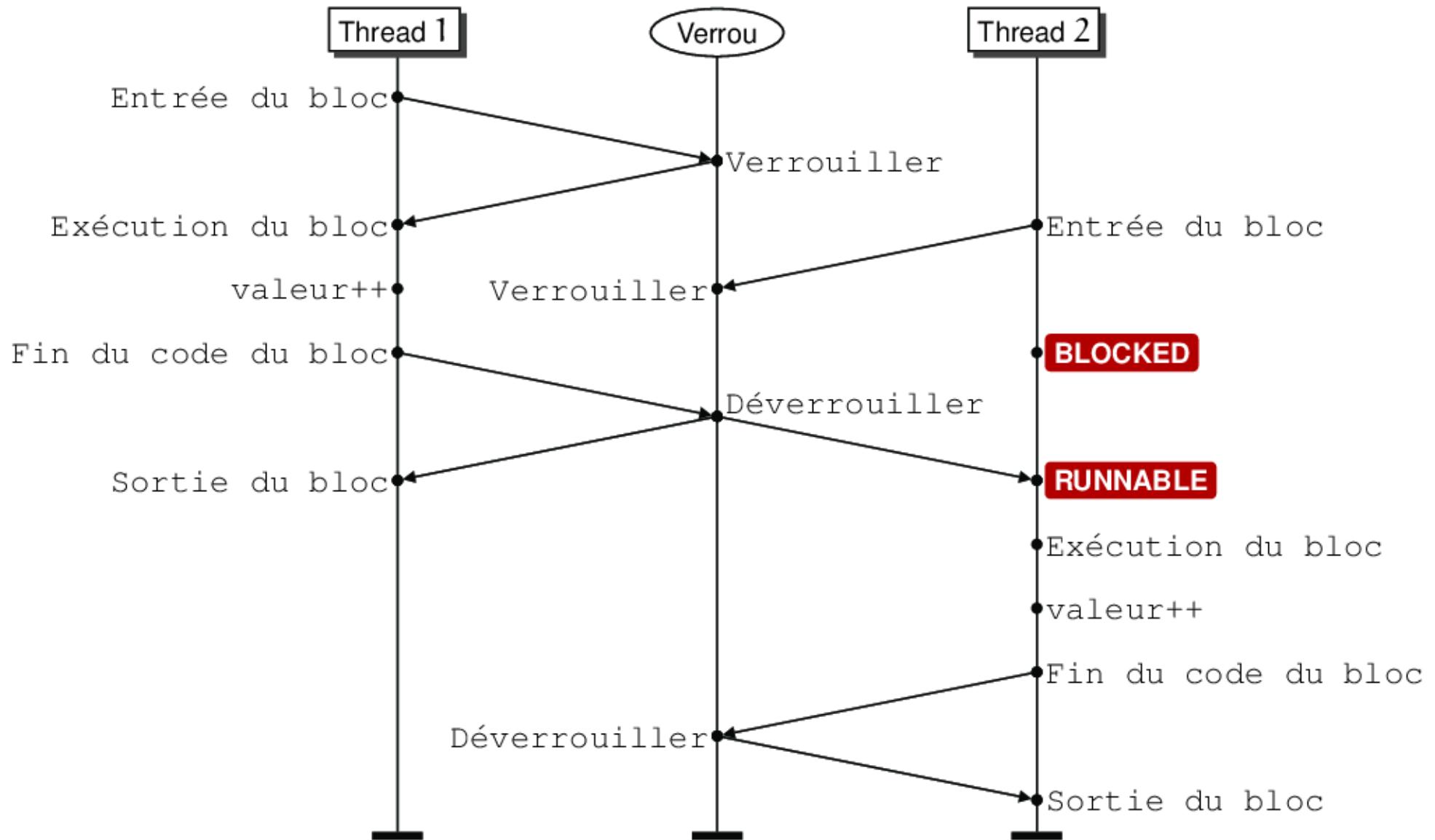
```
$ java Compteur  
La valeur finale est 20000  
...
```

Il y a désormais « exclusion mutuelle » des accès à la variable **valeur** : un seul thread incrémentera cette variable à chaque instant : celui qui possède 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)**.

Fonctionnement d'un bloc << synchronized >>



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 !

Un bloc **synchronized** requiert avant l'exécution de son code l'acquisition du *verrou intrinsèque* de l'objet indiqué en paramètre. Dans notre exemple, c'est le verrou de la classe **Compteur.class**.

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

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.

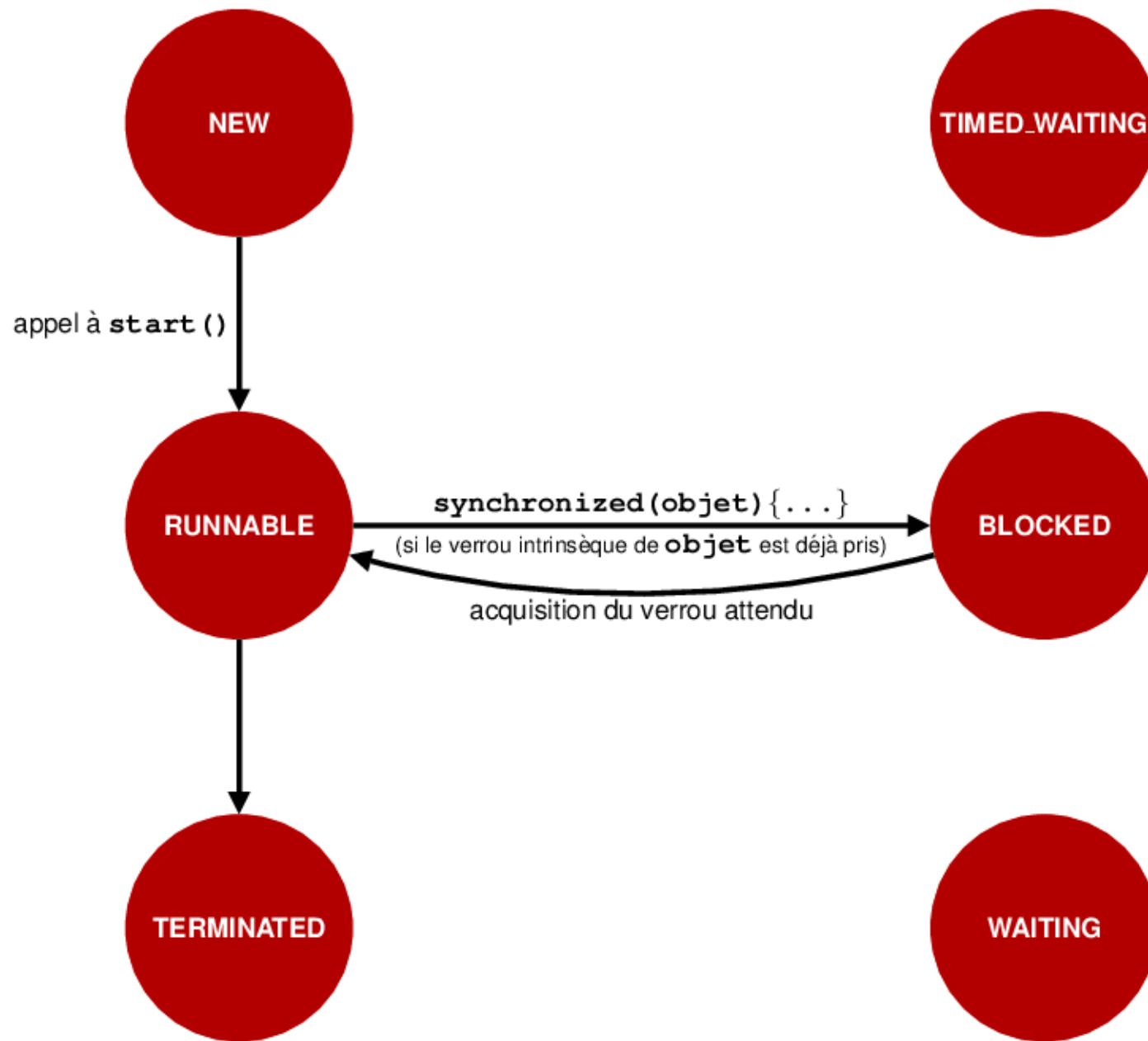
- ~~> C'est le principe du verrou : le thread prend le risque d'être bloqué.

