TP2

Hugo Jacotot : 71802786

Matthieu Le Franc : 71800858

Exercice 1

1/. Est ce que vous observez que la thread Lecteur ne termine pas ?

Effectivement, la thread lecteur ne termine pas car le thread lecteur et modifie thread lecteur qui ne va jamais relire pendant ce temps.

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 ?

Si le main s'exécute d'abord elle termine car on lit la valeur en mémoire partagée et sinon elle ne termine pas. Donc pas 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 ?

La thread lecteur ne termine pas car elle ne relit pas donc ça ne change rien de synchronize sur la thread qui écrit, il faut le faire sur la thread de lecture.

4/. On remplace `public static boolean fait = false;` par `public static volatile boolean fait = false;` Est ce que la thread Lecteur termine toujours ?

Oui car volatile permet de lire la valeur en mémoire partagée, la dernière valeur écrite. Donc quand la valeur est mise à jour, le thread peut la lire.

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 ?

Ca ne termine pas car on ne relit pas la valeur de t[0] donc on ne voit pas la modification.

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 ?

Oui car volatile permet de lire la valeur en mémoire partagée, la dernière valeur écrite. Donc quand la valeur est mise à jour, le thread peut la lire.

7/. Est ce que la thread Lecteur termine toujours ?

```
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");
    }
}
```

Oui car volatile permet de lire la valeur en mémoire partagée, la dernière valeur écrite. Donc quand la valeur est mise à jour, le thread peut la lire.

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 ?

Non car lt n'est pas volatile donc on ne lit pas la valeur en mémoire partagée.

9/. Est ce que la thread Lecteur termine toujours ?

Non car la thread lecteur ne voit pas la dernière modification donc elle ne termine pas.

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 ?

```
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");
        }
    }
```

Ne termine pas car x n'est pas volatile donc on ne lit pas la valeur en mémoire partagée.

Exercice 2

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.

Implémentation de l'interface Lock :

```
public class Exo2Q1 implements Lock, Runnable {
    private boolean[] flag;
    private int nb;
    volatile int victim;

    public Exo2Q1() {
        this.nb = 0;
        this.flag = new boolean[2];
    }

    public void run() {
        while (nb < 20) {
            lock();

            // Section critique
            try {
                System.out.println("la thread entre en section critique pour la "
+ nb + " fois");
                nb++;
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
```

```
                unlock();
            }
        }

        // Simulation en dehors de la section critique
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void lock() {
        int i = ThreadId.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i ;
        while (flag[j] && victim == i) {}; // wait
    }

    @Override
    public void unlock() {
        int i = ThreadId.get();
        flag[i]=false; // I'm not interested
    }

    @Override
    public void lockInterruptibly() throws InterruptedException {}

    @Override
    public boolean tryLock() {return false;}

    public static void main(String[] args) {
        Exo2Q1 lock = new Exo2Q1();
        for (int i = 0; i < 10; i++) {
            new Thread(lock).start();
        }
    }

    @Override
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method 'tryLock'");
    }

    @Override
    public Condition newCondition() {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method
'newCondition'");
    }
}
```

En utilisant `java.util.concurrent.locks.ReentrantLock` :

```
public class ThreadCrit {
    public static class Critique implements Runnable {
        private Lock lock;
        private int nb;
        private int id;

        public Critique(Lock lock, int id) {
            this.lock = lock;
            this.nb = 0;
            this.id = id;
        }

        public void run() {
            while (nb < 20) {

                lock.lock();

                // Section critique
                try {
                    System.out.println("la thread " + id + " entre en section
critique pour la " + nb + " fois");
                    nb++;
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    lock.unlock();
                }
            }

            // Simulation en dehors de la section critique
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        Lock lock = new ReentrantLock();
        for (int i = 0; i < 10; i++) {
            new Thread(new Critique(lock, i)).start();
        }
    }
}
```

2/. Utiliser l'implémentation de Java java.util.concurrent.locks.ReentrantLock de l'interface Lock pour exécuter votre programme (avec 10 threads).

