



Ceci est un extrait électronique d'une publication de  
Diamond Editions :

<http://www.ed-diamond.com>

Ce fichier ne peut être distribué que sur le CDROM offert  
accompagnant le numéro 100 de **GNU/Linux Magazine France**.

La reproduction totale ou partielle des articles publiés dans Linux  
Magazine France et présents sur ce CDROM est interdite sans accord  
écrit de la société Diamond Editions.

Retrouvez sur le site tous les anciens numéros en vente par  
correspondance ainsi que les tarifs d'abonnement.

Pour vous tenir au courant de l'actualité du magazine, visitez :

<http://www.gnulinuxmag.com>

Ainsi que :

<http://www.linux-pratique.com>

et

<http://www.miscmag.com>



Par :: Philippe Prados :: [site@philippe.prados.name](mailto:site@philippe.prados.name) ::  
 :: [press@philippe.prados.name](mailto:press@philippe.prados.name) :: [www.philippe.prados.name](http://www.philippe.prados.name) ::

# Modèle mémoire en Java

Cet article explique les problèmes que rencontrent des JVM en fonctionnement multiprocesseurs. Il indique les évolutions du modèle mémoire permettant de garantir une écriture unique, et une exécution sur de multiples plateformes.

Depuis la première version de Java, ce langage propose des mécanismes et des syntaxes pour gérer le multi-tâches. Des verrous peuvent être posés sur chaque objet afin d'interdire les accès concurrents.

Ainsi, les méthodes `synchronized` manipulent les uns après les autres les attributs d'un objet. Lorsque Java est sorti, cela constituait une révolution. C'était la première fois qu'un langage intégrait un mécanisme similaire.

Malheureusement, les choses sont beaucoup plus complexes que ce que les concepteurs imaginaient initialement. En effet, les technologies multiprocesseurs intègrent des caches d'accès à la mémoire, violant parfois les intuitions des développeurs.

L'accès à la mémoire était décrit dans le chapitre dix-sept des spécifications. Celui-ci vient d'être revu entièrement. Après trois ans de discussion, le JSR 133 s'en est chargé.

Où est la difficulté ? Dans une architecture avec plusieurs processeurs, il faut organiser l'accès à la mémoire commune. Des caches sont présents pour optimiser cela.

Les données ne sont pas écrites directement en mémoire, mais dans un cache intermédiaire. Charge à lui de choisir le meilleur moment pour modifier réellement la mémoire de la machine, rendant ainsi visibles les modifications aux autres processeurs.

Il peut décider de réorganiser les lectures et les écritures, afin d'optimiser l'utilisation des verrous et de réduire les points de contention entre les différents processeurs. Les données peuvent être écrites et visibles par un autre processeur dans un ordre différent du cheminement normal du programme. Les caches de chaque processeur ne sont pas toujours cohérents entre eux. Si une donnée est modifiée dans un cache, elle n'est pas toujours visible dans un autre cache. Cela permet d'améliorer considérablement la vitesse d'exécution.

Toutes les assertions s'appuyant sur un ordonnancement des lectures et des écritures en mémoires sont erronées.

Pourquoi faut-il définir un modèle d'accès à la mémoire ? Les compilateurs organisent le code pour améliorer les performances. Une donnée peut être gardée dans un registre du processeur pour améliorer une boucle, par exemple.

La donnée n'est donc pas présente en mémoire, et ne peut être vue par un autre processeur. Les compilateurs JIT, les caches ou les processeurs font de même.

Il est alors très difficile de savoir exactement quand une donnée est écrite en mémoire et peut être visible par tous.

Sur une architecture avec un seul processeur, ces optimisations sont transparentes. Ce n'est pas le cas avec des architectures plus complexes.

Les JVM ont certaines libertés pour manipuler la mémoire. Trois règles permettent les synthétiser :

>■ Le compilateur est libre de réorganiser les traitements si cela ne modifie pas la sémantique ;

>■ Le processeur peut modifier l'ordre d'exécution des traitements dans certaines situations (parallélisme interne, optimisation, etc.) ;

>■ Les caches sont généralement utilisés pour retarder l'écriture en mémoire, avec éventuellement une réorganisation des accès.

