Synchronisation de threads en Java
TP2 et TP3 : Pratique

V. Danjean R. Lachaize

Généralités

Les parties 1 et 2 sont des rappels de cours avec la présentation de la syntaxe Java pour la synchronisation. La partie 3 contient un ensemble de petits exercices pour apprendre à synchroniser les threads en Java. La partie 4 est dédiée au problème des lecteurs/rédacteurs. La partie 5 est un rappel sur la syntaxe Java pour manipuler les threads.

Table des matières

1 Moniteurs Java

La synchronisation en Java est faite en utilisant les moniteurs. En effet, chaque objet Java a un moniteur (c'est-à-dire un mutex et une condition) qui lui est associé.

Pour entrer dans un moniteur (c'est-à-dire pour prendre le mutex), il suffit d'exécuter une méthode qui possède le mot-clé `synchronized`. On sort du moniteur (on relâche le mutex) lorsque l'on sort de la méthode. Cela signifie qu'il n'est pas possible pour deux threads différents d'exécuter simultanément des méthodes marquées avec le mot-clé `synchronized` sur un même objet.

Le moniteur java est réentrant. Cela signifie qu'un thread peut réacquérir un mutex qu'il possède déjà. En pratique, cela signifie qu'une méthode avec le mot-clé `synchronized` peut appeler une autre méthode du même objet qui possède aussi le mot-clé `synchronized` sans créer un interblocage.

```
class Compte {
    private double solde;
    Compte(double i) {solde = i;}

    synchronized void  deposer(double montant) {
        solde = solde + montant;
    }
    synchronized void  retirer(double montant) {
        solde = solde - montant;
    }
    double consulter() {return solde;}
}
```

Ce code spécifie que les méthodes `deposer` et `retirer` ne peuvent pas être exécutées simultanément par deux threads différents pour une même instance de `Compte`. Par exemple, si un thread exécute `deposer` sur un objet de la classe `Compte`, alors tous les threads essayant d'exécuter `deposer` et `retirer` *sur ce même objet* seront mis en attente.

Les threads `ThreadDeposer` et `ThreadRetirer` modifient tous l'attribut solde. Mais comme ces modifications ne sont faites que par l'intermédiaire des méthodes avec le mot clé `synchronized`, cela assure que les modifications soient assurées de manière atomique.

2 Conditions : wait et notify/notifyAll

En plus du mutex associé à chaque objet Java, il y a une condition. Les opérations *wait*, *signal* et *broadcast* décrite en cours sont invoquée en Java à l'aide respectivement des primitives `wait()`, `notify()` et `notifyAll()`. On peut noter que ces primitives n'ont pas de paramètre puisqu'elles manipulent le mutex et la condition associés à l'objet courant.

2.1 Attention aux détails suivants

- lors du réveil l'ordre FIFO n'est pas garanti ;
- en Java, les moniteurs définissent une file d'attente (une condition) unique ;
- Les threads doivent coopérer i.e. si des threads appellent `wait()`, d'autres threads doivent appeler `notify()`/`notifyAll()`.

2.2 Méthode wait

Le fonctionnement de la méthode `wait` est le suivant :

1. nécessite que le thread possède le moniteur i.e on est dans un bloc synchronized ;
2. *de manière atomique*, bloque le thread en relâchant le moniteur (comme cela d'autre threads peuvent l'acquérir) ;
3. une fois réveillé, réacquiert le moniteur ;

L'utilisation correcte de wait est généralement la suivante :

```
while (!test) {
                    wait();
}
```

L'utilisation avec un `if` est généralement incorrecte puisque, entre le moment où un thread est réveillé et le moment où il réaquiert le moniteur, il est possible que le test soit changé (par exemple parce qu'un autre thread a pris le moniteur entre-temps).

```
if (!test) wait(); // GÉNÉRALEMENT FAUX
```

3 Exercices simples

3.1 exempleThread6.java

Dans cet exemple, nous reprenons l'exemple de la semaine précédente de manipulation de comptes par des threads `ThreadRetirer` et `ThreadDeposer`.

Dans cet exemple, nous avons modifié le programme de manipulation de compte pour interdire le retrait s'il n'y a pas assez d'argent. Quand le solde est insuffisant, les threads `ThreadRetirer` attendent (`wait` dans la méthode retirer), ils sont réveillés par les threads `ThreadDeposer` qui appellent la méthode `deposer` contenant un appel à `notify`.

Compiler et observer l'exécution de threadExemple6.

Essayer de trouver un cas où un appel à retirer est fait avec un solde 0. Que se passe-t-il ?

3.2 exempleThread7.java

Ce programme est quasi identique au programme précédent. La différence tient dans le fait que pour la vérification de la condition avant wait, nous avons remplace le while par un if. Compiler et exécuter. Y a-t-il des cas où le solde devient négatif ? Expliquer.

