2019

Compte rendu de TD : Comprendre la concurrence

Concurrence E/S – TD n°2

Jonathan CRÉTÉ – Filière Informatique

Exercice 0 - When things add up

1) Récupérez la classe Counter.java et exécutez-laplusieurs fois.

Essayez d'expliquer ce que vous observez.

Les valeurs obtenues sont toutes différentes et supérieures à 10_000.En mémoire, le champ value vaut 10_000 (la résultat varie à chaque exécution)

La valeur de value est 0. La thread 1 lit cette valeur et il est déschedulé. Le thread2 est schédulé et il termine son runnable. Donc la valeur de value est 10_000

Le thread t1 est reschedulé et il vaut la valeur de 1.

Puis il termine son runnable et à la fin value vaut 10_000.

2) Est-il possible que ce code affiche moins que 10000 ? Expliquer précisément pourquoi.

On ne verra jamais moins de 10_000 dans la console, démontrons-le avec un scénario possible:

- Si t1 **lit en mémoire** 0, il **incrémente et écrit en mémoire** 3000 (*par exemple*).
- t1 est **déschédulé**.
- t2 est **schédulé,** lit 3000 sur le tas et **incrémente** jusqu'à 5000. Il est **déschédulé**.
- t1 est **re-schédulé**. il **lit** 3000 en mémoire, incrémente et **stocke 10_000 en mémoire**.
- t2 est **re-schédulé** et "écrase" en écrivant 5000 en mémoire. Il **incrémente** en imaginant qu'il reste 8000 tour de boucles.
- t2 **écrit en mémoire** 13_000
- t2 est déschédulé.
- la JVM s'arrête si tous les threads sont **morts.**

En fin de compte, 10_000 est le minimum possible en mémoire quoi qu'il arrive (le processus arrivera à son terme). Si un thread est déschedulé alors qu'il a déjà lu la valeur, alors on perd la valeur de value qui a été mise à jour (l'incrémentation n'est pas une opération atomique)

Exercice 1 - Stop now !

1) Récupérer la classe StopThreadBug.java. Avant de l'exécuter, essayer de comprendre quel est le comportement espéré. Où se trouve la data-race ?

La data-race (*zone mémoire partagée par deux thread*) est stop. En effet, Le main et le thread essaie d'écrire le stop.

2) Exécuter la classe plusieurs fois. Qu'observez-vous ?

On a aucun contrôle sur l'ordre d'exécution des threads.

3) Modifiez la classe StopThreadBug.java pour supprimer l'affichage dans la boucle du thread. Exécuter la classe à nouveau plusieurs fois. Essayez d'expliquer ce comportement.

println (E/S) est une opération lente. Le JIT attend un certain nombre de boucles pour optimiser (l'optimisation coûte cher).

Ici, Le JIT a optimisé le code. il créer une variable locale stop initialisée à false, et comme la variable est toujours à false, on ne sort jamais de la boucle. On doit donc arrêter le thread (Celui implémentant Runnable) à la main.

4) Le code avec l'affichage va-t-il toujours finir par arrêter le thread ?

Pas forcément, on ne sait pas comment le scheduler va se comporter.

Exercice 2 – Les fourberies du JIT

1) Quand on exécute le code précédent, quels peuvent être les différents affichages constatés ?

a = 0 b = 0 -> Le main s'est éxécuté en entier, le main puis le thread

a = 1 b = 2 -> Le thread puis main

a = 1 b = 0 -> Le main jusqu'a après a = 1 puis thread

a = 0 b = 2 -> main jusqu'à b = 2 puis thread (*Le JIT a changé l'ordre des affectations*)

2) Quand on exécute le code précédent, quels peuvent être les différents affichages constatés ?

On observe une data-race sur le champ l. L'affectation se fait en deux étapes ce qui peut être interrompu avec le scheduler.

Long signifie potentiellement deux opérations (*Pas atomique*).

Valeurs possibles :

FFFF FFFF

0000 FFFF

FFFF 0000

0000 0000

Exercice 3 – When things pile up

1) Recopiez cette classe dans une nouvelle classe HelloListBug puis modifiez-la pour y ajouter les nombres dans la liste au lieu de les afficher. Faites afficher la taille finale de la liste, une fois tous les threads terminés.

On modifie le code :

```
...
var list = new ArrayList<Integer>(5000 * nbThreads);
...
list.add(i *nbThreads);
...
```

2) Exécuter le programme plusieurs fois et noter les différents affichages.

On note les différents affichages sur 4 exécutions :

--> 19037

--> 20000

--> 9229

--> 13655

3) Expliquer comment la taille de la liste peut être plus petite que le nombre total d'appels à la méthode add.

On a un problème de data-race avec la size de ArrayList (C'est globalement le même problème que pour l'incrémentation)

Imaginons un scénario possible :

Deux threads t1 et t2 sont en exécution et appellent la méthode add(...) de ArrayList

- *Le thread 1* récupère le prochain index libre de la liste (index 0)
- Le *thread 1* est **deschedulé**, on passe au thread 2
- Le thread 2 récupère le prochain l'index libre de la liste soit l'index 0 (le même qu'a récupéré le *thread 1*, car ce dernier n'a pas eu le temps de mettre quelque chose à jour car il vient d'être **deschedulé**)
- *Le thread 2* ajoute l'integer dans la liste à l'index 0
- Le *thread 2* est **deschedulé**, on passe au thread 1
- Le *thread 1* ajoute l'integer dans la liste à l'index 0 (*et écrase du coup la valeur qu'avait ajoutée thread 2*)

On observe donc que la liste à une taille de longueur 1 alors que la méthode *add(...)* a été appelée 2 fois

4) Modifier votre code pour ne par fixer la capacité initiale de la liste.

```
var list = new ArrayList<Integer>();
```

5) Exécuter le programme plusieurs fois. Quel est le nouveau comportement observé ? Expliquer quel est le problème. Là encore, il faut regarder le code de la méthode ArrayList.add.

Une Exception *ArrayIndexOutOfBoundsException* est levée car on a dépassé la taille de notre objet ArrayList. *« Index 6790 out of bounds for length 4164*»

Comme abordé dans le scénario précédent, on cherche à insérer à l'index 6790 qui n'existe pas dans notre ArrayList, car l'appel à la méthode add(...) entre les deux threads n'est pas synchronisé. En revanche, la taille de la liste **à bien augmentée (**4164, une copie de la liste avec une nouvelle capacité est effectuée**), mais reste inférieur à la l'index demandée** (dû à la méthode *grow() appelée par add)* .