Toutes ces situations entraînent qu'à un instant T, la vision que possède une tâche de la mémoire peut être différente de la vision d'une autre tâche.

Certains traitements permettent aux développeurs de prendre la main sur ces processus, et d'exiger certains comportements.

Les mots-clefs `synchronized` et `volatile` servent à cela. En dehors de ces traitements particuliers, la mémoire peut être vue différemment pour les différentes tâches. Un programme correctement rédigé, respectant le modèle d'accès à la mémoire de java, fonctionne quelle que soit l'architecture utilisée, mono ou multi-processeurs. Qui sait rédiger un programme respectant cela ?

Plusieurs problèmes existent dans le modèle d'accès à la mémoire traditionnelle.

## Les différents risques

Plusieurs difficultés apparaissent, lorsqu'on exige une qualité irréprochable du programme : l'ordre d'écriture n'est pas garanti ; les objets immuables ne le sont pas ; l'accès aux variables volatiles n'est pas synchronisé avec l'accès aux autres variables ; les écritures des attributs 64 bits ne sont pas atomiques.

## L'ordre d'écriture n'est pas garanti

Très souvent, les développeurs utilisent le modèle « Singleton » pour garantir qu'une seule instance d'une classe est présente. Ce modèle est utilisé à tort et

à travers, avec des impacts négatifs sur les performances et des risques quant à l'accès multiprocesseurs.

```
class Singleton
{
    static private Singleton singleton;
    private Singleton()
    {
    }
    public Singleton getSingleton()
    {
        if (singleton==null)
        {
            synchronized(Singleton.class)
            {
                if (singleton==null)
                {
                    singleton=new Singleton();
                }
            }
        }
        return singleton;
    }
    public void maMethode()
    {
    }
}
```

Pour éviter de déclarer la méthode `getSingleton()` en `synchronized`, un mécanisme de double vérification est utilisé. Un premier test permet de voir si la variable `singleton` est initialisée. Si c'est le cas, il est possible de retourner directement sa valeur. Sinon, nous entrons dans un cas critique. C'est normalement le premier appel à la méthode.

Que se passe-t-il si deux tâches invoquent cette méthode pour la première fois au même moment ? La variable `singleton` est à `null`, le bloc `synchronized` est alors ouvert. Mais, juste avant, l'autre tâche peut avoir valorisé la variable. Entre le premier test et l'ouverture du bloc `synchronized`, il peut se passer beaucoup de choses. Pour détecter cela, une deuxième vérification synchrone est effectuée. Si, et seulement si, les deux tests sont positifs, la variable est initialisée avec une instance.

Comme l'ordre d'écriture des variables n'est pas garanti entre les différentes tâches, il est possible que le deuxième test soit valide, alors qu'une autre tâche a déjà initialisé l'instance. Deux instances seront alors construites. C'est un comble pour un Singleton !

### Les objets immuables ne le sont pas

Les objets immuables sont des objets qui n'évoluent plus après leur construction. Ils possèdent généralement des attributs finaux. C'est le cas de la classe `String`. Ils sont supposés ne pas avoir besoin

de synchronisation car les accès multiples en lecture ne posent pas de problèmes d'accès. Mais, à cause des problèmes d'ordonnement des accès, il est possible, bien qu'extrêmement rare, qu'une donnée soit lue par une tâche avant d'avoir été initialisée par une autre.

Comment cela peut-il arriver ? Les chaînes de caractères de java possèdent trois attributs : un tampon de caractère, un index de début et un index de fin délimitant la chaîne dans le tampon. Pourquoi utiliser des index pour les chaînes ?

Cela permet de partager les tampons entre plusieurs instances `String` et/ou des instances `StringBuffer`. Une chaîne construite à partir d'une autre partage son tampon. La méthode `substring` retourne une instance `String` pointant sur le tampon de la chaîne originale, mais avec des index différents.

Pour éviter la duplication souvent inutile du tampon, la conversion d'une instance `StringBuffer` en `String` permet à la chaîne de voler le tampon du `StringBuffer`. Dans ce cas, un drapeau est levé dans l'instance `StringBuffer` pour dupliquer le tampon lors de la première méthode désirant le modifier. Généralement, une chaîne est construite dans un `StringBuffer`, puis convertie en `String` et enfin, le `StringBuffer` est abandonné. Dans ce cas classique, le tampon n'est jamais dupliqué en mémoire.

