TD et TP n° 10 : Concurrence

I) Accès en compétition

Exercice 1 : Hello World et accès en compétition

Dans le cours sur la concurrence, vous avez vu l'exemple suivant qui permet de lancer un thread, à partir d'un `Runnable`, qui affiche le message `"hello world!"` :

```java
public class HelloRunnable implements Runnable {
   public void run() {
      System.out.println("Hello world!");
   }
}
public class Main {
   public static void main(String[] args) {
      (new Thread(new HelloRunnable())).start();
   }
}
```

1. Implémentez la méthode statique `void hellofromThreads(int n)` dans la classe `Main` qui permet de lancer `n` threads qui affichent le même message `"hello world!"`.
Appelez cette méthode à partir de `main` pour un `n` $\geqslant 6$.

2. Modifiez la classe `HelloRunnable` tel que chaque thread a un identificateur unique `id` de type `int`.
Adaptez le le code pour que chaque thread affiche son `id` avec le message. Par exemple, le thread avec `id` $= 2$ affiche `"Thread 2: Hello world!"`. Testez votre programme plusieurs fois, est-ce que les messages sont toujours affichés avec le même ordre.

On veut maintenant que parmi les `n` $\geqslant 6$ threads créés seulement 3 threads arrivent à afficher leurs messages.

3. Définissez une variable statique qui compte le nombre de messages affichés (à la fin de l'exécution, ce compteur doit être égal au nombre de threads `n` passé comme argument à la méthode `hellofromThreads`).
Sans utiliser `synchronized`, modifiez la méthode `run` de `HelloRunnable` tel que le message est affiché seulement si le compteur de messages est inférieur à 3.
Testez votre programme. qu'est-ce que vous remarquez. comment vous expliquez le phénomène que vous avez remarqué.

4. Corrigez le problème de la question précédente en utilisant un verrou que vous avez vu en cours.
Testez votre programme plusieurs fois, est-ce que c'est toujours les mêmes threads qui affichent leurs messages.

II) Profiter de concurrence

Exercice 2 : Test de primalité (en concurrence)

L'algorithme suivant permet de vérifier est-ce qu'un nombre est premier ou non :

Algorithm 1 Test de primalité

```
1:  procedure isPrime(n)
2:      if n = 0 then
3:          return false
4:      end if
5:      for i ← 2 to ⌊√n⌋ do
6:          if n mod i = 0 then
7:              return false
8:          end if
9:      end for
10:     return true
11: end procedure
```

Le but de cet exercice est d'implémenter l'algorithme au-dessus en deux versions : une version séquentielle et une version concurrente paramétrable par le nombre de threads à utiliser. La version concurrente sera implémentée différemment que l'exemple vu en cours avec les Workers. On va comparer le temps de l'exécution de la version séquentielle et la version concurrente (avec 1,2,4,8,16 et 32 threads) pour remplir le tableau suivant :

version	`2`	`9999999929L`	`1262182800679059439L`	`8223335555577777779L`	`9181531581341931811L`[1]
	premier	premier	non premier	premier	premier
séquentielle (Question 1)					
parallèle threads = 1 (Question 2)					
parallèle threads = 2 (Question 2)					
parallèle threads = 4 (Question 2)					
parallèle threads = 8 (Question 2)					
parallèle threads = 16 (Question 2)					
parallèle threads = 32 (Question 2)					

1. `9181531581341931811L` est le plus grand entier premier qu'on peut stocker dans une variable de type **`long`**.