- 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.

Les six états d'un thread (suite)



Modificateur **synchronized** d'une méthode

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

- Il y a :
- acquisition du verrou de **this** en entrée de bloc.
 - relâchement du verrou de **this** en sortie de bloc.

```
synchronized void inc() {  
    compteur++;  
}
```

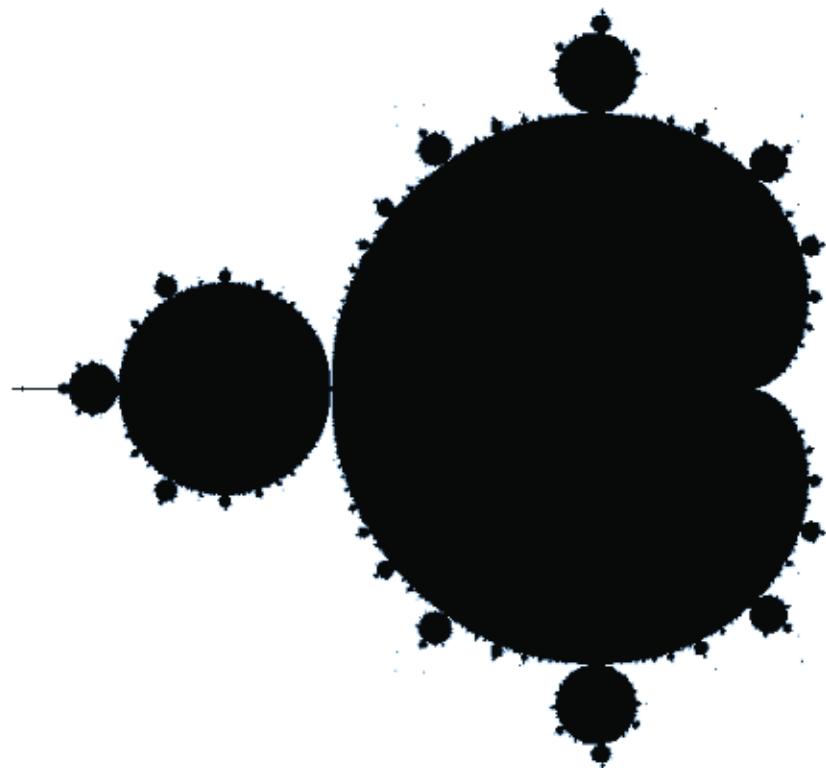
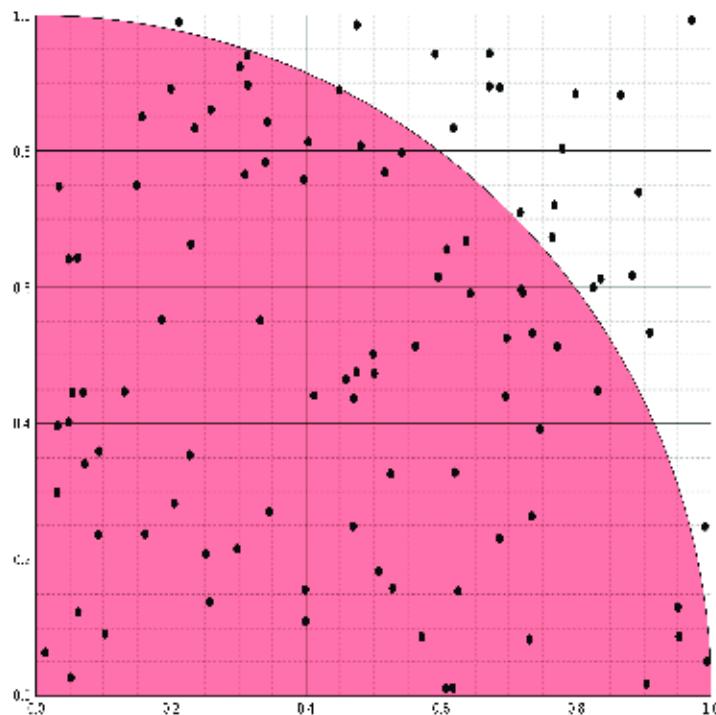
```
void inc() {  
synchronized(this) {  
    compteur++;  
}
```

Ces deux méthodes sont équivalentes.

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).

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**.



Des exercices plus complexes viendront ensuite...

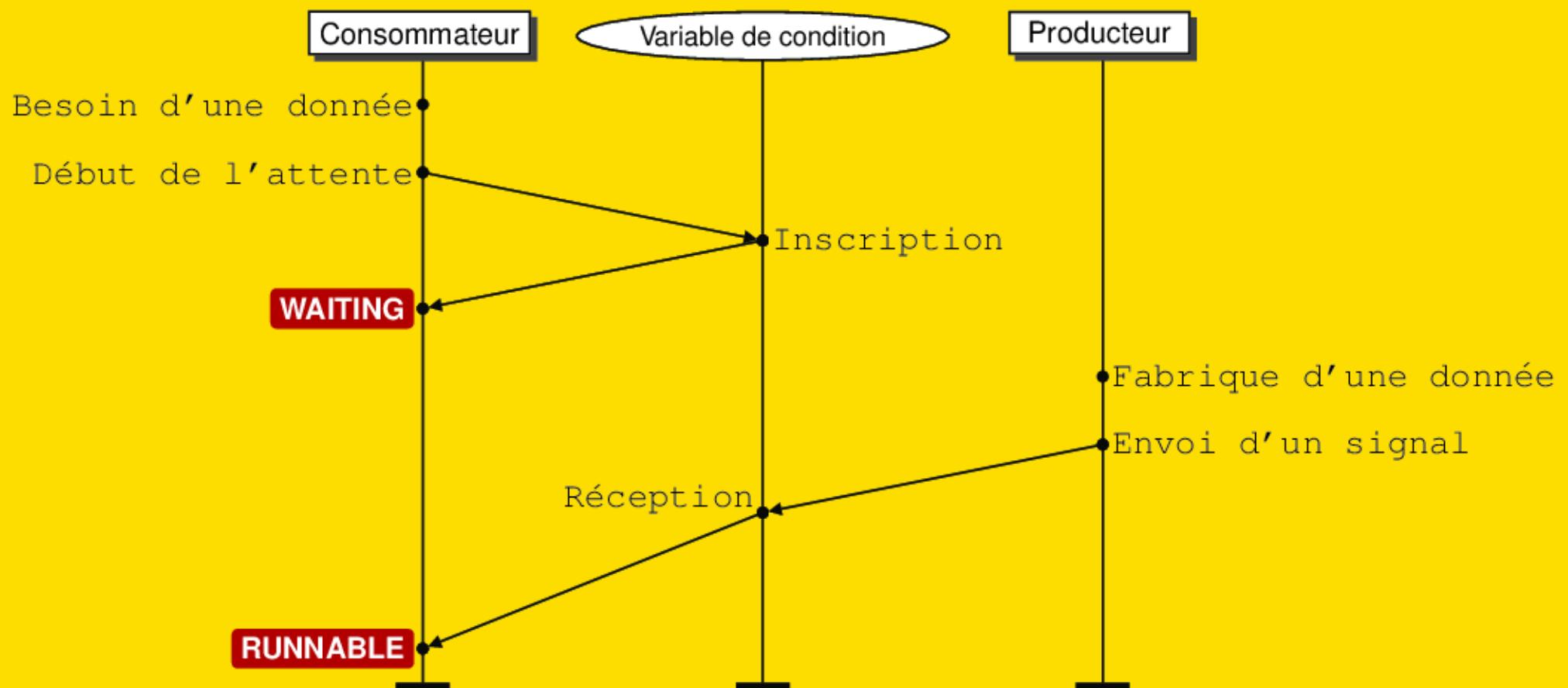
Exercice : que va afficher ce programme ? (Réponse en TD et test en TP)

```
class SyncTest extends Thread {  
    String msg;  
    private static Object unObjet = new Object() ;  
    public SyncTest(String s) { msg = s; start () ; }  
    public void run() {  
        synchronized (unObjet) {  
            System.out.print("[" + msg);  
            sleep(1000);  
            System.out.println("]");  
        }  
    }  
    public static void main(String [] args) {  
        new SyncTest("Hello");  
        new SyncTest("Synchronized");  
        new SyncTest("World");  
    }  
}
```

