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 $\mathtt{NF}$ se compose de deux notes

- une note d'examen terminal : $\mathtt{ET}$
- une note de projet : $\mathtt{P}$

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

La note de projet $\mathtt{P}$ 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 $\mathtt{ET'}$.

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

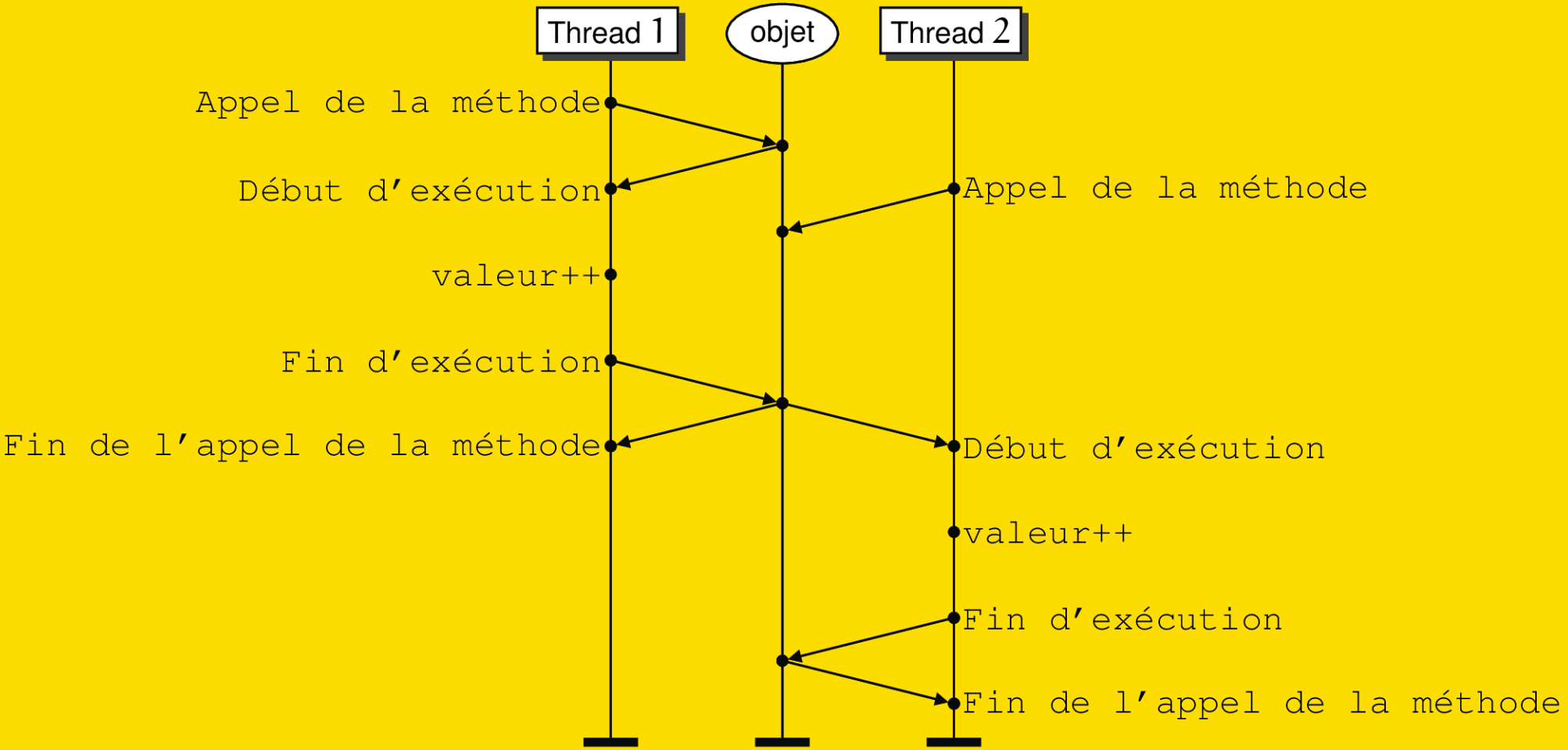
