Université Paris Cité

Compléments en Programmation Orientée Objet
TP nº 10 : *Multithreading* (primitives de synchronisation)

À finir, si ce n'est pas déjà fait : TP09

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 lamain 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 ;
   - une méthode `debutLecture()`, bloquante, exécutée par un lecteur pour demander d'accéder en lecture à une ressource partagée,
   - une méthode `debutEcriture()`, bloquante, exécutée par un rédacteur pour demander d'accéder en écriture à la ressource partagée,
   - une méthode `finLecture()`, exécutée par un lecteur pour indiquer qu'il a terminé la lecture de la ressource partagée ;
   - une méthode `finEcriture()`, exécutée par un rédacteur pour indiquer qu'il a terminé de modifier la ressource partagée.
2. Écrire une classe `ThreadSafeReadWriteBox`, encapsulant une ressource de type `String` et une instance de verrou `ReadWriteLock`. Utilisez le verrou pour définir le getteur et le setteur de la resource, afin que l'accès à celle-ci se fasse selon le schéma décrit précédemment.
3. Écrivez une classe de test dont le `main()` manipule une instance de `ThreadSafeReadWriteBox` contenant initialement 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'écriture et 10 de lecture.

   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 la méthode `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.

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.

4. `ReadWriteLock`, ainsi que tous les verrous explicites fournis par le package `java.util.concurrent.locks` du JDK ont 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*.

   En écrivant une classe comme `ThreadSafeReadWriteBox`, qui encapsule un `ReadWriteLock`, on a proposé une API plus sûre, où il est impossible pour l'utilisateur d'oublier de libérer un verrou. Mais cette classe a un objectif très spécialisé.

   Pourriez-vous proposer une nouvelle interface (ou tout du moins un ensemble de méthodes publiques avec leurs signatures), basée sur des fonctions d'ordre supérieur, qui n'ait pas ce problème ?

   Écrivez une classe `SafeReadWriteLock` implémentant cette interface (ou méthodes) en se basant sur une instance privée de `ReadWriteLock`.

5. Même question, mais avec une autre technique.

   Au lieu de passer les lectures et écritures via des lambdas, on peut les mettre dans des blocs *try-with-resource* (documentez-vous sur le sujet) : `try (Resource r = ...) { doSomething() }`. Pour cela, il suffit que la nouvelle classe contienne juste une méthode d'acquisition de verrou de lecture, resp. d'écriture, retournant un nouveau jeton de lecture (classe à définir), resp. d'écriture, qui implémente `Autocloseable`. On fait alors la libération du verrou dans la méthode `close` du jeton correspondant afin qu'elle ait lieu automatiquement à la sortie du `try`.

6. Comment modifieriez-vous `ReadWriteLock` pour donner la priorité aux lecteurs, de telle sorte qu'aucun rédacteur en attente ne se voie accorder le droit d'écriture avant que tous les lecteurs actuels et ceux en attente aient fini de lire ?

   Quel problème pourrait se poser alors en utilisant une telle classe (notamment si de nouveaux lecteurs arrivent très souvent) ?

Quelques conseils :

- Sous NetBeans vous pouvez contrôler le temps d'exécution directement dans la console, si votre IDE ne le fait pas, il vous faudra utiliser `System.nanoTime()`.
- 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.

Exercice 2 : Moniteurs et les sections critiques

Une section critique est un fragment de code qui doit être exécuté par un seul *thread* en même temps. Plus exactement s'il y a plusieurs *threads* qui veulent exécuter la section critique un seul y accède et les autres sont mis en attente.
De manière plus large nous pouvons avoir des fragments de code qui modifient les données partagées ce qui implique que les *threads* qui exécutent ces fragments ne peuvent pas être exécutés en parallèle. Nous appelons ces fragments de code « `writeSection` ».
D'autres fragments de code accèdent aux données partagées seulement en lecture, appelons-les « `readSection` ».
Le but de l'exercice est d'écrire une classe `SectionsCritiques` avec deux méthodes :

```
1    public boolean writeSection(Runnable r)
2    public boolean readSection(Runnable r)
3
```