- ✓ *Construction d'un thread*
- ✓ *Propriétés d'un thread*
- ✓ *État d'un thread Java*
- ✓ *Synchronisation et verrous*
- 👉 *Signaler et attendre en Java*

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. »



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.

Trois 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.
- ③ La fonction **empty (cv)** permet de déterminer si la file associée 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 !

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

Le thread attend donc sur un objet précis !

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

notifyAll() Redémarre tous les threads qui ont appelé **wait()** sur cet objet.

Un exemple de moniteur qui protège une donnée (1/3)

```
public class WaitNotify {  
  
    public static void main(String[] args) {  
        Moniteur moniteur = new Moniteur();  
        Afficheur afficheur = new Afficheur(moniteur);  
        Travailleur travailleur = new Travailleur(moniteur);  
        afficheur.start();  
        travailleur.start();  
    }  
}
```

Un objet de la classe **Moniteur** a été créé ; les deux threads lancés peuvent agir sur cet objet, passé en paramètre.

Un exemple de moniteur qui protège une donnée (2/3)

```
class Afficheur extends Thread {  
    Moniteur moniteur;  
    public Afficheur(Moniteur moniteur) {this.moniteur = moniteur;}  
    public void run(){  
        System.out.println("Le résultat est " + moniteur.recuperer());  
    }  
}  
  
class Travailleur extends Thread {  
    Moniteur moniteur;  
    public Travailleur(Moniteur moniteur) {this.moniteur = moniteur;}  
    public void run(){  
        moniteur.initialiser();  
    }  
}
```

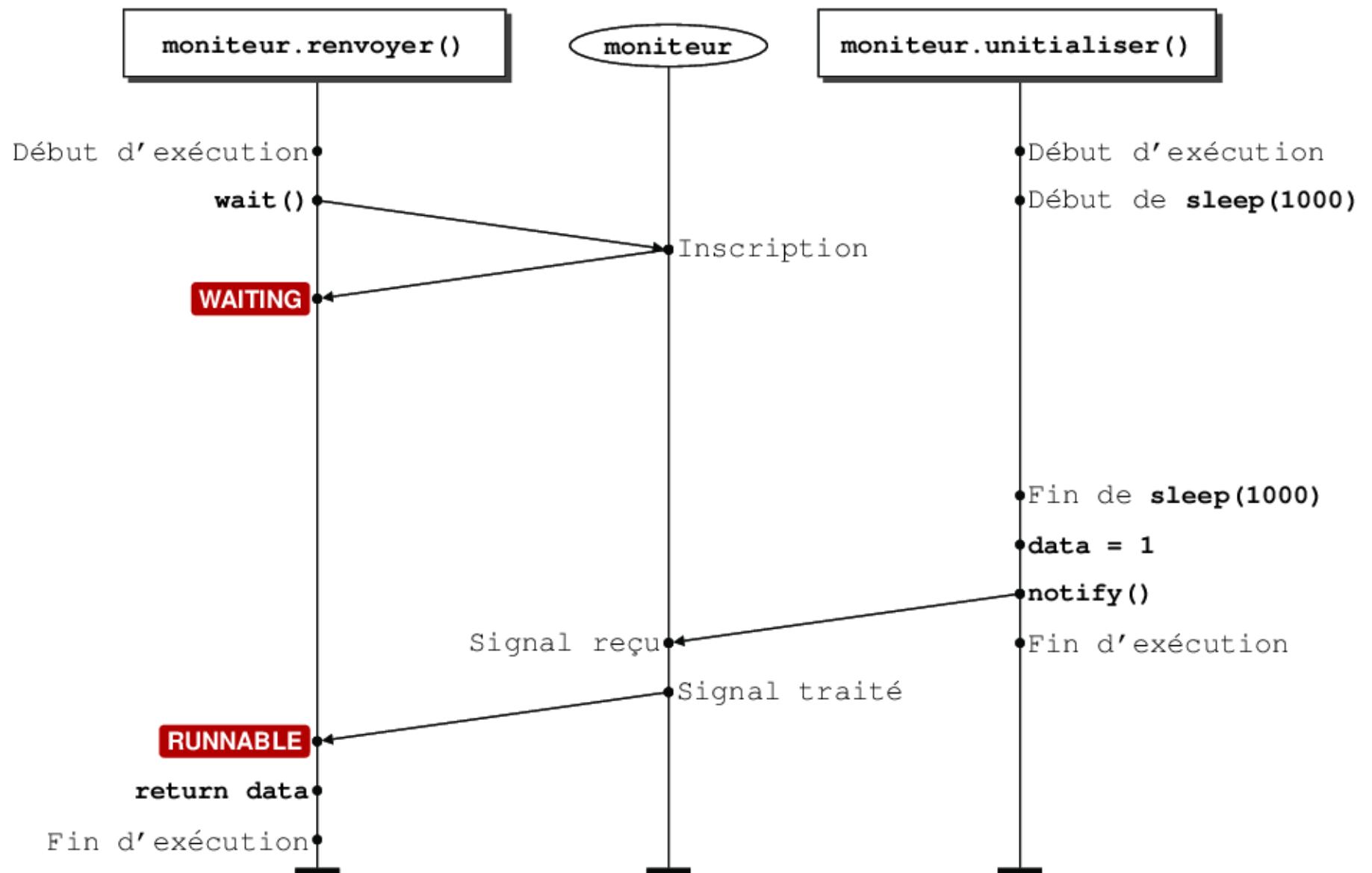
Chaque thread appelle une méthode sur le même objet moniteur.

Un exemple de moniteur qui protège une donnée

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



Principe approximatif de l'exécution



Ce scenario est-il vraiment envisageable ?

Disciplines de signalisation : deux types de variables de condition

Pour diverses raisons, une variable de condition est toujours associée à un *verrou, qu'il faut posséder pour pouvoir la manipuler.*

C'est une contrainte adoptée par Java !

