

STI 2^e année – POO avancée: Java

TD: Thread et deadlocks

1 Les premières « threads »

Exercice 1 Définir une classe extension de `Thread` qui reçoit en paramètre de construction un nom et un entier n . Son corps est une boucle, parcourue 20 fois, qui affiche son nom sur la sortie standard puis se met en attente pendant n secondes.

Exercice 2 Ecrire un programme principal qui reçoit en argument de la ligne de commande deux entiers n_1 et n_2 . Ce programme crée 2 threads du type précédents, en passant n_1 à la première et n_2 à la seconde. Puis il lance les deux threads et attend leur fin.

Exercice 3 Etudier les effets des variations relatives de n_1 et n_2 sur l'ordre d'exécution des deux threads.

2 Ordonnancement des « threads »

Cet exercice simple vise à montrer que les threads Java sont effectivement ordonnancées : elles peuvent être préemptées à tout moment par le système ce qui provoque un entrelacement (apparemment spontané) de leur exécution. En fait l'ordonnancement se fait par « tranche de temps »¹ : on accorde une durée d'exécution continue maximale à chaque thread ; quand cette durée est atteinte, la thread est préemptée, pour laisser leur chance à d'autres threads de priorité égale ou supérieure ; elle sera reprise plus tard, bien sûr.

Exercice 4 On reprend la structure à deux threads de l'exercice précédent que l'on modifie de la façon suivante :

- le programme principal ne reçoit cette fois qu'un seul argument, le nombre n de tours de boucle que chacune des deux threads doit effectuer (c'est le même nombre pour les deux) ;
- on retire l'instruction d'attente (`sleep()`) de la boucle du corps de la méthode `run()` ; les threads vont donc se dérouler en continu, sans auto-préemption.

En faisant croître n de manière (vaguement) exponentielle on montrera qu'à partir d'une certaine valeur, les deux threads ne s'exécutent plus l'une après l'autre mais entrelacent leur exécution.

Si l'on a pris soin de ne pas écrire sur la sortie standard autre chose que ce qui est demandé ici, on pourra compter le nombre de « changements de main » en filtrant la sortie standard du programme par le script shell `nContextSwitches` suivant :

```
#!/bin/sh
# Count the number of time when a line from stdin differs
# from the previous line
awk '{if (prev != $0) {prev = $0; print "changement"}}' | wc -l
```

1
2
3
4
5
6

1. Dans les premières versions de Java, ce comportement n'était pas garanti sur tout système d'exploitation ; depuis Java 2 (version 1.2 et supérieur du JDK) il l'est.

3 Barrières

Question 5 Testez les *CyclicBarrier* pour synchroniser (rendez-vous) vos threads précédents. Affichez un "J'attends les autres" avant la barrière et un "On est tous débloqué!" juste après.

4 Deadlock

Le but de cet exercice est de créer un deadlock (une fois n'est pas coutume!).

Question 6 Créez une classe *Ressource* ne contenant rien. Créez deux objets *r1* et *r2* dans votre main et passez ces deux objets à la construction des deux Threads précédents. Vous stockerez ces deux objets en attributs privés des Threads.

Question 7 Réalisez alors l'algorithme suivant dans le Thread 1 :

```
prendre un moniteur sur r1;  
attendre 3s;  
prendre un moniteur sur r2;  
afficher "Je suis 1 et j'ai les deux ressources !"
```

Question 8 Réalisez alors l'algorithme suivant dans le Thread 1 :

```
attendre 1s;  
prendre un moniteur sur r2;  
attendre 3s;  
prendre un moniteur sur r1;  
afficher "Je suis 2 et j'ai les deux ressources !"
```

Exercice 9 Constatez le deadlock.

Exercice 10 Supprimez les temporisations. Vous constaterez que le deadlock disparaît car la probabilité qu'il y ait entrelacement de la prise des deux moniteurs est faible.

Exercice 11 Faites des boucles très longues faisant itérer la prise de ressources. Un deadlock devrait survenir à plus ou moins long terme.

Question 12 Résolvez le deadlock en changeant d'implémentation et en utilisant un objet implémentant l'interface *Lock* au lieu d'utiliser **synchronized**, par exemple un *Lock* réentrant.