Collections Concurrentes
TP 03 - Memory Model, Opération atomique, CompareAndSet, Réimplantation de locks

https://github.com/MelissaDaCosta/structures_concurrentes

Mélissa DA COSTA

Ré-entrant et non ré-entrant

Ré-entrant :
Le fait qu'une fonction puisse se rappeler elle-même.
- exclusion mutuelle
- mécanisme de verrouillage récursif
- Peut-être bloqué plusieurs fois par le même processus sans créer d'interblocage (deadlock).

Non ré-entrant :
Un lock non ré-entrant est un lock que l'on prend qu'une seule fois.

Généralement, seulement les locks sont toujours ré-entrant.
=> Permet de définir des sections critiques

Thread.onSpinWait()

Dans la méthode **lock**, si le lock est déjà pris, on doit attendre. **Thread.onSpinWait** permet de résoudre l'attente active : Interdit au thread d'être schédulé par l'os.

```
public void lock() {
  while (isLocked) {
    Thread.onSpinWait();
  }
 // pas thread-safe : peut être interrompue ici
  isLocked = true;
```

Cette méthode n'est pas **ré-entrant** car si on appelle 2 fois la méthode lock, on est **bloqué** a vie dans la méthode.

SpinLock non ré-entrant

Pour rendre la méthode **lock thread-safe**, il faut utiliser un **compareAndSet :**

```
  public void lock() {
    // attendre tant que le lock est déjà pris -> attente active
    while (!HANDLE.compareAndSet(this, false, true)) { // passer de faux à vrai
      Thread.onSpinWait();// résout l'attente active
    }
    // si CaS renvoie vrai -> on a le lock, sinon on boucle : on se met en
pause
  }

  public boolean tryLock() {
    return HANDLE.compareAndSet(this, false, true);
    // renvoie true si ca a marché et prend aussi le lock
    // si CaS renvoie false, tryLock renvoie aussi false et on a pas le lock
  }

  public void unlock() {
    this.isLocked = false; // écriture volatile
  }
```

compareAndSet

On utilise un **compareAndSet** lorsque plusieurs **threads** peuvent faire la même opération en même temps.

C’est le cas de la méthode **lock**. Le **compareAndSet** rend la méthode **thread-safe** car il rend l’accès à une variable **volatile (isLocked) atomique.**

Si une seule **thread** réalise l’opération, il n’y a pas besoin du **compareAndSet**.
C’est le cas pour la méthode **unlock**. Un thread à la fois appelle **unlock** car il faut qu’il ai pris le **lock** avant.

ReentrantSpinLock : lock

```
public void lock() {
    // on récupère la thread courante
    var currentThread = Thread.currentThread();
    while (true) {
      // si lock est == à 0, on utilise un CAS pour le mettre à 1 et
      if (HANDLE.compareAndSet(this, 0, 1)) {
        // on sauvegarde la thread qui possède le lock dans ownerThread.
        this.ownerThread = currentThread;   // ca marche !
        // cette écriture peut ne pas être vue par d'autre thread
        // mais c'est pas grave car elle est lue par le thread courant !
        return;
      }
      // sinon on regarde si la thread courante n'est pas ownerThread,
      if (this.ownerThread == currentThread) {   // lecture volatile
        // si oui alors on incrémente lock.
        // une seule thread peut rentrer dedans donc on peut faire ++
        this.lock++; // écriture volatile : écritures précédentes faites en RAM
        return;
      }
      Thread.onSpinWait();
    }
  }
```

ReentrantSpinLock : unlock

```
public void unlock() {
    // si la thread courante est != ownerThread
    if (this.ownerThread != Thread.currentThread()) {
      throw new IllegalStateException();
    }
    // ici on est le owner thread, pas d'autre thread qui peuvent passer
    var lockVolatile = this.lock; // lecture volatile
    // pour éviter de faire plein de trafic en lecture
    // et écriture volatile qui coûtent cher
    // on stock dans un variable intermédiaire pour éviter ca
    // si lock == 1
    if (lockVolatile == 1) {
      // on remet ownerThread à null
      this.ownerThread = null; // rend le lock
      // écriture volatile: donc pas besoin de ownerthread volatile
      // car garantie que les écritures d'avant ont été faites en RAM avant
      this.lock = 0;
      return;
    }
    // on décrémente lock
    this.lock = lockVolatile - 1; // écriture volatile
  }
```

Double-Checked Locking

Double-Checked Locking / Singleton paresseux : Design pattern de concurrence hérité du C++.
Consiste à essayer d'initialiser un **singleton** de façon **paresseuse** (lazy).

Un **singleton** est **global**, **unique** et **accessible** à n'importe qu'elle endroit du programme.

Si l'on appelle pas le **singleton**, il n'est pas **initialisé** → **paresseux.**

Double-Checked Locking

```
public class Utils {
    private static Path HOME;
    private final static Object lock = new Object();
    public static Path getHome() {
      if (HOME == null) {
        // peut être dé-schédulé ici
        synchronized(lock) {
          if (HOME == null) {
            return HOME = Path.of(System.getenv("HOME"));
          }
        }
      }
      return HOME;
    }
  }
```

Rajouter un **if** dans le block **synchronized** = **double check**
Il y a toujours un problème de **publication** : L'objet **Path** crée par **Path.of** peut être vu alors qu'il n'a pas été entièrement **initialisé.**
=> Mettre HOME volatile

getAcquire et setRelease

Éviter les lectures/écritures dans des champs **volatiles** car cela réduit les **performances**.

getAcquire garantie que les instructions suivantes ne seront pas exécutées avant. Elles peuvent être ré-ordonnées mais ne seront jamais exécutées avant le **getAcquire.**

setRelease garantie que les instructions précédentes ne seront pas exécutées après. Elles peuvent être ré-ordonnées mais ne seront jamais exécutées après le **setRelease.**

Ces garanties sont **moins** fortes que pour un **volatile** qui garantie des lectures/écritures en RAM.

Mais ces garanties sont suffisantes car nous souhaitons juste s'assurer de **l'ordre** de certaines instructions.

Initialization-on-demand holder

```
public class Utils {
    private Path HOME2;
    private static class LazyHolder {
        static final Path HOME2 = Path.of(System.getenv("HOME"));
    }
    public static Path getHome2() {
        return LazyHolder.HOME2;
    }

    public static void main(String[] args) {
        System.out.println(Utils.getHome2());
    }
}
```

Le design pattern **Initialization-on-demand** est plus efficace que l'utilisation des **VarHandle** car la classe **LazyHolder** est exécutée seulement quand **getHome2** est appelée.

Cette méthode peut être utilisée seulement si le constructeur de l'objet à initialiser (ici Path) garanti de ne pas échouer.