Multi-programmation

Introduction au lock-free programming

Malek.Bengougam@gmail.com

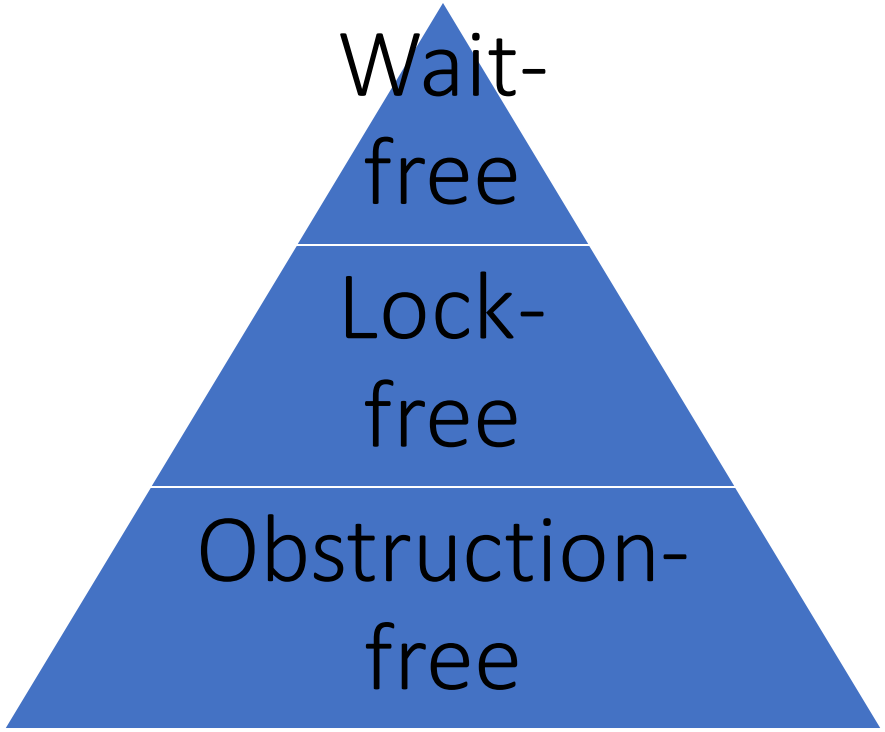
Un algorithme de lock est bon si...

- Il offre une exclusion mutuelle
  - Sécurité des transactions
- Il permet la progression du processus
  - On a toujours un thread qui s'exécute
- L'attente est limitée
  - Pas de famines (*starvation*)
- Idéalement, juste (*fair*) dans sa répartition
  - Préserve l'ordre des requêtes

Problèmes des locks

- Si le nombre de locks est insuffisant on risque des data races
- Si le nombre de locks est grand, on a des contentions importantes
  - Scalabilité réduite. Plus le nombre de thread concurrent augmente plus le risque qu'un thread soit bloqué en attente est fort.
- Situations de vérrouillage complet (deadlock) possibles
  - Verrouillage/déverrouillage dans le mauvais ordre, Cf. Diner des philosophes
- Difficile voire impossible de contrôler le comportement de tout le code en situation de vérrouillage (oubli, mauvais lock, exceptions, bibliothèques externes non thread-safe...)
- Les conditions variables posent également un problème, risque de perte de notification, spurious wakeup...
- https://en.wikipedia.org/wiki/Lock_(computer_science)#Disadvantages

Qu'est-ce que le Lock-free ?



- Une première acceptation de "lock-free" est "sans mutexes"
  - Autre acceptation: non-bloquant

- En fait, il y'a plusieurs formes de "lock-free"
- Un traitement fait un progrès dans l'exécution si

- Sans obstructions: aucun thread concurrent ne participe au traitement (isolé)
- Sans blocages (lock-free): au moins un thread fait avancer le traitement
- Sans attentes (wait-free): tous les threads effectuent leur traitement en un nombre fini d'étapes
  - Un algorithme "wait-free" est nécessairement "lock-free" (et donc "obstruction-free")
  - Un sémaphore peut être wait-free tant que l'on n'a pas excédé le nombre de token/slots autorisés !