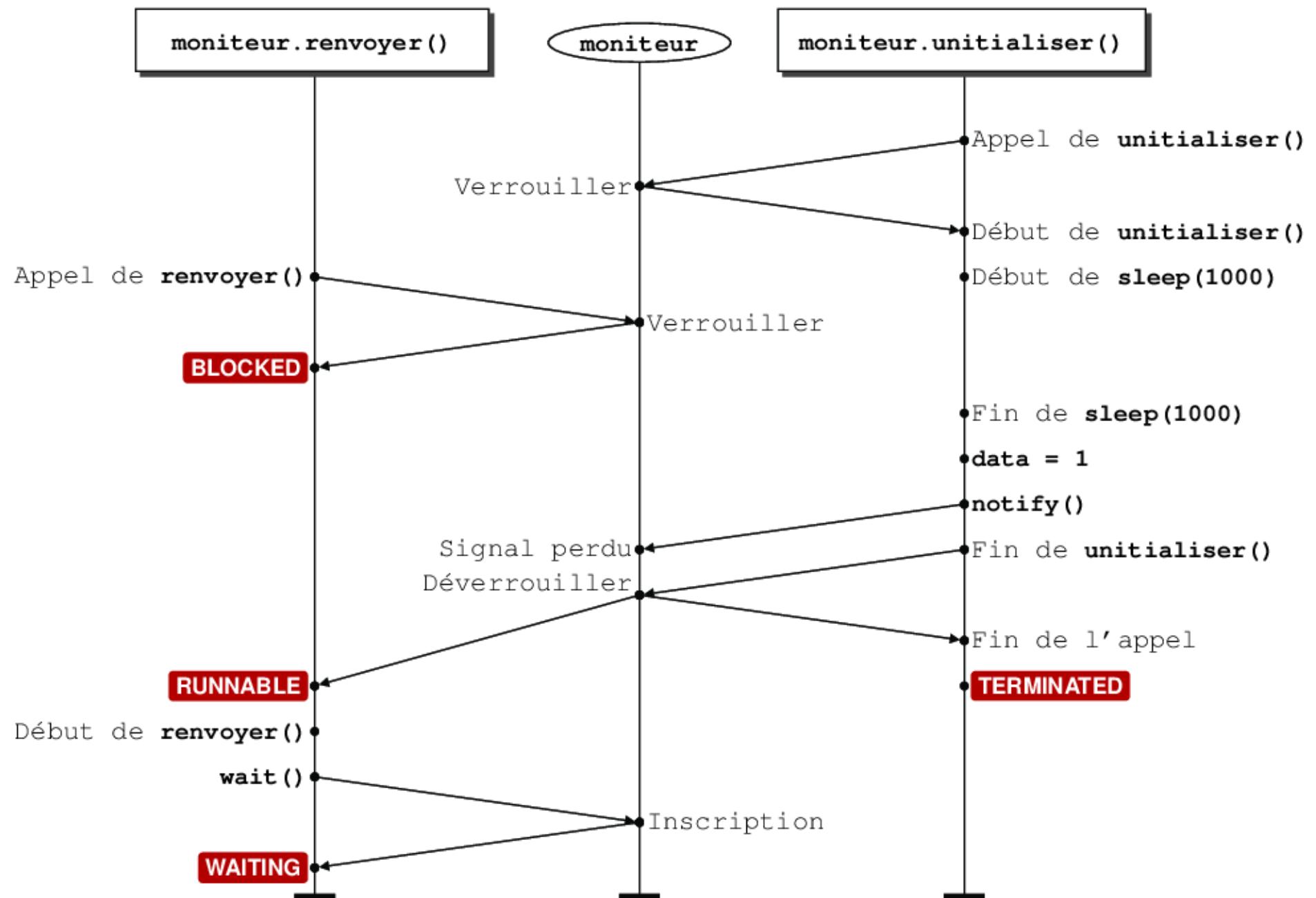
Naturellement, le verrou est relâché lorsqu'un thread se place en attente.

Deux approches sont possibles en général pour définir l'effet de **signal (cv)**, en tenant compte du fait que *le processus qui exécute cette opération est celui qui possède le verrou.*

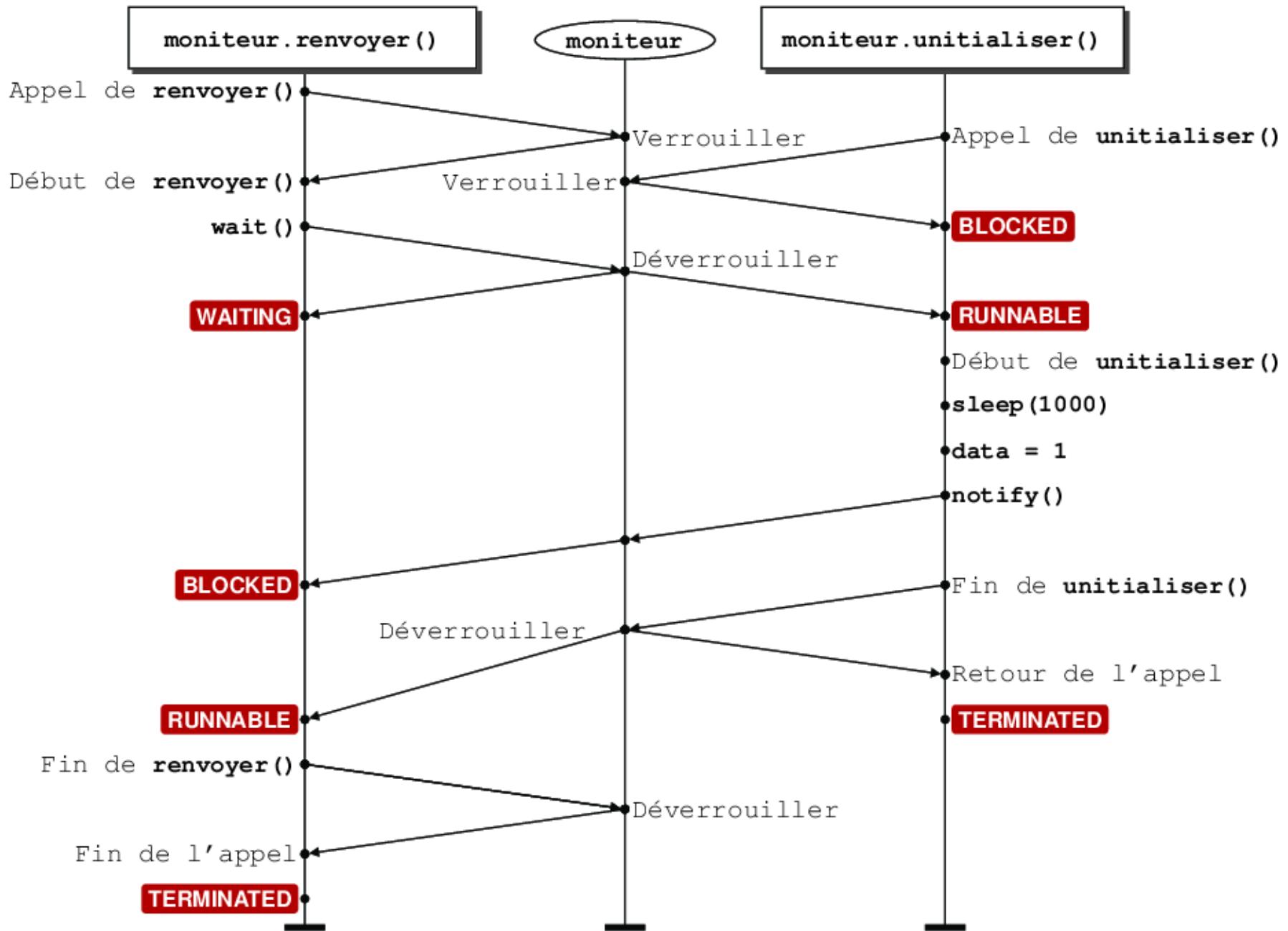
- ① **Signaler et Attendre** : le processus qui signale cède le verrou à celui qui vient d'être activé.
- ② **Signaler et Continuer** : le processus qui exécute l'instruction **signal (cv)** *conserve le verrou* (et poursuit donc son exécution). Le processus qui a reçu le signal sera exécuté plus tard, *lorsqu'il aura obtenu le verrou.*

La seconde discipline est une approche non-préemptive ; elle est plus récente, mais c'est celle la plus couramment utilisée (Unix, Java, Pthreads).