Voir ci-dessous ou `Exo2Q2.java`

3/. Quelles sont les differences lors de l'execution sans demande d'équité ou avec demande de verrou équitable ?

Avec demande d'équité :

- Les threads sont exécutées dans l'ordre d'arrivée (approche FIFO)
- Coût de performances car gestion plus complexe de la file d'attente (changements de contexte plus fréquents)
- Tous les threads finiront par obtenir le verrou (pas de starvation)
- Permet d'avoir un comportement plus prévisible

Sans demande d'équité :

- Le thread accèdant au verrou est le plus prêt donc pas forcément celui qui attend depuis le plus longtemps (approche LIFO)
- Coût de performances réduit car gestion plus simple de la file d'attente (moins de changements de contexte)
- Possibilité de starvation (un thread peut attendre indéfiniment)

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/.

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 `labelt[i]<labelt'[j]` ?

Oui car `label[i]` est incrémenté à chaque fois qu'une thread entre en section critique donc on a toujours `labelt[i] < labelt'[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 `labelt[i]<labelt'[j]`

- Condition Initiale : Au moment t, la thread i écrit label[i], et à cet instant flag[j] = false.
- Ensuite, à un instant t' > t, la thread j écrit label[j].
- Lorsque la thread i écrit label[i] à l'instant t, elle se prépare à entrer en section critique et flag[j] = false signifie que la thread j ne se prépare pas à entrer en section critique.
- Lorsque la thread j décide d'entrer en section critique, elle met flag[j] = true

Donc la réponse est oui car même si `flag[j] == false`, lorsque i a écrit `label[i]`, on aura toujours `labelt[i] < labelt'[j]` lorsque j écrit ensuite `label[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.

- La thread i a écrit label[i] à l'instant t et se prépare à entrer en section critique mais elle doit attendre que toutes les threads avec des labels inférieurs terminent leur exécution en section critique.
- Si d'autres threads commencent le processus pour entrer en section critique après t mais avant que i entre, on leur attribue des labels supérieurs à celui de i.

Exemple d'exécution avec 4 threads :

- à l'instant t, la thread 1 écrit.
- avant l'instant t, les threads 0, 2 et 3 ont déjà écrits leurs labels mais tous leurs flags étaient false donc elles n'attendaient pas activement pour entrer en section critique.
- à t+1, les threads 0, 2 et 3 mettent leurs flags à true et se préparent à entrer en section critique.
- même si la thread 1 a écrit son label avant les autres, elle devra attendre que les threads 0, 2 et 3 terminent leur exécution en section critique avant de pouvoir y entrer elle même.

6/. Ecrire l'implémentation de Lock par cet algorithme. Vous justifierez votre implémentation et fournirez des exemples d'exécution de l'algorithme.

```
public class Exo2Q6 {
    // Flag de chaque thread
    private boolean[] flag;
    // Label de chaque thread
    private int[] label;
    // Nombre de threads
    private int n;
```

```
    public Exo2Q6(int n) {
        this.n = n;
        flag = new boolean[n];
        label = new int[n];
    }

    public void lock() {
        int i = ThreadId.get();
        System.out.println("Thread " + i + ", n = " + n);
        flag[i] = true;
        label[i] = max(label) + 1;
        for (int j = 0 ; j < n ; j++) {
            while (j != i && flag[j] && (label[j] < label[i] || (label[j] ==
label[i] && j < i)));
        }
    }

    public void unlock() {
        flag[ThreadId.get()] = false;
    }

    private int max(int[] tab) {
        int max = tab[0];
        for (int i = 1; i < tab.length; i++) {
            if (tab[i] > max) {
                max = tab[i];
            }
        }
        return max;
    }
}
```

Exemple d'exécution avec 4 Threads :

- Thread 1 appelle lock() et entre en section critique (son label est 1)
- Thread 2 appelle lock() et entre en section critique (son label est 2)
- Thread 3 appelle lock() et entre en section critique (son label est 3)
- Thread 1 appelle unlock() et sort de la section critique
- Thread 4 appelle lock() et entre en section critique (son label est 4)
- Thread 2 appelle unlock() et sort de la section critique
- Thread 3 appelle unlock() et sort de la section critique
- Thread 4 appelle unlock() et sort de la section critique

7/. L'implémentation de Lock par l'algorithme de Lamport donne-t-il un Lock Réentrant ? Quelle propriété n'est plus assurée ?

