UNIVERSITÉ LIBRE DE BRUXELLES ULB

FACULTÉ DES SCIENCES
DÉPARTEMENT D'INFORMATIQUE

Les threads en Java8 et C++17

Nicolas Potvin *

Lundi 19 février 2018

Résumé

To be done

1 Introduction

2 A propos des threads

2.1 Qu'est-ce qu'un thread ?

Un thread, dans le jargon informatique, est ce qu'on appelle en français un "fil d'exécution". Il s'agit de la succession d'opérations nécessaires à l'exécution d'un programme (ou d'une partie d'un programme). Lorsqu'un programme s'exécute, nous parlons de processus. Un processus contient toujours au moins un thread qui représente le fil d'exécution principal du programme. Cependant, il peut aussi contenir plusieurs threads lorsqu'une parallélisation du traitement des données est possible. A titre d'exemple imaginons le processus « Changer une roue de voiture », un seul garagiste peut changer les quatre roues l'une après l'autre. Mais deux garagistes iraient deux fois plus vite car il n'est pas nécessaire d'avoir fini de changer la première roue pour changer la deuxième. Dans cet exemple les garagistes représentent les threads. Nous pourrions même aller jusque quatre threads pour changer toutes les roues sur le temps qu'il faut pour n'en changer qu'une.
Lorsque nous programmons, c'est ainsi que nous réfléchissons, nous pensons faire des programmes rapides et performants grâce à la parallélisation. Malheureusement, dans un ordinateur, les choses ne sont pas toujours aussi simples. Il est tout à fait possible qu'un programme multi-thread prenne en réalité le même temps qu'un programme mono-thread, par exemple à cause de l'architecture du processeur. Les parties des différents threads seraient alors exécutées les unes après les autres, et la parallélisation n'aurait pas réellement lieu.
Lorsque nous parlons de performances, ce genre de considération entre en jeu, mais dans cet article nous nous concentrerons sur la manière dont les threads sont implémentés en Java et en C++. L'exécution réelle est laissée au système d'exploitation qui décidera que faire en fonction de la machine sur laquelle le programme s'exécute.
Dans tous les langages, les threads se manipulent de manières différentes, mais il y a cependant des points communs. Lorsqu'un thread est créé, il faut démarrer son exécution, généralement via une méthode *start*. De manière équivalente, il existe souvent une méthode opposée *stop* ou autre pour arrêter un thread. Il est important de noter que stopper un thread ne devrait se faire que lorsque le processus est sur le point de s'arrêter pour éviter les problèmes d'inconsistance (source ?). Une dernière méthode importante est la méthode *join* qui sert à fixer ce qu'on appelle un "rendez-vous". Imaginez l'exemple du changement de roues dans le cas d'une course de formule 1, si le pilote et les techniciens représentent tous des threads, il faut que le pilote attende que tous les techniciens aient fini avant de démarrer. Un rendez-vous est exactement ça, un thread qui attend qu'un (ou plusieurs) autre thread arrive à la fin de leur execution avant de repartir.

2.2 Problèmes de concurrence

Deux threads d'un même processus peuvent accéder tous les deux à des zones de la mémoire réservées à ce processus. Cette particularité est très pratique lorsque nous voulons que deux threads puissent communiquer. Cependant un problème peut survenir lorsqu'un thread lit l'information stockée en mémoire pendant qu'elle est modifiée par un autre thread. Ce scénario est ce qu'on appelle une "data race". D'autres scénarios problématiques sont possibles, mais ils ont tous comme point commun qu'ils sont causés par un manque de synchronisation des threads.
La solution au problème se trouve dans les "opérations atomiques". Une opération atomique est une opération qui est exécutée en entier par le

*Université libre de Bruxelles (ULB) <nicolas.potvin@ulb.ac.be>

processeur sans être interrompue, en d'autres mots, elle est incassable en plusieurs sous-opérations. Cette caractéristique permet de s'assurer que si un thread effectue une opération atomique, il sera le seul à l'effectuer.

