Programmation Objet Concurrente

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2018-2019

Version du 12 septembre 2018

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 : $\mathtt{P}$

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

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 $\mathtt{ET'}$.

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

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 élement 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 ;

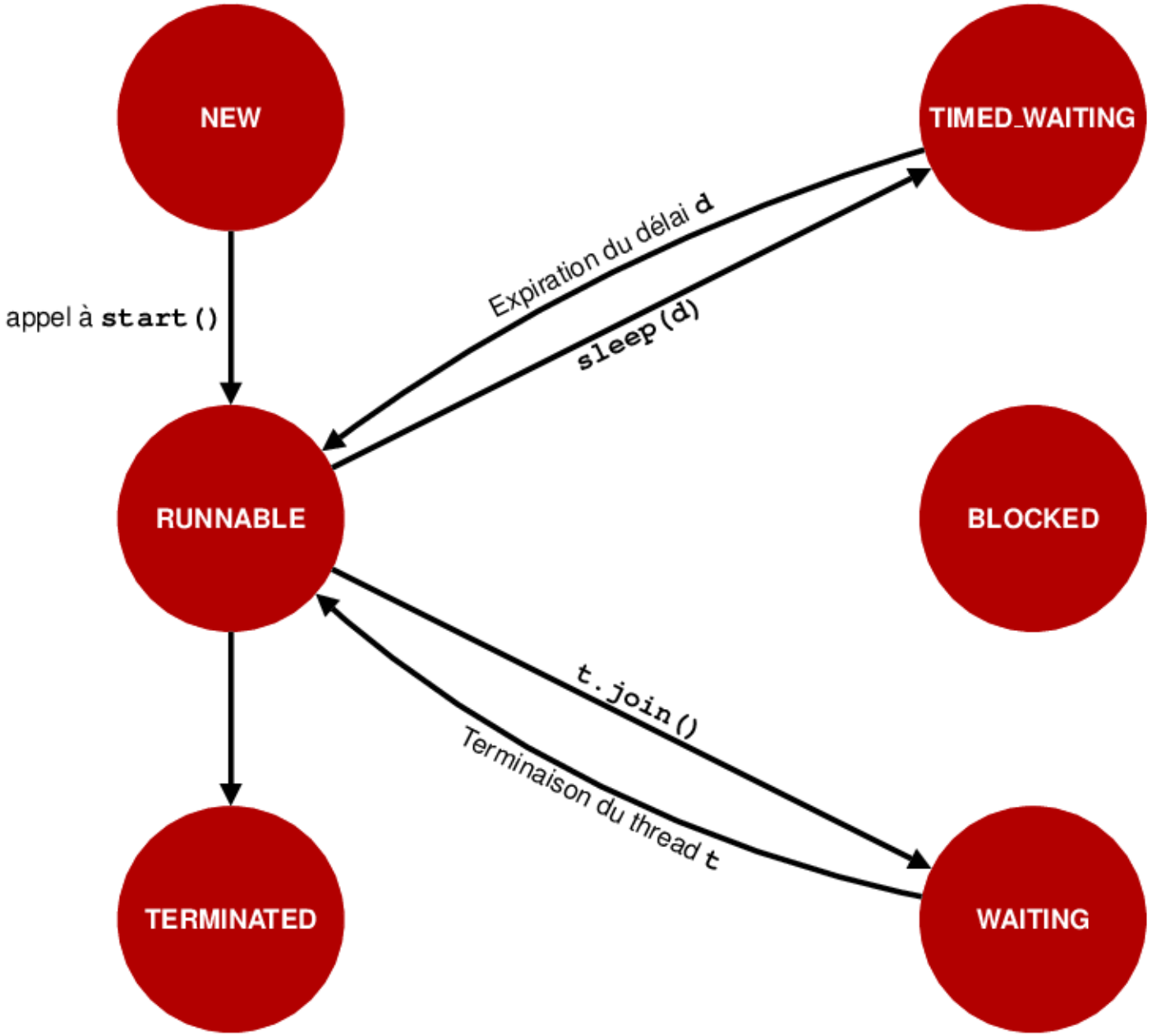
BLOCKED : il est bloqué en attente d'un *privilège logique*, par exemple de l'acquisition d'un *verrou* ;

