Synchronisation de threads en Java
TP2 et TP3 : Pratique

V.Marangozova-Martin

1 Moniteurs Java

La synchronisation en Java est faite en utilisant les moniteurs. Dans le cours, nous avons vu qu'un moniteur est un objet contenant des variables et des procédures qui ne permet qu'un processus actif à la fois. Ceci veut dire que si un processus exécute une procédure du moniteur, tous les autres processus qui veulent exécuter des procédures (la même ou d'autres), attendent.

En Java, un moniteur est attaché à un objet et ce moniteur est activé à l'aide du mot-clé `synchronized`.

Dans le code suivant :

```
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, quand un thread exécute la méthode `deposer`, aucun autre thread ne peut s'exécuter. En effet, le thread acquiert le moniteur en début de méthode `deposer` et le relâche à la fin de la méthode.

2 Conditions : wait et notify/notifyAll

Dans certains cas de programmes concurrents, il est possible que les réveils et les blocages de threads dépendent de conditions. Dans ce cas, pour bloquer un thread on utilise la primitive `wait()` et pour réveiller les threads on peut utiliser `notify` ou `notifyAll`. `notify` réveille un seul processus, alors que `notifyAll` réveille tous les processus bloqués.

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 :

- elle ne peut être appelée que si le thread possède le moniteur i.e est dans un bloc `synchronized`
- à l'appel, `wait` bloque le thread
- qui relâche le moniteur (comme cela d'autres threads peuvent l'acquérir)

L'utilisation correcte de `wait` est la suivante :

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

L'utilisation avec un `if` n'est pas correcte puisque, entre le moment où un thread est réveillé et le moment où il rentre en SC, il est possible que la condition change (*vol de cycle*).

```
if (!condition) wait(); //FAUX
```

3 Exercices

3.1 exempleThread6.java

Dans cet exemple, nous reprenons l'exemple précédent 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 si 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 ? Faites au moins une dizaine d'exécution pour faire apparaître le problème. Expliquer.

3.3 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.

Programmer les lecteurs rédacteurs. Notamment, écrire une première solution qui définit une classe `RessourcePartagée` et deux méthodes `read` et `write` en exclusion mutuelle (`synchronized`). Ces méthodes seront appelées par les threads lecteurs et rédacteurs. Avec cette solution, est-il possible d'avoir des lectures en parallèle ?

Ecrire une deuxième solution qui dispose de deux compteurs : le nombre de lecteurs et le nombre de rédacteurs et qui utilise wait et notify.

4 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 predefinies 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();       }
        }
    }
```