Lors de la création d'une instance `String`, le constructeur de `Object` est invoqué. À ce moment précis, les valeurs des index de début et de fin de la chaîne sont à zéro. Il est possible qu'une autre tâche exploite ces valeurs avant leur initialisation.

Dans l'exemple suivant, la variable `s2` partage le tampon avec la variable `s1`, et indique un décalage de cinq par rapport au début de celui-ci.

```
String s1="hello world";
String s2=s1.substring(5);
```

`S2` possède normalement la valeur « `world` ». Si une autre tâche utilise la variable `s2` avant la fin visible de l'initialisation, l'offset peut être à zéro. Dans ce cas, `s2` peut posséder la valeur « `hello` » ou « `hello world` ». Bien que les chaînes soient immuables, il existe une

période critique pendant laquelle l'instance n'est pas complètement initialisée. Si le compilateur java, le compilateur JIT de la JVM ou le cache du processeur décident de valoriser la variable `s2` avant de valoriser l'index de `s2`, ce risque existe.

### L'accès aux variables volatiles

Les variables déclarées volatiles ne sont pas placées dans des registres du processeur. Ainsi, les écritures sont visibles pour toutes les tâches. Le compilateur java, le compilateur JIT, le processeur ou le cache ne réorganisent pas les variables volatiles les unes par rapport aux autres.

Cela n'indique pas que les variables non volatiles sont écrites avant ou après, ou dans l'ordre décrit par le programme. Il n'est pas possible d'utiliser une variable `volatile` comme signal qu'un traitement est terminé.

```
Map configOptions;
char[] configText;
volatile boolean initialized = false;
...
// In thread A
configOptions = new HashMap();
configText = readConfigFile(fileName);
processConfigOptions(configText, configOptions);
configOptions=Collections.unmodifiableMap(java.util.
Map(configOptions);
initialized = true;
...
// In thread B
while (!initialized)
    sleep();
// use configOptions
```

Dans cet exemple, la variable `initialized` ne permet pas de garantir que les options de configuration sont bien présentes. Dans la tâche `B`, l'initialisation peut être déclarée comme terminée, alors que ce n'est pas encore le cas. L'idée est d'utiliser la variable `initialized` comme un gardien d'accès aux paramètres de configuration.

C'est une bonne idée, mais avec les anciennes JVM, cela n'est pas certain. La seule garantie que propose l'attribut `volatile` est une écriture directe en mémoire, indépendamment des autres écritures.



## L'accès aux variables composites

Les JVM de java s'appuient sur les processeurs d'accueil pour l'écriture en mémoire. Il est indiqué dans les spécifications que la lecture et l'écriture des types primitifs tenant dans 32 bits doivent être atomiques.

C'est-à-dire que le processeur ne peut interrompre une écriture d'une donnée 32 bits pour donner la main à une autre tâche.

En pratique, je pense que les JVM fonctionnant sur des processeurs 8 bits dans des cartes à puces par exemple, ne respectent pas cette contrainte.

Pour les données 64 bits, le processeur est obligé d'effectuer deux opérations élémentaires pour modifier la mémoire.

Par exemple, pour écrire une variable de type `long`, le processeur doit écrire dans un premier temps le poids fort, puis le poids faible – ou l'inverse suivant les architectures.

Entre les deux écritures, une interruption peut donner la main à une autre tâche. Si celle-ci consulte la variable de type `long`, elle obtient une valeur qui n'est ni la valeur avant l'écriture, ni la valeur de l'écriture, mais une valeur correspondant à une modification partielle.

Par exemple, la variable `v` de type `long` possède la valeur `0x1234567887654321` en hexadécimal. Nous souhaitons y placer la valeur zéro. Si le processeur écrit le poids faible dans un premier temps, il existe un moment très court où la variable `v` possède la valeur `0x1234567800000000`.

Si par malchance, la main est donnée à une autre tâche pendant ce moment critique, la valeur est erronée.

Par contre, si le processeur écrit le poids fort en premier, la variable `v` peut avoir la valeur `0x87654321`.

Pour mémoire, le petit robot qui a parcouru Mars avait un bogue de ce type. Une tâche de priorité faible s'occupant de la météo a planté le programme.



