Collections Concurrentes
TP 01 - Volatile, Opérations atomiques et CompareAndSet

Mélissa DA COSTA

Rappels de concurrence

```
public class Bogus {
  private boolean stop;

  private final Object lock = new Object();

  public void runCounter() {
     var localCounter = 0;
     for(;;) {
          synchronized(lock){
               if (stop) {
                    break;
               }
          }
          localCounter++;
     }
     System.out.println(localCounter);
  }

  public void stop() {
     synchronized(lock) {
          stop = true;
     }
  }
```

Le programme est infini car la JVM optimise le code.
Elle créée une nouvelle variable locale *stop* dans un registre pour plus de rapidité.
Mais la modification de *stop* dans la méthode *stop* modifie dans la RAM.

Il faut donc ajouter des blocs *synchronized* dans la méthode *stop* et dans *runCounter* pour forcer la lecture et l'écriture dans la RAM.

Volatile

En définissant un champ **volatile** : `private volatile boolean stop;`

On précise que ce champ ne doit pas être dans un registre mais dans la RAM.

L'accès aux champs est donc plus lent.

Cela permet de remplacer les blocs *synchronized*.

On appelle les implantation qui n'ont ni bloc *synchronized* ni *lock* des implantations **lock-free**.

Opérations atomiques

L’opération count++ n’est pas atomique.

Elle équivaut à trois opérations :

- Lecture en mémoire
- Incrémentation
- Écriture en mémoire

Les instructions assembleur sont atomiques.

Il est possible d'accéder à ces opérations atomiques grâce au package **atomic.**

On utilise l’objet **AtomicInteger**

On ne sais pas si les opérations atomiques sont disponible sur notre architecture.

On peut simuler les opérations atomiques des assembleurs.

CompareAndSet

La méthode compareAndSet permet de simuler une comparaison et une modification de façon atomique.

CompareAndSet(&field, expectedValue, newValue)->boolean

Le boolean de retour de la méthode CompareAndSet retourne *true* si le contenue de *field* est égal à *expectedValue.*

Pour que la classe soit thread-safe, on boucle jusqu'à ce que *compareAndSet* retourne *true,* ce qui signifie le succès de l'opération.

```
private final AtomicInteger counter = new AtomicInteger();

public int nextInt() {
    while (true) {
        var currentValue = counter.get();
        if (counter.compareAndSet(currentValue, currentValue + 1)) {
            return currentValue;
        }
        // else : on repasse dans la boucle
    }
}
```

getAndIncrement

La méthode getAndIncrement permet de simuler l'opération ++ de façon atomique.

```
private final AtomicInteger counter = new AtomicInteger();

public int nextInt() {
    return counter.getAndIncrement();
}
```

Cas d’une liste chaînée

```
private final AtomicReference<Entry<E>> head = new
AtomicReference<>();

public void addFirst(E element) {
    Objects.requireNonNull(element);

    while (true) {
        var currentHead = head.get();
        // nouvelle allocation nécessaire
        var newHead = new Entry<E>(element, currentHead);

        if (head.compareAndSet(currentHead, newHead)) {
            return;
        }
    }
}
```

Ici, la classe AtomicReference n’est pas super efficace car il y a une indirection en plus pour accéder à E.