3.3 Sémaphores en Java

Cet exercice est à faire initialement sur feuille papier, avant d'être programmé.

En utilisant les moniteurs Java i.e les mots clés `synchronized`, ainsi que les méthodes `wait` et `notify/notifyAll`, écrire une classe `Semaphore` qui définit une structure de sémaphore en Java. Le sémaphore contient un compteur (initialisé à une valeur positive ou nulle par le constructeur), ainsi que deux méthodes P et V.

3.4 Le problème du producteur/consommateur

Programmer le problème du producteur/consommateur (vu en cours) en utilisant les sémaphores. Pour cela, écrire une classe `Stockage` contenant un tableau d'Object ainsi que les méthodes `Object Consomme()` et `void Produit(Object)`.

Écrire un programme principal qui instancie un objet `Stockage` puis qui crée de nombreux threads dont certains vont produire et d'autres consommer des objets. Faites en sorte que l'on puisse observer le déroulement des synchronisations (par exemple avec des `System.println()` judicieusement placés). Pour cela, vous aurez probablement à ralentir vos threads (en insérant des appels à `sleep(int)` à l'intérieur des sections critiques ou entre les productions/consommations d'objets). Note : vous pouvez utiliser la classe `Random` (voir la javadoc) pour générer des temps d'attente aléatoires.

Gardez des traces montrant que certains threads sont effectivement endormis (et réveillés plus tard) lorsqu'ils essayent de consommer alors que le stockage est vide ou bien au contraire lorsqu'ils essayent de produire alors que le stockage est plein.

3.5 Le problème du producteur/consommateur (bis)

Réécrivez le problème du producteur/consommateur en utilisant directement les moniteurs Java (i.e. sans utiliser la classe `Semaphore`).

4 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 zones 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 (et après) un accès en lecture ou en écriture à la ressource.

4.1 Exemple d'utilisation

```
class Fichier {
  SharedRsc access;
  ...
  void Lecture() {
    access.begin_read();
    // Maintenant, lecture dans le fichier
    ...
    // D'autres threads peuvent être en train de lire en même temps
    // mais aucun ne peut être en train d'écrire
    access.end_read();
```

```
  }
  void Ecriture() {
    access.begin_write();
    // Maintenant, écriture dans le fichier
    ...
    // aucun autre thread ne peut être en train de lire ni d'écrire
    // en même temps que nous.
    access.end_write();
  }
}
```

4.2 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`). Faites en sorte que l'on puisse observer le déroulement des synchronisations (par exemple avec des `System.println()` judicieusement placés).

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

4.3 Pour aller plus loin

Écrire trois autres classes résolvant le problème des lecteurs/rédacteurs mais garantissant en plus d'autres propriétés telles que :

1. priorité aux lecteurs : un rédacteur ne pourra commencer à écrire que lorsqu'aucun lecteur ne sera présent (y compris en attente) ;
2. priorités aux rédacteurs : un rédacteur demandant à écrire sera autorisé à le faire dès que tout les lecteurs et/ou rédacteurs actuels auront fini leur section critique (i.e. aucun nouveau lecteur ne sera admis si un rédacteur désire écrire) ;
3. ordre FIFO garanti : les lecteurs et les rédacteurs passent dans l'ordre de leur demande (si plusieurs lecteurs arrivent à la suite, ils doivent bien évidemment passer ensemble).

5 RAPPELS : Threads en Java

Les threads permettent d'avoir plusieurs activités en parallèle dans un programme. Les threads Java peuvent être implémentés de deux manières.

Utilisation de la classe `Thread` La première méthode est d'étendre la classe prédéfinie `Thread` :

```
//classes et interfaces prédéfinies Java, JDK
    interface Runnable {
        void run();
    }
    public class Thread extends Object
                        implements Runnable {
        void run() {...}
        void start() {...}
        ...
```

```
}

public class Compteur extends Thread {
    public void run() {...}
    }
    public static main() {
        Compteur c = new Compteur();
        c.start();
    }
}
```

L'héritage à partir de `Thread` est contraignant car il empêche tout autre héritage (en Java, une classe ne peut hériter qu'une seule autre classe).

Utilisation de l'interface `Runnable` La deuxième manière de faire est d'implémenter l'interface `Runnable`. Ceci permet l'héritage d'autres classes et et l'implémentation d'autres interfaces.

```
interface Runnable {
    void run();
}
public class Thread extends Object
                    implements Runnable {
    void run() {...}
    void start() {...}
    ...
}

public class Compteur implements Runnable{
    public void run() {...}
    }
    public static main() {
        Compteur c = new Compteur();
        new Thread(c).start();      }
    }
}
```