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 Propriétés du snapshot

On suppose qu'on dispose d'une implémentation où les opérations **scan** et **update(x)** sont atomiques. La thread i exécute `update(i); T=scan(); affichage de i et de T`.

1. Lors de différentes exécutions peut-on observer les affichages suivants (le premier chiffre est le numéro de la thread qui affiche):
   (a) 3: -1, -1, -1,3, -1, -1
   (b) 3: -1, -1, -1,-1, -1, -1
   (c) 3: -1, -1, -1,3, -1, -1
   0: 0, -1, -1,3, -1, -1
   (d) 3: -1, -1, -1,3, -1, -1
   0: 0, -1, -1,-1, -1, -1
   (e) 3: 0, -1, -1,3, -1, -1
   0: 0, -1, -1,3, -1, -1
2. On notera $scan_i$ le résultat du *scan* effectué par la thread i. Dans 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 $A[i] \neq -1$ alors $A[i] = B[i]$)

2 Implémetation wait free

On utilise 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` (lecture séquentielle des éléments du tableau) si ils sont égaux, c'est la valeur retournée, si ils sont différents on prend comme résultat le snapshot associé à la première valeur différente. Est ce que le `scan` et le `update` terminent toujours? Cette implémentation n'est pas atomique, donnez un contre exemple.

2. 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 deuxième 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'implémention obtenu est atomique (faites une preuve ou donnez un contre exemple)?

3. Réalisez cette implémentation atomique wait free.