Matthieu Zimmer
Jean-Phillipe Mangeot

Programmation concurrente en Java

Exercice 1 [*Threads - attente*] Écrivez une classe qui démarre deux threads. Le premier thread doit afficher dix fois "Bonjour !" en laissant passer une seconde entre chaque affichage. Le deuxième thread doit afficher cinq fois "Salut !", en lassant passer deux secondes entre chaque affichage. Utiliser Thread pour un des deux threads et Runnable pour l'autre.

Indication : La méthode statique sleep de la classe Thread permet de mettre un thread en attente. La méthode run() est appelée par un autre fil d'exécution lors de l'appel à start(). La méthode join() attend que la méthode run() termine.

Exercice 2 [*Variable partagée entre threads - synchronized*] On souhaite trouver toutes les occurrences d'un élément dans un tableau donné. Pour cela, si le tableau est long (plus qu'une constante TAILLE MIN), on peut le couper en deux et effectuer la recherche indépendamment dans les deux moitiés, en confiant la seconde partie à un nouveau thread travaillant en parallèle. Écrire un programme effectuant la recherche de cette manière ; le programme devra afficher un message pour chaque occurrence trouvée, puis afficher le nombre total d'occurrence l'élément dans le tableau.

Indication : Le mot-clé synchronized permet de protéger une section critique pour être sur qu'un seul fil d'exécution à la fois ne soit à l'intérieur.
synchronized(unObjet){
//section critique exécutée par un seul thread à la fois
}
Lorsque l'objet qui sert de verrou pour la synchronisation est this, et qu'il englobe tout le code d'une méthode, on peut mettre le mot-clé synchronized dans la signature de la méthode.

```
void methode() {
    synchronized(this) {
        //section critique
    }
}
```

```
synchronized void methode() {
    //section critique
}
```

Matthieu Zimmer
Jean-Phillipe Mangeot

Exercice 3 [*Synchronisation entre threads - notify/wait*] Il s'agit ici d'implémenter un système de producteurs/consommateurs. Ces producteurs/consommateurs communiquent par un buffer partagé pouvant contenir une chaîne de caractères. Si le buffer est vide, un producteur a le droit de le remplir et il signale ensuite à tout le monde qu'il l'a rempli, au contraire si le buffer contient une chaîne de caractères (c'est-à-dire qu'il est plein), le producteur doit attendre qu'un consommateur consomme ce qu'il y a dans le buffer. Quant au consommateur, il essaie de lire ce qu'il y a dans le buffer, si il y arrive, il vide le buffer et il signale cela à tout le monde, sinon il attend qu'un producteur produise une donnée dans le buffer.

1. Dans un premier temps, les classes Consommateur et Producteur en utilisant la classe ArrayBlockingQueue pour le buffer et testez les avec par exemple deux producteurs et deux consommateurs.
2. Dans un deuxième temps, réécrivez les classes Consommateur et Producteur avec cette fois-ce une classe Buffer implémentée par vos soins.

Indication : Pour cet exercice, on utilisera les méthodes wait() et notifyAll() de la classe Thread. 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 ;

Lorsque la méthode wait() est invoquée à partir d'une méthode synchronized, en même temps que l'exécution est suspendue, le verrou posé sur l'objet par lequel la méthode a été invoquée est relâché. Dès que la condition de réveil survient, le thread attend de pouvoir reprendre le verrou et continuer l'exécution.

L'utilisation correcte de wait() est généralement la suivante :

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

La méthode notify() réveille un seul thread.
Si plusieurs threads sont en attente, c'est celui qui a été suspendu le plus longtemps qui est réveillé. Lorque plusieurs threads sont en attente et qu'on veut tous les réveiller, il faut utiliser la méthode notifyAll.

Matthieu Zimmer
Jean-Phillipe Mangeot

Exercice 4 [*Problème des lecteurs-rédacteurs - lock*] 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.

Exemple d'utilisation :

```
class Fichier {
      SharedRsc access;

      void lecture () {
            access.begin_read();
            // Maintenant, lecture dans le #chier
            …
            // 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();
      }
}
```

Indications :
Utiliser ReadWriteLock en instanciant un ReentrantReadWriteLock.
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html