Matthieu Le Franc : 71800858

Hugo Jacotot : 71802786

TP5

Exercice 1

On notera scani le résultat du scan effectué par la thread i. Parmi les propriétés suivantes, lesquelles sont vraies ? (démonstration ou contre-exemple)

- **a). Pour tout i: scani[i] = i**, vrai car chaque thread exécute d'abord update(i), ce qui signifie qu'elle écrit sa propre valeur i à l'indice i dans le tableau partagé. Ensuite, lors du scan, chaque thread récupère la dernière valeur écrite à son indice, qui est précisément i dans ce cas.
- **b). Pour j != i, scanj [i] = i ou scanj [i] = −1**, vrai car pour toute thread j différente de la thread i, scanj[i] sera i (si la thread i a écrit à cet indice) ou -1 (si la thread i n'a pas encore écrit à cet indice).
- **c). Pour j != i, si scanj [i] = i alors scani[j] = j**, faux car si thread i update et scan et que scanj update et scan, on peut très bien, selon l'ordre d'exécution, avoir comme résultat de la snapshot de la thread i : [i: i, j: -1] et comme résultat de la snapshot de la thread j : [i: i, j: j].
- **d). Pour j != i, si scanj [i] = i alors scani[j] = −1**, faux car si :
    - la thread i exécute update(i), écrivant i à l'indice i,
    - la thread j exécute update(j), écrivant j à l'indice j,
    - la thread j effectue scan et obtient scanj[i] = i,
    - la thread i effectue scan et obtient scani[j] = j,

  on a alors scanj[i] = i et scani[j] = j étant donné que les threads i et j ont écrit à leurs indices respectifs.
- **e). Pour j != i, scanj [i] = i ou scani[j] = j** vrai :
    - Si scanj[i] = i, cela signifie que la thread j a lu la valeur i à l'indice i lors de son scan. Cela ne peut se produire que si la thread i a effectivement écrit à l'indice i. Ainsi, la dernière valeur écrite à l'indice i est i. Donc, lorsque la thread i effectue son scan, scani[j] devrait également être égal à j, car la dernière valeur écrite à l'indice j est j.
    - pour scani[j] = j, le raisonnement est le même pour scanj[i] = i et donc même conclusion.

  Donc scanj[i] = i ou scani[j] = j car si une thread j observe la valeur i à l'indice i, alors la thread i observera la valeur j à l'indice j.
- **f). Pour j != i, scanj ⊆ scani ou scani ⊆ scanj (la relation ⊆ est A ⊆ B si et seulement si "pour tout i, si A[i] = −1 alors A[i] = B[i]")**, vrai car l'opération est atomique donc si la thread j scan avant la thread i, alors scanj ⊆ scani, et si la thread i scan avant la thread j, alors scani ⊆ scanj.

Exercice 2

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

- **a). Toutes les exécutions de ce programme donnent-elles les mêmes affichages ?**
    - Non, les affichages ne sont pas les mêmes à chaque exécution car ils dépendent de l'ordonnancement des threads. Par exemple, si la thread 0 scan avant que les autres threads n'aient effectué leur update, alors le scan de la thread 0 affichera des valeurs potentiellement différentes que si elle avait scan après que les autres threads aient effectué leur update. Les mises à jour ne sont pas synchronisées entre les threads, ce qui signifie qu'elles pourraient être intercalées avec les opérations de la thread principale.
- **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.**
    - Avec update on réalise l'atomicité car même sans gestion de la concurrence, chaque thread modifie sa case du tableau à son id. Par contre, pour scan, pas de gestion de la concurrence et on parcours tout le tableau, ce qui peut être problématique si d'autres threads font par exemple un update pendant qu'on scan, la snapshot ne sera pas représentative du nouvel état, on ne garantit pas l'atomicité avec un scan.

  Exemple :

    - La première thread commence une opération scan
    - La deuxième thread commence une opération update et écrit une valeur à son indice
    - La première thread copie le tableau partagé dans sa copie locale alors que T2 modifie une valeur
    - La copie retournée par la première thread contient l'ancienne valeur, avant l'update de la deuxième thread
- **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))😉 ?**
    - Si une thread écrit deux fois la même valeur, alors la dernière valeur écrite à l'indice de la thread sera la valeur 1. Lorsque la thread principale effectue son scan, elle obtiendra la valeur 1 à l'indice de la thread qui a écrit deux fois la valeur 1. En bref, la dernière valeur écrite à l'indice de la thread sera la valeur qui sera retournée par le scan.

2/. Afin de réaliser une implémentation atomique, on associe une estampille à chaque écriture. On utilise la classe AtomicStampedReference 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 locking? wait-free?)**
    - Une exécution ne termine pas si une opération update est effectuée en permanence par une thread, car le scan ne termine que lorsque deux lectures successives sont identiques. Si une thread effectue en permanence des opérations update, alors le scan ne pourra jamais terminer. Avec obstruction-free, on assurerait que l'exécution pourra terminer sans être bloquée par une autre thread.

