Java Avancé

Fiche 8 : Threads

Table des matières

1 Objectifs

ID	Objectifs
AJ01	Ecrire des threads avec la classe Thread et l'interface Runnable
AJ02	Résoudre les problèmes de concurrence à l'aide du "synchronized"
AJ03	Se rendre compte des problèmes de deadlock

2 Concepts

2.1 Introduction

La programmation que nous avons abordée jusqu'à présent repose sur un seul **flux d'exécution**. Cette approche pose des limites dans certains cas :

- Lorsque le programme interagit avec le monde extérieur, il se retrouve souvent à attendre que ce monde extérieur réagisse ; par exemple, pour écrire un bout de fichier sur un disque ou encore pour obtenir une réponse d'un utilisateur.
- De plus en plus les ordinateurs sont équipés de plusieurs processeurs. Chaque processeur peut traiter son propre flux d'exécution, mais si le programme n'en possède qu'un, un seul processeur sera réellement utilisé et le programme n'utilise qu'une fraction de la puissance de la machine.
- Certains programmes s'expriment naturellement beaucoup mieux à l'aide de plusieurs flux d'exécution. Un serveur web pourra traiter chaque requête dans un flux indépendant des autres requêtes en cours de traitement.

Utiliser des processus du système d'exploitation pour résoudre ceci génèrerait notamment beaucoup de problèmes de collaboration entre eux car ils seraient indépendants et gérés par le système d'exploitation.

2.2 Thread

En programmation, peu importe le langage, on peut procéder à l'exécution de différents codes dans différents flux d'exécution en même temps (ou à des moments différés). Un flux d'exécution est appelé un **thread** en informatique.

Un thread est une version allégée de **processus**. L'idée est qu'un seul processus exécutera directement plusieurs threads. L'avantage de cette approche est multiple :

- Les threads sont légers comparés aux processus : beaucoup plus rapides à démarrer, beaucoup moins couteux en mémoire.
- Les threads partagent le même espace d'adressage et peuvent donc se partager des données et du code. Donc le coût de communication entre les threads est relativement bas par rapport à celui entre les processus.
- Le fait pour un processeur de passer de l'exécution d'un thread à un autre (context switch en anglais) est moins coûteux que pour les processus.

3 Exercices

3.1 Introduction

L'implémentation des Threads en Java a été vue au cours de COO. Au besoin, une fiche de théorie est disponible comme support ; elle s'intitule « Concurrence ».

Pour les exercices, téléchargez et ouvrez le projet « workshop 8 » fourni sur Moodle.

3.2 Thread et Runnable

On va implémenter une classe dont l'objectif est de faire une course de compteurs. Un compteur a un nom et permet de compter de 1 à max (nombre entier positif fourni lors de la création du compteur). Entre chaque nombre, le compteur marque une pause de 10 millisecondes. Il affiche alors son nom et le nombre (par exemple, "Bolt : 7"). À la fin, il affiche un message du type "Bolt a fini de compter jusqu'à xx". Les classes à compléter se trouvent dans le package `compteur`.

a) Classe Thread

La première façon d'implémenter un thread est d'étendre la classe prédéfinie Thread :

1. Complétez la méthode `run()` de la classe `CompteurThread` qui hérite de Thread. Le nom du compteur qui a gagné doit être affiché (attendre que tous aient fini). Le compteur gagnant doit en plus afficher un message du type "Bolt a gagné". Il faut conserver pour cela l'instance gagnante dans un attribut de classe `CompteurThread`. Mais attention cet attribut peut être lu/modifié par plusieurs threads en même temps.
2. Complétez la classe de tests `TestCompteurThread` qui lance plusieurs compteurs qui comptent jusqu'à 10. Voyez celui qui a fini le plus vite.

b) Interface Runnable

L'héritage à partir de Thread est contraignant car il empêche tout autre héritage. Une deuxième manière de faire est d'implémenter l'interface Runnable. Ceci permet l'héritage d'une autre classe et l'implémentation d'autres interfaces.

1. Complétez la classe `CompteurRunnable` qui étend l'interface Runnable. Cette fois-ci, on ne cherche plus le gagnant. À la place, chaque compteur va devoir afficher son ordre d'arrivée : le message de fin est du type : "Bolt a fini de compter en position 3". La position pourrait être conservée dans un attribut de classe de type `int`. Utilisez la classe `TestCompteurRunnable` qui lance plusieurs compteurs qui comptent jusqu'à 10 pour voir le résultat.

3.3 Race Condition

Classe Thread

Reprenez la classe compteurs à base de Thread du point précédent.

Faites la modification suivante à votre code :

Nous vous demandons de faire une pause de 10 ms entre chaque incrémentation du compteur et lors de l'enregistrement du gagnant d'attendre 10 ms avant de l'enregistrer.

Un problème devrait apparaître. Quel est-il, d'où vient-il ?

Corrigez ce problème. Au besoin relisez la section « Race condition » de la théorie.