Vingt milliers d'incrémentations anarchiques par deux threads

```
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

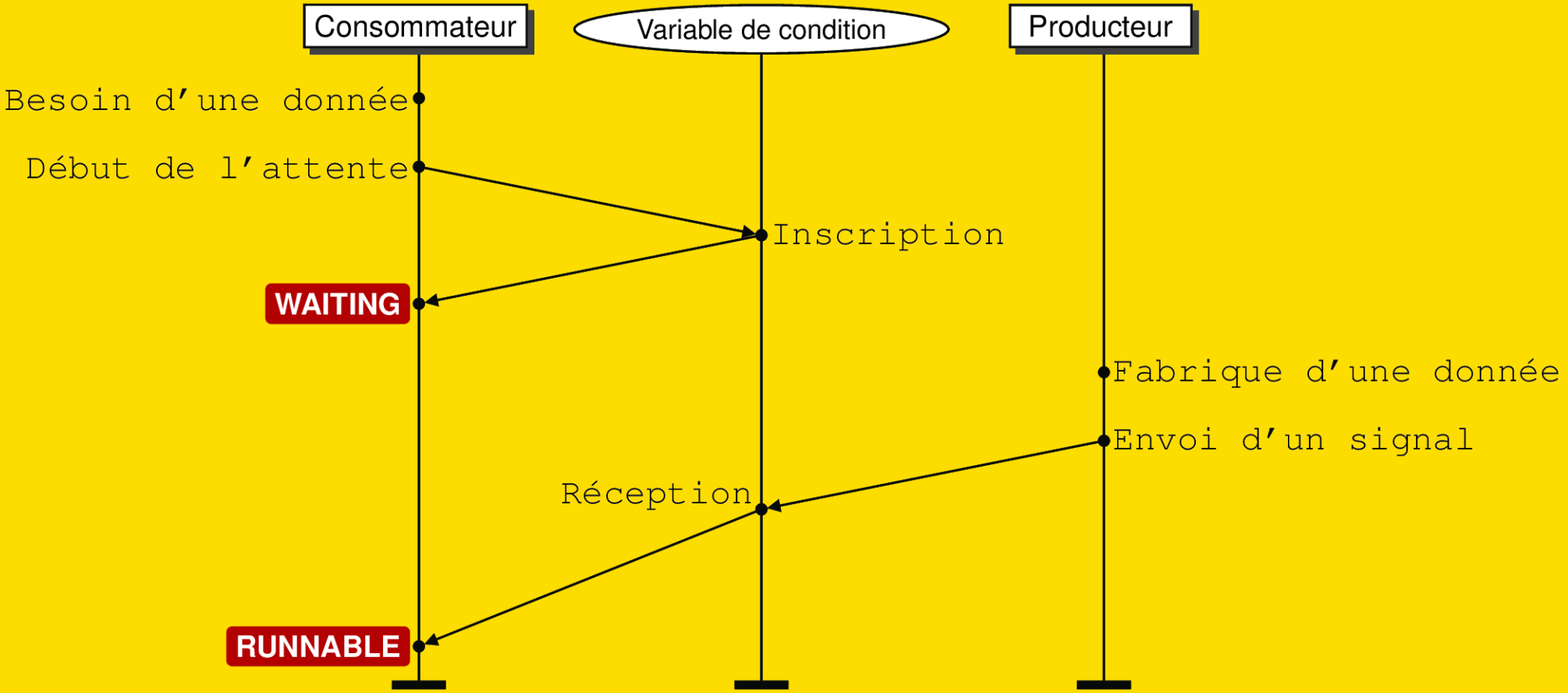
Lorsque deux threads peuvent accéder en même temps à un même objet (ou une variable partagée), il est *très souvent nécessaire* qu'ils ne le fassent qu'*un seul à la fois*.

Les accès à l'objet (ou à la variable partagée) doivent alors être *totalement ordonnés*.

