Multi-programmation

Partie 4 - modèle mémoire

Malek.Bengougam@gmail.com

Ordre mémoire ?

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

- Ces différentes valeurs permettent de définir l'ordre des opérations qui se trouvent séquentiellement (dans le programme) à proximité d'une opération pour laquelle un de ces paramètres est spécifié.

Pourquoi changer l'ordre mémoire ?

- **Sequentially_consistent** :
  - les accès atomiques sont **fortement synchronisés** pour limiter les réordonnancements de tout type d'accès (atomiques ou non)
  - Une **écriture** d'une variable atomique dans un thread implique que toutes les variables modifiées précédemment dans le thread courant seront **visibles** par un autre thread au moment de la lecture d'une variable atomique

  - *seq_cst* défini un **ordre total** conservateur :
  - ➢ Les accès atomiques inter-threads sont toujours synchronisés et séquentiels
  - ➢ Le contenu de la mémoire est toujours le même pour tous les threads
  - ➢ Les load d'un thread voient toujours des données à jour

- A contrario**, Relaxed** :
  - les variables atomiques agissent comme des variables non-atomiques. Cependant les **opérations atomiques** sont garanties d'agir de façon atomiques.

C'est une optimisation

- Contrairement à des langages comme le C# ou le java, le modèle mémoire du C++11 permet un contrôle plus fin de la synchronisation.

- On sait que trop de synchronisation entre les thread engendre de la contention.
  - On pourrait se contenter de mettre des mutex partout.

- C'est notamment important pour les algorithmes dit "lock-free" mais plus globalement cela permet de réduire le temps effectif pris par les contentions.

Qui peut réordonnancer ? (1)

- Le compilateur est le premier maillon du réordonnancement.
  - Il peut combiner plusieurs instructions, modifier l'ordre des instructions etc...

- On peut inhiber localement le réordonnancement par l'usage de barrières (*barrier* aussi appelée *fence*)
  - Spécifiquement, ici on parle de ***compiler fence***

- Ces barrières sont proposées traditionnellement sous la forme de fonctions de l'OS (ou des fonctions intrinsèques)
  - Par exemple les intrinsics _readBarrier() et _writeBarrier() de l'API Windows
  - De nos jours il est préférable d'utiliser les fonctions du C++11

- Cet ordre n'est garantie que dans le fil (thread) courant d'exécution !
  - Ou si tous les threads s'exécutent sur le même coeur (par affinité ou limite de la machine)

Qui peut réordonnancer ? (2)

- Le CPU peut également réordonnancer les instructions dans le fil (thread) d'exécution courant.

- Les différentes valeurs de l'enum *memory_order* insèrent différentes formes de barrières mémoires -qui peuvent être des instructions spécifiques, des variantes d'instructions ou des modificateurs au niveau des mnémoniques
  - On les appelle également ***memory fence*** ou ***CPU fence***.

- Une *memory fence* implique forcément une *compiler fence.*

- *En l'occurrence, seq_cst implique une **barrière complète** (__mm_mfence) et donc un ordre total (sauf si un load ou un store n'est pas seq_cst).*
  - *Sur les architectures x86_64 les loads peuvent être réordonnancés avec des stores qui précédent si les adresses mémoires sont différentes, sauf si mfence est spécifiée.*

L'implémentation de l'ordre mémoire

- Techniquement, l'ordre mémoire est dépendant de l'architecture CPU

Allowed Memory reordering in some architectures

Type	ARMv7	x86	AMD64
Loads reordered after Loads	Yes	No	No
Loads reordered after Stores	Yes	No	No
Stores reordered after Stores	Yes	No	No
Stores reordered after Loads	Yes	Yes	Yes

- Cela signifie que contrairement aux architectures ARM, les architectures x86 (IA32) et x86_64 (AMD64) dans la majorité des cas ont un modèle mémoire CPU **strict**
- ce qui implique qu'il n'est pas nécessaire de générer des instructions supplémentaires (sauf pour un cas).

Les fences en C++11

- `atomic_signal_fence(memory_order Order)`
- `atomic_thread_fence(memory_order Order)`
  - La forme standard d'une barrière qui peut être en lecture (*acquire* ou *consume*), en écriture (*release*), ou les deux à la fois (*acquire-release*)

- La différence entre *compiler fence* et *memory fence* est difficile à bien cerner en C++11.
- Pratiquement, si l'une des variables avant ou après la fence est atomique on a à faire à une *memory fence*, autrement c'est une *compiler fence*.