La signature de la version séquentielle de l'algorithme est la suivante où `n` est le nombre qu'on veut vérifier :

```
public static boolean isPrime(long n);
```

La version concurrente aura la signature suivante où `n` est le nombre qu'on veut vérifier et `t` le nombre de threads à utiliser :

```
public static boolean isPrime(long n, int t);
```

Pour le temps de l'exécution, on utilisera le code suivant dans le main qui va afficher si le nombre est premier et le temps d'exécution en second (on prend trois chiffres après la virgule, par exemple, si le programme affiche "PT2.029466S", on notera dans le tableau 2.029).

```
1  ...
2  Instant start = Instant.now();
3  System.out.println(isPrime(9181531581341931811L));
4  Instant end = Instant.now();
5  Duration timeElapsed = Duration.between(start, end);
6  System.out.println(timeElapsed);
7  ...
```

1. Version séquentielle :
   - Implémentez la version séquentielle de l'algorithme.
   - Évaluez votre implémentation avec les nombres donnés dans le tableau, et remplissez-le par les temps que vous avez obtenus.

2. Version concurrente fonctionne comme suite :

Pour `n` $= 221$ et `t` $= 2$, on calcule $\sqrt{221} = 14.866...$ puis on divise l'intervalle $[\![2, 14]\!]$ par `t` $= 2$ pour obtenir deux intervalles $[\![2, 8]\!]$ et $[\![9, 14]\!]$ qu'on associe à chaque thread. Le premier thread, par exemple, vérifie est-ce qu'il existe un $i \in [\![2, 8]\!]$ tel que $n \bmod i = 0$. Si le nombre `n` est premier, quand les deux threads terminent, on retourne **`true`**. Si le nombre `n` n'est pas premier, l'un des deux threads trouve un diviseur de `n`, il interrompt l'autre thread et il annonce qu'il a trouvé un diviseur (dans une variable) et il termine son exécution. On vérifie à la fin si l'un des threads a trouvé un diviseur, si c'est le cas, on retourne **`false`**.

- Écrivez la classe `Interval` avec deux attributs `debut` et `fin` de type **`long`**.
- Implémentez une méthode `List<Interval> diviser(int t)` dans la classe `Interval` qui permet de diviser l'interval actuel en t intervalle comme expliqué dans l'exemple au-dessus.
- Écrivez la classe `IsPrimeRunnable` qui implémente `Runnable` et qui prend dans son constructeur le nombre `n` et un `Interval`. Dans la méthode `run` de cette classe, vérifiez s'il existe un entier dans l'intervalle donné qui divise `n`, si c'est le cas, l'attribut boolean de cette classe `isNotPrime` est mis à **`true`** (`isNotPrime` initialement = **`false`**).
- Quand un `IsPrimeRunnable` découvre que le nombre `n` n'est pas premier, il doit être capable d'interrompre tous les autres threads pour retourner le résultat. Ajoutez un attribut `List<Thread> threads` à `IsPrimeRunnable` avec la méthode **`void`** `addThread(Thread thread)` qui permet d'ajouter un thread à cette liste. Modifier `run` tel qu'une fois on découvre que `n` n'est pas premier, on interrompt les threads stockés dans `threads`. (Assurez que si deux threads parmi plusieurs trouvent que n n'est pas premier au même temps, ils n'interrompent l'un l'autre et ils lassent les autres en exécution).
- Dans la méthode statique **`boolean`** `isPrime(long n, int t)`, créez l'intervalle approprié et divisez le avec la méthode `diviser(t)` ; pour chaque sous intervalle, instanciez un `IsPrimeRunnable`. Puis créez un thread pour chaque runnables que vous avez créé. Ajouterz à chaque runnable les threads des autre runnables (on ne veut pas qu'un runnable interrompt lui-même). Commencez les threads que vous avez créés et attendez qu'ils terminent leurs exécution (`join()`). Si à la fin, l'un des runnables a son attibut `isNotPrime = true`, on retourne **`false`** sinon **`true`**.
- Evaluez votre implémentation avec les nombres donnés dans le tableau pour `t` $\in \{1, 2, 4, 8, 16, 32\}$, et remplissez-le par les temps que vous avez obtenus.
- Quelles sont les conclusions qu'on peut tirer à partir de tableau.