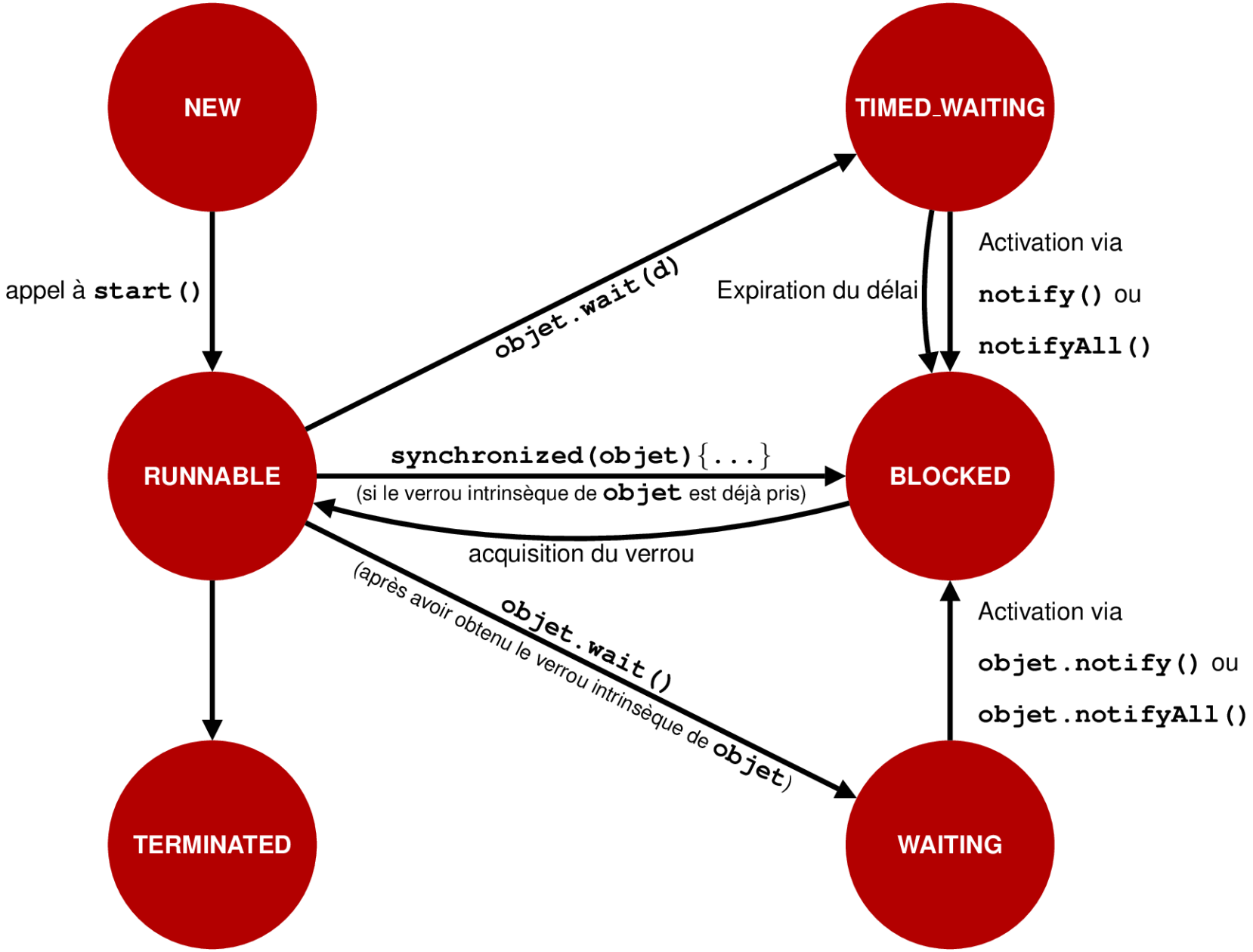


Second problème de synchronisation

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. »



Les six états d'un thread (fin)



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.

