Université Paris Cité

Master 2 - Année 2023-24

Programmation répartie

Exercice 1.— ThreadLocal

On propose la classe suivante pour donner un identifiant à chaque thread que l'on crée

```
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);
        }
}
```

Et deux programmes utilisant cette classe

```
public class TPC2 {
    public static void main(String[] args) {
        MyThread2 TH[]= new MyThread2[10];
        for( int i=0;i<10; i++) TH[i]=new MyThread2("nom"+i);
        try{
            for(int i=0;i<10; i++) TH[i].start();
            for(int i=0;i<10; i++) TH[i].join();
        }
        catch(InterruptedException e){};
    }
}

public class MyThread2 extends Thread{
   public int nbwrite; int x;
   public ThreadID tID;
   public int myid;
   public MyThread2(String name){
       this.nbwrite=nbwrite;
       tID=new ThreadID();
       myid=tID.get();
    }
    public void run(){
  // tID=new ThreadID();
  // myid=tID.get();
    System.out.println("la thread "+tID.get() + " "+myid);
    try{this.sleep(10);}
       catch(Exception e){e.printStackTrace();      }
    System.out.println("la thread " +tID.get() + "apres le sommeil  ");
```

```
}
}

public class MyThread2 extends Thread{
   public int nbwrite; int x;
   public ThreadID tID;
   public int myid;
   public MyThread2(String name){
              this.nbwrite=nbwrite;
       // tID=new ThreadID();
       //  myid=tID.get();
    }
    public void run(){
         tID=new ThreadID();
         myid=tID.get();
         System.out.println("la thread "+tID.get() + " "+myid);
         try{this.sleep(10);}
         catch(Exception e){e.printStackTrace();      }
         System.out.println("la thread " +tID.get() + "apres le sommeil  ");
}
}
```

1. Exécuter TCP2 avec la première version de MyThread2 et avec la 2ème.
2. Comment expliquez vous les différences de comportement?

Exercice 2.— ThreadLocal et atomicité
On définit la classe MonObjet grâce à laquelle les threads partagent un objet.

```
public class MonObjet {
    ThreadLocal<Integer> last;//nb ecriture de chaque thread
    double value;//valeur commune
    double valuebis;//valeur commune
    public MonObjet(int init){
        value=init;
        last=new ThreadLocal<Integer>(){
            protected Integer initialValue() {return 0;}};
    };
    public double read(){ return value;}
    public void add( ){
        last.set(new Integer(last.get()+1));
        value=value +1;
        valuebis=valuebis +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(){
        for(int i=0;i<nbwrite;i++)
        {
          o.add();
          this.yield();
          }
       System.out.println("la thread "+ThreadID.get()+" a pour last "+o.last.get());
}
}



public class TPC2 {

    public static void main(String[] args) {
        MonObjet  o= new MonObjet(0);
        MyThread2 W;
        MyThread2 R;
        W= new MyThread2(o,1000);
        R= new MyThread2(o,5000);
        W.start();
        R.start();
        try{
           R.join();
           W.join();} catch(InterruptedException e){};
        System.out.println("value" + o.value+", "+ "valuebis" + o.valuebis+" et "+
              " last "+ o.last.get());
    }
}
```

- Après avoir exécuté le Main2.main quelles sont les valeurs affichées?
- Comment assurer qu'à la fin du Main2.main on ait o.value=o.valuebis= nombre totale d'écritures?
- Quelle est la difference entre les variables *value* et *last*?

Exercice 3.— (Atomicité) On considère le programme suivant:

```
public class resultat {
    int [] tab;
    public resultat(int x){ tab=new int [x];}
}
public class sched extends Thread{
    resultat res=new resultat(5);
    public sched( resultat t){res=t;}
    public void echange( int i,int j){
        int tmp=res.tab[i];
        res.tab[i]=res.tab[j];
        res.tab[j]=tmp;};
```

```
    public void run (){
        for ( int k=0; k<10000 ; k++){
            int a= (int)(Math.random()*5);
            int b= (int)(Math.random()*5);
            echange ( a,b);
        }
    }

    public static void main(String[] args) {
        resultat t=new resultat(5); int i=0;int k;
        for(k=0;k<5;k++) t.tab[k]=k;
        for( k=0;k<5;k++) System.out.print(t.tab[k]);
        System.out.println();
        Thread  th1=new sched(t); th1.start();
        Thread  th2=new sched(t); th2.start();
        Thread  th3=new sched(t); th3.start();
        try{th1.join();th2.join();th3.join();}
        catch(Exception e){e.printStackTrace();};
        for(k=0;k<5;k++) System.out.print(t.tab[k] );
     }
}
```

1. Exécutez plusieurs fois le programme, est-ce que t contient toujours à la fin de l'exécution les valeurs 0,1,2,3,4 (dans un ordre quelconque) ?
2. Comment l'expliquez vous?
3. Est ce que si on synchronise la methode echange on aura après les échanges toujours les valeurs 0,1,2,3,4 dans le tableau .?
4. Proposer une solution pour faire en sorte que après les échanges il y ait toujours les valeurs 0,1,2,3,4 dans le tableau .