III) Synchronisation et moniteurs

Exercice 3 : Compteurs

On considère la classe `Compteur`, que nous voulons tester et améliorer :

```
1  public class Compteur {
2     private int compte = 0;
3     public int getCompte() { return compte; }
4     public void incrementer() { compte++; }
5     public void decrementer() { compte--; }
6  }
```

1. À cet effet, on se donne la classe `CompteurTest` ci-dessous :

```
1  public class CompteurTest {
2     private final Compteur compteur = new Compteur();
3
4     public void incrementerTest() {
5        compteur.incrementer();
6        System.out.println(compteur.getCompte() + " obtenu après incrémentation");
7     }
8
9     public void decrementerTest() {
10       compteur.decrementer();
11       System.out.println(compteur.getCompte() + " obtenu après décrémentation");
12    }
13 }
```

   Écrivez un main qui lance sur une seule et même instance de la classe `CompteurTest` des appels à `incrementerTest` et `decrementerTest` depuis des threads différents. Pour vous entraîner à utiliser plusieurs syntaxes, lancez en parallèle :

   - une décrémentation à partir d'une classe locale, dérivée de `Thread` ;
   - une décrémentation à partir d'une implémentation anonyme de `Runnable` ;
   - une incrémentation à partir d'une lambda-expression obtenue par lambda-abstraction (syntaxe `args -> result`) ;
   - une incrémentation à partir d"une lambda-expression obtenue par référence de méthode (syntaxe c`ontext::methodName`).

2. On souhaite maintenant qu"il soit garanti, même dans un contexte *multi-thread*, que la valeur de compte (telle que retournée par `getCompte`) soit toujours égale au nombre d'exécutions d'`incrementer` moins le nombre d'exécutions de `decrementer` ayant terminé avant le retour de `getCompte` (rappel : l'incrémentation `compte++` et la décrémentation `compte--` ne sont pas des opérations atomiques).
   Obtenez cette garantie en ajoutant le mot-clé **`synchronized`** aux endroits adéquats dans la classe `Compteur`.

3. Est-ce que les modifications de la question précédente assurent que `incrementerTest` et `decrementerTest` affichent bien la valeur du compteur obtenue après, respectivement, l'appel à `incrementer` ou à `decrementer` fait dans chacunes des deux méthodes de test ?
   Comment modifier `CompteurTest` pour que ce soit bien le cas ?

4. On veut ajouter à la classe Compteur la propriété supplémentaire suivante : « compte n'est jamais être négatif ». Celle-ci peut être obtenue en rendant l'appel à `decrementer` bloquant quand compte n'est pas strictement positif. Modifiez la classe `CompteurTest` en introduisant les `wait()` et `notify()` nécessaires.

Exercice 4 : Verrou Lecteur/Rédacteur

Le problème lecteurs-rédacteur est un problème d'accès à une ressource devant être partagée par deux types de processus :

- les lecteurs, qui consultent la ressource sans la modifier,
- les rédacteurs, qui y accèdent pour la modifier.

Pour que tout se passe bien, il faut que, lorsqu'un rédacteur a la main sur la resource, aucun autre processus n'y accède « simultanément » [1]. En revanche, on ne veut pas interdire l'accès à plusieurs lecteurs simultanés.
Malheureusement, les moniteurs de Java ne gèrent directement que l'exclusion mutuelle [2]. Pour implémenter le schéma lecteurs-rédacteur, il faut donc une classe dédiée.
Nous allons procéder en trois étapes :

- définition d'une classe verrou,
- association d'un verrou et d'une ressource,
- mise en place d'un test de lectures écritures concurrentes.

Il est probable que vous oublierez des choses au départ. Vous y reviendrez et procéderez aux ajustements au moment des tests. Vous trouverez également quelques conseils en fin d'exercice.

1. Définissez une classe, dont les objets seront utilisés comme des verrous, nous l'appellerons `ReadWriteLock`. Ils contiennent :
   - un booléen pour dire si un écrivain est actuellement autorisé ;
   - le nombre de lecteurs actuellement actifs sur la ressource ;
   - la méthode `dropReaderPrivilege()` qui décrémente le nombre de lecteurs actuels ;
   - la méthode `dropWriterPrivilege()` qui libère la ressource de son rédacteur ;
   - les méthodes `acquireReaderPrivilege()` et `acquireWriterPrivilege()`, bloquantes sur le moniteur du verrou, pour demander un droit d'accès en lecture ou en écriture.

