Université Paris Diderot
M2
Programmation Répartie

TP32

Moniteurs et Lecteurs Redacteurs

On rappelle que dans ce problème, lecteurs et écrivains veulent accéder à une "ressource". 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 accédent à la ressource en utilisant le readLock et les ecrivains le writeLock.

Le but de ce TP est d'implementer cette interface avec différents priorités pour les lecteurs et les écrivains. en utilisant les moniteurs sur objet de Java (synchronized, wait, notify)

Exercice 1.— Utilisation de la classe Reentrant ReadWriteLock

On considère la classe suivante:

```
import java.util.concurrent.locks.*;
public class BD {
    int tab[];
    ReadWriteLock lock;
    public BD( int l){
        tab=new int[l];
          lock=new ReentrantReadWriteLock(true);
    }
}
```

des threads lectrices et rédactrices utilisent cette classe:

```
 public class Lect extends Thread{
    BD base;
    public Lect(BD b)
            {base=b;}

    public void run(){
        for (int tour=0; tour<10;tour++){
            base.lock.readLock().lock();
            try{
               System.out.print( " lecteur "+ThreadID.get()+" " );
               for (int i=0; i< base.tab.length;i++) System.out.print( base.tab[i]+" " );
               System.out.println(" '" );
               Thread.sleep((long)Math.random()*base.tab.length*1000);
            }
```

```
            catch(InterruptedException e){System.out.println( "interompu "+
                                                        ThreadID.get());break;}
            base.lock.readLock().unlock(); System.out.println( "verrou enleve"+
                                                        ThreadID.get()+" sort");
            try{
                this.sleep((long)Math.random()*base.tab.length*100);
            }
            catch(InterruptedException e){System.out.println( "interompu
                                                en dehors"+ ThreadID.get());}
            }
        }
    }
-------
public class Red extends Thread{
    BD base;
    public Red(BD b)
            {base=b;}
    public void run(){
        for (int tour=0; tour<10;tour++){
            base.lock.writeLock().lock();
            try{
                int j= (int) (Math.random()*100);
                System.out.println( " ecrivain "+ThreadID.get()+"ecrit" +j);
                for (int i=0; i< base.tab.length;i++) base.tab[i]=j;
                Thread.sleep((long)Math.random()*base.tab.length*1);
            }
            catch(InterruptedException e){System.out.println( " interompu "+
                                                        ThreadID.get()); break;}
            base.lock.writeLock().unlock();
            System.out.println( " verrou ecrivain enleve"+ThreadID.get());

            try{
              Thread.sleep((long)Math.random()*base.tab.length*100);
            }
            catch(InterruptedException e){System.out.println( "interompu en
                                                        dehors"+ ThreadID.get());}
            }
        }
    }
---
public class LectRed {
    public static final int TAILLE=2;

    public static void main(String[] args) {
        BD base=new BD(TAILLE);
        Lect lecteur[]=new Lect[TAILLE];
        Red ecrivain[]=new Red[TAILLE];
        for(int i=0;i<TAILLE;i++) {
             lecteur[i]=new Lect(base);
             ecrivain[i]=new Red(base);
        }
```

```
        for(int i=0;i<TAILLE;i++) {
            lecteur[i].start();
            ecrivain[i].start();
          }
        //com
    }
}
```

- Peux on avoir plusieurs écrivains en même temps dans la base de données?
- Peux on avoir plusieurs lecteurs en même temps dans la base de données?
- Peux on avoir des lecteurs et des ecrivains en même temps dans la base de données?
- Que se passe-t-il si dans le main de la classe LectRed à la place de

  `//com`

  on a :

  `lecteur[0].interrupt();`
- Modifier le code des classes Lec et Red afin que les verrous soient toujours relachés même en cas d'interruption.