- Un algorithme est "lock-free" lorsqu'à tout instant, au moins un thread effectue un progrès dans son traitement.
  - Ce n'est pas tant éviter les locks que l'on a besoin mais éviter la contention

Particularité des primitives "lock-free"

- Typiquement, une primitive lock-free est considérée plus rapide que qu'une primitive non "lock-free".
- En partie parcequ'une primitive lock-free supporte moins de cas d'utilisations, mais surtout parceque l'on évite les mutexes de l'OS
  - Cependant, les effets de bord des caches (par ex. *false sharing*) ont plus d'impact sur les performances qu'une contention entre deux threads

- De plus, une primitive lock-free ne peux pas être soumis à des deadlocks et son interruption (par la terminaison d'un thread par exemple) ne provoque aucun effet de bord sur les autres thread

- En C++11 ces primitives offrent également la possibilité de spécifier un ordre mémoire (qui est seq_cst par défaut).

Primitive atomiques : echanges atomiques

- Read-Modify-Write (RMW) atomique
- En C++11 :

  T z = x.**exchange**(y)

```
T exchange(T new)
{
 T old = load();
 store(new);
 return old;
}
```

- L'échange est aussi implicite dans les fonctions fetch_***() par exemple :

  int z = **fetch_add**(x, y);

  - La variable z contient alors la valeur précédente de x.

Primitives atomiques : Test-And-Set (TAS)

- En C++11, Test-and-Set est implémenté via **std::atomic_flag** qui est le seul type atomique pour lequel le C++11 garanti un fonctionnement "lock-free"
- Pour tous les autres types, le C++11 fourni la fonction **is_lock_free**() au **runtime** (et pas à la compilation).
  - La problématique est essentiellement spécifique à la plateforme, mais cela concerne l'alignement des données et l'état des caches

- La particularité d'atomic_flag est d'implémenter la fonctionnalité **test_and_set**() qui renvoi la valeur précédente du flag.

- Une variante du spinlock classique avec TAS: `while(flag.test_and_set() == true);`
- Le premier thread qui accède à flag, le "set" à vrai mais renvoi faux.
- Les autres threads sont bloqués tant que flag == true, c'est une forme de mutex !
  - flag.clear() remet la valeur à faux, ce qui équivaut ici à un unlock !

L'exemple du signal avec atomic_flag

CPU 0

```
int compteur = 0;

atomic_flag updated =
ATOMIC_FLAG_INIT;

…

compteur = 42;

updated.test_and_set();
```

CPU 1

```
…



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

- test_and_set() effectue une transaction de type seq_cst par défaut, acquire_release est sans doute suffisant ici.
- On a donc des garanties fortes en termes de réordonnancement et de synchronisation
- Il faut penser au reset() également

Spin lock avec TestAndSet

```
struct spin_lock
{
 void lock() {
   while (m_flag.test_and_set(acq_rel)) ; // ou bien acquire
 }
 void unlock() {
   m_flag.clear(); // ou bien release
 }
 private:
   std::atomic_flag m_flag = ATOMIC_FLAG_INIT;
}
```

- On peut éventuellement "optimiser" la contention du while par une forme de "(Binary) Exponential Backoff"
  - Ca peut être un std::yield par exemple, ou une attente dont le délai s'incrémente graduellement

Problématiques des précédentes primitives

- Test-and-Set, Fetch-and-Add ou Exchange sont considérées insuffisantes pour implémenter des primitives "lock-free".

- Ces primitives ne sont pas justes (fair) dans leur implémentation (les sémaphores et mutexes de l'OS maintiennent une liste chaînée des threads), difficilement récursive (voire pas du tout) et sans doute difficilement scalable.

- Ces primitives atomiques sont cependant intéressantes pour implémenter des algorithmes comme les mutex légers (spinlock avec Test-and-Set).

Compare-And-Swap (CAS)