- Un *thread* qui exécute `writeSection(Runnable r)` demande à exécuter la méthode `run` du `Runnable r` mais il doit se mettre en attente tant que d'autres *thread* exécutent soit `writeSection` soit `readSection` (avec d'autres `Runnable`s). La méthode `writeSection()` retourne **`false`** si l'exécution de `Runnable` n'a pas eu lieu par exemple à cause d'exception (par exemple `InterruptedException` survenu avant qu'on puisse exécuter `run()` du `Runnable`), sinon la méthode retournera **`true`**.
- Un *thread* qui exécute `readSection( Runnable r)` demande à exécuter la méthode `run` du `Runnable r` mais il doit se mettre en attente tant qu'il y a un autre *thread* qui exécute `writeSection()` (avec un `Runnable` quelconque).
  La méthode `readSection()` retourne **`false`** si l'exécution de `Runnable` n'a pas eu lieu par exemple à cause d'exception (par exemple `InterruptedException` survenu avant qu'on puisse exécuter `run()` du `Runnable`).

La classe `SectionsCritiques` doit être écrite de façon à ce que les conditions suivantes soient satisfaites :

(1) On suppose que les *threads* qui exécutent `readSection()` s'exécutent en boucle et exécutent périodiquement `readSection()` (pas forcement avec le même `Runnable`).

(2) Les *threads* qui exécutent `writeSection()` peuvent aussi s'exécuter en boucle mais cela n'est pas nécessaire.

(3) Supposons qu'un *thread* a exécuté `writeSection(Runnable r)`. Le `Runnable r` en paramètre a modifié des données partagées par les threads. Les nouvelles tentative de l'exécution de `writeSection()` doivent être bloquées tant que tous les *threads* lecteurs n'exécutent `readSection(Runnable r)` qui est sensé de lire les données modifiées.

(4) Quand tous les lecteurs ont exécuté `readSection()` et s'il y a des *threads* rédacteurs en attente sur `writeSection()` c'est le *thread* Rédacteur qui attend le plus longtemps qui doit être débloqué.

En résumant, l'exécution du programme qui utilise `SectionCritique` se déroule de façon suivante :

(1) un *thread* rédacteur exécute `writeSection()`

(2) tous les *thread* lecteurs exécutent, peut-être en parallèle, `readSection()`

(3) s'il y a des rédacteurs en attente sur `writeSection()` celui qui attend le plus longtemps est réveillé et on passe à (1).