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* ;

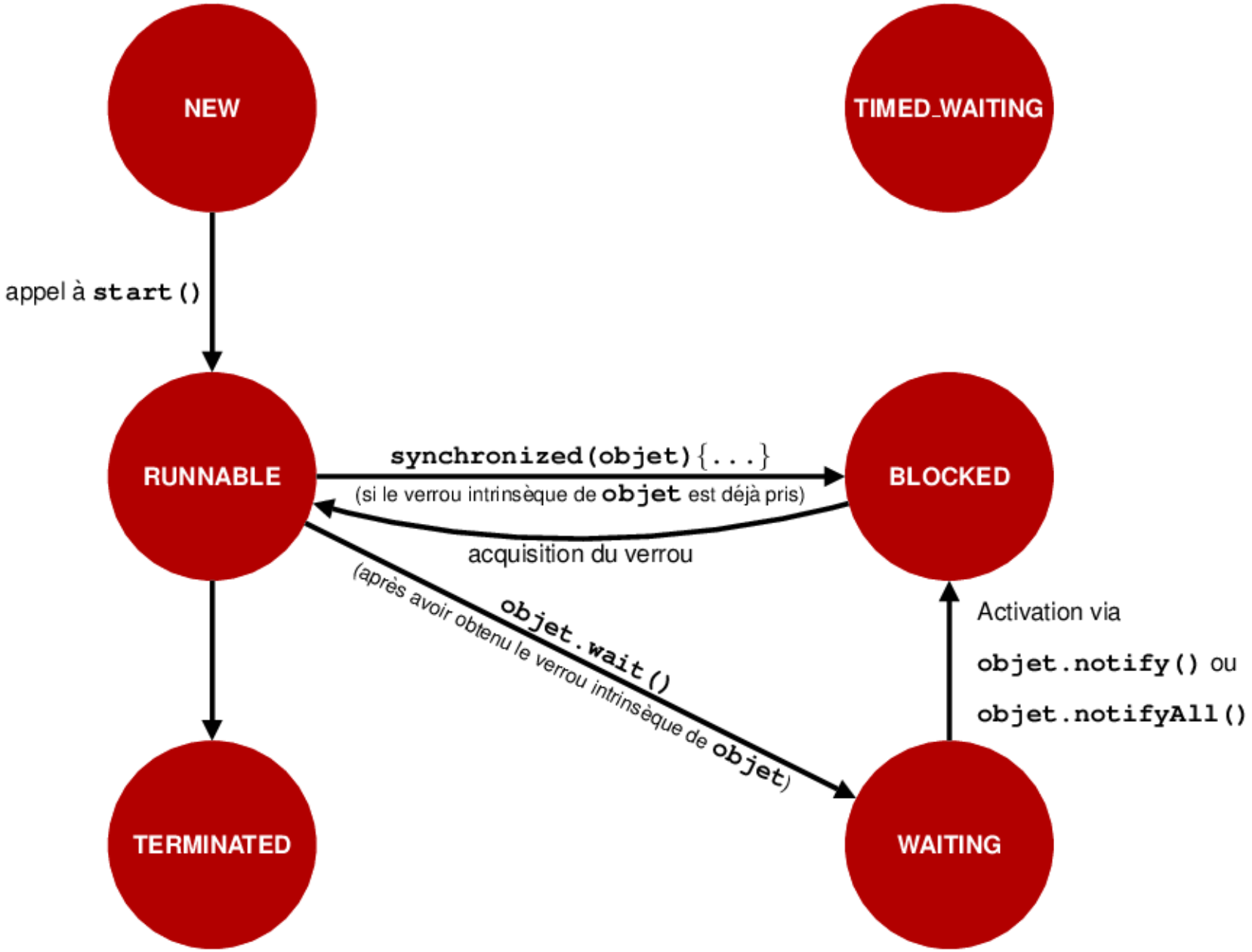
TIMED_WAITING : il attend qu'une durée s'écoule ou, éventuellement, qu'un évènement provoqué par un autre thread survienne ;

TERMINATED : il a fini d'exécuter son code.

