Université Paris 7
M2 II
Programmation Répartie

TP4

Snapshot

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();
}
```

La spécification séquentielle est qu'un scan retourne pour chacun des éléments la dernière valeur écrite (la valeur initiale si il n'y pas eu d'écriture).

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.

```
public class SimpleSnap<T> implements Snapshot<T> {

    private T[] a_table;

public SimpleSnap(int capacity, T init){
    a_table= (T[]) new Object[capacity];
    for (int i=0;i<capacity;i++)a_table[i]=init;
}

    public void update(T v) {
        int me=ThreadID.get();

        a_table[me]=v;
        //ne fait pas partie de l'implémentation
        try{  MyThread.sleep(1);} catch(InterruptedException e){};
         MyThread.yield();
    }
```

```
    private T[] collect() {
        T[] copy= (T[]) new Object[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(){
        T[] result;
        result=collect();
        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) L'implémentation réalise-t-elle l'atomicité des opérations **update** et **scan**? Si oui justifier, si non donner un exemple.

2. Afin de réaliser une implémentation atomique, on associe une estampille à chaque écriture. On utilise la classe `AtomicStampedReference<T>` qui contient la référence d'un objet et un entier (l'estampille) qui sont mis à jour atomiquement. 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) Dans quelle cas une exécution ne termine pas? Quelle condition de progression assure cette implémentation (obstruction free? non blocking? wait free?)

   (b) Justifier le fait que cette implémentation est atomique. Est ce que ce serait encore le cas si la classe `AtomicStampedReference<T>` était remplacer par une classe

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

   (c) Réaliser cette implémentation du snapshot.