- **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;
}
```

  - Cette implémentation avec la classe `AtomicStampedReference<T>` est atomique car, lorsqu'on update, on met une estampille puis, grâce à deux scan successif, on va collecter les estampilles qu'on compare pour s'assurer qu'elles sont égales avant de retourner la valeur. Si la classe `AtomicStampedReference<T>` était remplacée par la classe `Stamped<T>`, cette implémentation resterait atomique car même sans un `int stamp` atomique (c'est à dire que maintenant chaque thread a sa propre estampille), on s'assure lors d'un snapshot, grâce à deux scan successifs, que l'estampille de chaque case est égale entre les scans avant de retourner la valeur.

- **c). Réaliser cette implémentation du snapshot**

  - Ci-dessous l'implémentation, les fonctions modifiées sont : `scan`, `run` et `main`. Voici le résultat de l'affichage : `scan de 0: 3 3 3 3 3 3 3 3 3 3 3 3 3 3`

```
class ThreadID {
    private static volatile int nextID = 0;

    private static class ThreadLocalID extends ThreadLocal<Integer> {
        protected synchronized Integer initialValue() {
            return nextID++;
        }
    }

    private static ThreadLocalID threadID = new ThreadLocalID();

    public static int get() {
        return threadID.get();
    }

    public static void set(int index) {
        threadID.set(index);
    }
}

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

public interface Snapshot<T> {
    public void update(T v);
```

```
    public T[] scan();
}

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[] result1 = collect();
        T[] result2 = collect();
        for (int i = 0 ; i < result1.length ; i++) {
            Stamped<Integer> stamp1 = (Stamped<Integer>)result1[i];
            Stamped<Integer> stamp2 = (Stamped<Integer>)result2[i];
            if (stamp1.stamp != stamp2.stamp) {
                return scan();
            }
        }
        return result1;
    }
}

public class MyThread extends Thread {
    public SimpleSnap<Stamped<Integer>> partage;
    public int nb;
```

```
    public MyThread( SimpleSnap<Stamped<Integer>> partage, int nb){
        this.partage=partage;
        this.nb=nb;
    }

    public void run(){
        if (ThreadID.get()!=0) {
            Stamped<Integer> first = new Stamped<Integer>();
            first.reference = 1;
            first.stamp++;
            partage.update(first);

            Stamped<Integer> second = new Stamped<Integer>();
            second.reference = 2;
            second.stamp++;
            partage.update(second);

            Stamped<Integer> third = new Stamped<Integer>();
            third.reference = 3;
            third.stamp++;
            partage.update(third);
        } else {
            Object [] O=new Object[nb];
            O=partage.scan();
            System.out.print("scan de "+ThreadID.get() + ": ");
            for(int i=0;i<nb;i++){
                Stamped<Integer> stamp = (Stamped<Integer>)O[i];
                System.out.print(stamp.reference+" ");
            }
            System.out.println();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        int nb=15;
        SimpleSnap<Stamped<Integer>> partage = new
SimpleSnap<Stamped<Integer>>(nb, new Stamped<Integer>());
        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){};
    }
}
```

Exercice 3

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.

1/. On modifie maintenant le scan: on fait 2 collect (lectures séquentielles des éléments du tableau). Le scan retourne le résultat de ces deux collects s'ils sont égaux, et sinon le snapshot associé à la première valeur différente dans les deux collects (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.

Oui le scan et le update terminent toujours car il n'y a pas de situation bloquante. Pour update, on modifie toujours l'indexe du tableau correspondant au thread courant et pour le scan, s'il n'y a pas de différences entre les résultats des deux collectes, on retourne ce résultat, sinon on retourne le snapshot associé à la première valeur différente dans les deux collectes (les deux cas de figures sont couverts).

Cette implémentation n'est pas atomique car on ne garantit pas que le snapshot retourné est représentatif de l'état du tableau à un instant t. Par exemple, si une thread effectue un update, puis une autre thread effectue un update, puis la première thread effectue un scan, le snapshot retourné ne sera pas représentatif de l'état du tableau à un instant t.

2/. Même question si on prend celle dans le deuxième collect.

Dans ce cas là les deux fonctions terminent également car pas plus de conditions bloquantes que dans le cas précédent.

Cette implémentation n'est pas atomique car : si on fait deux update en même temps, dans deux threads i et j distincts, chaque update va faire son scan, ses deux collect, mais ils n'auront pas la valeur d'update de l'autre. la thread i aura l'ancienne valeur d'update de la thread j à l'indice j et réciproquement pour j.

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) ?

Scan et update terminent toujours car si trois valeurs sont différentes ont sort, autrement, au bout de 2 collectes sans modifications (donc 2 collectes identiques), on sort. On couvre suffisament de situations pour ne pas se retrouver dans une situation bloquante.

Dans le pire cas, pour chaque case du tableau, on fait 2 collect 3 fois. Donc au plus : $2*3n$ collect.

Il y a dans cette implémentation des lectures/écritures concurentes sur les éléments du tableau. Individuellement, les opérations sont atomiques mais dans un contexte global, on ne le garantit pas.