Les six états d'un thread



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

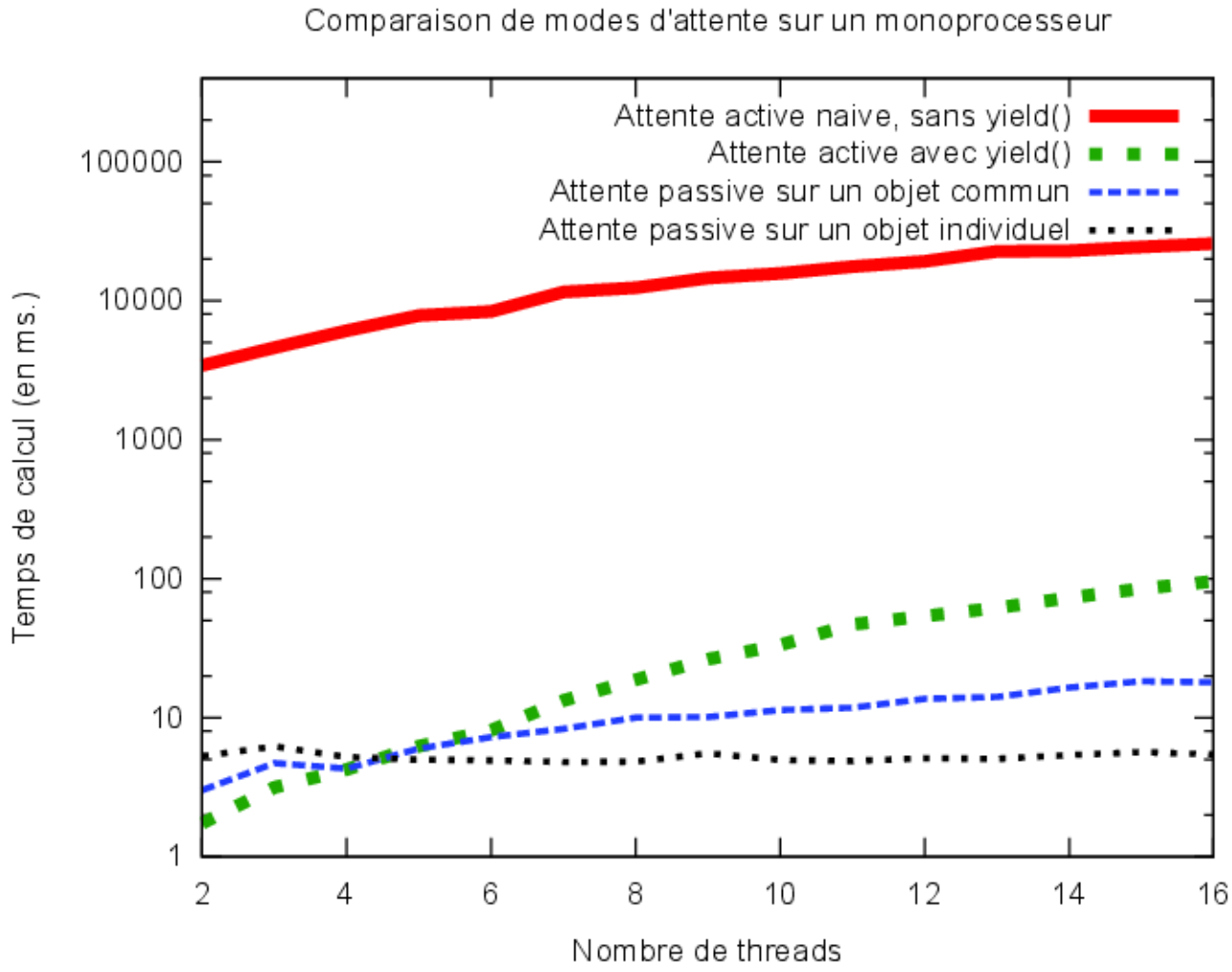
Les *verrous* associés aux objets en Java sont un outil fondamental pour écrire un programme correct en Java. La syntaxe de **`synchronized`** assure que chaque verrou pris sera relâché (à la fin du bloc).

`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.

