Threads en Java
TP3 : Synchronisation (suite)*

L'objectif de ce sujet est d'approfondir les notions de programmation concurrente en Java étudiées dans le cadre du TP précédent.

1 Erreur `IllegalMonitorStateException`

Étudier le code des fichiers fournis (`ExempleThread8` et classes annexes), qui implémente une variante d'un exemple étudié au TP précédent.
Ce code est incorrect. Expliquer pourquoi.
On pourra remarquer que ce code compile sans erreurs mais que des exceptions sont générées lors de son exécution.
Pour bien comprendre le problème, il est conseillé de lire la documentation des classes `java.lang.Object` [1] (notamment les méthodes `wait`, `notify` et `notifyAll`), `java.lang.IllegalMonitorStateException` [2] et `java.lang.RuntimeException` [3].

2 Le problème des lecteurs-rédacteurs

Il s'agit d'accès concurrents à une ressource partagée par deux types d'entités : les lecteurs et les rédacteurs. Les lecteurs accèdent à la ressource sans la modifier. Les rédacteurs, eux, modifient la ressource. Pour garantir un état cohérent de la ressource, plusieurs lecteurs peuvent y accéder en même temps mais l'accès pour les rédacteurs est un accès exclusif. En d'autres termes, si un rédacteur travaille avec la ressource, aucune autre entité (lecteur ou rédacteur) ne doit accéder à celle-ci. Le problème des lecteurs-rédacteurs est un problème classique de synchronisation lors de l'utilisation d'une ressource partagée. Ce schéma est typiquement utilisé pour la manipulation de fichiers ou de structures de données en mémoire.

Chercher une solution au problème des lecteurs rédacteurs en définissant une classe `SharedRsc` et quatre méthodes `begin_read`, `end_read`, `begin_write` et `end_write` qui réaliseront toute la synchronisation nécessaire.
Ces méthodes seront appelées par les threads lecteurs et rédacteurs avant (`begin`) et après (`end`) un accès en lecture ou en écriture à la ressource.

2.1 Exemple d'utilisation

*Cet énoncé est basé sur un document initialement rédigé par V. Danjean et V. Marangozova-Martin.
1. `https://docs.oracle.com/javase/8/docs/api/`
2. `https://docs.oracle.com/javase/8/docs/api/java/lang/IllegalMonitorStateException.html`
3. `https://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html`

```
class Fichier {
  SharedRsc access;
  ...
  void lecture() {
    access.begin_read();
    // Maintenant, lecture dans le fichier
    ...
    // D'autres threads peuvent etre en train de lire en meme temps
    // mais aucun ne peut etre en train d'ecrire
    access.end_read();
  }
  void ecriture() {
    access.begin_write();
    // Maintenant, ecriture dans le fichier
    ...
    // aucun autre thread ne peut etre en train de lire ni d'ecrire
    // en meme temps
    access.end_write();
  }
}
```

2.2 Exercice

Écrire la classe `SharedRsc`. Ensuite, écrire un programme principal qui instancie cette classe puis qui crée de nombreux threads dont certains vont lire (appels à `begin_read` puis `end_read`) et d'autres vont écrire (appels à `begin_write` puis `end_write`). Faire en sorte que l'on puisse observer le déroulement des synchronisations (par exemple avec des appels à `System.out.println` judicieusement placés).

Garder des traces montrant que certains threads sont effectivement endormis (et réveillés plus tard) lorsqu'ils essayent d'accéder à la ressource alors qu'ils n'y ont pas droit.

Remarque : Le concept de verrou lecteurs-rédacteurs existe dans la bibliothèque standard Java (avec une API légèrement différente). Voir notamment la documentation de l'interface `ReadWriteLock`[4] et de la classe `ReentrantReadWriteLock`[5]. Le but de l'exercice ci-dessus est d'implémenter vous-même ce concept.

3 Moniteurs et variables de condition

Comme étudié au cours des TP précédents, un moniteur Java n'est associé qu'à une seule variable de condition, ce qui peut être limitant dans certaines situations. Le but de cet exercice est d'étudier comment on peut contourner cette restriction.

Une solution possible consiste à utiliser le package `java.util.concurrent.locks`, qui fournit l'interface `Lock`. À partir d'un objet qui implémente cette interface (voir notamment la classe `ReentrantLock`), il est possible de créer un ou plusieurs objets `Condition`. Chaque objet `Condition` correspond à une variable de condition sur laquelle on peut bloquer/débloquer des threads.

4. `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html`
5. `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html`

3.1 Tampon producteurs-consommateurs

Un premier cas d'étude considéré sera le tampon producteurs-consommateurs développé au cours du TP précédent (version avec utilisation directe des moniteurs, sans sémaphores). L'objectif est d'obtenir une implémentation correcte de la synchronisation des accès au tampon sans utiliser `notifyAll` (l'utilisation de cette primitive est souvent inefficace car elle provoque de nombreux réveils inutiles).

Programmer une nouvelle version du tampon producteurs-consommateurs basée sur l'utilisation des interfaces `Lock` et `Condition`.

Remarque : Avant de commencer, il est recommandé de lire très attentivement la documentation Java concernant l'interface `Lock`[6] et l'interface `Condition`[7].

3.2 Lecteurs-rédacteurs

Programmer ensuite une nouvelle version du problème des lecteurs-rédacteurs, en utilisant les interfaces `Lock` et `Condition`.

4 Compléments sur les lecteurs-rédacteurs

Écrire le code de trois autres variantes qui résolvent le problème des lecteurs/rédacteurs (cf. première section) mais garantissent également d'autres propriétés :

1. **Priorité aux lecteurs :** Un rédacteur ne peut commencer à écrire que lorsqu'aucun lecteur n'est en train de manipuler la ressource (ou en attente pour le faire)
2. **Priorité aux rédacteurs :** Un rédacteur demandant à écrire est autorisé à le faire dès que tous les lecteurs et/ou rédacteurs actuels auront fini leur section critique (aucun nouveau lecteur n'est admis en section critique si un rédacteur désire écrire) ;
3. **Ordre FIFO garanti :** Les lecteurs et les rédacteurs passent dans l'ordre de leur demandes (si plusieurs lecteurs arrivent à la suite, ils doivent bien évidemment passer ensemble, sans attendre inutilement).

5 (Optionnel) API BlockingQueue

Écrire un programme Java qui met en œuvre des interactions entre des threads producteurs et des threads consommateurs via un tampon partagé. Contrairement aux exercices du TP précédent, il n'est pas demandé d'écrire le code de gestion du tampon mais au contraire d'utiliser une implémentation fournie par la bibliothèque Java : interface `BlockingQueue` et classe `ArrayBlockingQueue`. Lire la documentation correspondante et tester les différentes sémantiques pour les opérations de dépôt d'un élément (`add`, `offer`, `put`, `offer` avec timeout) et de retrait d'un élément (`remove`, `poll`, `take`, `poll` avec timeout).

6. `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html`
7. `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html`

Annexe A : Pour aller plus loin

Pour approfondir l'étude des mécanismes de synchronisation avancés disponibles dans la bibliothèque Java, vous pouvez consulter les liens suivants :

- `https://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/package-summary.html`
- `http://tutorials.jenkov.com/java-util-concurrent/index.html`
- `http://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html`