

Fiche 8 : Threads

Table des matières

1	Objectifs.....	2
2	Concepts.....	2
2.1	Introduction.....	2
2.2	Thread.....	2
3	Exercices.....	3
3.1	Introduction.....	3
3.2	Thread et Runnable	3
a)	Classe Thread.....	3
b)	Interface Runnable	3
3.3	Race Condition	3
3.4	Race condition 2	4
3.5	Deadlock	4
3.6	Tri fusion.....	5

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 coûteux 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 « Concurrency ».

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](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html).

3.4 Race condition 2

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

- a. `Compte` : cette classe contient un solde et un historique des dépenses. 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

Dans l'antique et mystérieuse Académie de la Magie d'Etheria, trois Magiciens Élite s'apprêtaient à réaliser chacun une cérémonie puissante, connue seulement par ceux qui ont maîtrisé les anciens grimoires. Chaque cérémonie nécessitait l'usage exclusif de trois incantations sacrées pour invoquer un sortilège de grande envergure. Mais, un problème ancien guettait, connu sous le nom de "Deadlock des Anciens".

Lors de la grande conjonction des astres, les Magiciens Élite commencèrent leurs rituels :

- Le premier Magicien invoqua le "Symposium des Illusions Prismatiques", un sortilège de tromperie et de mirages.
- Le deuxième Magicien déploya l'"Assemblée des Ondes Frissonnantes", capable de geler le cours même du temps.
- Le troisième Magicien entama le "Rituel du Crépuscule Abyssal", sombre et mystérieux, pour voiler le monde dans les ténèbres.

Mais leurs pouvoirs étaient tellement grands et interconnectés que les incantations se bloquèrent mutuellement. Les énergies commencèrent à s'entremêler et à créer une tension dans l'air, menaçant l'existence même de l'Académie.

Les apprentis de l'Académie ont découvert que les Magiciens ne pouvaient terminer leurs rituels, car chacun attendait qu'un autre libère son incantation avant de pouvoir continuer. C'était un "Deadlock des Anciens", un cercle de magie fermé où chaque pouvoir en attendait un autre.

Chers étudiants, c'est maintenant à vous de briser ce cercle vicieux et de libérer les Magiciens de ce deadlock. Vous devez plonger dans le code du Grimoire et des Cérémonies se trouvant dans le package `deadloc_magic` pour trouver une solution qui permettra à toutes les cérémonies de se dérouler sans encombre.

Modifiez le code pour que les Magiciens Élite puissent tous lancer leurs cérémonies sans que leurs pouvoirs ne se bloquent mutuellement. La gloire et la reconnaissance de l'Académie d'Etheria attendent celui ou celle qui résoudra ce mystère!

Résolvez le problème en tenant compte du fait qu'ils doivent absolument commencer leurs cérémonies en même temps. En revanche, l'ordre des incantations n'est pas importante. Vous ne pouvez donc pas changer la classe `MagicTheGathering` en revanche vous pouvez modifier le Grimoire.

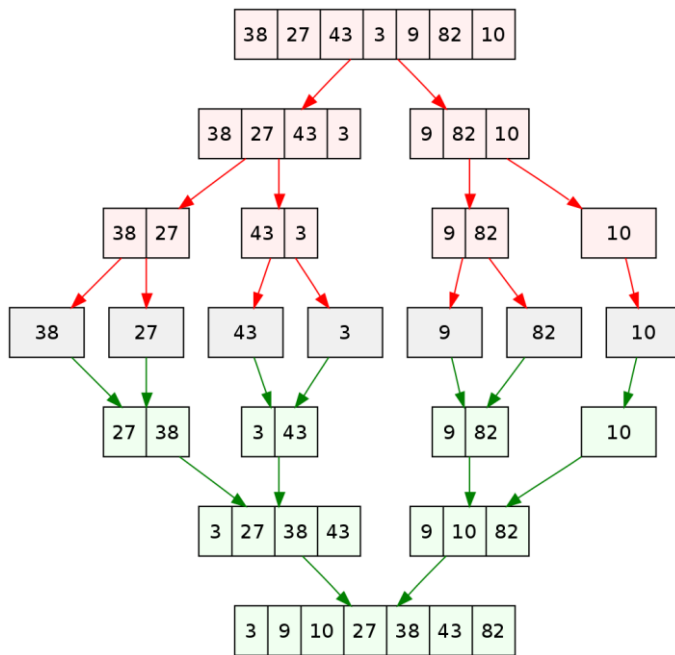
Veillez à expliciter la stratégie utilisée pour éviter les deadlock.

