Collections Concurrentes

Lab 2 : **Modèle de mémoire, publication et Opérations atomiques**

Problème de publication, Opérations atomiques, VarHandle et CAS

SORAN Altan - ESIPE INFO 2020

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

Introduction : Modèle de mémoire

Un des avantages de Java est que le code produit peut s'exécuter sur plusieurs systèmes d'exploitation (donc différentes architectures). On peut le qualifier de *WORA* (Write-Once-Run-Everywhere).

Le code est donc “générique”, mais les architectures peuvent fonctionner différemment: gros problème! C’est pour cela que le langage à mit en place des mécanismes pour s’abstraire de cette notion.

Un des problèmes possibles, est que certains processeurs (qui ne sont pas sous le mode TSO) peuvent intervertir l’ordre dans lequel se font les instructions assembleurs.

Mise en place de garantie

Pour palier à ce problème, Java à mit en place certaines garanties sur l'ordre d'exécution:

- Lorsque l'on démarre un thread avec *`Thread::start`*, on est garantie que la thread va voir les écritures fait avant son démarrage.
- *`Thread::join`* garantie que les écritures faites dans la thread, seront visible dans la thread courante.

```
var a = new ValueHolder(4);
var t = new Thread(() -> {
    System.out.println(a); // Garantie que a est non null
    a.value = 3;
});
t.start(); t.join();
System.out.println(a.value); // Garantie que value vaut 3
```

Mise en place de garantie

On peut aussi garantir des ordres d'exécution avec un *bloc statique*, les initialisations seront visibles par toutes les threads qui utilisent la classe.

Ou encore avec des blocs synchronized, en effet:

- L’entrée garantie la relecture des champs depuis la RAM
- La sortie garantie que toutes les écritures seront visibles en RAM

Problème de publication

Un autre gros problème qui existe et ce qu'on appelle le problème de publication.

Dans un context multi-thread, il est possible d'accéder à la valeur d'un champs d'un objet alors que celui-ci n'a pas été initialisé et donc voir les valeurs par défaut.

```
class A {
   int value;
   A(int value) {
        this.value =
value;
   }
```

Soit cette classe *A*, faire *new A()* correspond à faire 2 choses, crée l'objet *A* en mémoire, puis assigner son champ *value*.

Un autre thread peut essayer de lire la valeur du champs, met verra 0.

Exemples de problème de publication

```
class A {
    int value;
    final int value2;
    A(int value) {
        this.value = value;
        value2 = 3;
        new Thread(() -> {
            this.value2; // 0 ou 3 🤷
        }).start();
    }
    public static void main(String[] args)
{
        var a = new A(4);
        new Thread(() -> {
            System.out.println(a.value);
                // 0 ou value 🤷
        }).start();
    }
}
```

Plusieurs problèmes ici:

- Il ne faut pas démarrer un thread dans un constructeur car même quand le champ est final, on n'as pas de garantie sur this
- On n'est pas sur que dans le thread lancé par le main, le champs value s'est initialisé, ou pas

Mot clé final

Le mot clé final permet de déclarer une variable “constante”, c’est à dire que ne va pas changer de valeur/référence après s’être initialisé.

Ce mot clé garantit aussi que la référence de l’objet ne sera publié qu’une fois que ce champs (tous les champs finals du coup) sera initialisé.

```
class A {
   final int value;
   A(int value) { this.value = value; }
}
var a = new A(4);
new Thread(() -> {
   System.out.println(a.value); // Garantie que a.value vaut 4
}).start();
```

Mot clé volatile

Un champ déclaré volatile à plusieurs effets:

- Écrire dans un champs volatile force l'écriture en RAM de toutes les écritures précédentes.
- La lecture oblige les prochaines lectures à s'effectuer depuis la RAM
- Les écritures sont atomiques (même un long de 64 bits sur une archi 32 bits se fera de manière atomique !)

```
class A {
   int value;
   volatile boolean done
   A(int value) {
     this.value = value;
     this.done = false;
     }
}
```

```
var a = new A(4);
new Thread(() -> {
   a.value = 3;    // Les 2 écritures
   a.done = true; // sont maintenant en RAM
}).start();
if(a.done) System.out.println(a.value);
// Si l'écriture de a.done s'est réalisé, alors
a.value VAUT 3
```