```
bool compare_exchange(T expected,
T desired)
{
 T current = load();
 if (current == expected) {
   store(desired);
   return true;
 } else {
   expected = current;
   return false;
 }
}
```

- Effectue un échange conditionnel
- En C++11 : bool success = x.**compare_and_exchange_strong**(y, z)
  - Si x==y, x = z et success = true
  - Sinon y = x et success = false

- CAS permet de déterminer si x a changé (généralement y = la précédente valeur de x).
- Si la valeur n'a pas changé, on peut procéder à l'échange de x et z (qui peut être une simple affectation au final)

- Une variante **compare_and_exchange_weak**() peut retourner *false* même si x==y (*spurious failure*). La raison est que le hardware n'a pu accéder atomiquement en écriture dans un délai imparti, le "micro-lock" a généré un time-out.
  - (en x86 c'est le préfixe lock qui indique la requête d'un accès exclusif)

Usage classique de CAS

- Une utilisation classique de CAS est le spinlock (ou spin-wait).
- Il s'agit d'une forme dite "légère" du mutex où l'on fait du busy-wait sur le résultat de CAS (on boucle tant que success == false).
  - En fait, les boucles sont souvent utilisées dans les algorithmes lock-free

```
while (value.compare_exchange_weak(expected, desired));
```

- Notez que l'on utilise de préférence la version _weak dans une boucle

- Cependant le spin-lock n'est évidemment pas "wait-free" car précisément il est nécessaire de boucler sur la valeur de retour de CAS.
- Si on peut se passer d'une boucle, `compare_exchange_strong()` peut être une primitive idéale pour un algorithme "wait-free"

Le problème ABA

- Il s'agit d'une problématique d'aliasing.
- Prenons l'exemple d'une pile (stack) utilisant CAS comme primitive de synchronisation. Deux threads accèdent de manière concurrente à cette pile.
- Cette pile contient 3 valeurs A (top), B et C (bottom).

- Le thread1, pop A : "top" doit pointer alors sur B.
- CAS est utilisé pour vérifier qu'un autre thread n'a pas modifié "top" et faire top=B.
  - Cependant le thread1 est préempté à ce moment là (avant le CAS)
- Le thread2, pop A et B et re-push A
- Lorsque le thread1 s'exécute de nouveau, la comparaison de CAS ne voit pas de changement sur la pile (car l'ancienne valeur pointée par "top" est la même que l'actuelle)
- Le thread1 va donc pop A et faire pointer "top" sur B … qui n'est plus là.

Solutions possibles pour ABA

- La solution la plus simple consiste à ajouter un tag (versioning).
- Pour une pile, on va compter le nombre de pops.

- Le CAS doit alors être utilisé également sur le tag afin de vérifier si le compteur à changer
- … seulement le premier CAS va bloquer l'exécution sans pouvoir effectuer le second CAS.

- Il faudra pouvoir tester deux variables à la fois. C'est ce que font les primitives CAS2 (test et echange deux variables contigües en mémoire) ou DCAS (Double-CAS, test et échange deux variables non contigües).
  - Encore faut-il que ces primitives soient disponibles sur la plateforme...

- Une autre primitive alternative à CAS est LL/SC (load-link/store-conditional) que l'on retrouve sur la plupart des architectures RISC – qui ne disposent pas de CAS d'ailleurs.
  - Le découplage des load et des stores fait que le problème ABA n'apparait pas ici.

Solution possible en C++11

- Pour que le compilateur puisse générer une instruction de type CAS2 (comme cmpxchg8/16 ou équivalent) il faut un moyen de rendre deux variables atomiques ensemble.
  - https://www.felixcloutier.com/x86/cmpxchg8b:cmpxchg16b

- Théoriquement une struct atomique composée de deux variables (non atomiques) adjacentes en mémoire devrait suffir.
  - La seule difficulté vient du fait qu'il faut maintenant load/store la structure complète, pas possible de modifier un seul élément d'une struct atomique indépendamment
  - En pratique il est parfois également nécessaire d'aligner manuellement la structure sur la taille de ses membres avec alignas afin d'éviter que la struct soit splittée sur plusieurs lignes de caches

- En pratique il est (encore?) nécessaire de recourir à des hacks qui ne marchent pas sur tous les compilateurs comme utiliser un union entre une struct atomique et une struct de variables atomiques, cf https://stackoverflow.com/questions/38984153/how-can-i-implement-aba-counter-with-c11-cas

- Contre-intuitivement, un std::mutex (une courte section critique) pourrait être plus rapide que CAS2
  - Le coût d'un spinlock avec CAS2 peut s'avérer prohibitif

Solution alternative

- Plutôt que d'utiliser CAS2 avec deux 32 bits ou deux 64 bits, on peut éventuellement exploiter la structure comme une union entre un int/long int et un bitfield.

- Dans l'exemple de la pile, on peut ainsi réserver 16 bits pour le compteur de "pop" et 16 bits pour un index à la place d'un pointeur (ou d'un int).
  - A supposer que l'on ne peut pas avoir plus de 2^16 éléments dans la pile.