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

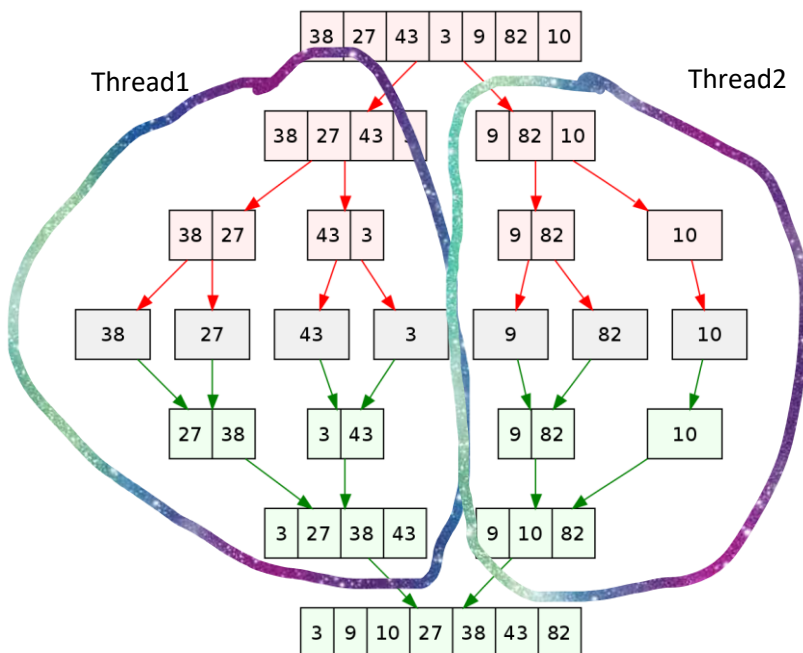
3.6 Tri fusion

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 :

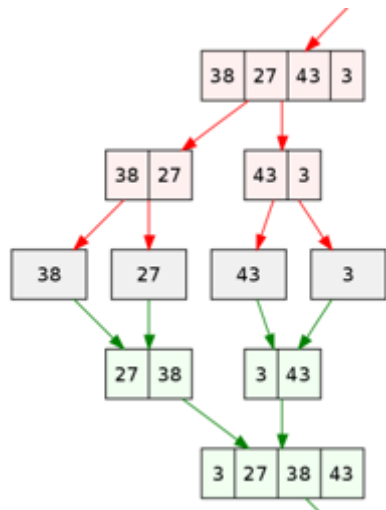


Si nous disposons de deux threads, nous pouvons séparer le tri en deux de la façon suivante :

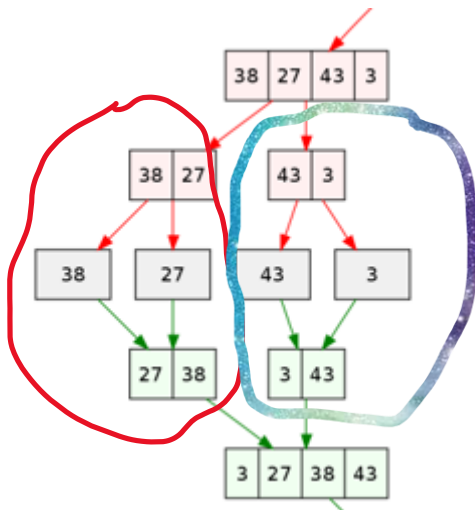


Nous voyons que les parties de tableaux manipulées par ces threads sont séparées. Il n'y a donc pas de problèmes de synchronisation. Par contre, le merge se fait dans le thread principal. Pour cela il faudra attendre que les deux threads aient fini de tirer leur partie de tableau.

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.