## Les solutions : JSR 133

Le problème résulte de la visibilité des modifications d'une variable par différentes tâches. Il peut y avoir de nombreuses raisons pour qu'une donnée modifiée par une tâche ne soit pas visible par une autre tâche.

Le compilateur peut organiser le code différemment pour optimiser les performances ; il peut placer la variable dans un registre du processeur ; le compilateur JIT de la JVM peut utiliser des optimisations agressives ; le cache du processeur peut attendre avant d'écrire la donnée dans la mémoire physique ; il peut posséder une ancienne valeur.

Les nouvelles spécifications donnent des sémantiques précises aux adjectifs `synchronized`, `volatile` et `final`.

### Synchronized

La première précision concerne l'utilisation des blocs `synchronized`. Maintenant, tous les caches doivent être vidés à la fin d'un bloc `synchronized`. Cela concerne les variables présentes dans les registres et les caches du processeur.

Cela garantit que les variables écrites dans un bloc `synchronized` soient toutes visibles lorsqu'une autre tâche prend la main sur le moniteur. En dehors, rien n'est garanti.

### Volatile

La sémantique des variables volatiles évolue également. Déjà, les variables non atomiques comme `double` ou `long`, le deviennent si elles sont `volatile`.

Cela oblige la JVM à interdire une interruption du processeur lors de l'écriture des poids faibles et forts de ces variables.

Cela a un impact sur les performances. Le processeur doit exécuter une instruction supplémentaire pour se placer dans un mode sans interruption, puis doit exécuter une autre instruction pour retourner dans le mode classique.

Cela permet de résoudre le problème, mais c'est à utiliser avec discernement.

Comme nous l'avons vu, l'ordre d'écriture des variables volatiles est garanti, mais pas des autres variables. La nouvelle sémantique impose que toutes les variables en cours doivent être écrites en mémoire physique, lors de la manipulation d'une variable volatile.

Cela a un impact important sur le code existant. En effet, les compilateurs avaient la liberté de réorganiser le code comme ils le souhaitaient, du moment que la sémantique était sauvegardée. Maintenant, ils doivent limiter la réorganisation aux codes présents avant l'écriture d'une variable `volatile`.

Il faut normalement recompiler toutes les applications avec un compilateur récent, afin de respecter ces nouvelles spécifications.

Afin de conserver l'utilisation habituelle d'une variable volatile comme jalon à des traitements plus complexes, le comité a décidé de pénaliser les performances pour corriger le code existant.

### Final

Les nouvelles spécifications garantissent que les attributs finaux seront correctement initialisés et visibles au terme d'une initialisation.



Attention, cela ne concerne pas l'utilisation par une autre tâche de `this`, avant la fin du constructeur. Ainsi, les variables finales sont considérées comme sûres, et ne nécessitent pas d'accès concurrent. Le compilateur peut les cacher dans des registres s'il le souhaite.

### A lieu avant

Le comité a également défini précisément l'ordre d'exécution suivant différentes situations :

- >■ Toutes les actions dans une tâche ont lieu avant les actions arrivant après dans l'ordre du programme ;
- >■ Le déblocage d'un moniteur arrive avant le blocage du même moniteur ;
- >■ L'écriture d'une variable volatile arrive avant la lecture de cette variable ;
- >■ Un appel à `Thread.start()` sur une tâche, arrive avant toutes actions sur cette tâche ;

➤ Toutes les actions dans une tâche arrivent avant qu'une autre tâche les rejoigne par `Thread.join()`.

Ces règles permettent de régler le problème de l'initialisation de l'exemple ci-dessus. Comme la valorisation de la variable `initialized` arrive après l'initialisation de l'instance, les nouvelles règles garantissent que toutes les informations seront visibles avant la modification de la variable.

Est-ce que cela règle le problème de la double vérification, utilisée pour initialiser un singleton ? En déclarant la variable `singleton` en `volatile`, toutes les modifications précédant la valorisation sont visibles pour les autres tâches. Le problème est alors réglé. Mais, c'est au prix d'un coût important en performance. Il est nettement préférable d'éviter l'utilisation absurde du singleton en déclarant des méthodes statiques à la place.

```
final public class Singleton
{
    public static void maMethode()
    {
        ...
    }
    ...
    Singleton.maMethode();
}
```

Le singleton ne se justifie que s'il existe plusieurs sous-classes possibles, et que des paramètres d'initialisation permettent de sélectionner la bonne sous-classe à utiliser. Par exemple, s'il existe les sous-classes `Windows` et `XWindows` à la classe `Singleton`, et qu'un paramètre permet d'initialiser la classe avec l'une ou l'autre instance.