- A noter que dans le cas des compiler fences:
  - Il n'y a pas de différences entre *consume* et *acquire*
  - il n'y a pas de différences entre *acquire-release* et *sequentially-consistent*

  - https://en.cppreference.com/w/cpp/atomic/atomic_thread_fence

Typologie de l'ordre mémoire (1)

- **Acquire** : garantie que dans le même thread tous les load et store **après** l'usage de *acquire* ne s'exécutent pas avant (*happens-after*).
  - *Attention : les lectures et écritures ayant eu lieu avant peuvent par contre être ordonnancées après la barrière*

- Spécifier *acquire* signifie également que l'on souhaite que les variables soient à jour.

- **Consume** : est moins fort que *acquire* mais introduit une notion de **dépendance**. Seules les opérations qui dépendent spécifiquement des variables concernées par le consume suivent la règle *happens-after*.
  - std::kill_dependancy() met un terme à la chaîne de dépendance.

Ordre mémoire et multithreading (1)

- Avec plusieurs thread une fence implique une synchronisation des états (***commit***).
- Un *acquire* (ou un *consume*) force donc la lecture des variables d'un *commit* (***pull***).
- Les écritures de variables utilisées dans d'autres threads sont visibles dans le thread courant si il y'a une fence spécifiée *release (**push**)* après l'écriture.
- Il y'a donc une relation de synchronisation (***synchronizes-with***) entre les threads par l'usage de *fences*.
  - Plus spécifiquement, on a une relation ***happens-before*** entre les threads.

Exemple introductif

{A}

atomic_thread_fence(acquire)

{B}

atomic_thread_fence(release)

{C}

- Les opérations de {B} s'exécuteront toujours après le acquire et avant le release (on a une exécution de type ***happens-before***)
    - des opérations de {A} peuvent avoir lieu après le acquire (mais pas après release) car elles ne sont pas concernées par le acquire
    - des opérations de {C} peuvent avoir lieu avant le release (mais pas avant acquire) car elles ne sont pas concernées par le release

- Un thread X procédant à un release implique qu'un autre thread Y procédant acquire sera à jour avec les données de X (***synchronises-with***). Cela dit, rien ne permet aux opérations de {B} d'éviter des *data races*.
- X et Y peuvent traiter {B} en même temps et modifier les résultats de chacun.

Typologie de l'ordre mémoire (2)

- **Release** : implique une relation de type ***happens-before*** dans l'ordonnancement des lectures et écritures.
- Dans le cas où le thread courant exécute un *acquire*, l'ensemble des variables modifiées avant *release* dans un autre thread seront visibles et à jour dans le thread courant.
- A noter que si le release concerne un accès de type *consume* seules les variables qui ont une dépendance sur cette variable seront mises à jour.
  - Ceci étant purement théorique, les chaînes de dépendance ne semblent toujours pas gérées par les compilateurs.

- **Acquire_release** : les accès de type lecture puis écriture (*read-modify-write*) sont séquencés dans le même thread de sortes à préserver l'ordre séquentiel.
- Cela reste moins contraignant que *seq_cst* dans le sens où un ordre total n'est pas imposé.

Exemple introductif : acquire_release

{A}
atomic_thread_fence(acquire_release)
{B}
atomic_thread_fence(acquire_release)
{C}

- Les opérations de {B} s'exécuteront toujours après le acquire et avant le release (on a une exécution de type ***happens-before***)
- Les opérations de {A} s'exécuteront toujours avant le release (on a une exécution de type ***happens-before***)
- Les opérations de {C} s'exécuteront toujours avant le acquire (on a une exécution de type ***happens-before***)

Les fences n'empêchent pas les data race

CPU 0

```
int compteur = 0;
bool updated = false;
…
compteur = 42;
atomic_thread_fence(release);
updated = true;
```

CPU 1

```
…

bool ok = updated;
atomic_thread_fence(acquire);
if (ok)
   assert(compteur > 0);
```

- "compteur = 42" est exécuté avant "updated = true" (***happens-before*** garanti par *release*).
- Compteur est toujours à jour dans les deux thread après l'écriture (***synchronises-with***) et avant lecture
- updated est lu avant compteur (***happens-before*** de acquire) … mais updated n'a pas encore été modifié !

