TP n° 10 bis : Verrou Lecteur/Rédacteur
(Correction)

Exercice 1 : Un peu plus loin que les moniteurs

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(); lock.acquireReaderPrivilege();
            lock.acquireReaderPrivilege();
```

   mais celle-ci, oui :

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

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

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

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

   Correction :

```
1  public class ReadWriteLock {
2     private boolean pris = false;
```

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.

```
3     private int nbLecteur = 0;
4
5     public synchronized void dropReaderPrivilege() {
6         if (nbLecteur > 0) nbLecteur--; // il faut notifier, car qq'un peu attendre
7         // on devrait lever une exception pour un nb <0 "Mauvaise utilisation"
8         notify(); // simple suffit
9     }
10
11    public synchronized void dropWriterPrivilege() {
12        pris = false; // il faut notifier
13        notifyAll(); // penser à notifyAll, avec notify un seul lecteur qui attend serait
              débloqué
14    }
15
16    // bloquant jusqu'à  l'obtention du privilège
17    public synchronized void acquireReaderPrivilege() throws InterruptedException {
18        while (pris) wait();
19        nbLecteur++;
20        // notify(); ne servirait à rien
21    }
22
23    // bloquant jusqu'à  l'obtention du privilège
24    public synchronized void acquireWriterPrivilege() throws InterruptedException {
25        while (pris || nbLecteur > 0) wait();
26        pris = true;// notify(); ne servirait à rien
27    }
28
29 }
```

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).

Correction :

```
1  public class ThreadSafeReadWriteBox {
2     private final ReadWriteLock lock;
3     private String content;
4
5     public ThreadSafeReadWriteBox(String x) {
6         content = x;
7         lock = new ReadWriteLock();
8     }
9
10    private static void attendExact(int x) {
11        try {
12            Thread.sleep(x);
13        } catch (InterruptedException ex) {
14            System.out.println("InterruptedException non traitée");
15        }
16    }
17
18    private static void attendJusquA(int x) {
19        attendExact((int) (x * Math.random()));
20    }
21
22    public void set(String v) {
23        try {
24            lock.acquireWriterPrivilege();
25            attendExact(1000); // attendre 1 secondes
26            content = v;
27        } catch (InterruptedException ex) {
28            System.out.println("InterruptedException non traitée");
29        } finally { // non, probablement que ce n'est pas dans un finally, en cas
```

```
            d'interruption, ca ferait même une libération de trop (mais dans le monde des
            booléans ca n'a pas d'importance)
30          lock.dropWriterPrivilege();
31      }
32  }
33
34  public String get() throws InterruptedException {
35      lock.acquireReaderPrivilege();
36      try {
37          attendJusquA(2000); // attendre jusqu'à  2 secondes
38          return content;
39      } finally {
40          lock.dropWriterPrivilege();
41      } // le finally est nécessaire à cause de l'ordre return, puis libération
42  }
43 }
```

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.

Correction :

```
1  public class TestReadWriteLock {
2      public static void main(String[] args) {
3          ThreadSafeReadWriteBox x = new ThreadSafeReadWriteBox("Init");
4
5          new monThreadWrite(x, "A").start(); // remplace Init par A;
6          new monThreadWrite(x, "B").start(); // remplace Init par B;
7
8          for (int i = 0; i < 10; i++) new monThreadRead(x).start(); // effectue une lecture
               (et affiche)
9      }
10
11     static class monThreadWrite extends Thread {
12         String val;
13         ThreadSafeReadWriteBox b;
14
15         monThreadWrite(ThreadSafeReadWriteBox x, String y) {
16             b = x;
17             val = y;
18         }
19
20         public void run() {
21             b.set(val);
22         }
23     }
24
25     static class monThreadRead extends Thread {
26         ThreadSafeReadWriteBox b;
27
28         monThreadRead(ThreadSafeReadWriteBox x) {
29             b = x;
30         }
31
32         public void run() {
33             try {
```