Exercice 4.— Volatile
Exécutez le programme suivant soit avec la déclaration
`public  static volatile int check=0;`
soit avec
`public  static  int check=0;`
Comment expliquez vous ces comportements?

```
public class Main {
/* 2 versions avec ou sans volatile*/
        public  static volatile int check=0;
        public static void main(String[] args) {
                 MyObject object = new MyObject();
                 Stop s = new Stop();
                 s.start();
                 object.start();
         }
}

public class MyObject extends Thread {
          public void work()
```

```
         { int cur;
         for(cur=Main.check;cur!=10;) {
                  if (Main.check>cur)
                         { System.out.println("check = "+Main.check+ " cur = "+cur);
                           cur=Main.check;
}
                 }
        }
public void run() {
                 super.run(); work(); System.out.println("coucou");
        }
}

public class Stop extends Thread {
        private boolean finish;
        public Stop() { finish = false; }
        public void incr()
        {
                 Main.check++;
                 if (Main.check == 11 )
                  {      System.out.println("received 11 stop");
                         this.finish=true;
                   }
          }
         public void run() {
                 super.run();
                 while(!finish)
                   {
                         try { sleep(1000);
                          } catch (InterruptedException e) { e.printStackTrace();}
                         incr();
                     }
       }
}
```

Exercice 5.— Exclusion mutuelle
Dans le package `java.util.concurrent.locks` se trouve l'interface Lock.

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

On veut implementer l'interface Lock par l'algorithme de Peterson pour 2 threads, puis la généraliser à n threads. Ecrire le code d'une thread qui rentre régulièrement en section critique en utilisant un verrou (la section critique et la section non critique pourront être simulées par une mise en sommeil de la thread).

1. Exclusion mutuelle pour 2 threads. On rappelle le code de l'algorithme de Peterson.

```
boolean[] flag= new boolean[2];
int victim;
```

```
  public Peterson(){
      for(int i=0 ; i<flag.length ; ++i)
          flag[i] = false;
  }

  public void lock() {
      int i = ThreadID.get();
      int j = 1 - i;
      flag [i]=true; // I'm interested
      victim = i ; // you go first
      while ( flag[j] && victim == i) {}; // wait
  }

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

Ecrire l'implémentation de Lock par cet algorithme.

2. Exclusion mutuelle pour n threads

   Pour implémenter le verrou on propose d'utiliser l'algorithme de Peterson généralisé à n threads dont on rappelle le code ci dessous:

```
class Filter implements Lock {
     int IDLE = -1;
     int[] level;
     int[] victim;
     int size;


  public Filter(int threads) {
    size   = threads;
    level  = new int[threads];
    victim = new int[threads - 1];
    for (int i = 0; i < size; i++) {
      level[i]  = 0;
         }

  }

  public void lock() {
    int me = ThreadID.get();
    for (int i = 1; i < size - 1; i++) {
      level[me]  = i;
      victim[i] = me;
      // spin while conflicts exist
      while (sameOrHigher(me, i) && victim[i] == me) {};
    }
    level[me] = size - 1;
```

```
  }
  // Is there another thread at the same or higher level?
  private boolean sameOrHigher(int me, int myLevel) {
    for (int id = 0; id < size; id++)
      if (id != me && level[id] >= myLevel) {
        return true;
      }
    return false;
  }
  public void unlock() {
    int me = ThreadID.get();
    level[me] = IDLE;
  }
}
```

Ecrire l'implémentation de Lock par cet algorithme.

En supposant que l'algorithme est utilisé par 10 Threads. Répondre aux questions suivantes (si la réponse est positive donnez un exemple d'exécution (ou indiquez quelle exécution donnerait ce résultat) si la réponse est négative expliquez pourquoi)

La thread d'identifiant id est au niveau i si $level[id] = i$

(a) Peut il y avoir 10 Threads au niveau 1 simultanément?

(b) Peut il y avoir 10 Threads au niveau 2 simultanément?

(c) Si il y a au moins 1 thread au niveau 1 et aucune au niveau 2 (et supérieur) est ce qu'une des threads de niveau 1 peut passer au niveau 2?

(d) Peut il y avoir 9 Threads au niveau 2 simultanément?

(e) Peut il y avoir 9 Threads au niveau 3 simultanément?

(f) Si une thread accède à la section critique, à quels niveaux sont les autres threads ?

(g) Montrer la propriété suivante: pour tout j, $1 \leq j \leq n-1$, il y a au plus $n-j+1$ threads au niveau j.

(h) Montrer que une thread qui demande l'accès à la section critique parviendra en section critique.