Ce système peut bloquer ! (mais ce n'est pas un interblocage)



Ce système termine parfois...



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

Les méthodes `wait()` et `notify()` doivent 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 qui entre alors dans l'état **WAITING** ;
- ~~ *relâche le verrou intrinsèque* de `a`.

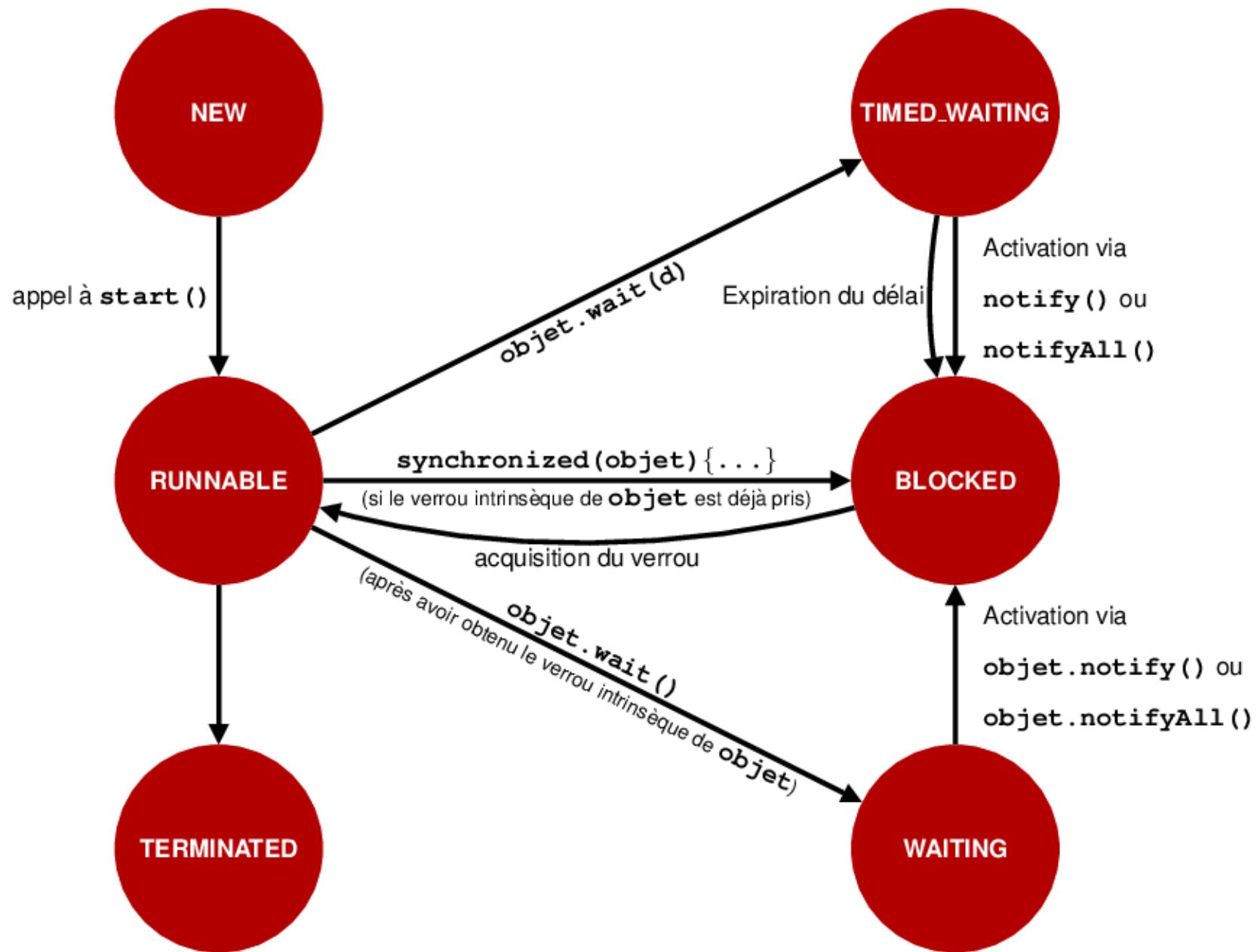
`a.notify()`

- ~~ *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 dans l'état **BLOCKED** et doit récupérer le verrou avant de redémarrer.

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

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

Les six états d'un thread (fin)



- ✓ *Construction d'un thread*
- ✓ *Propriétés d'un thread*
- ✓ *État d'un thread Java*
- ✓ *Synchronisation et verrous*
- ✓ *Signaler et attendre en Java*
- 👉 *Le mot-clé « volatile »*

Un problème de vue (à tester chez soi)

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



Ce programme termine-t-il ? Pourquoi ?

Solution : le modificateur **volatile**

```
static class A extends Thread {  
    public volatile boolean fin = false;  
    public void run() {  
        while(! fin) ; // Attente active  
    }  
}
```



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

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

Ce qu'il faut retenir

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

Les champs d'un objet susceptibles d'être accédés par plusieurs threads doivent *a priori* être déclarés **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 **notify()** ou **notifyAll()** sur cet objet.

