Collections Concurrentes
TP 02 - Modèle de mémoire, publication et Opérations atomiques

https://github.com/MelissaDaCosta/structures_concurrentes

Mélissa DA COSTA

Modèle de mémoire (Memory Model)

JAVA : **W**rite **O**nce **R**un **A**nywhere

Un même code peut s'exécuter sur plusieurs architectures/OS différents.

Le **Modèle de mémoire** permet de garantir une même exécution grâce à des règles sur l'ordre d'exécution du code.

Une opération voie l'état d'une opération précédente : « happen before »

Total Store Order (TSO)

Le **Total Store Order** garantit que l'écriture en RAM se fait dans le même ordre que le code assembleur.

TSO	Pas TSO
intel	ARM
sparc	Power

Les architectures **non** TSO consomme plus que les architectures TSO.

Les garanties pour les threads

Pour **un même thread**, la lecture voit toujours la dernière écriture.

Il n’y a aucune garantie entre des threads différents.

thread.start() garantie que toute les écritures précédentes seront visibles par la thread qui démarre.

thread.join() garantie que les écritures faites dans la thread “jointe” seront visible à la thread courante

Les initialisations faites dans un **bloc statique** sont visibles par toutes les threads qui utilisent la classe.

Synchronized et lock

- L'entrée dans un bloc synchronized oblige la **relecture** des champs à partir de la RAM.

- En sortie d'un bloc synchronized, toutes les **écritures** sont visibles en RAM.

– Les écritures peuvent être visible en RAM avant la sortie.

Problème de publication

Un problème de publication survient lorsqu'un thread voit un objet avec des champ non initialisés.

Un champ déclaré **final** est visible par toutes les threads après la **fin du constructeur**.

Exemple sans problème de publication car les champs sont déclarés **final** :

```
public class Person {
	private String name;
	private final int age;

	public Person(String name, int age) {
	    this.name = name;
	    this.age = age;
	}
}
```

Les effets de volatile

- L'écriture dans un champ volatile **rend visible toutes les écritures précédentes**.

– La lecture dans un champ volatile oblige les lectures suivantes a être **rechargé à partir de la RAM**.

– L'écriture d'une valeur 64 bits (long ou double) est vu comme **atomique** même sur une machine 32 bits.

Un champ volatile coûte plus cher qu’un champ final.

VarHandle

VarHandle permet de récupérer par référence (& en C) . Pour changer les valeurs d'une case mémoire la RAM.

On peut utiliser un seul VarHandle pour changer plusieurs case mémoire si elles sont du même type.

GetVolatile sur un VarHandle permet de récupérer la valeur de la variable avec les mêmes règles de lecture que si elle avait été déclarée **volatile**. (Les lectures suivantes sont faites à partir de la RAM.)

```
private static final VarHandle NEXT_HANDLE;

  static {
    var lookup = MethodHandles.lookup();
    try {
      // findVarHandle(Class<?> recv,String name, Class<?> type)
      NEXT_HANDLE = lookup.findVarHandle(Entry.class, "next", Entry.class);
    } catch (NoSuchFieldException | IllegalAccessException e) {
      throw new AssertionError(e);
    }
  }
```