

Des threads disposent d'un tableau de mémoire partagée (une thread i peut écrire dans l'entrée d'indice i de ce tableau).

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();  
}
```

Dans sa spécification séquentielle, un **scan** retourne pour chacun des éléments la dernière valeur écrite (la valeur initiale si il n'y pas eu d'écriture), un **update(v)** réalisé par le thread i écrit v dans l'entrée i du tableau.

Exercice 1.— Propriétés du snapshot

On suppose qu'on dispose d'une implémentation où les opérations **scan** et **update(x)** sont atomiques. La valeur initiale de chaque entrée est -1. La thread i exécute **update(i); T=scan(); affichage de i et de T.**

1. On notera $scan_i$ le résultat du **scan** effectué par la thread i . Parmi les propriétés suivantes, lesquelles sont vraies ? (démonstration ou contre-exemple)
 - (a) Pour tout i : $scan_i[i] = i$
 - (b) Pour $j \neq i$, $scan_j[i] = i$ ou $scan_j[i] = -1$
 - (c) Pour $j \neq i$, si $scan_j[i] = i$ alors $scan_i[j] = j$
 - (d) Pour $j \neq i$, si $scan_j[i] = i$ alors $scan_i[j] = -1$
 - (e) Pour $j \neq i$, $scan_j[i] = i$ ou $scan_i[j] = j$
 - (f) Pour $j \neq i$, $scan_j \subseteq scan_i$ ou $scan_i \subseteq scan_j$ (la relation \subseteq est $A \subseteq B$ si et seulement si "pour tout i , si $A[i] \neq -1$ alors $A[i] = B[i]$ ")

Exercice 2.— Implémentation non-blocking

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();}
        } catch(InterruptedException e){};
    }
}
-----

```

- (a) Toutes les exécutions de ce programme donnent-elles les mêmes affichages?
 - (b) En supposant que toutes les valeurs écrites dans chaque entrée du tableau sont différentes, l'implémentation réalise-t-elle l'atomicité des opérations `update` et `scan`? Si oui justifiez, si non donnez un exemple.
 - (c) Que se passe t-il si une thread écrit 2 fois la même valeur
(ex `partage.update(new Integer(1)); partage.update(new Integer(2));`
`partage.update(new Integer(1));`)
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 de façon atomique. 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 quel 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 remplacé par une classe

```

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

```

- (c) Réaliser cette implémentation du snapshot.

Exercice 3.— Implémentation wait-free du snapshot.

On souhaite réaliser une implémentation wait-free d'un atomique snapshot. On utilisera pour cela des registres atomiques estampillés contenant un tableau et une valeur.

On modifie le `update`, en écrivant un snapshot en même temps que la nouvelle valeur et que la nouvelle estampille.

```

public class Elem <T> {
    public T value;
    public T[] snap;
    public Elem(T value){
        this.value=value;
        snap=null;
    }
    public Elem(T value, T[] snap){
        this.value=value;
    }
}

```

```

        this.snap=snap;
    }
}

public class WaitFreeSnap<T> implements Snapshot<T> {

    private AtomicStampedReference <Elem<T>>[] a_table;
    public WaitFreeSnap(int capacity, T init){
        a_table= new AtomicStampedReference[capacity];
        T[] initsnap= (T[])new Object[capacity];
        for (int i=0;i<capacity;i++) {initsnap[i]=init;}
        for (int i=0;i<capacity;i++) {Elem e=new Elem(init,initsnap);
            a_table[i]= new AtomicStampedReference <Elem<T>> (e,0);}
    }
    public void update(T w) {
        int me=ThreadID.get();
        int st = a_table[me].getStamp();
        T[] lscan=scan();
        Elem<T> v=new Elem<T>( w, lscan);
        a_table[me].set(v,st+1);
    }
}

```

1. On modifie maintenant le **scan**: on fait 2 **collect** (lectures séquentielles des éléments du tableau). Le scan retourne le résultat de ces deux collect s'ils sont égaux, et sinon le snapshot associé à la première valeur différente dans les deux collect (celle dans le premier collect). Est-ce que le **scan** et le **update** terminent toujours? Cette implémentation n'est pas atomique, donnez un contre exemple.
2. Même question si on prend celle dans le deuxième collect.
3. On modifie maintenant le **scan**: on fait des **collect** jusqu'à ce que deux **collect** soient égaux ou bien que pour un indice *i* on ait vu 3 valeurs différentes. On retourne le snapshot associé à la troisième valeur différente. Combien fait-on au plus de **collect** pour réaliser un **scan**? Est-ce que le **scan** et le **update** terminent toujours? Est-ce que l'implémentation obtenue est atomique (faites une preuve ou donnez un contre-exemple)?
4. Réalisez cette implémentation atomique wait-free.