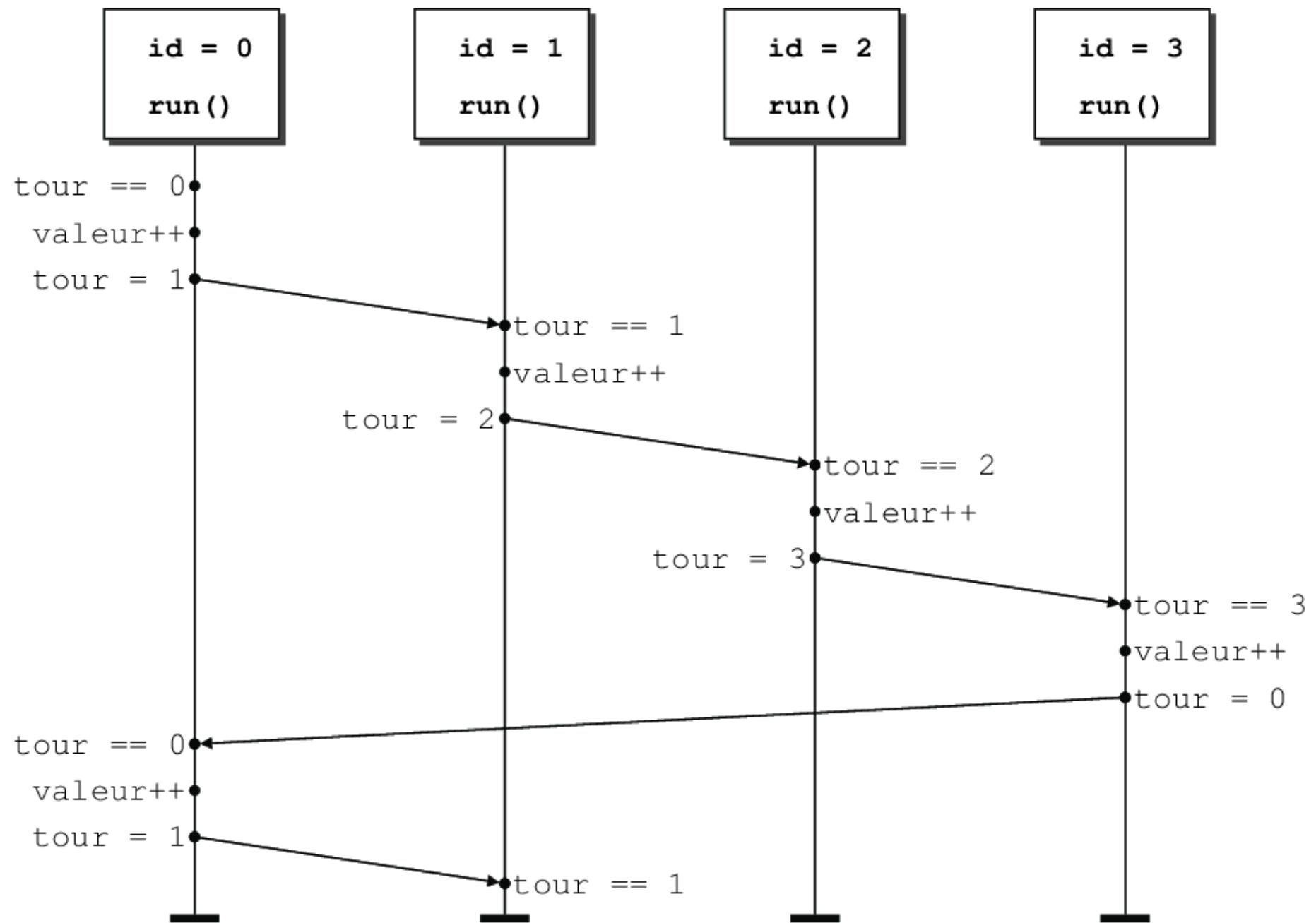
La méthode **wait()** nécessite d'acquérir au préalable le verrou intrinsèque de l'objet sur lequel elle est appliquée, à l'aide de **synchronized**. *Mais ce verrou est alors relâché !*

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 2 — 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;
    }
}
```

Le benchmark adopté

```
public class Compteur extends Thread {  
    static volatile int valeur = 0;  
    static final int valeur_finale = 840;  
    static int nombre_de_compteurs;  
    static volatile int tour = 0;  
    final int identité; // Chaque thread possède une identité  
    final int part;  
    Compteur(int identite) {  
        this.identite = identite ;  
        this.part = valeur_finale / nombre_de_compteurs ;  
    }  
}
```

Il y aura **nombre_de_compteurs** threads ; chacun réalise **part** incrémentations de **valeur**. La variable **tour** désigne le thread qui doit réaliser la prochaine incréméntation.

Le benchmark adopté

```
public static void main(String[] args) throws exceptions {
    nombre_de_compteurs = Integer.parseInt(args[0]);
    Thread[] T = new Thread[nombre_de_compteurs];
    for(int id=0; id<nombre_de_compteurs; id++) {
        T[id] = new Compteur(id);
    }
    for(int id=0; id<nombre_de_compteurs; id++) T[id].start();
    for(int id=0; id<nombre_de_compteurs; id++) T[id].join();
}
public void run() {
    ...
}
```

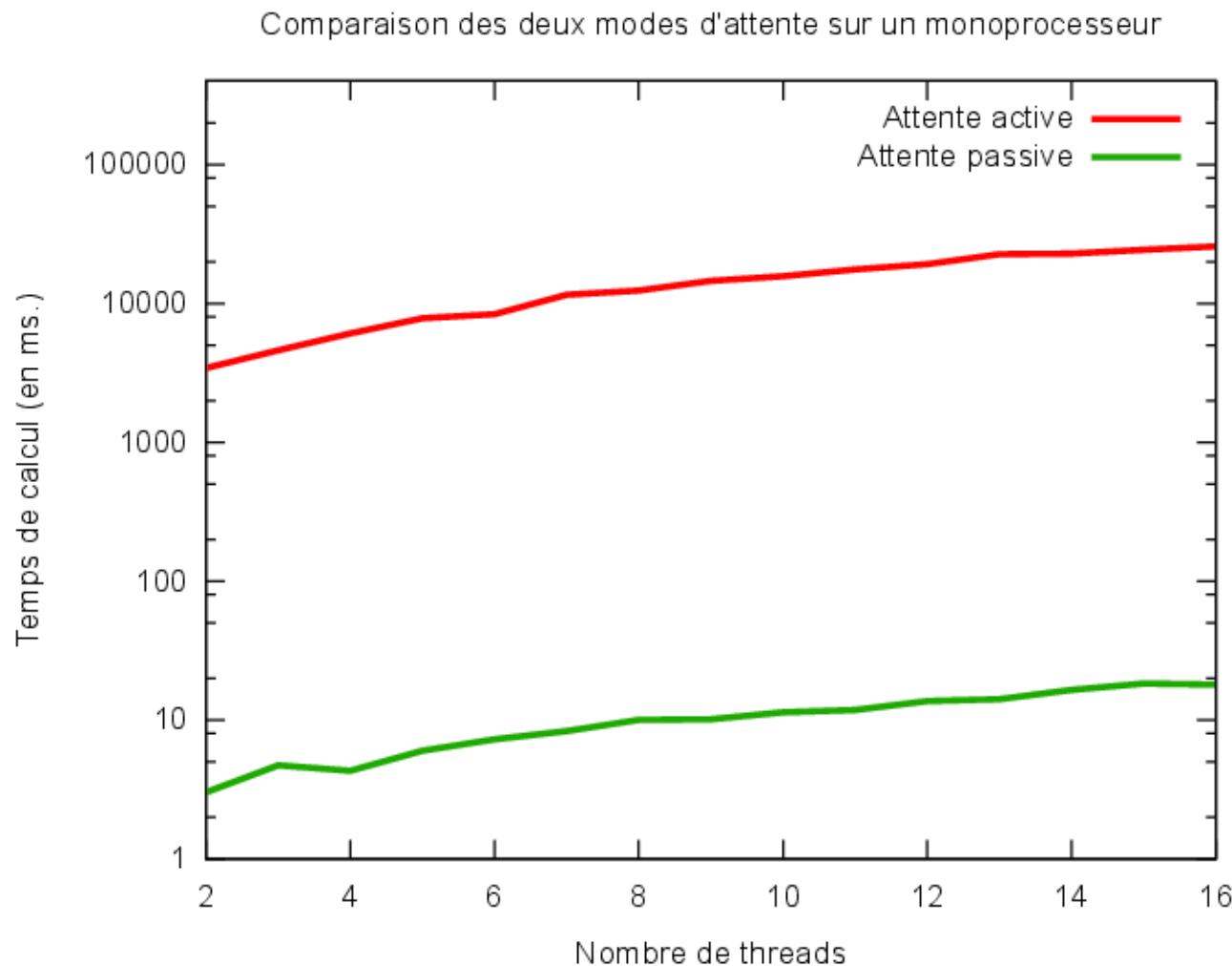
Alternative à l'attente active : l'attente passive

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();  
        }  
    }  
}
```