Interface Runnable

Procédez à la même modification qu'au point précédent. Résolvez les problèmes. Pour résoudre les problèmes sur le compteur de position des Runnables, des types spéciaux, dits atomic, peuvent être utilisés dans ce genre de cas. Pour plus d'information, rendez-vous sur la documentation du package java.util.concurrent.atomic.

3.4 Race condition 2

Dans le package `abonnements`, vous trouverez 4 classes :

a. `Compte` : cette classe contient un solde et un historique des dépense. Sa méthode `depenser` permet d'enregistrer une dépense et de modifier son solde. Sa méthode `verifier` permet d'afficher l'historique pour vérifier que les opérations sont bien légales.
b. `Depense` : cette classe représente une dépense. Elle a un nom et un montant.
c. `Abonnement` : un abonnement est une dépense régulière (ici mensuelle) à enregistrer sur un compte. Les abonnements sont des threads qui s'exécutent tant que le solde du compte est suffisant.
d. `TestAbonnements` : cette classe crée 5 abonnements pour un compte, lance les threads, attend la fin de leur exécution et puis vérifie l'état du compte.

Le solde du compte ne peut pas être négatif puisqu'un abonnement s'arrête si le solde n'est pas suffisant pour le payer. Malheureusement, on peut voir que ce n'est pas le cas ici : des abonnements réussissent à prendre de l'argent malgré tout et le solde final est bien souvent négatif !

Modifiez le programme pour éliminer ce problème de race condition.

Votre solution doit être équilibrée : il faut éviter qu'un abonnement ne vide le compte à lui tout seul !

3.5 Deadlock

Lors de la réunion annuelle des magiciens, ceux-ci se réunissent et doivent lancer ensemble trois sorts. Pour pouvoir lancer ces sorts ils ont besoins d'ingrédients magiques qui n'existent qu'en un seul exemplaire. Ces ingrédients peuvent toutefois être réutilisés dans plusieurs incantations. Les magiciens sont obligés

- de prendre ces ingrédients un par un
- de méditer après avoir pris chaque ingrédient
- de conserver ces ingrédients jusqu'à la fin de l'incantation

L'ordre dans lequel ils prennent ces ingrédients n'a pas d'importance, vous pouvez donc les intervertirs. Une fois tous les ingrédients nécessaires réunis ils peuvent lancer l'incantation.

Par exemple voici l'incantation de « Calme infini » par Gandalf :

LARME_DE_FEE pris par Gandalf. Je vais méditer

POIL_DE_TROLL pris par Gandalf. Je vais méditer

CUISSE_DE_GRENOUILLE pris par Gandalf. Je vais méditer

Calme infini lancé !

Vous trouverez dans le package `deadlock_magic` une classe `Grimoire` qui permet de lancer les incantations ainsi qu'une classe MagicTheGathering lançant les incantations.

Malheureusement les magiciens se marchent sur les pieds les uns des autres en prenant les ingrédients et se retrouvent coincés. Quel est le problème ? Au besoin relisez la section sur les deadlocks dans la théorie.

Résolvez le problème en tenant compte du fait qu'ils doivent absolument commencer à préparer les incantations en même temps. Vous ne pouvez donc pas changer les classe `MagicTheGathering` en revanche vous pouvez modifier le `Grimoire`.

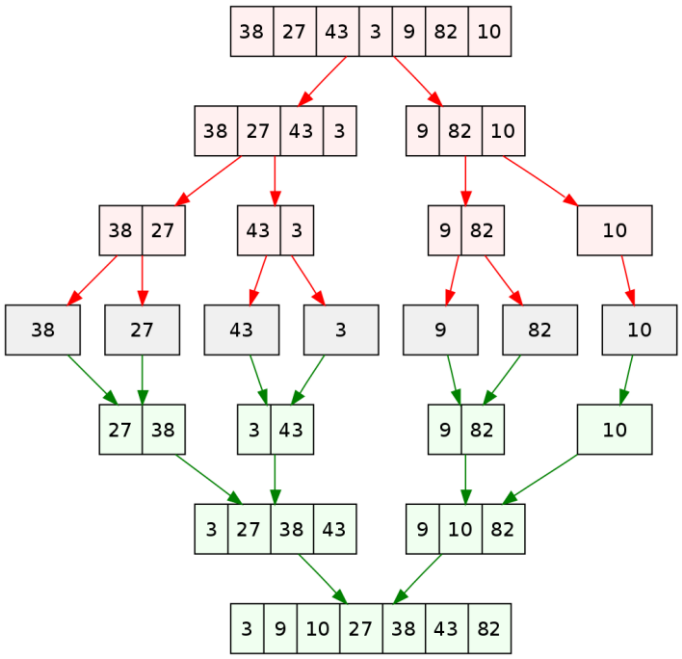
Veillez à expliciter la stratégie utilisée pour éviter les deadlock.

Vous devez à la fin avoir le messager « Toutes les incantations ont réussies !! ».

