Université Paris-Diderot
M2 II
Programmation Répartie

TP

Producteur-Consommateur
et
Lecteur-Ecrivain

Vous devez déposer sur Didel le code Java de ce tp pour le 20 février. Vous pouvez travailler en groupe jusqu'à 3 et rendre un TP par groupe de 3.

Dans le package `java.util.concurrent.locks` se trouve l'interface Lock.

```
public interface Lock{
 public void lock();
 public void unlock();
 public void lockInterruptibly();
 public Condition newCondition();
 public boolean tryLock();
 public boolean tryLock(long time, TimeUnit unit);
 }
```

L'interface `Condition`, permet elle de définir les méthodes `await`, `signal` and `signalAll`. La classe `ReentrantLock` implémente l'interface `Lock`.

1 Producteur-consommateur

Le but de ce TP est d'utiliser l'interface `Lock` (et `Condition`) pour résoudre le problème du producteur-consommateur.

Les threads producteur créent des produits (par exemple un entier) et les placent dans une file d'attente. Chaque thread consommateur retire de la file d'attente un item produit par un producteur et le consomme (par exemple l'affiche). La file d'attente est bornée: lorsque la file d'attente est pleine un producteur se met en attente. Lorsqu'elle est vide un consommateur se met en attente.

Pour les besoins du TP on permettra de donner un débit différent aux producteurs et aux consommateurs (chaque thread fera de façon aléatoire des mises en sommeil (`sleep`) entre les productions et les consommations).

On a devra définir trois classes: `File` pour la file, `Prod` pour le producteur et `Cons` pour le consommateur.

La classe `File` contiendra les deux méthodes `mettre` et `enlever`. On pourra supposer que le producteur produit des objets de la classe `Integer` qui seront consommés par le consommateur.

1. Donner une implémentation de `File` en utilisant la classe `ReentrantLock` (on demande de ne pas utiliser directement de méthodes sur les objets comme synchronized, wait, ...).

2. Ecrire des classes `Prod`, `Cons` utilisant la classe `File` ainsi qu'une classe `Main` permettant de tester ces classes. On doit pouvoir lancer simultanément plusieurs producteurs et consommateurs.

3. Pour la remise du TP, fournir en plus du texte des programmes un exemple significatif d'exécution du programme et un fichier .jar qui devra pouvoir s'exécuter.

2 Lecteurs/Rédacteurs

On rappelle que dans ce problème, lecteurs et écrivains veulent accéder à une "ressource partagée". Un écrivain travaille en exclusion mutuelle avec les lecteurs et les écrivains. Un lecteur est en exclusion mutuelle avec les écrivains, mais plusieurs lecteurs peuvent accéder en même temps à la "ressource".

Dans le package `java.util.concurrent.locks` se trouve l'interface ReadWriteLock.

```
public interface ReadWriteLock{
 public Lock readLock();//Lock utilise pout lire
 public Lock writelock();//Lock utilise pour ecrire
 }
```

Un `ReadWriteLock` permet de résoudre le problème des lecteurs écrivains, lorsque les lecteurs accèdent à la ressource en utilisant le `readLock` et les ecrivains le `writeLock`.

Le but de ce TP est d'implementer cette interface avec différentes priorités pour les lecteurs et les écrivains en utilisant l'interface `Lock`.

1. Pour pouvoir tester l'implémentation du `ReadWriteLock` écrire le code d'un thread lecteur qui consulte et d'un thread écrivain qui modifie une base de données (représentée par un tableau d'entiers) en utilisant l'interface `ReadWriteLock`. Testez ces classes avec l'implémentation `ReentrantReadWriteLock` de l'interface `ReadWriteLock` .

2. Réalisez l'implémentation de `ReadWriteLock` avec `Lock` et `Condition` en assurant une *priorité des lecteurs sur les rédacteurs*: Si un lecteur se présente alors que des lecteurs sont en train de lire la base de données, le lecteur peut consulter la base. Les lecteurs peuvent donc occuper indéfiniment la base de données. On peut donc avoir une famine des écrivains. On fournira en plus du texte des programmes un exemple significatif d'exécution du programme utilisant l'implémentation de la question précédente et un fichier .jar qui devra pouvoir s'exécuter.

3. Réalisez l'implémentation de `ReadWriteLock` avec `Lock` et `Condition` en assurant une *priorité des rédacteurs sur les lecteurs*: si un lecteur utilise la base de données, tous les lecteurs nouveaux peuvent y accéder jusqu'à l'arrivée d'un rédacteur. A partir de ce moment tous les nouveaux arrivant attendent. Quand le dernier lecteur a fini, il réveille l'écrivain en attente. Quand cet écrivain a terminé il réveille un des processus en attente. On fournira en plus du texte des programmes un exemple significatif d'exécution du programme utilisant l'implémentation de la première question et un fichier .jar qui devra pouvoir s'exécuter.