Université Paris 7
M2 II
Programmation Répartie

TP

Mémoire partagée
Implémentation sans verrou

Des threads disposent d'une mémoire partagée. On suppose que les threads sont numérotés de 0 à $nb-1$. La thread i peut lire la mémoire(**scan**) et écrire dans un des éléments de celle-ci (**update(x)** écrit x à l'indice i). L'interface est la suivante

```
public interface Snapshot<T> {
    public void update(T v);
    public T[] scan();
}
```

1. On réalise une première implémentation de **scan** et de **update** et on utilise dans le programme suivant des threads qui ne font qu'écrire et une thread qui lit.

```
import java.util.concurrent.atomic.AtomicStampedReference;
public class SimpleSnap<T> implements Snapshot<T> {
    private AtomicStampedReference<T> [] a_table;
public SimpleSnap(int capacity, T init){
    a_table=(AtomicStampedReference<T> [])new AtomicStampedReference [capacity];
    for (int i=0;i<capacity;i++)a_table[i]=new AtomicStampedReference<T>(init,0);
}

    public void update(T v) {
        int me=ThreadID.get();
        AtomicStampedReference<T> nouv=
         new  AtomicStampedReference<T>(v,a_table[me].getStamp()+1);
        a_table[me]=nouv;
        //ne fait pas partie de l'implémentation
        try{  MyThread.sleep(1);} catch(InterruptedException e){};
         MyThread.yield();
    }

    private AtomicStampedReference<T>[] collect() {
        AtomicStampedReference<T>[] copy= (AtomicStampedReference<T>[])
                new AtomicStampedReference [a_table.length];
```

```
        for(int j=0;j<a_table.length;j++){ copy[j]=a_table[j];
        //ne fait pas partie de l'implémentation
         try{  MyThread.sleep(3);} catch(InterruptedException e){};
         MyThread.yield();
          }
        return copy;
    }

    public T[] scan(){
        AtomicStampedReference<T>[] copy;
        T[] result=(T[]) new Object[a_table.length];
        copy=collect();
        for(int j=0;j<a_table.length;j++) result[j]=copy[j].getReference();
        return result;
        }
    }
 --------------------
public class MyThread extends Thread{
 public SimpleSnap<Integer> partage;
 public int nb;
 public MyThread( SimpleSnap<Integer>  partage, int nb){
        this.partage=partage;
        this.nb=nb;

    }
 public void run(){
    if (ThreadID.get()!=0){
          partage.update(new Integer(1));
          partage.update(new Integer(2));
          partage.update(new Integer(3));

    }
  else {
    Object [] O=new Object[nb];
    O=partage.scan();
    System.out.print("scan de "+ThreadID.get() + ": ");
    for(int i=0;i<nb;i++){
         System.out.print((Integer)O[i]+" ");
    }
    System.out.println();
  }
}
 ---------------
public class Main {
    public static void main(String[] args) {
        int nb=15;
```

```
        SimpleSnap<Integer>  partage= new SimpleSnap<Integer>(nb,new Integer(0));
        MyThread R[]=new MyThread[nb];
        for (int i=0;i<nb;i++)  R[i]= new MyThread(partage,nb);
        try{
            for (int i=0;i<nb;i++){R[i].start();if (i!=0)R[i].join();}
            R[0].join();
        } catch(InterruptedException e){};
    }
}
---------------
```

(a) Toutes les exécutions de ce programme donnent elles les mêmes affichages?

(b) Parmi les affichages possibles, avez vous observé l'affichage suivant:

```
scan de 0: 0 3 3 2 2 1 1 0 0 0 0 0 0 0 0
```

(c) L'implémentation réalise-t-elle l'atomicité des opérations **update** et **scan**?

2. Afin de réaliser une implémentation atomique, on associe une estampille à chaque écriture. Le **scan** réalise des lectures de la mémoire tant que deux lectures successives sont différentes. Quand elles sont identiques le résultat est la dernière lecture faite.

   (a) Réaliser cette implémentation où le **scan** peut ne pas terminer.

   (b) Dans quelle cas une exécution ne termine pas?

   (c) Votre implémentation resterait-elle atomique si au lieu d'utiliser une `AtomicStampedReference<T>` vous utilisiez une classe:

   ```
   class Stamped<T>{
   T reference;
   int stamp;
   }
   ```

3. Réaliser enfin l'implémentation où le **scan** termine en utilisant l'implémentation du **scan** vu en cours.