Université Paris-Diderot
M2 II
Programmation Répartie

TP1

Rappel Java et Atomicité

On demande pour ce TP de rendre les programmes (avec les commentaires appropriés) correspondant à la section 4. Pour les autres sections on demande de faire des réponses succinctes. Ce TP doit être rendu sur le site du cours sur Moodles PROGREP pour le 1 février. Il peut être rendu par groupes d'au plus 4 personnes

1 Class ThreadLocal<T>

On propose la classe suivante pour donner un identifiant à chaque thread que l'on crée. Cette classe sera utilisée dans la suite. Vérifier que si n threads appellent la méthode `get` elles obtiendront une identité entre 0 et n. Qu'est ce que `ThreadLocal<Integer>`? Que peut-il se passer si on supprime les mots clés `volatile`, `synchronized`?

```
public class ThreadID {
    private static volatile int nextID=0;
    private static class ThreadLocalID extends ThreadLocal<Integer>{
        protected synchronized Integer initialValue(){
            return nextID ++;
        }
    }
    private static ThreadLocalID threadID =new ThreadLocalID();
    public static int get(){
        return threadID.get();
    }
    public static void set (int index){
        threadID.set(index);
        }
}
```

2 Atomicité et méthodes *synchronized*

On considère l'implémentation en Java d'une file:

```
public class FileConcur {
    final static int QSIZE=20000;
    int head=0;
    int tail=0;
    int[] cell= new int [QSIZE];
    public void enq(int o){
        cell[(tail++)%QSIZE]=o;
    }
    public int deq(){
       return  cell[(head++)%QSIZE];
    }
}
```

Avec:

```
public class MyThread extends Thread{
    public FileConcur f;
    public int nb;
    public MyThread( int nb, FileConcur f){
        this.nb=nb;
        this.f=f;
    }
    public void run(){
        System.out.println("Je suis la thread " +ThreadID.get());
        int k=ThreadID.get()+1;
        for (int i=0;i<nb;i++)
            { f.enq(k);
          this.yield();
        }
        int x=0;
        for (int i=0;i<nb;i++)
            {x=x + f.deq();
            this.yield();
            }
        System.out.println("Thread "+ThreadID.get()+ " somme " +x);

    }
}

public class Main {
    public static void main(String[] args) {
        FileConcur f= new FileConcur();
        MyThread W= new MyThread(10000,f);
        MyThread R= new MyThread(10000,f);
        W.start();R.start();
        try{R.join();W.join();} catch(InterruptedException e){};
        System.out.println( "file utilisée par deux threads "+
                                        f.head +" tete/fin "+f.tail);

    }
}
```

- Si les instructions cell[(tail++)%QSIZE]=o; et cell[(head++)%QSIZE]; étaient atomiques la classe FileConcur implémenterait-elle correctement une file utilisée par plusieurs threads (on suppose qu'on ne met pas plus de QSIZE éléments dans la file et qu'on enlève uniquement des éléments quand la file n'est pas vide)
- Si les instructions cell[(tail++)%QSIZE]=o; et cell[(head++)%QSIZE]; étaient atomiques quelles seraient les valeurs de head et de tail affichées lors de l'exécution de Main.main.
- Après exécution de Main.main quelles sont les valeurs de head et de tail effectivement affichées. Qu'en déduisez vous sur l'atomicité en java.
- L'exécution précédente est elle encore possible si on ajoute "synchronized" lors des définitions de `enq` et `deq` dans FileConcur?

3 Atomicité et blocs *synchronized*

On définit la classe MonObjet grâce à laquelle les threads partagent un objet.

```
public class MonObjet {
    ThreadLocal<Integer> last;//nb ecriture de chaque thread
    int value;//valeur commune
    int value2;//valeur commune
    public MonObjet(int init){
        value=init;value2=init;
        last=new ThreadLocal<Integer>(){
            protected Integer initialValue() {return 0;}};
    };
    public int read(){ return value;}
    public void add( ){
        last.set(new Integer(last.get()+1));
        value=value +1;
        value2=value2 +1;
         }
    }
```

avec

```
public class MyThread2 extends Thread{
    public MonObjet  o;
    public int nbwrite;
    public MyThread2( MonObjet  o,int nbwrite){
        this.o=o;
        this.nbwrite=nbwrite;
    }
    public void run(){
        System.out.println("Je suis la thread " +ThreadID.get());
        for(int i=0;i<nbwrite;i++)
        { i
          o.add();
          this.yield();
          }
        System.out.println("la thread "+ThreadID.get()+
             " valeurs  "+ o.value+" et " +o.value2 +
              " pour sa "+ o.last.get()+" eme ecriture (derniere) ");
}
}
 public class Main2 {
  public static void main(String[] args) {
        MonObjet  o= new MonObjet(0);
        MyThread2 W,R;
        W= new MyThread2(o,10000);
        R= new MyThread2(o,10000);
        W.start();R.start();
        try{R.join();W.join();} catch(InterruptedException e){};

    }
```

```
}
```

- Si l'instruction x=x+1 était atomique, qu'elles devraient être les valeurs de o.value et o.value2 affichées lors de l'exécution de Main2.main?
- Après avoir exécuté le Main2.main qu'elles sont les valeurs affichées?
- L'instruction x=x+1 était-elle atomique en java?
- Comment assurer qu'à la fin du Main2.main on ait o.value=o.value2 égal au nombre totale d'écritures?

4 Lock-Condition

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);
}
```

La classe ReentrantLock implémente cette interface.

Le but de ce TP est d'utiliser l'interface Lock pour résoudre le problème du producteur consommateur. Un thread producteur crée des produits (Object)) et les place dans une file d'attente. Un thread consommateur retire le produit de la file d'attente et le consomme (par exemple l'affiche). La file d'attente est bornée: lorsque la file d'attente est pleine le producteur se met en attente. Lorsqu'elle est vide le consommateur se met en attente.

Pour les besoins du TP on simulera des vitesses différentes de production et de consommation (chaque thread s'endormira un certain temps entre chaque itération)

On a trois classes: File pour la file d'attente d'attente, Prod pour le producteur et Cons pour le consommateur.

La classe File a deux méthodes mettre et enlever.

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