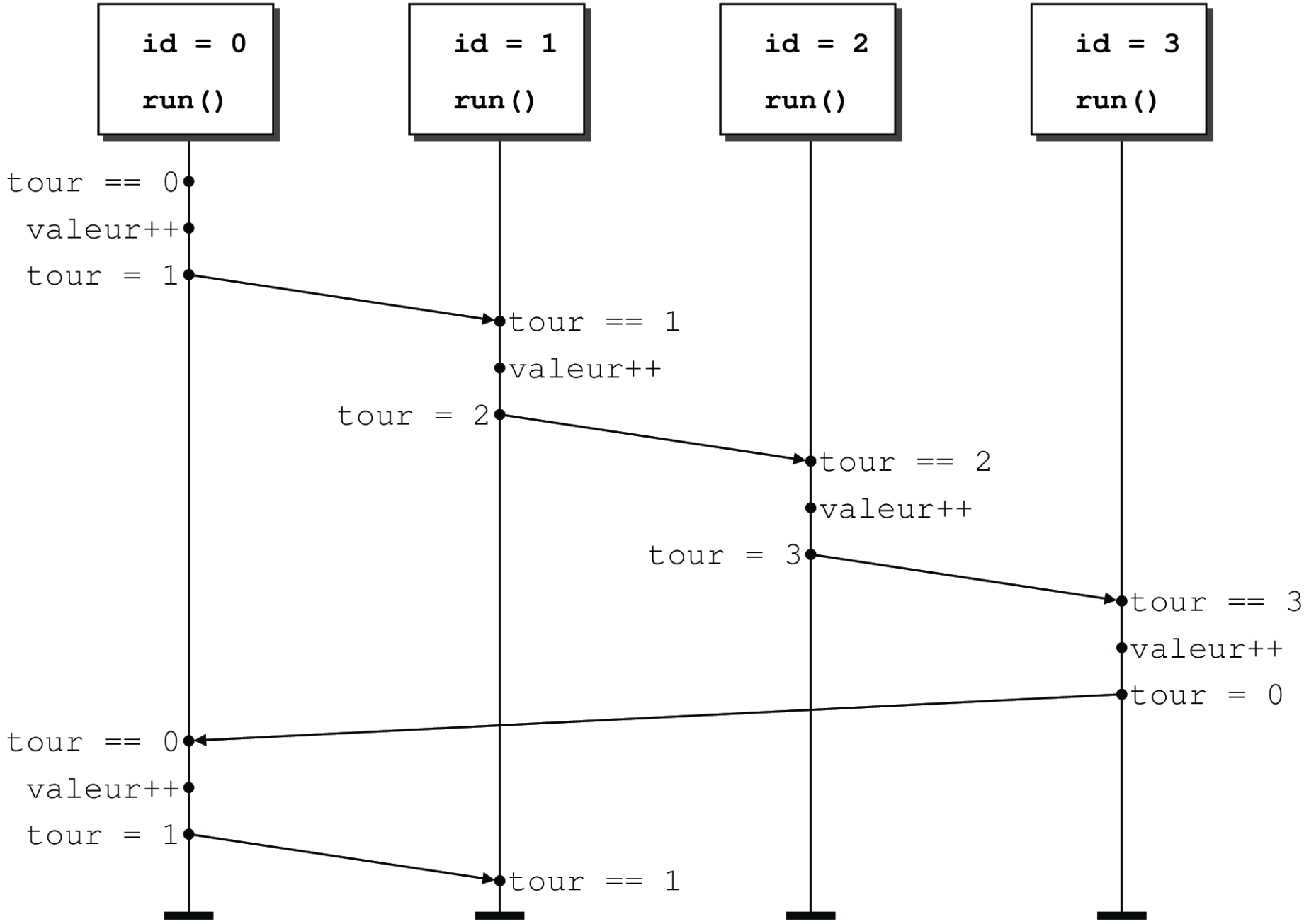
Les *verrous* associés aux objets en Java sont un outil fondamental pour écrire un programme correct en Java. La syntaxe de **`synchronized`** assure que chaque verrou pris sera relâché (à la fin du bloc).