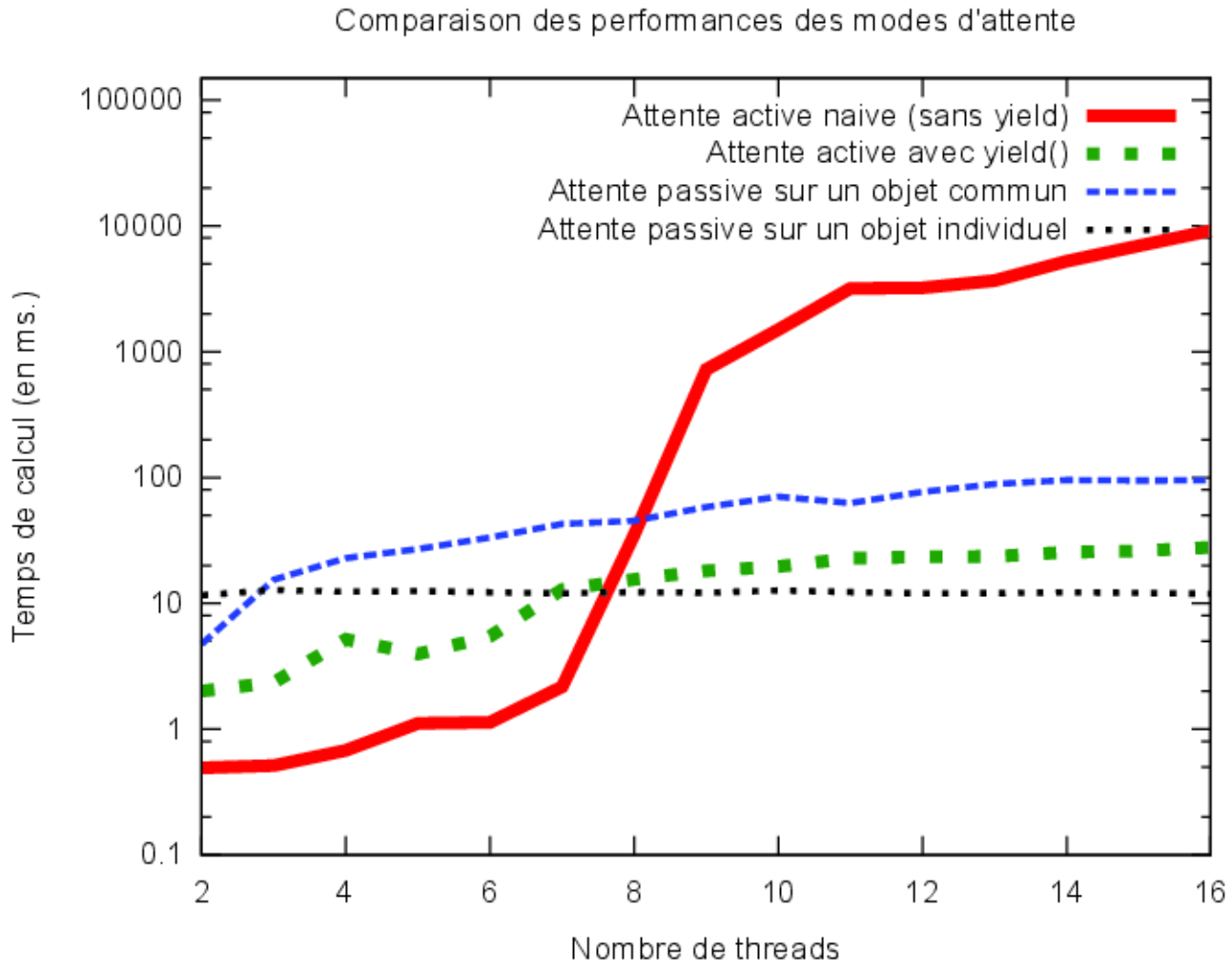
La méthode **`wait()`** nécessite d'acquérir au préalable le verrou intrinsèque de l'objet sur lequel elle est appliquée, à l'aide de **`synchronized`**. *Mais ce verrou est alors relâché !*
Les méthodes **`notify()`** et **`notifyAll()`** nécessitent également au préalable **`synchronized`**, mais *ces méthodes ne relâchent pas le verrou !*

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 faudra 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 à condition de ne pas insérer l'instruction **`yield()`**.

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.