Les mutex

Pour garantir la consistance de la mémoire, une solution est l'utilisation des *mutex*. Un mutex se comporte un peu comme le verrou d'une cabine. Lorsque quelqu'un entre, il ferme le verrou, empêchant quiconque de rentrer. Il ne l'ouvrira que lorsqu'il aura terminé et qu'il sortira, laissant la place à une éventuelle autre personne attendant au dehors. Nous pouvons protéger un accès à une zone de la mémoire grâce à ce genre de mécanisme, les mutex présentent toujours au moins deux opérations *lock* et *unlock* (ou *release*) qui permettent de fermer le verrou et de l'ouvrir. Le plus souvent, les mutex proposent aussi une troisième opération de base *try-lock* permettant d'essayer de rentrer, et si le verrou est fermé, le thread n'attend pas (ne reste pas bloqué) et peut faire autre chose. Toutes ces opérations ont la particularité qu'elles sont atomiques, condition *sine qua non* du bon fonctionnement d'un mutex.
Plusieurs types différents de mutex existent, en plus des mutex "simples" décrits précédemment, nous parlerons des mutex récursifs. Ceux ci se comportent de la même manière à la différence qu'ils comptent le nombre de fois que l'opération *lock* leur aura été appliquée et ne se libéreront qu'une fois qu'ils auront reçu un nombre égal d'opérations *unlock*. En reprenant l'analogie de la porte à verrou, un mutex récursif serait une porte avec plusieurs verrous.

Les sémaphores

Jusqu'à présent nous avons considéré un mutex (simple) comme un verrou. Nous pouvons adopter un autre point de vue et le voir comme un compteur représentant le nombre de places libre. Il vaudrait 1 lorsque la place est libre, et 0 lorsque la place est prise. Les threads ne peuvent donc accéder à la zone protégée que lorsque le mutex vaut 1. Mais parfois, il est possible qu'il y ait plus qu'une place possible. Le sémaphore fonctionne exactement ainsi. Lorsqu'il vaut 0 et qu'un thread essaie d'entrer, il devra attendre. Sinon, il décrémentera le compteur et entrera. Lorsqu'un thread a fini son travail dans la zone protégée, il sortira en incrémentant le compteur.

2.3 Les threads en C++17

C++ fait correspondre les threads du programme avec les threads du système d'exploitation. Intuitivement, il pourrait s'agir de la manière la plus efficace de procéder. Ainsi les différents fils d'exécution sont gérés directement pas l'OS. (p1362)

Classe thread

Dans le langage C++, les threads sont manipulés à travers des objets de la classe thread. Il est important de garder à l'esprit que toute instanciation de la classe thread ne représente pas un fil d'exécution si il est créé via le constructeur par défaut. De manière équivalente, un objet thread utilisé pour lancer un thread ne représentera un fil d'exécution réel que tant que ce fil n'a pas terminé son exécution (qu'il a des tâches à effectuer) ou que le thread n'est pas détaché. Un thread est détaché lorsque le fil d'exécution poursuit son exécution mais n'est plus représenté par un objet thread instancié.
De manière générale, nous pouvons voir ces objets thread comme une interface entre l'OS et le programmeur pour manipuler facilement les différents fils d'exécution. La méthode *join* expliquée précédemment est implémentée. La méthode *start* n'est pas implémentée explicitement, lorsqu'un thread est créé via un constructeur spécialisé ou un constructeur de transfert, il est automatiquement lancé. La méthode *stop* n'est pas implémentée du tout. Nous pourrions penser que le destructeur pourrait tenir ce rôle, mais il ne détruit l'objet thread que si il est détaché ou si le fil d'exécution est terminé. Le programmeur qui souhaite tout de même avoir un contrôle sur l'exécution du thread doit implémenter lui même un *stop* (par exemple via un flag booléen partagé qui serait vérifié régulièrement par le thread secondaire qui mettrait fin à son exécution lui même si le flag venait à changer). L'avantage qu'en tire le programmeur est qu'il gère lui même la manière dont le thread s'interrompt, évitant qu'il soit brutalement interrompu, risquant de laisser la mémoire dans un état inconsistant.