```
34                 System.out.println(b.get());
35             } catch (InterruptedException e) {
36                 System.out.println("Thread interrompu.");
37             }
38         }
39     }
40 }
```

4. `ReadWriteLock` (comme les verrous explicites fournis par le 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`.

   Correction : Une version avec des fonctions d'ordre supérieur prenant des `Runnable` comme argument, exécuté dans un bloc **`try`**/**`finally`** :

```
1  import java.util.Optional;
2  import java.util.function.Supplier;
3
4  public final class SafeReadWriteLock {
5      private final ReadWriteLock lock = new ReadWriteLock();
6
7      public void read(Runnable r) throws InterruptedException {
8          lock.acquireReaderPrivilege();
9          try {
10             r.run();
11         } finally {
12             lock.dropReaderPrivilege();
13         }
14     }
15
16     public void write(Runnable r) throws InterruptedException {
17         lock.acquireWriterPrivilege();
18         try {
19             r.run();
20         } finally {
21             lock.dropWriterPrivilege();
22         }
23     }
24 }
```

   Une autre version sans fonction d'ordre supérieur, mais en retournant des « permis » implémentant `Autocloseable`, ce qui permet l'utilisation dans un bloc *try-with-resource* :

```
1  public final class SafeReadWriteLock2 {
2      private final ReadWriteLock lock = new ReadWriteLock();
3
4      public ReadToken getReadToken() throws InterruptedException {
```

```
5           return new ReadToken();
6       }
7
8       public WriteToken getWriteToken() throws InterruptedException {
9           return new WriteToken();
10      }
11
12      public class ReadToken implements AutoCloseable {
13          private ReadToken() throws InterruptedException {
14              lock.acquireReaderPrivilege();
15          }
16
17          @Override
18          public void close() {
19              lock.dropReaderPrivilege();
20          }
21      }
22
23      public class WriteToken implements AutoCloseable {
24          private WriteToken() throws InterruptedException {
25              lock.acquireWriterPrivilege();
26          }
27
28          @Override
29          public void close() {
30              lock.dropWriterPrivilege();
31          }
32      }
33
34  }
```

Exemple d'utilisation :

```
1   public class SRWLDemo {
2       public static void main(String[] args) throws InterruptedException {
3           SimpleBox y = new SimpleBox();
4           SafeReadWriteLock l1 = new SafeReadWriteLock();
5           SafeReadWriteLock2 l2 = new SafeReadWriteLock2();
6
7
8           // utilisation de l1 en lecture :
9           l1.read(() -> {
10              System.out.println(y.content);
11          });
12          // utilisation de l1 en écriture :
13          l1.write(() -> {
14              y.content = "C";
15          });
16
17          // utilisation de l2 en lecture (avec try-with-resource) :
18          // (équivalent à un try/finally dont le finally appellerait token.close())
19          try (SafeReadWriteLock2.ReadToken token = l2.getReadToken()) {
20              System.out.println(y.content);
21          }
22
23          // utilisation de l2 en écriture (avec try-with-resource) :
24          // (équivalent à un try/finally dont le finally appellerait token.close())
25          try (SafeReadWriteLock2.WriteToken token = l2.getWriteToken()) {
26              y.content = "D";
27          }
28
29      }
30
31      static class SimpleBox {
32          String content = "Init";
33      }
34  }
```

> Autre possibilité : on peut rendre les choses encore plus sûres en forçant les accès en écriture à passer par un permis d'écriture, en faisant en sorte qu'un verrou ne puisse servir qu'à contrôler l'accès à un certain attribut modifiable encapsulé (même principe que `ThreadSafeReadWriteBox`, mais en version générique). Contrairement à `ThreadSafeReadWriteBox` ici, il n'y aurait pas de setteur mais, à sa place, une méthode prenant en paramètre une `Function<T, T>` qui servirait à modifier la donnée encapsulée : `content = f.apply(content)`.

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.