Non, l'implémentation de Lock par l'algorithme de Lamport ne donne pas un Lock Réentrant car il n'est pas possible pour une thread de réacquérir le verrou si elle l'a déjà acquis. Si un thread détient déjà le verrou, pour le réacquérir il doit le libérer le même nombre de fois qu'il a été acquis pour que le verrou soit réellement libéré.

Exercice 3

Peut on avoir plusieurs rédacteurs qui écrivent en même temps dans base.tab ?

- Non car red.java prend le verrou en écriture lors de son entrée en section critique

Peut on avoir plusieurs lecteurs qui lisent en même temps dans base.tab ?

- Oui car le verrou en lecture n'est pas

Peut on avoir des lecteurs et des rédacteurs qui accèdent en même temps à base.tab ?

- Non car le verrou en écriture est en exclusion mutuelle avec les verrous en lecture

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();`

- On obtient un deadlock lors de l'exécution car lorsque le thread est interrompu il ne relâche pas son verrou, ce qui empêche tous les autres de rentrer en section critique

Modifier le code des classes Lec et Red afin que les verrous soient toujours relâchés même en cas d'interruption.

- J'ai retiré le `break` qui provoquait la sortie de la boucle lors d'une iterruption d'un thread. Ainsi le thread continue l'exécution de la boucle for et n'est pas bloqué lors de la prochaine acquisition du lock, car celui-ci est réentrant.

Exercice 4

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 ?

Tous les threads seront en exclusion mutuelle car plus aucune différence n'est faite entre les verrous en lecture et les verrous en écriture.

En utilisant `lock=new TropSimple();` :

- On assure toujours l'exclusion entre rédacteurs car `TropSimple` utilise un seul `ReentrantLock` donc un seul rédacteur peut détenir le verrou à la fois, ce qui empêche d'autres rédacteurs d'entrer dans les sections critiques en même temps.
- On assure toujours l'exclusion entre lecteurs et rédacteurs un lecteur tenant le verrou empêche un rédacteur d'entrer dans la section critique et inversement.
- Plusieurs lecteurs ne peuvent pas lire en même temps car plusieurs threads peuvent acquérir le verrou de lecture tant qu'aucun thread n'a acquis le verrou d'écriture.

Exercice 5

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

-Entre rédacteurs : oui car les rédacteurs ne peuvent pas rentrer en section critique si il existe déjà un lecteur ou un rédacteur étant en section critique

- Entre lecteurs et rédacteurs : oui car les lecteurs ne peuvent pas rentrer en section critique si il existe déjà un rédacteur en section critique et les rédacteur doivent être les seuls étant en section critique.
- Plusieurs lecteurs peuvent lire en même temps car les lecteurs ne se préoccupent que de savoir s'il existe déjà un rédacteur en section critique.

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 ?

- B rentre en premier car plusieurs lecteurs peuvent avoir accès à la ressource en même temps
- Il y a des exécutions dans lesquelles un rédacteur n'a jamais accès à la base de données car les lecteurs rentrent directement. Si des lecteurs continuent d'affluer, le rédacteur ne pourra jamais rentrer en section critique

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.

Nous avons ajouté un booléen indiquant si un rédacteur a demandé l'accès à la section critique sans l'avoir obtenu, ainsi les rédacteurs seront prioritaires lorsque tous les lecteurs auront quitté la section critique.

Voir code répertoire exo5/