1. Compléter le code de la méthode `main()` (disponible sur moodle) :

```
1 import java.util.List;
2 import java.util.Random;
3
4 public class Main {
5     static BoiteMessages boite = new BoiteMessages();
```

```
6     static SectionsCritiques sectionsCritiques = new SectionsCritiques();
7
8     public static void main(String[] args) {
9         Messages messages1 = new Messages(new String[]{"In", "Sunt", "dicere", "cuius",
10                "homines", "notas", "rebus", "praecurrit", "dicere", "omnis", "praecurrit",
11                "enim", "est", "cuius", "parandis"});
12
13        ProducerThread w1 = new ProducerThread(sectionsCritiques, messages1);
14
15        Messages messages2 = new Messages(new String[]{"W", "Szczebrzyszynie", "chrzaszcz",
16                "brzmi", "trzcinie", "slynie", "wol", "pyta"});
17        ProducerThread w2 = new ProducerThread(sectionsCritiques, messages2 );
18
19        ConsumerThread r1 = new ConsumerThread(sectionsCritiques, "r1");
20        ConsumerThread r2 = new ConsumerThread(sectionsCritiques, "r2");
21        ConsumerThread r3 = new ConsumerThread(sectionsCritiques, "r3");
22        ConsumerThread r4 = new ConsumerThread(sectionsCritiques, "r4");
23
24        List<Thread> listThread = List.of(w1, w2, r1, r2, r3, r4);
25        for (Thread t : listThread) t.start();
26    }
27
28    static class BoiteMessages {
29        private String message;
30        public String getMessage() {
31            return message;
32        }
33        public void setMessage(String message) {
34            this.message = message;
35        }
36    }
37
38    static class Messages {
39        private String[] msg;
40        private Random random = new Random(System.currentTimeMillis());
41        public Messages(String[] msg) {
42            this.msg = msg;
43        }
44        public String getNewMessage() {
45            return msg[random.nextInt(msg.length)];
46        }
47    }
48
49    static class ProducerThread extends Thread {
50        private Messages m;
51        private SectionsCritiques sectionsCritiques;
52
53        public ProducerThread(SectionsCritiques sectionsCritiques, Messages m) {
54            this.sectionsCritiques = sectionsCritiques;
55            this.m=m;
56        }
57
58        @Override
59        public void run() {
60            super.run();
61            while (true) {
62                sectionsCritiques.writeSection(
63
64               //TODO
65
66                );
67                if (isInterrupted())  return;
68            }
69        }
70    }
71
72    static class ConsumerThread extends Thread {
73        private SectionsCritiques sectionCritiques;
74
75        public ConsumerThread(SectionsCritiques sectionsCritiques, String name) {
76            super(name);
77            this.sectionCritiques = sectionsCritiques;
78            sectionsCritiques.registerReader(this, true);
```

```
79          }
80
81          @Override
82          public void run() {
83              super.run();
84              while (true) {
85                  sectionCritiques.readSection(() -> {
86
87              //TODO
88
89                  });
90
91                  if (isInterrupted())   return;
92              }
93          }
94      }
95  }
```

.

`main()` est une méthode de test qui lance deux *threads* rédacteurs `ProducerThread` et quatre *threads* lecteurs `ConsumerThread`.

Les *threads* agissent sur l'objet `BoiteMessage boite` (ligne 5)

(a) `ProducerThread` passe dans `writeSection()` un `Runnable` implémenté par une lambda expression. Le `Runnable` commence par exécuter `Thread.sleep(100)` et ensuite met dans la `boite` un nouveau message `m.getNewMessage()`.

(b) `ConsumerThread` passe dans `readSection()` un `Runnable` implémenté par une lambda expression. Le `Runnable` commence par exécuter `Thread.sleep(50)` et ensuite écrit sur la sortie standard le message contenu dans la `boite` précédé par le nom du *thread* (`Thread.getName()`).

2. Pour implémenter `SectionsCritiques` nous aurons besoin d'un verrou booléen. Ecrire la classe

```
1       public class BooleanLock{
2         public synchronized void lock() throws InterruptedException
3         public synchronized void unlock()
4       }
5
```

avec le comportement suivant : un seul *thread* peut acquérir le verrou en exécutant `lock()`, tous les autres seront bloqués dans `lock()` jusqu'à ce que le *thread* possédant le verrou exécute `unlock()`.

Il est temps de commencer le code de `SectionsCritiques`.
La classe maintiendra trois attributs,

```
1 private final BooleanLock booleanLock = new BooleanLock();
2 private final Map<Thread, Boolean> readers = new HashMap<>();
3 private final List<Thread> writers = new LinkedList<>();
```

La liste `writers` permet de mémoriser les *threads* Rédacteurs qui demandent à exécuter `writeSection()` et se trouvent bloqués faute de droit : soit parce que un autre *thread* est en train d'exécuter une section critique soit parce que tous les lecteurs n'ont pas encore exécuté `readSection()` après le dernier `writeSection()`.

Le `Map readers` sert à enregistrer tous les *threads* lecteurs. En effet on ne pourra pas savoir si tous les Lecteurs ont exécuté `readSection()` après le dernier `writeSection()` si on ne sait pas quels sont les *threads* Lecteurs ! Donc on enregistre les thread Lecteur (comme clé) avec une valeur booléenne (comme value) dans un `Map`. La valeur booléenne associée au *thread* Lecteur indique si ce *thread* a déjà exécuté `readSection()` après le dernier `writeSection()` (**`true`** si c'est le cas). Donc si aucun booléen **`false`** alors un *thread* Rédacteur peut exécuter `writeSection()`.