```
union {
    struct {
        uint32 pops : 16;
        uint32 index : 16;
    };
    uint32 casValue;
};
```

- Ainsi CAS est suffisant car l'entier 32 bits ne peut être identique lorsque le nombre de dépillement est pisté par pops.

Pendant qu'on parle des caches

- Le plus gros problème en termes de performance est lié aux caches
- Le *false sharing* est une problématique connue qui apparait lorsque plusieurs threads accèdent à une même variable partagée (accès global)
  - La solution radicale et préférable est d'éviter d'avoir des variables partagées (par exemple en thread-local storage, mais pas toujours possible).

- Dans le cas de CAS2 il est important de s'assurer que les 2 int ou long int sont sur la même ligne de cache (via alignas si le compilateur a du mal)

- Cependant, dans la plupart des cas, les problèmes viennent du fait que des variables partagées sont sur la même ligne de cache alors qu'elles ne sont pas utilisées dans le même contexte.
  - On subit donc un false-sharing alors que les threads n'accèdent pas aux mêmes variables!
  - Là aussi la solution consiste à ajouter du padding (alignas de la taille d'une ligne de cache) entre ces variables

références

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

- https://en.wikipedia.org/wiki/Load-link/store-conditional
- https://liblfds.org/mediawiki/index.php?title=Article:CAS_and_LL/SC_Implementation_Details_by_Processor_family

- https://stackoverflow.com/questions/35830641/can-x86-reorder-a-narrow-store-with-a-wider-load-that-fully-contains-it?noredirect=1&lq=1
- https://preshing.com/20150402/you-can-do-any-kind-of-atomic-read-modify-write-operation/

- https://fgiesen.wordpress.com/2014/07/07/cache-coherency/

- https://github.com/CppCon/CppCon2014/tree/master/Presentations/Lock-Free%20Programming%20(or%2C%20Juggling%20Razor%20Blades)
- https://github.com/CppCon/CppCon2015/tree/master/Presentations/C%2B%2B11%2C%2014%2C%2017%20Atomics%20-%20the%20Deep%20Dive
    - En rappel des principes déjà vu https://github.com/CppCon/CppCon2014/tree/master/Presentations/Overview%20of%20Parallel%20Programming%20in%20C%2B%2B

Software Transactional Model (STM)

Modèle transactionnel (ACID)

- Les règles des transactions concurrentes sont les suivantes:

- **Atomicité** : exécution complète (ou autrement pas d'exécution)
- **Consistance** : la transaction passe d'un état consistant à un autre état consistant (donne l'impression de s'exécuter les unes à la suite des autres)
- **Indépendance (ou isolation)** : le résultat d'une transaction n'est visible qu'une fois *commit* (les opérations restent correctes si d'autres transactions concurrentes ont lieu).
- (**Durabilité** : optionnellement, les effets sont persistents)

- Les **fences** ne disposent d'aucune de ces garanties
- Les **mutexes** répondent au modèle transactionnel pour l'ensemble de la section critique.
- Les opérations **atomiques** répondent au modèle transactionnel pour une instruction.