TRAINA Tanguy
INFO2 – Logiciel

Structures et Algorithmes concurrents

Lab 3 - Memory Model, Publication et Lock

Github : https://github.com/furtiveJack/Collections_Concurrente

Réentrance

- **Fonction non-réentrante** : *lorsqu'un thread est dans une fonction, aucun autre thread n'a le droit d'y rentrer (pas même le thread courant)*

- **Lock non-réentrant :** *un lock non-réentrant est un lock qui ne peut être prit qu'une fois*

SpinLock non-réentrant

- **Concept :**
  - on essaie d'acquérir le *lock*
  - s'il n'est pas disponible, on attend qu'il le soit
  - quand on l'a, on effectue l'action souhaitée

- **Problème** : boucle tant que le lock n'est pas obtenu -> attente active

SpinLock non-réentrant

- **Solution :** on utilise la méthode *Thread.onSpinWait()*.

- -> Indique à l'environnement *run-time* que le thread courant est en train d'attendre dans une boucle
  - -> Donne la possibilité, quand l'environnement le permet, d'optimiser l'attente.

- Exemple :

```
public void lock() {
    while (! HANDLE.compareAndSet( ...args: this, false, true)) {
        Thread.onSpinWait();
    }
}
```

SpinLock réentrant

```java
public class ReentrantSpinLock {
    private volatile int lock;
    private volatile Thread ownerThread;
```

```java
public void lock() {
    var current = Thread.currentThread();
    while (true) {
        if (HANDLE.compareAndSet( ...args: this, 0, 1)) {
            ownerThread = current;
            return;
        }
        if (ownerThread == current) {
            lock++;
            return;
        }
        Thread.onSpinWait();
    }
}
```

```java
public void unlock() {
    if (ownerThread != Thread.currentThread()) {
        throw new IllegalStateException();
    }
    var lock = this.lock; //volatile read
    if (lock == 1) {
        ownerThread = null;
        this.lock = 0; //volatile write
        return;
    }
    this.lock = lock - 1; //volatile write
}
```

- **Concept :**
  - **P**lusieurs threads peuvent rentrer dans la méthode *lock()* mais seul celui possédant le jeton pourra effectuer une action, les autres attendront.
- **Remarque:**
  - On utilise *compareAndSet* dans la méthode *lock()* car plusieurs threads peuvent rentrer dans cette méthode sans nécessairement avoir le *lock*.
  - Pas besoin de faire ça dans *unlock()* car on a la garantie que le seul thread qui peut y accéder est celui qui possède le lock.

```java
public class ReentrantSpinLock {
    private volatile int lock;
    private /*volatile*/ Thread ownerThread;
```

```java
public void lock() {
    var current = Thread.currentThread();
    while (true) {
        if (HANDLE.compareAndSet( ...args: this, 0, 1)) {
            ownerThread = current;
            return;
        }
        if (ownerThread == current) {
            lock++;
            return;
        }
        Thread.onSpinWait();
    }
}
```

```java
public void unlock() {
    if (ownerThread != Thread.currentThread()) {
        throw new IllegalStateException();
    }
    var lock = this.lock; //volatile read
    if (lock == 1) {
        ownerThread = null;
        this.lock = 0; //volatile write
        return;
    }
    this.lock = lock - 1; //volatile write
}
```

Optimisations avec volatile

- L'écriture *volatile* empêche la réorganisation des instructions par la JVM.
- **Idée :** dans la méthode *unlock()*
  - quand on fait une écriture volatile, on obtient la garantie que tous les champs de l'objet ont été relus et mis à jour en RAM.
    - c'est pourquoi on update *ownerThread* avant d'écrire le volatile.
  - quand on fait une lecture volatile, on obtient la garantie que les autres champs seront lus en RAM ensuite.
- Le champs *ownerThread* peut donc ne pas être déclaré *volatile*, car le champs *lock* l'étant déjà, on peut s'assurer de toujours avoir la bonne valeur de *ownerThread* (en organisant les instructions de la bonne manière).

Initialisation paresseuse d'un singleton

- **Rappel :** un **singleton** est un objet qui ne doit initialisé (instancié) qu'une seule fois.
- **Proposition :** on veut une manière d'avoir un champ d'une classe qui ne sera initialisé qu'une seule fois, le plus tard possible (seulement quand on a besoin de sa valeur), et de manière concurrente
- Pour cela, 3 idées de solution :
    1. on utilise un lock associé au mot-clé *synchronized,* et on teste si le champ (déclaré *static volatile*) a déjà été initialisé; s'il ne l'a pas été, on l'initialise ; enfin on renvoie sa valeur.
        - problème de publication si l'on ne déclare pas le champ volatile
    2. Même idée mais en utilisant un *VarHandle* et ses méthodes *getAcquire()* et *setRelease()* pour manipuler le champ (déclaré *static*) : ces méthodes coûtent moins cher que des lectures/écritures volatiles, on a donc un léger gain de performance.
    3. On utilise le pattern "*initialization on demand holder*" : utilisation d'une classe interne *static* contenant le champ (déclaré *static final*)
- La méthode 3 est plus simple à mettre en place que la 1 et 2, et la JVM garantie que l'initialiation de la classe interne ne sera faite qu'au moment où l'on aura besoin de la valeur de la variable qu'elle contient.