Formation : INFO2

INFO
INGÉNIEUR
ESIPE

Rapport du TD n°02
Concurrence

Guillaume CAU
09/10/2019

UNIVERSITÉ PARIS-EST
MARNE-LA-VALLÉE

Esipe
INGÉNIEURS PARIS EST

INGENIEURS
2000
L'EXCELLENCE AU SERVICE DE L'ALTERNANCE

Exercice 00 :

Après plusieurs exécution de la méthode main de la classe *Counter*, je remarque que la valeur du compteur change entre chaque exécution. Les valeurs obtenues oscillent entre 10 000 et 20 000.

Ce comportement est en quelque sorte normal. Si plusieurs threads utilisent la même variable, ils peuvent s'emmêler les uns dans les autres.
Dans notre situation, il y a deux threads qui utilisent la variable *value*. Ces deux threads itèrent *value* de 0 à 10 000.
Mais, étant donné que ces itérations ne sont pas atomiques, nous ne pouvons pas savoir dans quel ordre ni comment la variable *value* sera modifiée.

Dans ma compréhension du problème, il y a trois cas de figure :

1. Le premier thread récupère la valeur de *value* puis le deuxième thread récupère la valeur de *value*. La main est ensuite donné au thread 1 qui incrémente *value* de 10 000 puis c'est au tour du thread 2 qui incrémente lui aussi *value* de 10 000.
   Et enfin, **T1**(thread 1) réécrit *value* dans le tas puis **T2**(thread 2) réécrit lui aussi *value* dans le tas. La valeur de **T1** est donc écrasée par celle de **T2**.
   La valeur de *value* affichée à la fin est donc : 10 000.
2. **T1** lis, incrémente, puis réécrit *value* dans le tas puis passe la main à **T2** qui fait la même chose.
   La valeur de *value* affichée à la fin est donc : 20 000.
3. Lors du dernier cas, **T2** lis la valeur de *value* lorsque **T1** a été interrompu en plain milieu de son incrémentation.
   La valeur de *value* affichée à la fin est donc : 10 000 < *value* < 20 000.

Il n'est théoriquement pas possible que la valeur de *value* soit inférieure à 10 000. En effet, lorsqu'un processus lit *value* pour la première fois, sa valeur est forcément comprise entre 0 et 10 000 (Sauf si autre mécanisme que je ne connais pas).
Le thread incrémente ensuite la valeur qu'il a lu de 10 000. La valeur obtenue est donc forcément supérieure ou égale à 10 000.

Exercice 01 :

Le comportement attendu de la méthode main de la classe *StopThreadBug* est le suivant :
Le thread **Main** va créer un nouveau thread **T1** qui va boucler à l'infini tant que la variable *stop* ne prend pas la valeur **false**.
C'est après avoir attendu 5 secondes que le thread **Main** va passer la valeur de la variable stop à **false** et donc normalement stopper le programme.

Le comportement observé n'est malheureusement pas celui attendu. En effet, **T1** ne s'arrête jamais.
J'en conclus donc que **T1** peut lire la variable *stop* une unique fois au début du programme. Il la place ensuite sur sa pile d'exécution et n'utilisera plus que celle-ci.

Lorsque nous ajoutons l'appel à la fonction println, il s'avère que **T1** finis par s'arrêter. Il est possible que l'interruption système oblige les thread à ré-accéder à la valeur dans le tas.

Il est également possible que le compilateur optimise la boucle infinie sans le println en un while(1). Mais nous ne saurons pas quelle option est la bonne.

Exercice 02 :

Lors de l'exécution de la méthode **Main** de la classe *ExempleReordering*, les différents affichage attendus sont : {(a = 0, b = 0), (a = 1, b = 0), (a = 1, b = 2)}.

Il peut cependant arriver que les valeurs (a = 0, b =2) soient affichées. Il s'agit du JIT qui a réorganiser l'ordre des instructions du code.

Il faut donc faire attention, si le comportement mono-thread du programme est inchangée lors de l'inversion de deux affectation de variable, le JIT est susceptible de le faire.

Pour la deuxième partie de l'exercice, je n'obtiens que des 0. Il s'agit d'une machine 64bits donc les long n'ont pas besoin d'être fragmenté.
Mais, si la machine fait 32bits, il faudra faire l'affectation en deux étapes. L'opération n'est donc plus atomique.

Sachant que -1 en hexa est représenté par FFFFFFFF, et 0 par 00000000, les valeurs possibles sont (le comportement est identique à la première partie):

1. FFFFFFFF
2. 00000000
3. FFFF0000
4. 0000FFFF

Exercice 03 :

Dans l'exercice 03, n thread insèrent 5000 *Integer* dans une *ArrayList*. Pour comprendre pourquoi à la fin de l'exécution du programme, la taille de la liste est inférieure au nombre de add effectués, il est nécessaire de comprendre comment fonctionne la méthode add.

Lorsque la méthode add() est appelée sur une ArrayList, *add* insère le nouvel élément à la taille list.size() du tableau puis effectue l'opération size++.
Au vu des exercices précédent, il est facile de remarquer que l'opération *add* n'est pas atomique et, donc, que le *add* va poser des problèmes en multithreadé.

Dans la deuxième partie de l'exercice, lorsque la capacité de l'*ArrayList* n'est pas précisée, des ***java.lang.ArrayIndexOutOfBoundsException*** peuvent survenir.
Ce comportement est possible car si la taille est modifiée par un thread avec une valeur inférieure à la taille actuelle du tableau et qu'on essaie ensuite d'insérer un nouvel élément dans ce tableau, la fonction *add* s'en rend compte et throw l'exception pour éviter de faire planter le programme (segfault).