Matthieu Le Franc : 71800858

Hugo Jacotot : 71802786

TP4

Exercice 1

Que se passe-t- il si on réalise l'écriture en effaçant d'abord l'écriture précédente ?

La lecture peut potentiellement terminer sur une erreur si la lecture de l'indice i intervient entre la suppression de l'ancien et l'criture du nouveau.

Que se passe-t-il si on réalise l'écriture en écrivant d'abord la nouvelle valeur puis en effaçant ensuite l'écriture précédente ?

On suppose que la lecture du nombre intervient entre les deux étapes de l'écriture du nouveau nombre.

La lecture parcourant le tableau de gauche à droite et s'arrêtant à la première valeur lue, on distingue deux cas:

- Le nouveau nombre écrit est plus petit que l'ancienne valeur
- Le nouveau nombre est plus grand que l'ancienne valeur

Si le nouveau nombre est plus petit, la nouvelle valeur sera lue.

Si le nouveau nombre est plus grand, l'ancienne valeur sera lue.

En déduire qu'il ne faut pas toujours effacer les précédentes écritures. Comment réalise-t-on une écriture ?

L'ordre des deux étapes d'écriture d'une nouvelle doit être différent en fonction de si la nouvelle valeur est plus petite ou plus grande que l'ancienne.

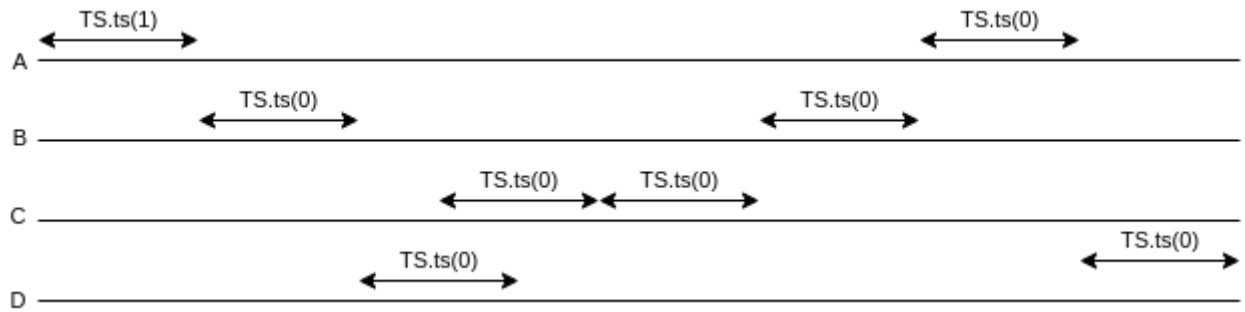
Implémenter la classe RegInt en utilisant un tableau de AtomicBoolean.

```
import java.util.concurrent.atomic.AtomicBoolean;
import java.lang.UnsupportedOperationException;

public class RegInt {

    private AtomicBoolean tab[];

    public RegInt(int max) {
        tab = new AtomicBoolean[max + 1];
    }

    public int read() {

        int i = 0;
        while (tab[i].get() == false && i < tab.length) {
```

```
            i++;
        }

        if (i == tab.length) {
            throw new UnsupportedOperationException();
        }

        return i + 1;
    }

    public void update(int value) {

        int i = 0;
        while (tab[i].get() == false && i < tab.length) {
            i++;
        }

        if (i == tab.length) {
            throw new UnsupportedOperationException();
        }

        if (i + 1 > value) {
            tab[value + 1].set(true);
            tab[i + 1].set(false);
        } else {
            tab[i + 1].set(false);
            tab[value + 1].set(true);
        }
    }

}
```

Exercice 2

Donner un exemple d'exécution de 4 threads partageant un objet TS et faisant chacune 2 appels a ts() de cet objet.



A l'aide de synchronized donnez une implémentation linéarisable de cet objet.

```
public class SynchronizedTS {
```

```
    private int value;

    public SynchronizedTS() {
        value = 1;
    }

    public synchronized int ts() {

        if(value == 1) {
            value = 0;
            return 1;
        }
        return 0;
    }
}
```

En utilisant un AtomicBoolean du package java.util.concurrent.atomic, implémenter un objet TS.

```
public class AtomicBooleanTS {

    AtomicBoolean bool;

    public AtomicBooleanTS() {
        bool = new AtomicBoolean(true);
    }

    public int ts() {

        if(bool.compareAndSet(true, false)) {
            return 1;
        }

        return 0;
    }
}
```

En utilisant un AtomicInteger du package java.util.concurrent.atomic, implémenter un objet TS.

```
public class AtomicIntegerTS {

    AtomicInteger atint;

    public AtomicIntegerTS() {
        atint = new AtomicInteger(1);
    }

    public int ts() {

        if(atint.compareAndSet(1, 0)) {
```

```
                return 1;
            }

            return 0;
        }
    }
```

Exercice 3

Donner un exemple d'exécution de 4 threads partageant un objet compteur et faisant chacune 2 appels add() de cet objet.

...

Cette première implémentation est-elle linéarisable ?

Non, car il n'existe aucun mécanisme permettant d'assurer que l'appel à add retournera une valeur cohérente.

Cette seconde implémentation est-elle linéarisable ?

Oui, car les différentes implémentations de TS permettent d'assurer la linéarisabilité de l'exécution.