Java Avancé

Memory Model, Opération atomique, CompareAndSet, Réimplantation de locks

Guillaume Cau

Récapitulatif

- Un lock permet de délimiter une "section critique" à l’intérieur d’un programme.
- Une section critique est une suite d’instructions exécutable par un Thread à la fois.

L’attente dans les programmes

- Dans un programme, nous pouvons attendre de deux façon différente :
  - En utilisant l’attente active :

    Le programme va vérifier à chaque cycle si une condition à évoluée.
    Ex : while(timestamp < timestamp + timeout)

  - En utilisant une attente optimisée :

    Le programme est interrompu à l’aide de certains mécanismes permettant d’éviter la vérification des conditions à chaque cycle.

    Ex : Thread.onSpinWait() ;

Utilisation du onSpinWait

- Pour que le onSpinWait conserve le comportement souhaité, il faut le positionner à l'intérieur d'une boucle :
  - while (<condition>) {
    Thread.onSpinWait()
    }

- De cette manière, si le thread poursuit son exécution malgré la condition qui n'est plus respectée, le thread est rendormi.

Fonctionnement d’un lock

- La structure d’une section critique protégée par un lock est la suivante :
  - 1) Prendre le lock
    - Si le lock est libre, le Thread le prend, sinon le thread attend qu’il soit disponible
  - 1.bis) Essayer de prendre le lock
    - Si le lock est libre, la méthode tryLock récupère le lock et renvoie true, sinon, elle renvoie false.
  - 2) Executer la section critique
  - 3) Libérer le lock

Le reentrantLock

- Le problème d’un lock normal, c’est que le même thread bloque lors de l’acquisition du même lock. Ce comportement pose problème lorsque des algorithmes récursifs doivent être utilisés.
- Le comportement de l’acquisition du lock et de sa restitution doit donc être modifié pour permettre une acquisition "multiple".

Le mode Release/Acquire (RA)

- Le fonctionnement de mode RA est simple à comprendre :
  Si je prépare le dîner, que je dis que le dîner est prêt et que vous m'entendez, vous pouvez être sûr que le dîner existe et qu'il est prêt.

- L'utilisation du mode RA nécessite d'utiliser les méthode setRelease() et getAcquire().

- Nous pouvons éviter les problèmes de publication en utilisant ce principe :
  - Si getAcquire() est fait avant le setRelease(), getA() renverra null
  - Si getAcquire() est fait après le setRelease(), alors le dîner est prêt et getA() renverra l'objet entièrement initialisé.

Le double-check locking

- Le double-check locking est un patron de conception qui suit la structure suivante :

```
var home = this.home;
if ( home == null ) {          /* Vérifie si "home" a été initialisée */
    synchronized(lock) {       /* Si ce n'est pas, on se met en mode threadsafe */
        home = this.home;
        /* On reverrifie, la valeur de home peut être modifiée par un autre thread */
        if ( home == null )
            return this.home = Path.getHome(); /* Si elle est toujours null, on la set. */
        }
}
```

- Le code vérifie de manière unsafe si la variable est initialisée, sinon, il rentre dans la section critique, refait le test de manière safe puis fait l'affectation si besoin.

- Le problème est que ce patron ne marche pas pour les objets en Java. On utilise donc plutôt le support d'initialisation à la demande.

Initialization-on-demand holder idiom

- Le "Initialization-on-demand holder idiom" est un patron de conception qui suit la structure suivante :

```
private static class Holder {
    static final Singleton S = new Singleton();
}
public static Singleton getSingleton() {
    return Holder.S;
}
```

- A la première invocation de la méthode, la JVM chargera et initialisera la classe Holder et, étant donné que S est final, elle est initialisée qu’une seule fois.