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

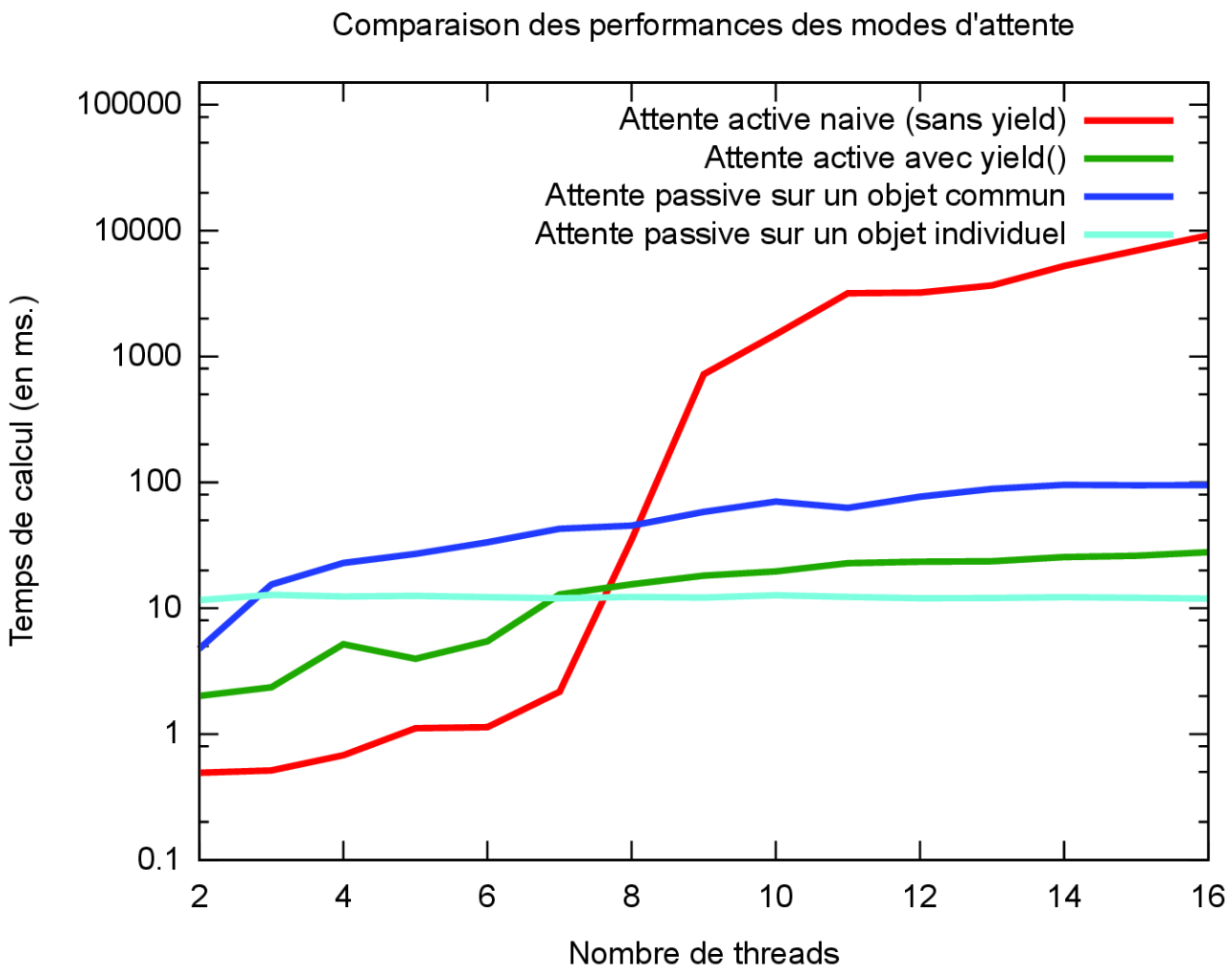
`wait()` permet à un thread *d'attendre sur un objet* jusqu'à ce qu'un autre thread lui lance un *signal*, via un appel **`notify()`** ou **`notifyAll()`** sur cet objet.

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 !*

Le benchmark adopté : les compteurs en rond



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.