

# Notes de cours de Programmation objet concurrente

Yann Miguel

28 septembre 2020

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Cours 1</b>	<b>3</b>
2.1	Un thread . . . . .	3
2.2	Éléments de synchronisation . . . . .	4
2.2.1	Synchronisation par verrou . . . . .	4
2.2.2	Synchronisation par signaux . . . . .	4
<b>3</b>	<b>Cours 2</b>	<b>5</b>
3.1	L'indépendance . . . . .	5
3.2	La notion d'atomicité . . . . .	5
3.3	Les variables atomiques . . . . .	5
<b>4</b>	<b>Informations importantes</b>	<b>6</b>

# 1 Introduction

Cette UE est importante car savoir utiliser plusieurs processeurs et threads est important dans l'informatique moderne.

Les threads sont aussi appelés processus légers, à l'opposés des processus lourds, qui contiennent leur propres données.

Les erreurs sont beaucoup plus communes en programmation parallèle.

Il y aura 5 cours de 2h:

1. les threads en java
2. le problème de l'atomicité
3. présentation des outils en java
4. programmation optimiste
5. aperçu du modèle mémoire java

## 2 Cours 1

### 2.1 Un thread

Un thread est un objet qui exécute du code. Afin d'écrire cela proprement, il faudra écrire des classes qui correspondront aux threads.

Le moyen le plus simple de faire ça est de créer une classe qui hérite de la classe Thread, ce qui oblige à créer une méthode run, qui sera exécutée par le thread. **IMPORTANT!** il faut utiliser start et non run pour lancer le thread.

Une autre approche, qui est peu recommandée, est de créer une classe qui hérite de Runnable, plutôt que d'hériter de Thread.

Une classe de Thread ou de Runnable peut contenir le main, et on peut lancer un thread en le créant.

Il existe une méthode nommée getName(), qui permet de récupérer le nom d'un thread. La méthode setName() permet de changer son nom. Pour accéder à un thread, on a besoin de sa référence, et non de son nom.

Les priorités d'un thread n'ont que peu d'utilité. Il servent à prioriser les threads, si jamais le système à besoin ne peut pas gérer tous les threads.

Un thread peut rencontrer des instructions qui le feront sortir du runnable, par exemple un verrou.

Si un thread est dans un verrou, il doit attendre l'autorisation pour continuer.

Il entrera dans un état TimeWaiting si il doit attendre un certain temps, ou bien dormir (méthode sleep(int ms)).

La méthode join() permet de mettre en pause un thread tant qu'un autre n'a pas fini.

Les méthodes sleep() et join() peuvent soulever une exception.

La méthode yield() pousse un thread à laisser d'autres s'exécuter. Son usage est déconseillé, car cela diminue la propreté du code.

Afin qu'un thread puisse s'exécuter, il faut qu'il ait accès à un cœur. Il n'y a à priori aucun intérêt d'avoir plus de threads que de cœur, sauf si cela peut simplifier le code.

## 2.2 Éléments de synchronisation

### 2.2.1 Synchronisation par verrou

Si plusieurs threads doivent toucher les mêmes données, on peut verrouiller les données à chaque fois qu'un thread les change. Cela permet de faire en sorte que seul un thread change les données à la fois.

Le mot clé "synchronized" permet de ne faire exécuter du code que si on a accès au verrou. Si un thread a besoin d'un verrou, mais ne peut pas y accéder, il passe en état blocked. Il repassera en état runnable que lorsqu'il aura accès au verrou.

Prendre un verrou est appelé "verrouiller", et libérer un verrou est appelé "déverrouiller".

En java, il est impossible de tenter de prendre un verrou si on utilise "synchronized". C'est à dire, que "synchronized" ne permet pas d'exécuter du code si le verrou n'est pas libre lorsqu'on le demande.

Lorsqu'une méthode est déclarée "synchronized", elle ne peut pas être exécutée si le thread qui veut l'exécuter n'a pas le verrou.

Si une méthode est synchronisée et statique, on demande le verrou de la classe plutôt que celui de l'objet.

### 2.2.2 Synchronisation par signaux

Le principe de la programmation par signaux est de régler les problèmes de type producteur-consommateur. Par exemple, le cas dans lequel un thread a besoin d'une donnée produite par un autre, il doit alors attendre que le thread "producteur" produise la donnée.

## 3 Cours 2

### 3.1 L'indépendance

La notion d'indépendance a été théorisée en 1966. Lorsqu'on a deux parties du programmes, on sépare chaque partie en deux parties:

- les variables lues par la partie
- les variables écrites par la partie

Pour que les parties soient indépendantes, il faut que:

- la partie 1 ne lise aucune variable écrite par la partie 2
- la partie 2 ne lise aucune variable écrite par la partie 1

Donc, que les deux parties ne touchent pas des variables similaires.

### 3.2 La notion d'atomicité

Dans ce cours, une instruction est dite atomique si et seulement si les variables lues ou modifiées par cette instruction ne peuvent pas être lues ou modifiées par le reste du programme durant son exécution. Avoir des instruction atomiques n'oblige pas le reste des threads à ne rien faire dans le programme.

Il y a deux types d'atomicité:

- les instructions atomiques de base du langage
- la création d'instruction atomiques avec un verrou rajouté par le codeur

En java, les écritures de références sont toujours atomiques.

Dans le cas ou on mets plusieurs verrous, afin d'éviter les inter-blocages, il faut décider un ordre des verrous dans le programme, et ne jamais le changer.

### 3.3 Les variables atomiques

Si la variable est déclaré comme atomique, AtomicInteger au lieu d'Integer, par exemple, cela permet à la variable d'avoir les avantages de l'atomicité, sans devoir créer de verrou dans le code.

## 4 Informations importantes

Dans cette UE, le projet sera les rendus de TP, et comptera pour  $\frac{1}{4}$  de la note finale.

Les  $\frac{3}{4}$  suivant seront l'examen final.

Durant la seconde session, la note de projet sera gardée seulement si elle augmente la note finale.

MCC session2:  $\max(\frac{1}{4} \text{ TP} + \frac{3}{4} \text{ ET}, \text{ ET})$

L'examen terminal sera sur papier. Les compte rendus de TP seront un exercice à rendre.