Collections Concurrentes

Lab 1 : **Volatile, Opérations atomiques et CompareAndSet**

Algorithme lock-free et opérations atomiques

SORAN Altan - ESIPE INFO 2020

https://github.com/asoran/Structures-et-Algo-Concurentes

Rappel : Section Critique

En java une section critique c’est un endroit dans le code qui va être exécuté par 2 threads ou plus qui vont essayer de mettre à jour de la data partagé non-mutable en même temps. Il y a de la concurrence.

Le comportement du programme en plus d’être mauvais, est imprévisible.

Pour corriger ce problème il existe des mécanismes en Java par exemple les blocs synchronisées ou les locks.

Mais on va voir qu’il y a un autre mécanisme un peu plus bas niveau: le mot clé *volatile*.

Volatile

En java chaque thread à une mémoire séparée du nom de working memory qui lui est propre pour effectuer des opérations. Après avoir effectué des opérations, les threads copient les valeurs de ces variables dans la mémoire principale.

On peut mettre le mot clé *volatile* devant la déclaration d'une variable et forcé les écritures et lecture directement en mémoire principale pour cette variable.

Déclarer des variables en volatile peut regler certains petit problèmes de concurrences mais pas tous, il n'est pas fait pour complètement remplacer le système de *synchronized*.

java.util.concurrent.atomic

Dans le package *java.util.concurrent.atomic* il y a des classes qui proposes des méthodes pour faire des manipulations thread-safe, lock-free et atomique sur des variables (pour les types primitifs et un peu plus).
Pour cela les méthodes de la classe VarHandle sont utilisées.

Ces classes sont

- Thread-safe (Pas de problème de concurrence)
- Atomic (Opérations atomic, ne peut pas être interrompue pendant son exécution)
- Lock-free (pas de synchronized / lock)

AtomicInteger

Un exemple de classe du package *java.util.concurrent.atomic* : AtomicInteger

Cette classe fournit des méthodes pour manipuler un integer de façon atomique. Exemples de méthodes utiles:

- *getAndIncrement(newValue)*, permet de récuperer la valeur d'une variable puis de l'incrémenter
- *compareAndSet*(*expectedValue, newValue*), permet de modifier la valeur d'une variable si la valeur actuelle de cette variable est de *expectedValue*.

On peut donc utiliser cette classe pour sécuriser les accès concurrent à une variable dans un context multi-threads.

java.util.concurrent.atomic

Dans le package *java.util.concurrent.atomic* il y a des classes qui proposes des méthodes pour faire des manipulations thread-safe, lock-free et atomique sur des variables (pour les types primitifs et un peu plus).
Pour cela les méthodes de la classe VarHandle sont utilisées.

Ces classes sont

- Thread-safe (Pas de problème de concurrence)
- Atomic (Opérations atomic, ne peut pas être interrompue pendant son exécution)
- Lock-free (pas de synchronized / lock)

AtomicReference

On peut aussi appliquer cette technique pour tout type de variables en utilisant sa référence et la classe java.util.concurrent.atomic.AtomicReference.

Comme pour AtomicInteger, on peut modifier la référence de la variable (donc indirectement la “valeur” de la variable) avec *compareAndSet(expectedValue, newValue)*.

Exemple d’utilisation avec une implémentation de LinkedList

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

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

    while(true) {
        var current = head.get();
        if(head.compareAndSet(current, new Entry<E>(element, current))) {
            return;
        }
    }
}
```

java.lang.invoke.VarHandle

Les VarHandle sont des références typées d'une variable. Cette variable peut être un membre statique ou d'instance de la classe, ou bien même un élément d'un tableau. Un VarHandle est immutable.

On peut par exemple récupérer la valeur de la variables avec la méthode *get()* et on peut changer la valeur avec *compareAndSet()*.

Pour créer un VarHandle, on peut passer par un *lookup* et utiliser la méthode *findVarHandle(Class<?> recv, String name, Class<?> type)* pour récupérer un VarHandle de la variable d'instance 'name' de type 'type' dans la classe 'recv'.