Le modèle Singleton est un artifice technique permettant de combler un manque de java : l'impossibilité de redéfinir une méthode statique. Le polymorphisme ne fonctionne pas avec les méthodes statiques. Pour contourner ce problème, une instance artificielle est utilisée, permettant de bénéficier du polymorphisme dans des méthodes statiques.

Ce modèle est né du C++ qui ne possède pas d'initialisation paresseuse. Cela permet de contrôler le moment d'une initialisation. Avec java, ce n'est pas nécessaire. En effet, les classes ne sont chargées en mémoire que si on les utilise.

Il ne devrait pas y avoir de méthodes d'instances publiques dans un singleton. Seul un accès `static` devrait être proposé. Même en cas d'héritage du singleton.

```
class Singleton
{
    static volatile private Singleton singleton;

    private Singleton()
    {
    }
    private Singleton getSingleton()
    {
        if (singleton==null)
        {
            synchronized(Singleton.class)
            {
                if (singleton==null)
                {
                    switch (...)
                    {
                        case Windows :
                            singleton=new WindowsSingleton();
                            break;
                        case Unix :
                            singleton=new UnixSingleton();
                            break;
                    }
                }
            }
        }
        return singleton;
    }
    private void maMethode()
    {
        ...
    }
    public static void maMethode()
    {
        getSingleton().maMethode();
    }
}
```

Cela permet de factoriser l'invocation de la méthode `getSingleton()`. Il n'est pas nécessaire de l'invoquer dans l'appelant à chaque fois que l'on souhaite utiliser `maMethode()`.

Comme nous l'avons vu, il est nettement préférable de s'en passer. Les variables volatiles sont très coûteuses, et exigent une recompilation de vos applications. Il existe pourtant une approche élégante et sûre : utiliser l'initialisation paresseuse naturelle de java. Pourquoi résoudre des problèmes du C++ dans Java ?

```
class Singleton
{
    static volatile private Singleton singleton;

    static
    {
        switch (...)
        {
            case Windows :
                singleton=new WindowsSingleton();
                break;
            case Unix :
                singleton=new UnixSingleton();
                break;
        }
    }
}
```

```
}
private Singleton()
{
}
private Singleton getSingleton()
{
    return singleton;
}
private void maMethode()
{
    ...
}
public static void maMethode()
{
    getSingleton().maMethode();
}
```

Java n'initialisera l'instance `singleton` que lors de la première invocation de `maMethode()`. Je trouve trop souvent du code bêtement appliqué, sans réfléchir. Les singletons sont légion. Les arguments avancés pour les justifier sont généralement les suivants : « c'est plus objet », « c'est un modèle connu. On le voit partout ». En quoi le modèle singleton rend l'application plus objet ? Si c'est le cas, il faut supprimer les méthodes statiques. Oui, mais le singleton a besoin de la méthode `getSingleton()` qui est statique ! Cet argument ne tient pas. Ce n'est pas parce que le modèle singleton est utilisé n'importe comment qu'il faut suivre le troupeau et l'appliquer à toutes les sauces. Les précisions sur le nouveau modèle mémoire entraînent des impacts particulièrement négatifs lors de l'utilisation de ce modèle. Pensez-y avant de foncer sur une route cabossée.

Gardez toujours à l'esprit le nouveau modèle mémoire, et les risques d'une mauvaise utilisation. Il est rare que les développeurs possèdent des machines avec de nombreux processeurs. Ils risquent de ne pas identifier les problèmes évoqués, et ne peuvent même pas les reproduire sur leurs postes de développement. Les précisions des JSR 133 sont à prendre en compte dès maintenant dans vos applications multi-tâches.

## Références

- <http://www.jcp.org/aboutjava/communityprocess/review/jsr133/>
- <http://www.philippe.prados.name/Langage/Java/index.html#Singleton>