Matthieu Le Franc : 71800858

Hugo Jacotot : 71802786

TP7

Exercice 1

On rappelle l'implémentation a un gros grain. Elle utilise la classe Node ci dessous. Pour simplifier on supposera que la clef (key) est l'entier (int) correspondant à l'Integer item

1/. Cette implementation est-elle linéarisable (si oui donnez les points de linéarisation et justifiez sinon donnez un exemple)

Cette implémentation est linéarisable, points de linéarisation :

- après avoir fait un `lock.lock();` le moment où on va effectivement obtenir le vérrou
- après avoir libéré le verrou, le moment où on aura effectivement libéré la ressource

2/. Est-elle wait-free ?

Non, l'implémentation n'est pas wait-free car elle utilise des locks. Si un Thread lock la ressource, les autres threads devront attendre que le lock soit libéré.

3/. Ecrire une thread qui utilise cette implémentation (le constructeur de la thread aura en paramètre un Set) Si cette thread a pour identité i (donne par ThreadID) elle ajoute i, 2 * i, i + 4 , i + 3 à l'ensemble, teste si i + 5 y appartient et enlève i + 4

```
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 MyThread extends Thread {
    private MonSet set;
```

```
    public MyThread(MonSet set) {
        this.set = set;
    }

    public void run() {
        int i = ThreadID.get();
        set.add(i);
        set.add(2 * i);
        set.add(i + 4);
        set.add(i + 3);
        set.contains(i + 5);
        set.remove(i + 4);
    }
}
```

4/. Ecrire un programme qui lance 3 threads qui partagent un Set

```
public class Main {
    public static void main(String[] args) {

        MonSet set = new Set();

        MyThread thread1 = new MyThread(set);
        MyThread thread2 = new MyThread(set);
        MyThread thread3 = new MyThread(set);

        thread1.start();
        thread2.start();
        thread3.start();

        try {
            thread1.join();
            thread2.join();
            thread3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Exercice 2

On veut implémenter cette interface en utilisant une liste chainée et une synchronisation à grain fin. On utilise pour cela la classe Node

```
class Node {
    Integer item;
    int key;
```

```
    Node next;
    ReentrantLock lock;
}
```

On supposera toujours que la clef (key) est l'entier (int) correspondant à l'Integer item.

1. Ecrire une implementation de Set en utilisant une synchronisation à grains fins

```
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 MonSet implements Set{
    private Node head;
    private Lock lock= new ReentrantLock();

    public MonSet() {
        head = new Node(Integer.MIN_VALUE);
        head.next = new Node(Integer.MAX_VALUE);
    }

    @Override
    public boolean add(Integer item) {
        int key = item.hashCode();
        Node pred, curr;
        lock.lock();
        try {
            pred = head;
            pred.lock.lock();
            curr = pred.next;
            curr.lock.lock();
            while (curr.key < key) {
                pred.lock.unlock();
                pred = curr;
                curr = curr.next;
                curr.lock.lock();
```

```
            }
            if (key == curr.key) {
                return false;
            } else {
                Node node = new Node(item);
                node.next = curr;
                pred.next = node;
                return true;
            }
        } catch (Exception e) {
            e.printStackTrace();
    }

    @Override
    public boolean contains(Integer item) {
        int key = item.hashCode();
        Node pred, curr;
        try {
            pred = head;
            pred.lock.lock();
            curr = pred.next;
            curr.lock.lock();
            while (curr.key < key) {
                pred.lock.unlock();
                pred = curr;
                curr = curr.next;
                curr.lock.lock();
            }
            return (key == curr.key);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            curr.lock.unlock();
            pred.lock.unlock();
        }
    }

    @Override
    public boolean remove(Integer item) {
        int key = item.hashCode();
        Node pred, curr;
        try {
            pred = head;
            pred.lock.lock();
            curr = pred.next;
            curr.lock.lock();
            while (curr.key < key) {
                pred.lock.unlock();
                pred = curr;
                curr = curr.next;
                curr.lock.lock();
            }
            if (key == curr.key) {
                pred.next = curr.next;
```

```
                return true;
            } else {
                return false;
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            curr.lock.unlock();
            pred.lock.unlock();
        }
    }
}
```

2/. Cette implementation est-elle linéarisable (si oui donnez les points de linéarisation et justifiez sinon donnez un exemple)

Cette implémentation est linéarisable, points de linéarisation :

- après avoir fait un `lock.lock();` le moment où on va effectivement obtenir le vérrou
- après avoir libéré le verrou, le moment où on aura effectivement libéré la ressource

3/. Est-elle wait-free ?

Non, l'implémentation n'est pas wait-free car elle utilise des locks. Si un Thread lock la ressource, les autres threads devront attendre que le lock soit libéré.

4/. Réutiliser le programme de l'exercice précédent qui lancent 3 threads et les threads qui ajoutent et enlèvent des éléments avec cette implémentation

```
public class MyThread extends Thread {
    private MonSet set;

    public MyThread(MonSet set) {
        this.set = set;
    }

    public void run() {
        int i = ThreadID.get();
        set.add(i);
        set.add(2 * i);
        set.add(i + 4);
        set.add(i + 3);
        set.contains(i + 5);
        set.remove(i + 4);
    }
}

public class Main {
    public static void main(String[] args) {

        MonSet set = new MonSet();
```

```
        MyThread thread1 = new MyThread(set);
        MyThread thread2 = new MyThread(set);
        MyThread thread3 = new MyThread(set);

        thread1.start();
        thread2.start();
        thread3.start();

        try {
            thread1.join();
            thread2.join();
            thread3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```