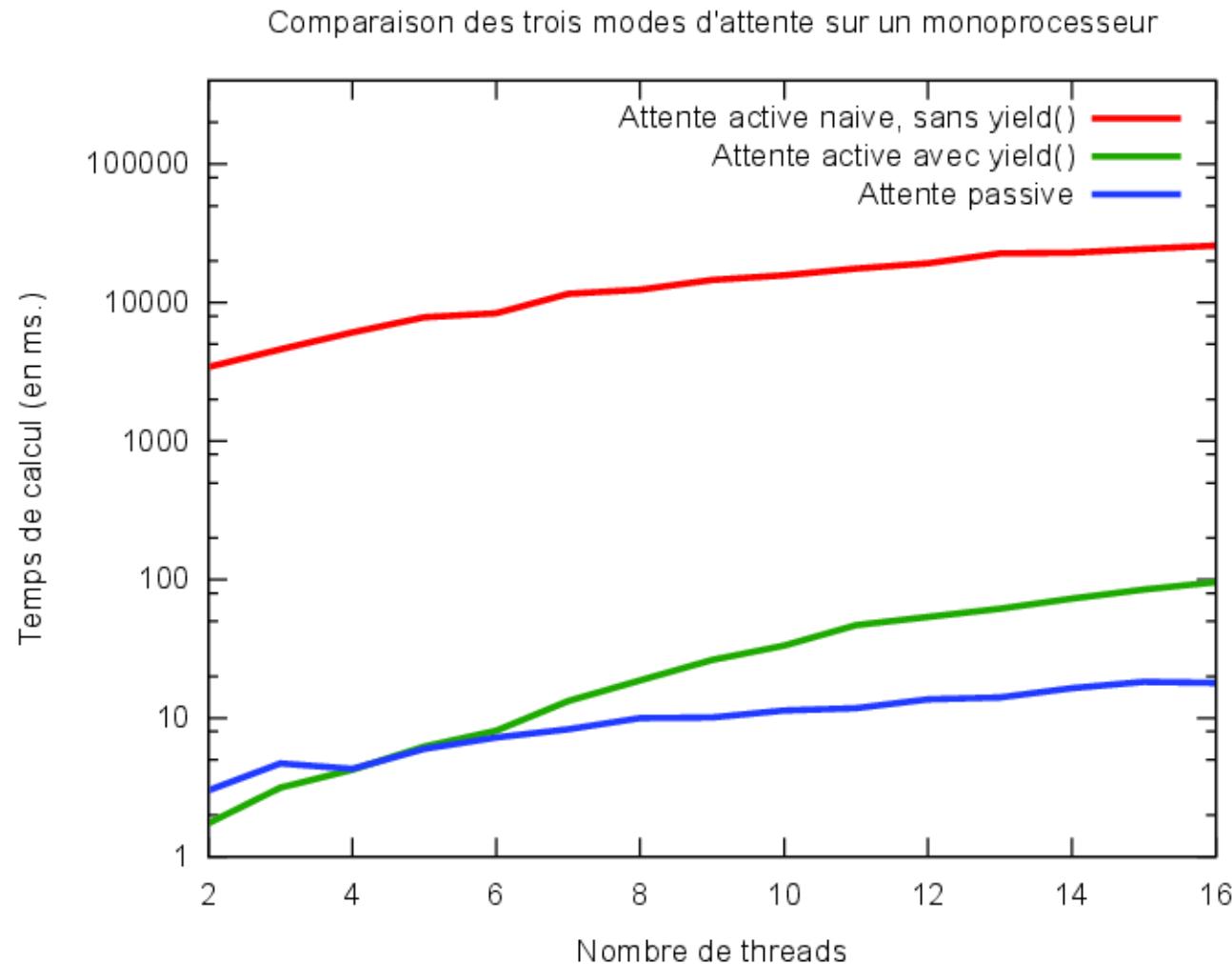
Pourquoi faut-il réveiller tout le monde ?

Premiers résultats, sur une machine monoprocesseur



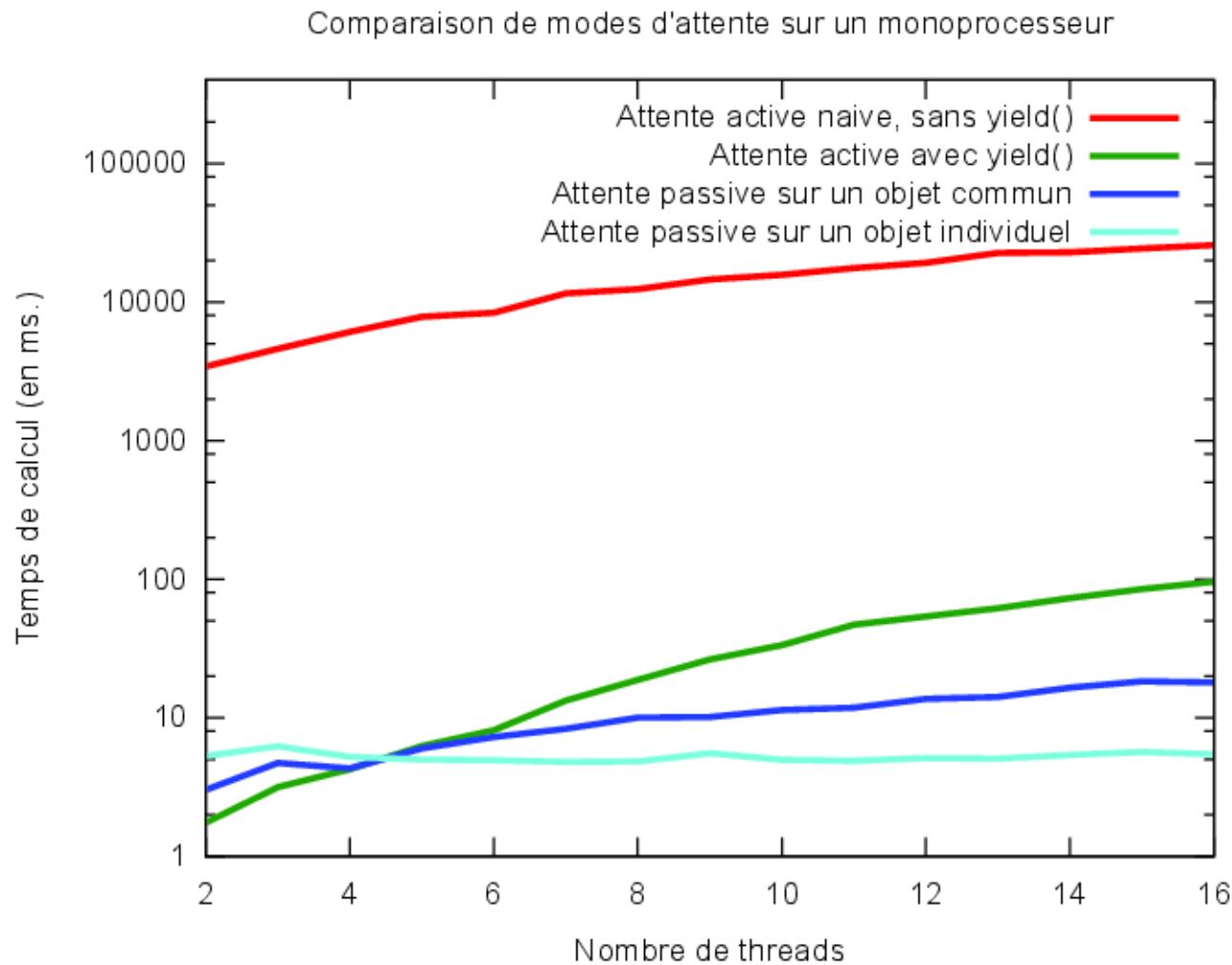
Le temps de calcul est bien moindre avec une attente passive. Il est un millier de fois plus élevé avec une attente active, car chaque thread actif sur le processeur consomme sa tranche de temps en entier (même si ce n'est pas son tour d'incrémenter).

