Université Paris Cité

Master 2 - Année 2023-24

Programmation répartie

TP2

Exclusion mutuelle

Exercice 1.— Volatile

```
public class Exo {

    public static boolean fait = false;
    public static int n;
//1
    public static class Lecteur extends Thread {
        public void run() {
            while (!fait);
            System.out.println(n);
        }
    }

   public static void main(String[] args) throws InterruptedException {
   //2
            new Lecteur().start();
            Thread.sleep(100);
            n = 150;
            fait = true;
            System.out.println("fait");
    }
}
```

Pour chaque question vous expliquerez pourquoi le comportement est possible ou impossible

1. Est ce que vous observez que la thread Lecteur ne termine pas ?
2. On ajoute en //1
   `public static Integer I=3;`
   et on remplace `while (!fait);` par `while (!fait) {synchronized(I){} ;}`
   Est ce que la thread Lecteur termine toujours ?
3. On ajoute en //1
   `public static Integer I=3;`
   et on remplace `fait = true;` par `synchronized(I){fait = true;}`
   Est ce que la thread Lecteur termine toujours ?
4. On remplace `public static boolean fait = false;` par
   `public static volatile boolean fait = false;`
   Est ce que la thread Lecteur termine toujours ?

5. On ajoute en //1

   `public static boolean [] t = new boolean[10];`

   et en //2 `t[0]=false;`

   et on remplace `while (!fait);` par `while (!t[0]);`

   et `fait=true;` par `t[0] = true;`

   Est ce que la thread Lecteur termine toujours ?

6. On ajoute en //1

   `public static volatile boolean [] t = new boolean[10];`

   et en //2 `t[0]=false;`

   et on remplace `while (!fait);` par `while (!t[0]);`

   et `fait=true;` par `t[0] = true;`

   Est ce que la thread Lecteur termine toujours ?

7. ```
   public class Exo2 {
       public static int n;
       public static volatile boolean [][] t = new boolean[10][20];

       public static class Lecteur extends Thread {
           public void run() {
           //1
          while (!t[0][0]);
               System.out.println(n);
           }
       }

      public static void main(String[] args) throws InterruptedException {
               t[0][0]=false;
               new Lecteur().start();
               Thread.sleep(100);
               n = 150;
               t[0][0] = true;
               System.out.println("fait");
       }
   }
   ```

   Est ce que la thread Lecteur termine toujours ?

8. On ajoute à Exo2 en //1 `boolean [] lt=t[0];`

   et on remplace `while (!t[0][0]);` par `while (!lt[0]);`

   Est ce que la thread Lecteur termine toujours ?

9. ```
   public class Exo {
       public static int n;
       public static class Essai{boolean a; int b;}
       public static class Essai2 {int c; Essai d;}
       public static volatile Essai2 fait=new Essai2();

       public static class Lecteur extends Thread {
   ```

```
        public void run() {
  //1
       while (! fait.d.a);
            System.out.println(n);
        }
    }

    public static void main(String[] args) throws InterruptedException {
            fait.d= new Essai();
            fait.d.a=false;
            new Lecteur().start();
            Thread.sleep(100);
             n = 150;
             fait.d.a = true;
            System.out.println("fait");

    }
}
```

Est ce que la thread Lecteur termine toujours ?

10. On ajoute en //1 `Essai x=fait.d;` et on remplace `while (! fait.d.a);` par `while (! x.a);`
    Est ce que la thread Lecteur termine toujours ?

Exercice 2.— Exclusion mutuelle pour n threads
Dans le package `java.util.concurrent.locks` se trouve l'interface Lock.

```
public interface Lock{
   public void lock();
   public void unlock();
    ........
}
```

1. Ecrire le code d'une thread qui rentre régulièrement en section critique en utilisant un verrou (une implementation de l'interface Lock). La section critique et la section non critique pourront être simulées par une mise en sommeil de la thread et l'impression d'un message indiquant le nombre de fois où cette thread est rentrée en section critique. Apres 20 entrées en section critique la thread s'arrête. Ecrire le code d'un programme qui lance un nombre variable de ces threads.
2. Utiliser l'implémentation de Java java.util.concurrent.locks.ReentrantLock de l'interface Lock pour exécuter votre programme (avec 10 threads).
3. Quelles sont les differences lors de l'execution sans demande d'équité ou avec demande de verrou équitable.
4. Pour implémenter le verrou on propose d'utiliser l'algorithme de la boulangerie de Lamport ci dessous (pseudo code) . (On a (i,j) $<<$ (k,l) si i<k ou si (i=k et j<l)).

```
class Backery implements Lock{
boolean flag[];
Label [] label;
public Bakery(int n){
      flag=new boolean [n];
      label=new Label[n];
```

```
        for (int i=0; i<n; i++){
              flag[i]=false; label[i]=0;}

    public void lock(){
        int i= ThreadId.get();
        flag[i]=true;
        label[i]= max (label[0],label[1],...label[n-1])+1;
        while (Il existe k avec flag[k]&&( label[k],k)<< (label[i],i));
        }
    public unlock(){
        flag[ThreadId.get()]=false;
    }
```

5. Répondre aux questions suivantes, si la réponse est positive donnez un exemple d'exécution qui donnerait ce résultat, si la réponse est négative expliquez pourquoi.
   (a) Si la thread i a écrit $label[i]$ à l'instant t et la thread j écrit $label[j]$ à l'instant $t' > t$, a-t-on toujours $label^t[i] < label^{t'}[j]$?
   (b) Si la thread i a écrit $label[i]$ à l'instant t et que à cet instant $flag[j] = false$ et la thread j écrit $label[j]$ à l'instant $t' > t$, a-t-on toujours $label^t[i] < label^{t'}[j]$?
   (c) Si la thread i a écrit $label[i]$ à l'instant t, combien de thread peuvent entrer en section critique après t et avant que la thread i n'entre en section critique. Donnez un exemple d'exécution où la thread laisse passer le plus grand nombre de thread avant de pouvoir rentrer en SC.
6. Ecrire l'implémentation de Lock par cet algorithme. Vous justifierez votre implémentation et fournirez des exemples d'exécution de l'algorithme.
7. L'implémentation de Lock par l'algorithme de Lamport donne-t-il un Lock Réentrant ? Quelle propriété n'est plus assurée.

Exercice 3.— Lecteurs et rédacteurs
Dans le problème " lecteurs et rédacteurs": lecteurs et rédacteurs veulent accéder à une "ressource". Un rédacteur travaille en exclusion mutuelle avec les lecteurs et les rédacteurs. Un lecteur est en exclusion mutuelle avec les rédacteurs, 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 rédacteurs le writeLock.
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
   }
}
```

- Peut on avoir plusieurs rédacteurs qui écrivent en même temps dans base.tab?
- Peut on avoir plusieurs lecteurs qui lisent en même temps dans base.tab?
- Peut on avoir des lecteurs et des rédacteurs qui accèdent en même temps à base.tab?
- Que se passe-t-il si dans le main de la classe LectRed on a à la place de

  `//com`

  on a

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

Exercice 4.— 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 rédacteurs? entre lecteurs et rédacteurs? Plusieurs lecteurs peuvent-ils lire en même temps?

Exercice 5.— On propose l'implémentation 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 rédacteurs? entre lecteurs et rédacteurs? Plusieurs lecteurs peuvent-ils lire en même temps?
- On suppose que des lecteurs lisent. Un rédacteur A demande l'accès 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 rédacteur n'a jamais accès 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 rédacteur 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 le rédacteur a eu accès à la base il n'y a pas de priorité entre les lecteurs ou les rédacteurs pour l'accès suivant.