# Collections Concurrentes

## Lab 3 : **Memory Model, Publication et Lock**

*Memory Model, Opération atomique, CompareAndSet, Réimplantation de locks*

SORAN Altan - ESIPE INFO 2020

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

# Rappel : ReentrantLock

En java, pour rendre nos programmes thread safe, on peut utiliser un ReentrantLock (depuis 1.5). Il y a plusieurs différences entre les lock et le mécanisme de synchronized (voir la doc).

- Avec la méthode `lock()` un thread peut prendre le verrou d'un bloc critique. Les autres threads qui essaieront de rentrer seront en attente (comme l'entré dans un bloc synchronized)
- Pour `tryLock()`, c'est comme faire un lock(), mais la fonction renvoie false si le verrou est déjà prit.
- Avec le méthode `unlock()`, on libère le verrou pour que le prochain thread puisse à son tour entrer dans la méthode (sortie d'un bloc synchronized)

# Recoder les ReentrantLock

On peut essayer de coder partiellement le mécanisme des lock.

On veut :

- Savoir quel thread possède le verrou: un champs de type Thread
- Savoir le nombre de fois où lock est appelé car un Thread peut passer par plusieurs verrous (par appel récursif par exemple): un champ int
- Quel la classe soit Thread-safe bien sûr
- Avoir les méthodes `lock` et `unlock` (et optionnellement `tryLock`)
- Faire attendre les autres threads une fois qu'un thread a acquit le jeton

Dans les prochaines slides seront expliqués pas à pas le codage de cette classe.

# CustomReentrantLock : Champs

```
public static final VarHandle HANDLE;
static {
    try {
        HANDLE = MethodHandles.lookup()
                 .findVarHandle(ReentrantSpinLock.class, "lock", int.class);
    } catch (NoSuchFieldException | IllegalAccessException e) {
        throw new AssertionError("Should not happen");
    }
}

private volatile int lock;
private Thread currentThread;
```

On déclare lock (le nombre d'appels) en volatile, et un HANDLE associé pour le gérer. Si on s'y prend intelligemment, on verra qu'on a pas besoin de déclarer currentThread en volatile !

# CustomReentrantLock : `lock()`

```
public void lock() {
    var t = Thread.currentThread();
    for (;;) {
        if (HANDLE.compareAndSet(this, 0, 1)) {
            currentThread = t; // ça marche
            return;
        } else {
            if (currentThread == t) {
                ++lock; // même thread donc OK
                return;
            }
            Thread.onSpinWait();
        }
    }
}
```

L'algo est simple:

- récupérer le thread courant:
    - si = current, on laisse passer et on incrément lock
    - sinon, on attend

L'attente se fait avec `onSpinWait()`, qui permet de faire de l'attente intelligente via une instruction assembleur spéciale (`PAUSE`)

Note: On voit que on a pas besoin de rendre currentThread concurrent car de tout façon le CaS ou le == vont raté (comparaison avec null)

# CustomReentrantLock : unlock()

```
public void unlock() {
    if (Thread.currentThread() != currentThread) {
        throw new IllegalStateException();
    }
    var l = lock;
    if (l == 1) currentThread = null;
    lock = l - 1;
}
```

Algo:
Récupérer le thread courant

- Si différent: Exception
- Si égal, décrémenter lock. Si lock vaut 0, alors libérer le verrou.

Note: Ici on effectue une écriture volatile (avec lock) après avoir changé la valeur de currentThread, l'écriture de currentThread est donc aussi recopier en mémoir principale. Ici non plus, pas besoin de rendre thread-safe (pas de problème concernant la lecture, il n'y as qu'un seul et même thread qui est autorisé a appelé unlock: celui qui a le verrou !

# Thread safe lazy singleton

Comme le titre de la slide l'indique, on veut pouvoir réaliser une classe singleton qui soit thread-safe et lazy (paresseuse).
On veut que les champs ne soient initialisés qu'au premier appel de ce champs. C'est pour gagner du temps et de la mémoire car on ne veut pas que tous les champs/constantes de cette classe s'initialise alors que l'on va peut être même pas les utilisées.

Ce genre de classe pourrait servir comme dictionnaire de constante par exemple.

Il existe un design pattern de concurrence hérité du C++ qui s'appelle "Double-Checked Locking". Il y a plusieurs façon de l'implémenter en Java, et on peut très facilement faire des erreurs en essayant car c'est compliqué.

# Initialization-on-demand holder

En java on peut très facilement implémenter le “Double-checked locking” en utilisant le pattern initialization-on-demand holder et quelque garanties de Java

```
public class Utils {
   private static class SomethingHolder {
     static final Something instance = /* ... */ null;
   }
   public static Something getHome() { return SomethingHolder.instance; }
}
```

Déclarer la variable en final nous garantit qu’elle est thread safe et qu’elle va être publié seulement une fois que l’objet a complètement charger.
L’encapsuler dans une classe garantit son caractère lazy car la variable ne être mise en mémoire uniquement quand la classe SomethingHolder va être appelé pour la première fois !