   Testez cette classe. Par exemple, la séquence suivante ne doit pas bloquer :

```
1 val lock = new ReadWriteLock();
2 lock.acquireReaderPrivilege();
3 lock.acquireReaderPrivilege();
```

   mais celle-ci, oui :

```
1 val lock = new ReadWriteLock();
2 lock.acquireReaderPrivilege();
3 lock.acquireWriterPrivilege();
```

   (On peut la débloquer en appelant `lock.dropReaderPrivilege()` dans un autre thread.)
   et celle-là aussi :

```
1 val lock = new ReadWriteLock();
2 lock.acquireWriterPrivilege();
3 lock.acquireReaderPrivilege()
```

   (On peut la débloquer en appelant lock.dropWriterPrivilege() dans un autre thread.)

1. On évite ainsi de créer des accès conflictuels non synchronisés, i.e. des accès en compétition.
2. Le moniteur n'appartient qu'à un seul thread en même temps, à l'exclusion de tout autre.

2. Écrire une classe `ThreadSafeReadWriteBox`, encapsulant une ressource de type `String` et une instance de verrou `ReadWriteLock`. Utilisez le verrou dans le getteur et le setteur de la ressource, afin de garder les accès en lecture et écriture (en acquérant le privilège pertinent avant l'accès ; puis en le libérant après l'accès).
3. Ecrivez une classe de test dont le `main()` manipule une instance de `ThreadSafeReadWriteBox` contenant la chaîne `"Init"`. Vous lancerez deux threads changeant la valeur de la ressource en `"A"` et `"B"` respectivement, et 10 autres threads qui se contenteront d'afficher la ressource. On aura donc 2 opérations d'écritures et 10 de lectures.
   Pour se rendre compte de l'ordonnancement et de la concurrence, modifiez la méthode `set` de `ThreadSafeReadWriteBox` pour qu'elle attende une seconde avant d'écrire. Modifiez également `get` pour qu'elle attende aléatoirement entre 0 et deux secondes.
   Etudiez les ordonnancements possibles des lectures et écritures et donnez une estimation du temps attendu. Vérifiez bien que votre test s'exécute dans ces délais.
4. `ReadWriteLock` (comme les verrous explicites fournis par la package `java.util.concurrent.locks` du JDK) a un défaut majeur par rapport aux moniteurs : rien n'oblige à libérer un verrou après son acquisition (pour les moniteurs, c'était le cas car l'acquisition se fait en entrant dans le bloc **`synchronized`** et la libération en en sortant). Un tel oubli provoquerait typiquement un *deadlock*.
   L'API de la classe `ThreadSafeReadWriteBox` qui encapsule un `ReadWriteLock`, est API plus sûre car il est impossible pour l'utilisateur d'oublier de libérer le verrou encapsulé (c'est géré par le getteur et le setteur). Mais cette classe est trop spécialisée (seulement lecture et affectation d'un `String`).
   Pourriez-vous proposer une nouvelle interface pour `ReadWriteLock` qui n'ait pas ce problème ? (pensez fonctions d'ordre supérieur ou bien alors, documentez vous sur les blocs *try-with-resource* et l'interface `Autocloseable`)
   Écrivez une classe implémentant cette API en se basant sur une instance encapsulée (privée) de `ReadWriteLock`.

Quelques conseils :

- On rappelle que pour utiliser et libérer une ressource (ici le verrou) la bonne façon de faire est de la forme `acquerir(R); try` { instructions } **`finally`** { `liberer(R);` } ainsi même s'il y a un return dans les instructions, la ressource est libérée.
- Pensez à distinguer `notify` et `notifyAll`, n'en ajoutez pas non plus partout. Justifiez bien leur écriture en vous demandant qui peut être en état d'attente.