Notes à propos du TP

Exercice 1

Question 1

Si l'instruction de type `x = x + 1` était atomique, alors nous aurions *value* et *valuesbis* aux valeurs suivantes:

```
value = 6000
valuebis = 6000
```

Question 2

Après avoir exécuté le programme, nous avons un résultat possible suivant:

```
value: 4051, valuebis: 4068 et last: 1000   // thread W
value: 5994, valuebis: 5994 et last: 5000   // thread R
value: 5994, valuebis: 5994 et last: 0
```

Donc, l'instruction de type `x = x + 1` n'est pas atomique.

Question 3

Il faudra utiliser un moniteur au niveau de la classe `MonObjet` de la manière suivante :

```
// Dans MonObjet
public synchronized void add() {
    last.set(new Integer(last.get() + 1));
    value = value + 1;
    valuebis = valuebis + 1;
}
```

De cette manière, on pourra faire l'incrémentation de manière concurrente.

Question 4

La variable `value` est partagée entre différents threads, tandis que `last` est locale à chaque thread.

Exercice 2

Remarques préliminaires

- On Définit une classe qui générera un identifiant pour chaque nouveau thread.
- L'identifiant du thread est local au thread.
- Les deux implémentations de *MyThread* (i.e `MyThread21` et `MyThread22`) ont le même code en commun. La différence est que dans la première version, l'identifiant du thread est initialisé **à l'exécution** du thread, tandis que dans la deuxième version, l'identifiant est initialisé **à la construction** du thread.

Je n'ai pas remarqué de différence entre les deux versions.

Exercice 3

On veut tester le comportement d'un programme multithread manipulant une variable de classe partagée issue de `Main`

Remarque préliminaire

Dans l'hypothèse où `cur` $\in]-\infty, +\infty[$, on peut dire que ce programme peut ne pas terminer. En effet, il existe une exécution tel que le thread `MyObject` ne termine pas.

Dans `MyObject`, l'exécution se termine lorsque `cur = 10`. Or, `cur` est réinitialisé à `Main.check` à chaque que la condition `Main.check > cur` est vrai dans la boucle *for* dans `MyObject`.

De plus, dans `Stop`, `Main.check` est incrémenté jusqu'à atteindre la valeur `11`.

Il suffit donc que le thread `Stop` se lance en tout premier, et ceux, durant toute son exécution, puis termine (avec en post-condition `Main.check = 11`), et que le thread `MyObject` se lance pour avoir l'absence de terminaison du programme (`cur != 10` ne sera donc jamais vrai dans cette exécution).

Analyse du comportement du programme

Avec le mot-clé `volatile`

Nous avons (toujours) systématiquement l'affichage suivant:

```
check = 1 cur = 0
check = 2 cur = 1
```

```
check = 3 cur = 2
check = 4 cur = 3
check = 5 cur = 4
check = 6 cur = 5
check = 7 cur = 6
check = 8 cur = 7
check = 9 cur = 8
check = 10 cur = 9
coucou
received 11 stop
```

Sans le mot-clé `volatile`

Le programme ne termine pas, et ceux, malgré l'appel de `Thread.sleep()`.

Explication

Cela est dû au fait que lorsqu'on utilise le mot-clé volatile, on indique que la variable sera modifié par plusieurs threads. De ce fait, la valeur de la variable `Main.check` ne sera pas placé dans le cache cache locale d'un thread.

De plus, l'accès à une variable volatile fonctionne plus ou moins comme si on la manipulait dans un bloc *synchronized*, mais appliqué à la variable elle-même.

Source - The volatile keyword in Java

Exercice 4

Remarque concernant l'énoncé:

Le code indiqué dans l'énoncé est incorrect. En effet, les variable membres de la classe Paterson et de Filter ne sont pas déclarées `volatile`. Nous avons dû mettre le mot-clé `volatile` pour que les deux programmes fonctionnent.

Hypothèse de travail:

- Nous supposons que nous avons 10 threads.
- Chaque instruction s'exécute de manière atomique.

Questions

(a)

Oui, c'est possible. Il suffit d'exécuter chaque thread jusqu'à la boucle *while* dans le premier tour de la boucle *for*.

(b)

Suposons que cela soit possible, c'est-à-dire qu'il y a 10 threads au niveau 1.

Donc ∀ thread d'identifiant *i*, i ∈ [0-9], level[i] = 1.

Cela signifie que le 10ème thread, ayant pour identifiant *j* (j ∈ [0-9]), lorsqu'il est allé au niveau 1, a vu, lorsqu'il était au niveau 0 :

```
sameOrHigher(j, 0) == false
```

ou bien

```
victim[0] != j
```

Donc, à un moment donné, le thread *j* a quitté la boucle

```
while (sameOrHigher(me, i) && victim[i] == me) // me == j et i == 0
```

En supposant que `victim[i]` ait changé, cela veut dire qu'un thread a fait le lock après le 10ème thread. Or, on n'a pas plus de 10 threads qui font `lock()`.

C'est ABSURDE.

On ne peut donc pas avoir 10 threads simultanément au niveau 1.

(c)

Oui cela est possible, car dans le cas ou plusieurs threads sont au niveau 0, et acun au niveau 1, au moins 1 thread sortira de la boucle *while* à cause de la condition :

```
victim[0] != me
```

au niveau 0, parmi tous les thread qui sont à ce niveau, il y a exactement un thread victime. Les autres threads, quant à eux, pourront sortir de la boucle *while* et passer au niveau 1.

(d)

Supposons que dans le cas où il y a au moins 1 thread au niveau 0 et au moins un thread au niveau 1, tous les threads de niveau 0 peuvent passer au niveau 1.

Dans cette question, je vais supposer que j'ai *n* threads, *m* threads au niveau 0, et *k* threads au niveau 1. (n, m et k sont des entiers strictement positifs)

Donc si tous mes threads de niveau 0 passent au niveau 1, cela veut dire que :

∀ thread i ∈ [0 - (k-1)]

```
sameOrHigher(i, 0) == false
```

ou bien

```
victim[0] != i
```

Cela signifie que les *k* threads qui sont passés au niveau 1 on vu au moins une des conditions indiquées tantôt.

Supposons que `sameOrHigher(i, 0)` soit vrai pour tout le monde et que `victim[0] != i` soit la condition vue par les *k* threads.

Donc il existe un thread *j* (j ∈ [0 - (k-1)]) parmi les k threads tel que `victim[0] == i`. Ce thread j, faisant parti des k threads, est donc une victime, et ne peut donc pas passer au niveau 1. Or, on supposé au tout début que tous les thread de niveau 0 (les *k* thread) passaient au niveau 1, le thread *j* inclus.

C'est ABSURDE.

Donc s'il y a au moins 1 thread au niveau 0 et au moins un thread au niveau 1, on ne peut pas avoir tous les threads de niveau 0 qui passent au niveau 1.

(e), (f) et (g)

L'idée de l'algorithme de Peterson, appliqué à *n* threads est faire en sorte à ce qu'à un niveau *j*, donné, on ait au plus *n - j* threads, jusqu'à ce qu'au niveau *n - 1* nous ayons un seul thread. Ce thread sera donc celui qui ira en section critique.

Les question e, f et g pourront être répondue à travers la question *i*

(h)

Si un thread est en section critique, alors tous les autres threads seront au plus au niveau *n - 2* (*n* étant le nombre de threads)