Gestion de la concurrence

C++ propose une classe *mutex* qui implémente des méthodes *lock*, *unlock* et *try_lock* décrites précédemment. Une classe *recursive_mutex* existe aussi, elle implémente les

trois même méthodes et fonctionne exactement comme les mutex récursifs expliqués ci-avant. Il existe également *timed_mutex* dont la seule différence par rapport à un mutex normal est que le programmeur peut utiliser la méthode *try_lock* avec un délai qui correspond à un temps d'attente pendant lequel le thread courant essaiera d'acquérir le mutex.
shared_mutex ?

2.4 Les threads en Java

Contrairement à C++, une fois compilé, le code Java ne tourne pas directement sur la machine, mais plutôt sur une machine virtuelle. Nous nous intéresserons d'abord au langage avant d'observer de plus prés ce qui se passe au niveau de la machine virtuelle.

Classe *Thread* et interface *Runnable*

En Java, il existe une classe *Thread* qui constitue le seul moyen offert par le langage pour manipuler des fils d'exécution. Cette classe implémente l'interface *Runnable* qui ne contient qu'une méthode, la méthode *run*. Nous pouvons voir cette méthode comme la méthode *main* d'un processus, il s'agit en quelque sorte du point d'entrée du *thread*. Le code exécuté par le thread sera celui écrit dans l'implémentation de *run*.
Pour instancier un *thread* il faut soit coder une sous classe de *Thread* soit instancier *Thread* directement en passant au constructeur un objet implémentant l'interface *Runnable*, dans ce cas, la méthode *run* du thread sera celle de l'objet. Ces deux méthodes ont leurs avantages et leurs inconvénients. Une classe héritée de *Thread* a l'avantage de pouvoir redéfinir les méthodes de *Thread*, permettant un contrôle plus fin du *thread*. Implémenter l'interface *Runnable* permet de le faire à la volée lors de la création du thread si les tâches à effectuer ne demandent pas trop de code, évitant de se perdre dans des fichiers contenant de nouvelles classes. Un autre avantage beaucoup plus pratique est qu'une classe anonyme créée à la volée au sein d'une autre classe permet d'accéder aux attributs de cette classe. Il est toujours possible de coder une nouvelle classe implémentant l'interface si le programmeur n'a pas besoin de redéfinir les autres méthodes de *Thread*. Dans ce cas les deux manières de faire se valent.
Lorsqu'un *thread* est créé, il faut explicitement le démarrer avec la méthode *start*. Il existe aussi une méthode *stop* mais elle est dépréciée, pour les raisons évoquées dans la section C++ ; arrêter un *thread* en pleine exécution risque de laisser des parties de la mémoire dans un état inconsistant, compromettant la bonne marche de tout le processus. La méthode *join* est implémentée et fonctionne exactement comme expliqué plus haut, Java propose également une surdéfinition de cette méthode avec une durée en argument, permettant d'attendre que le *thread* termine durant un certain temps sans bloquer éternellement.

Concurrence et synchronisation

Java introduit la notion de *moniteur* ; en pratique, un moniteur se manipule comme un *mutex récursif* via les méthodes *lock* et *unlock*. Chaque objet instancié en Java se voit associer un moniteur(p660), ce qui veut dire que chaque objet peut être utilisé comme un *mutex récursif*.
Il est ensuite possible de coder plus ou moins comme en C++ en utilisant les moniteurs des objets avec lesquels le programmeur travaille. Mais Java nous permet de gérer la synchronisation de manière plus sûre grâce au mot-clé *synchronized*. Ce mot du langage s'utilise juste avant un bloc d'instruction, avec en argument une référence vers un objet qui servira de moniteur pour ce bloc. Si il est utilisé dans la signature d'une méthode, l'objet qui implémente cette méthode sert de moniteur, si la méthode est statique, le moniteur utilisé est associé à la classe.(p660)
parler des wait sets, wait, notify et notifyAll ?
parler de volatile ?