Rôle et apport de l'instruction `yield()`



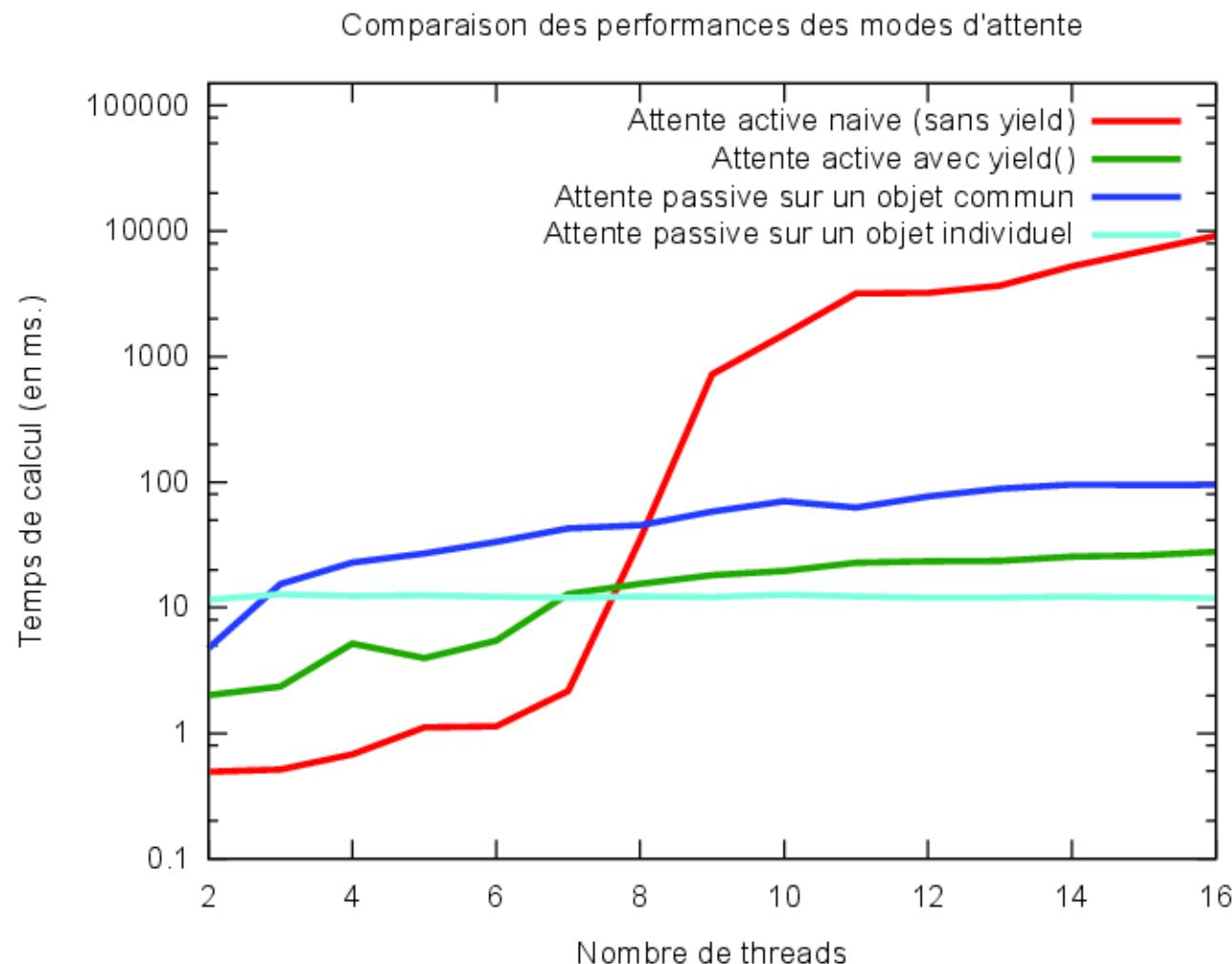
Attendre via `while (tour!=id) yield();` permet de relâcher le processeur pour ne pas gâcher la tranche de temps allouée ; néanmoins, sur cette machine le relâchement ne semble pas instantané puisque l'attente passive reste plus efficace s'il y a plus de 6 threads.

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 d'incrémenter, il faut appliquer **wait ()** et **notify ()** sur des objets distincts.

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



Tant que le nombre de threads est inférieur au nombre de coeurs, l'attente active naïve est la plus efficace. Sur cette machine, `yield()` semble plus rapide que le mécanisme des signaux envoyés à tous ; mais l'envoi d'un seul signal demeure l'implémentation la plus efficace.

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.

Interruptions et exceptions

Master Informatique — Semestre 2 — 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.

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.

Exemple peu lisible (comme promis) malgré une bonne indentation

```
Thread[] threads=new Thread[5];
for(int i=0;i<threads.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());
        }
    });
    threads[i]=t;
    t.start();
}
Thread.sleep(10);
for(int i=0;i<threads.length;i++) threads[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

✓ *Statut d'interruption*

(👉 *Les exceptions*

Rappel sur les exceptions

Les exceptions correspondent au mécanisme de gestion des erreurs du langage Java. Celui-ci utilise des objets représentant les erreurs produites, trois mots clés qui permettent de détecter et de traiter ces erreurs : **try**, **catch** et **finally** ainsi que le moyen de les lever ou les propager, à l'aide **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.



Pour faciliter la lecture et simplifier la présentation, les exemples indiqués jusqu'à présent n'ont pas pris en compte les exceptions.

Cependant les codes complets et corrects de ces exemples seront disponibles en Travaux Pratiques.

Exemples d'exceptions

sleep (delai) 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**.

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.

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**.

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

✓ *Statut d'interruption*

✓ *Les exceptions*

👉 *Un exercice pour terminer*

Exercice avec `interrupt()`

Partant du code peu lisible illustrant les interruptions, l'objectif est de ralentir l'affichage en faisant en sorte que chaque thread attende un temps aléatoire compris entre 1 et 5 ms, entre chaque itération.

Pour cela on ajoute un appel à une méthode nommée `slow()` dans la boucle de la méthode `run()` :

```
public void run() {  
    for(int j=0 ; !Thread.interrupted() ; j++) {  
        System.out.println(j + "ième_exécution_de_" + id);  
        slow(); // Ralentit l'affichage  
    }  
    ...  
}
```

Écrire une méthode `slow()` correcte !

Exercice avec `interrupt()` : les threads sont ralentis.

```
1ière exécution de 4
2ième exécution de 2
1ière exécution de 0
3ième exécution de 2
1ière exécution de 1
1ière exécution de 3
Fin d'exécution_du_code_1
false
Fin_d'exécution du code 4
Fin d'exécution_du_code_2
Fin_d'exécution du code 0
Fin d'exécution_du_code_3
false
false
false
false
```