Memory Model : C++ vs Java

Féaux de Lacroix Martin Herbert Damien

1 Modèle Mémoire :

Dans le contexte de système de mémoire partagé, le but d'un modèle mémoire est de définir les valeurs que peut renvoyé la lecture d'une mémoire partagé.

2 Consistance mémoire :

2.1 Consistance Séquentielle :

Dans un programme avec un seul thread, une lecture est sensée retourne la dernière valeur écrite à cette adresse, où "dernière" est définie uniquement par l'ordre du programme. Ceci permet au programmeur de voir les accès mémoires comme étant fait (atomiquement) un par un, dans l'ordre du programme. Cela permet aussi au compilateur de réordonner les accès tant que l'ordre du programme est préservé entre l'écriture et la prochaine lecture sur la même adresse.

Il peut être tentant d'utiliser le même raccourci pour les programmes à mémoires partagés, en regardant les accès mémoires comme des accès entrelacés qui sont effectués les uns après les autres, cependant ceci n'est pas vrai au niveau hardware dès qu'on a plusieurs coeurs sur son processeur et au niveau d compilateur des optimisations comme déplacer l'accès à un invariant de boucle avant la boucle, puisque tout changement est visible pour tout le programme et être ainsi contraire à la consistance séquentielle.

2.2 Consistance relaxé :

Ce genre de modèle mémoire autorise à rendre visible l'écriture à un emplacement dans des ordres inconsistants pour tout les threads. En réciproque,

la lecture d'un emplacement mémoire n'a pas à être consistant par rapport à l'ordre d'exécution des programmes même s'il existe une dépendance de la donnée à travers les threads.

Note : La majorité du travail effectué dans ce domaine concerne surtout des implémentations hardwares car difficiles d'interfacer pour un langage de programmation mais elles permettent aux compilateurs d'effectuer des optimisations aux processeurs et compilateurs.

2.3 Modèle mémoire "Data-Race-Free" :

Ce genre de modèle mémoire prennent une approche différente. Ils remarquent qu'une bonne approche en programmation est d'avoir des programmes bien synchronisé ou alors sans accès concourant. Tout accès en mémoire pour une adresse signifie que celle-ci est soit en lecture seulement ou alors synchronisé. Dans les autres cas, les accès mémoires ne sont pas visibles entre les différents threads ou alors la consistance mémoire n'est pas garantie par le modèle mémoire.

Ceci est le modèle de choix pour les langages de programmation mais lorsque ceux-ci souhaitent avoir des garanties de sécurités même pour les programmes ne respectant pas les conditions nommées ci-dessus (tel que Java), alors ceux-ci doivent aussi inclurent d'autres théories comme la consistance relaxée et ceux-ci génèrent des bugs toujours non résolus dans leurs modèles mémoires. Il existe d'autres raisons d'y inclurent d'autres théories comme des optimisations, tel qu'on peut en voir dans les programmes "wait-free et lock-free" .

3 API :

Pour pallier ce problème de mémoire partagée, Java et C++ mettent à disposition des API depuis les versions 1.5 et C++11 respectivement.

3.1 Visibilité

En Java et C++, une variable partagée peut être mise en cache dans la mémoire locale d'un thread. Le mot-clé "volatile" a été ajouté dans ces deux langages pour indiquer au processeur que la variable partagée ne doit pas être

mise en cache et donc doit toujours lire/écrire dans la mémoire partagée. Cela s'appelle la garantie de visibilité de la variable.

3.2 Atomicité

La visibilité de la variable n'est pas suffisante dans certains cas. Par exemple l'opération "i++" est en réalité deux opérations sur la mémoire partagée : lire la valeur de i puis écrire dans la variable i.

Lorsque deux threads font cette opération alors il est possible que les deux threads lisent la valeur de i en même temps et donc fassent l'opération "i++" avec la même valeur pour i, ce qui fait la même chose que si un seul thread avait fait l'opération. Pour empêcher l'accès à la variable i, il existe plusieurs solutions plus ou moins efficaces qui ont été ajouté dans Java et C++.

3.2.1 synchronized

Le mot-clé synchronized en Java permet d'empêcher l'accès à une partie du code à plusieurs threads en même temps, il n'existe pas d'équivalent en C++.

3.2.2 Verrou

Le verrou permet de ne laisser qu'un seul thread avoir l'accès à du code, il peut relâcher le verrou et ainsi laisser d'autres threads à avoir le verrou.

3.2.3 CAS

Une autre possibilité est d'utiliser une API en Java et C++ qui gère le cas particulier "i++" : c'est la classe AtomicInteger en Java et la classe atomic_int en C++, elles mettent à disposition des implémentations d'incrémentation atomiques (getAndIncrement en Java et operator++ en C++). Ces deux implémentations se basent sur le principe de CAS qui est une instruction présente dans certains processeurs, il existe une implémentation en Java et C++ le cas échéant.

Résultat de i++ à la fiin d'éxécution de 3 processus en Java					
itérations	ThreadTest	ThreadTestVolatile	ThreadTestSynchronized	ThreadTestLock	ThreadTestAtomic
10000000	22248422	19304346	30000000	30000000	30000000
100000000	111778396	203220207	300000000	300000000	300000000
1000000000	1066145197	1573786700	3000000000	3000000000	3000000000

Résultat de i++ à la fiin d'éxécution de 3 processus en C++					
itérations	Thread-test	Thread-test-volatile	x	Thread-test-lock	Thread-test-atomic
10000000	11096660	11060573	x	30000000	30000000
100000000	101148541	108286257	x	300000000	300000000
1000000000	1066402542	1175393691	x	3000000000	3000000000

Temps d'exécution du programme en Java					
itérations	ThreadTest	ThreadTestVolatile	ThreadTestSynchronized	ThreadTestLock	ThreadTestAtomic
10000000	0,3	0,55	0,94	0,54	0,59
100000000	1,62	2,46	6,28	2,63	3,17
1000000000	14,09	28,8	63,97	22,76	30,42

Temps d'exécution du programme en C++					
itérations	Thread-test	Thread-test-volatile	x	Thread-test-lock	Thread-test-atomic
10000000	0,28	0,27	x	1,07	0,35
100000000	2,97	2,82	x	10,23	3,44
1000000000	28,23	26,8	x	102,16	29,79

3.2.4 Rapidité

Parmi les différents choix vus pour implémenter i++, le verrou en Java et la classe atomic_int en C++ sont les plus rapides dans leur langage respectivement.

Voir l'implémentation.

4 Documentation :

Spécification du modèle mémoire pour le programmeur.

Spécification du modèle mémoire de oracle pour le programmeur.

Références

[1] Stackoverflow : C++11 Memory Model explanation

[2] Encyclopedia of Parallel Computin - Sarita V. AdveHans J. Boehm

[3] Oracle documentation on memory model
[4] A Primer on Memory Consistency and Cache Coherencel
[5] Memory Management in Java
[6] Volatile Vs Atomic
[7] Understanding Java Memory Model