Exercice 2.— Implementation de ReadWriteLock

On propose l'implementation suivante:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantLock;

public class TropSimple implements ReadWriteLock{
    private ReentrantLock l;
    public TropSimple()
    {
        l=new ReentrantLock();
    }
    public Lock readLock(){
        return l;
    }
    public Lock writeLock(){
        return l;
}
}
```

On remplace dans la classe BD

`lock=new ReentrantReadWriteLock(true);` par

`lock=new TropSimple();`

A-t-on toujours l'exclusion entre écrivains? entre lecteurs et écrivains? Plusieurs lecteurs peuvent-ils lire en même temps?

Exercice 3.— On propose l'implémetation suivante

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;

public class MonReadWriteLock implements ReadWriteLock{

private final MonReadWriteLock.ReadLock readerLock;
private final MonReadWriteLock.WriteLock writerLock;
private final Sync sync;
@Override
public MonReadWriteLock.WriteLock writeLock() { return writerLock; }
@Override
public MonReadWriteLock.ReadLock  readLock()  { return readerLock; }

public MonReadWriteLock(){
  sync=new Sync();
  readerLock=new MonReadWriteLock.ReadLock(this);
  writerLock=new MonReadWriteLock.WriteLock(this);
}

final static class Sync{
    private int readers  = 0;
    private int writers    = 0;
    public synchronized void lockR() {
        while(writers > 0 ){
            try{
                wait();
            }
            catch(InterruptedException e ){}
            }
        readers++;
        }
    public synchronized void unlockR() {
        readers--;
        notifyAll();
        }
    public synchronized void lockW() {
        while(readers > 0 || writers > 0){
            try{
                wait();
            }
            catch(InterruptedException e ){}
            }
        writers++;
        }
      public synchronized void unlockW() {
        writers--;
        notifyAll();
        }
}
```

```
public  static class ReadLock implements Lock{
    private final Sync sync;
    protected ReadLock(MonReadWriteLock lock) {
        sync = lock.sync;
    }
    @Override
    public void lock() {
        sync.lockR();
        }
    @Override
    public void unlock() {
        sync.unlockR();
        }
    @Override
    public Condition newCondition() {
        throw new UnsupportedOperationException();
        }
    @Override
    public void lockInterruptibly() throws InterruptedException {
        throw new UnsupportedOperationException("Not supported yet.");
        }
    @Override
    public boolean tryLock() {
        throw new UnsupportedOperationException("Not supported yet.");
        }
    @Override
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {
        throw new UnsupportedOperationException("Not supported yet.");
        }
}

public static class WriteLock implements Lock{
    private final Sync sync;
    protected WriteLock(MonReadWriteLock lock) {
          sync = lock.sync;
    }
    @Override
    public  void lock() {
        sync.lockW();
        }

    @Override
    public  void unlock() {
       // System.out.println( "sortie write "+ThreadID.get());
        sync.unlockW();
        }

    @Override
    public Condition newCondition() {
        throw new UnsupportedOperationException("Not supported yet.");
        }
```

```
   @Override
    public boolean tryLock() {
         throw new UnsupportedOperationException("Not supported yet.");
        }
    @Override
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {
        throw new UnsupportedOperationException("Not supported yet.");
        }
   @Override
    public void lockInterruptibly() throws InterruptedException {
        throw new UnsupportedOperationException("Not supported yet.");
        }
  }
}
```

Et on remplace dans la classe BD
`lock=new ReentrantReadWriteLock(true);` par
`lock=new MonReadWriteLock();`

- A-t-on l'exclusion entre écrivains? entre lecteurs et écrivains? Plusieurs lecteurs peuvent-ils lire en même temps?
- On suppose que des lecteurs lisent. Un écrivain A demande l'accés à la base de données puis un lecteur B. Dans cette implémentation A passera-t-il avant B? Y a t-il des executions dans lesquelles un écrivain n'a jamais accés à la base de données
- Ecrire une implémentation de ReadWriteLock dans laquelle il n'y a pas famine des écrivains i.e. quand des lecteurs lisent, si un écrivain A demande l'accès à la base de données plus aucun lecteur ne pourra accéder à la base avant qu'un écrivain n'y ait accédé. Une fois que l'écrivain a eu accès à la base il n'y a pas de priorité entre les lecteurs ou les écrivains pour l'accés suivant.
- On veut renforcer la priorité des écrivains. Non seulement quand des lecteurs lisent, si un écrivain A demande l'accès à la base de données plus aucun lecteur ne pourra accéder à la base avant qu'un écrivain n'y ait accédé. Mais si il y a un autre écrivain en attente ce sera un écrivain qui aura accés à la base. Y a t-il des executions dans lesquelles un lecteur n'a jamais accés à la base de données?

  Ecrire une implémentation de ReadWriteLock avec cette priorité des écrivains sur les lecteurs.