Threads en Java
TP2 : Synchronisation *

Vue d'ensemble

Les parties 1 et 2 sont des rappels/compléments 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.

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 variable de 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 définies avec le mot-clé `synchronized` sur un même objet.

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` ou `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 sont effectuées de manière atomique.

*Cet énoncé est basé sur un document initialement rédigé par V. Danjean et V. Marangozova-Martin.

2 Variables de condition : `wait` et `notify`/`notifyAll`

En plus du mutex associé à chaque objet Java, il y a une variable de condition. Les opérations *wait*, *signal* et *broadcast* décrites 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 variable de 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 variable de condition) unique ;
- les threads doivent coopérer : 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 : l'appel doit être fait au sein d'un bloc `synchronized` ;
2. *de manière atomique*, bloque le thread en relâchant le moniteur (comme cela d'autres 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 réacquiert le moniteur, il est possible que le test soit changé (par exemple parce qu'un autre thread a pris le moniteur entre-temps). De plus, dans des cas assez rares mais possibles en pratique, l'implémentation de `wait` peut être sujette à des réveils intempestifs (*"spurious wakeups"*), non déclenchés par un appel à `notify` ou `notifyAll`.

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

3 Exercices simples

3.1 ExempleThread6.java

Dans cet exemple, nous considérons un cas assez proche de la séance précédente (cf. classes `ExempleThread4` et `ExempleThread5` qui manipulaient un tableau de nombres). Nous considérons ici un objet représentant un compte bancaire. L'état du compte est mis à jour par plusieurs threads de type `ThreadRetirer` et `ThreadDeposer`.

Ici, 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 `ExempleThread6`.

Essayer de trouver un cas où un appel à retirer est fait avec un solde nul. 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 remplacé le `while` par un `if`. Compiler et exécuter. Y a-t-il des cas où le solde devient négatif ? Expliquer.

3.3 Retour sur ExempleThread6.java

Reprenons maintenant le programme `ExempleThread6`. Dans le programme tel qu'il est fourni, tous les dépôts et les retraits effectués correspondent exactement au même montant. Le code de synchronisation est-il correct dans le cas plus général où les montants des opérations de dépôt et de retrait peuvent varier ? Expliquer.

3.4 Le problème des producteurs/consommateurs — version avec Moniteurs Java

Programmer le problème des producteurs/consommateurs (vu en cours) en utilisant les moniteurs Java. Pour cela, écrire une classe `Stockage` contenant un tableau d'éléments (par exemple, de type `Object`) ainsi que les méthodes `Object Consommer()` et `void Produire(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 appels à `System.out.println` judicieusement placés). Pour cela, vous aurez probablement à ralentir vos threads (en insérant des appels à `sleep` à l'intérieur des sections critiques ou entre les productions/consommations d'objets). Note : vous pouvez utiliser la classe `Random` (voir la documentation Java) 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 des producteurs/consommateurs — version avec sémaphores n°1

Implémenter une variante du programme précédent en utilisant cette fois-ci des sémaphores plutôt que des moniteurs Java. Dans cette première version, il est demandé d'utiliser la classe `Semaphore` fournie par la bibliothèque Java (`java.util.concurrent.Semaphore`).

3.5.1 Sémaphores en Java

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

En utilisant les moniteurs Java (les mots clés `synchronized`, ainsi que les méthodes `wait` et `notify/notifyAll`), écrire une classe `MySemaphore` qui définit une structure de sémaphore en Java. Le code écrit dans cet exercice sera utilisé et validé dans l'exercice suivant.

Dans un premier temps, on peut éventuellement simplifier le problème en faisant l'hypothèse que l'implémentation des moniteurs Java n'est pas sujette au problème des réveils intempestifs ("spurious wakeups"). Proposer ensuite une deuxième solution qui prend en compte le problème des réveils intempestifs.

3.6 Le problème des producteurs/consommateurs — version avec sémaphores n°2

Réécrire le code du problème des producteurs/consommateurs en utilisant cette-fois la classe `Semaphore` que vous avez programmée à l'exercice précédent plutôt que celle de la bibliothèque Java. **Important :** Effectuer ensuite une campagne de tests conséquente pour vérifier le bon fonctionnement de la classe `MySemaphore`.