Tous les accès aux attributs `readers` et `writers` dans les méthodes `readSection` et `writeSection` doivent être protégés par le verrou `booleanLock`. Par accès je comprends toute lecture ou modification de `Map` et `List` référencés par `readers` et `writers`.

Les Rédacteurs et les Lecteurs doivent être suspendu (`wait()`) quand les conditions d'écriture ou lectures ne sont pas satisfaites. Cela pose la question de moniteurs utilisés par ces threads.
Rappel. Tout objet peut être utilisé comme moniteur et on peut utiliser un objet quelconque pour faire `objet.wait()` `objet.notify()` et `objet.notifyAll()`.

Moniteurs pour les Rédacteurs. Tous les Rédacteurs ne peuvent pas utiliser le même moniteur puisque à un moment donnée il faudra libérer le Rédacteur qui attend le plus longtemps c'est-à-dire celui qui est le premier sur la liste `writers`. Pour avoir le contrôle quel Rédacteur sera libérer on ne pourra pas utiliser un moniteur, il faut un moniteur séparé pour chaque Rédacteur. Mais nous avons déjà un objet unique pour chaque *thread* Rédacteur, c'est le *thread* lui-même qu'on récupère grâce à `Thread.currentThread()`. Donc un Rédacteur se met en attente avec `Thread.currentThread().wait()` quand les conditions d'écriture ne sont pas satisfaite. Et le premier *thread* sur la liste `writers` pourra être libéré grâce à `thread.notify()` où `thread` est le premier sur la liste `writers`.

Moniteur pour les Lecteurs. Les *thread* Lecteurs peuvent et doivent utiliser un seul moniteur. Et c'est l'objet `readers` qui peut servir comme le moniteur de Lecteurs. Donc `readers.wait()` exécuté par un *thread* Lecteur permet de le suspendre, et `readers.notifyAll()` exécuté par un Rédacteur libérera tous les Lecteurs suspendus.

3. Écrire la méthode `public void registerReader(Thread t, boolean bool)` qui ajoute un nouveau *thread* avec la valeur booléenne dans le `Map readers`. La méthode sera utilisée pour enregistrer les *threads* Lecteurs.
4. Écrire la méthode `private boolean readyToWrite()` qui retourne `true` si les valeurs booléennes dans le `Map readers` sont toutes `true` (donc il n'y a aucun `false`). La fonction sera utilisée pour vérifier si un Rédacteur sera autorisé à exécuter le `Runnable` passé en paramètre de `writeSection()`.
5. Écrire la méthode `public boolean writeSection(Runnable r)`.
   Le Rédacteur qui constate que `readyToWrite()` retourne `false` doit être se faire suspendre (`wait()`) sur un moniteur [3].
   Dans la méthode `writeSection(Runnable r)`, après avoir exécuté `r.run()`, le Rédacteur doit mettre la valeur `true` pour tous les *threads* Lecteurs dans le `Map readers` et les notifier (`notifyAll()`) pour qu'ils puissent avancer [4].
6. Écrire la méthode `public boolean readSection(Runnable r)`.
   Avant d'avoir exécuté `r.run()` le Lecteur qui constate que le booléen qui lui correspond dans le `Map readers` est `true` doit se faire suspendre (`wait()`) sur son moniteur [5].
   Après avoir exécuté `r.run()` le Lecteur doit mettre la valeur `true` pour lui-même dans le `Map readers` pour indiquer qu'il a réussi à s'exécuter. Et s'il constate que `readyToWrite()` retourne `true` il doit libérer (`notify()`) le premier thread Rédacteur sur la liste `writers` (et le supprimer de la liste).

3. voir les explications sur les moniteurs de Rédacteurs
4. voir les explications sur le moniteur de Lecteurs
5. voir les explications sur le moniteur de Lecteurs