Accès atomiques

- Les variables **atomiques** en C++11 ont un rôle triple:

- a.S'assurer que les opérations sont effectivement atomiques
  - On parle plus globalement de "transactions", on y reviendra

- b.Permettre la mise en place d'un ordre mémoire lors des load/stores

- c. Introduire une forme de synchronisation entre les threads

- La seule différence avec les fences du C++11 est le point (a)

Comment éviter les data race ? Exemple du signal

CPU 0

```
int compteur = 0;
atomic<bool> updated{false};
…
compteur = 42;
updated.store(true);
```

CPU 1

```
…


if (updated.load())
   assert(compteur > 0);
```

- Les fences sont implicites,
- comme nous sommes en seq_cst ***happens-before*** est garanti pour store (*release*).
- updated est lu avant compteur ***happens-before*** pour load (acquire) !
- compteur sera lu correctement après l'écriture (***synchronises-with***) car "updated = true" => "compteur > 0"

Quand utiliser relaxed ?

- Chaque fois qu'il n'est pas nécessaire d'avoir une synchronisation entre thread.
  - Car un store() atomique en seq_cst est coûteux du fait des primitives utilisées (interlock + fence)

- Lorsque seule la garantie d'atomicité est importante, tels les :
  - Compteur de référence (reference counter)
  - Compteurs d'évenement (event counter)

```
std::atomic<int> compteur = 0;
...
for (int loop = 0; loop < 42; loop++)
    compteur.fetch_add(1, memory_order_relaxed);
```

- Ce code, exécuté par N threads, produit toujours compteur = N*42

L'exemple du signal avec relaxed

CPU 0

```
int compteur = 0;
atomic<bool> updated{false};
…
compteur = 42;
updated.store(true, relaxed);
```

CPU 1

```
…


if (updated.load(relaxed))
   assert(compteur > 0);
```

- La seule garantie que l'on puisse avoir ici est que le store est totalement atomique,
- mais "compteur = 42" peut être exécuté après le store.
- De plus comme compteur n'est pas une variable atomique, l'assert peut s'enclencher alors que CPU0 est en train d'écrire 42 en mémoire !
- Si compteur est atomique on n'a plus de data race, mais les store peuvent être réordonnancés.

L'exemple du signal avec acquire et release

CPU 0

```
int compteur = 0;
atomic<bool> updated{false};
…
compteur = 42;
updated.store(true, release);
```

CPU 1

```
…


if (updated.load(acquire))
   assert(compteur > 0);
```

- On revient au cas initial avec les fences...
- … sauf que cette fois-ci il n'y a plus de data race!

Double-checked locking pattern : le retour de la revanche

```
std::mutex g_mtx;
// on suppose std::atomic<Singleton*> m_instance;
Singleton* Singleton::getInstance() {
    Singleton* instance = m_instance.load(acquire);
    if (instance == NULL) {
        std::lock_guard<std::mutex> lock(g_mtx);
        instance = m_instance.load(relaxed); // relaxed suffisant car mutex
        if (instance == NULL) {
           instance = new Singleton;
           m_instance.store(instance, release);
        }
    }
    return instance;
}
```

Ex : (shared) smart pointer

- Un smart pointer se caractérise par :
- Un compteur de référence
- La destruction automatique des objets si compteur nul (ou négatif)

- En programmation concurrente, la difficulté est ici de s'assurer que l'objet n'est plus utilisable une fois détruit.
  - S'assurer que les modifications du compteur (inc/dec) sont visibles
  - S'assurer qu'on ne tente pas de le détruire plusieurs fois

références

- https://en.cppreference.com/w/cpp/atomic/memory_order

- https://preshing.com/20120930/weak-vs-strong-memory-models
- https://preshing.com/20120913/acquire-and-release-semantics
- https://preshing.com/20120625/memory-ordering-at-compile-time
- https://preshing.com/20120515/memory-reordering-caught-in-the-act
- https://preshing.com/20120710/memory-barriers-are-like-source-control-operations

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2055r0.pdf
- https://www.think-cell.com/en/career/talks/pdf/think-cell_talk_memorymodel.pdf
- http://gee.cs.oswego.edu/dl/jmm/cookbook.html (java memory model, très intéressant)
- https://preshing.com/20141024/my-multicore-talk-at-cppcon-2014/