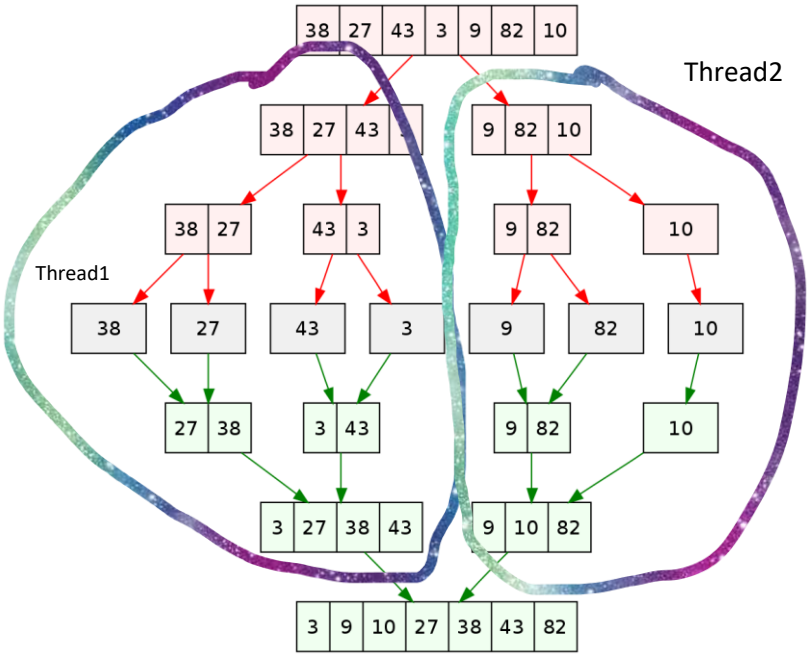
3.6 Tri fusion

Le package `tri_fusion` contient une version mono-thread d'un algorithme de tri bien connu : le tri fusion (merge sort).

Pour rappel, le merge sort consiste à diviser la table récursivement en 2 jusqu'à obtenir des tableaux triviaux à trier et d'ensuite de les fusionner (to merge) deux par deux lors de la remontée récursive comme l'illustre la figure ci-dessous :

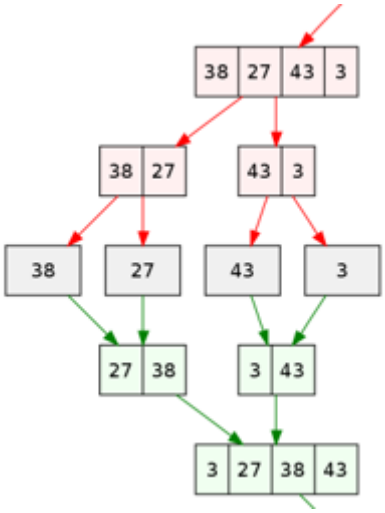


Si nous disposons de deux threads, nous pouvons séparer le tri en deux de la façon suivante :

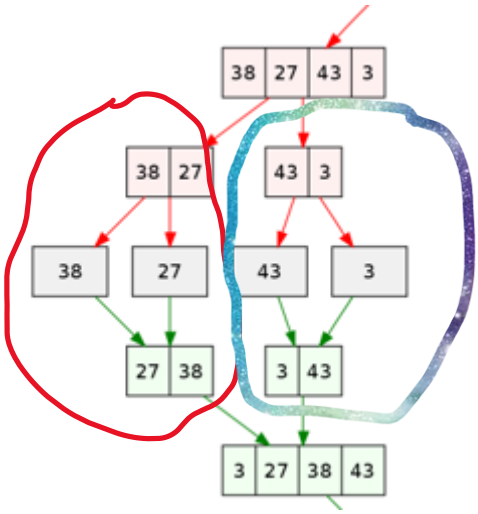


Nous voyons que les parties de tableaux manipulées par ces threads sont séparées. Il n'y a donc pas de problèmes de synchronisation. Par contre, le merge se fait dans le thread principal. Pour cela il faudra attendre que les deux threads aient fini de tirer leur partie de tableau.

Comme l'algorithme est récursif nous pouvons penser de façon récursive notre séparation en thread. Le thread 1 va faire les choses suivantes durant le tri :



On peut donc de nouveau créer récursivement deux nouveaux threads :



En utilisant la méthode `join()`, codez une version multi-thread de cet algorithme. Attendez la fin de l'exécution et affichez ensuite le résultat final.

Astuces :

- Le multi-threading devra être implémenté dans la classe Sorter.
- Pas besoin de modifier les méthodes `swap` et `mergeSort`.
- Ajouter un attribut `nbThreads` qui contiendra le nombre de « sous-threads workers » qui feront une partie du tri.

Une fois terminé, testez votre code en changeant légèrement la méthode main. Observez également ce qu'il se passe en modifiant les paramètres :

- Nombre de threads
- Taille du tableau

Comparez avec la version mono-thread.