

Multi-programmation

Partie 3 – lock programming

Malek.Bengougam@gmail.com

lock programming : Pourquoi verrouiller ?

- Certaines parties du code sont sensibles aux accès concurrents
- C'est-à-dire que le déterminisme (résultat correct d'un calcul par ex., accès inconstant à une ressource...) peut être affecté par l'usage du multi-processing
- Un tel cas de figure se caractérise par la présence d'une "**section critique**".
- Une section critique est nécessaire dès lors qu'une ressource est partagée par plusieurs thread
 - On souhaite en effet qu'**un seul thread à la fois** accède à une section critique
 - Cette ressource peut être une simple variable, un fichier, une zone mémoire...

Exclusion mutuelle

- Pour garantir l'accès d'un seul thread à la fois à une section critique il faut un mode opératoire garantissant que les autres thread n'accèdent pas à la section critique.
 - C'est une forme de "*gentlemen's agreement*": tous les thread sont d'accord pour ne pas déranger un thread dans la section critique.
- La façon la plus simple consiste à appliquer un verrou (lock) au moment où l'on accède à la section critique.
- Lorsque le verrou est bloqué (locked) les autres thread exécutant ce verrou sont bloqués jusqu'à ce qu'il soit déverrouillé (unlocked)
 - On parle alors d'une "**exclusion mutuelle**" ou **mutex**
 - Le nombre de thread débloqués dépend du type de verrou utilisé

Accès compétitifs

- Une ***race condition*** (ou situation de compétition) apparait lorsque des thread entrent en compétition des threads dans l'accès aux ressources.
 - Une ressource est ici tout élément dont a besoin le thread pour fonctionner
- On l'a vu dans les exercices précédents. Lorsque plusieurs thread tentent d'écrire dans la même variable globale, le résultat peut être faussé (pour des raisons que l'on verra plus tard).
 - Il en va de même dans le cas de tout accès à une ressource (comme l'exemple de `std::cout`).
- Les race conditions entraînent des bugs difficiles à trouver et fixer car ils dépendent de phénomènes non-déterministes (ordonnancement etc...).
 - C'est donc une forme d'*heisenbug*

Problématiques de l'exclusion mutuelle

- Dans notre exemple, appliquer un verrou au moment de l'écriture de la variable permet de garantir le bon fonctionnement de l'addition.
- Cependant, ce n'est pas toujours sans problèmes:
- On dit qu'il y'a **contention** lorsqu'un thread est bloqué dans son exécution par un accès concurrentiel, il est alors dans une forme d'**attente active**
 - Idéalement, la contention d'une exclusion mutuelle doit être la plus courte possible afin de ne pas impacter les performances ou les traitements des autres thread
- Il est aussi possible qu'une situation conduise au non déverrouillage de la section critique.
 - En effet, il y'a une forme de couplage ou dépendance entre les threads en compétition
 - Dans le pire cas cela produit une situation d'**interblocage** (**deadlock**)

Réentrance et Ownership

- <https://fr.wikipedia.org/wiki/R%C3%A9entrance>

Exclusion mutuelle en C++11

- La primitive la plus simple est représentée par `std::mutex`
- Pour verrouiller on appelle **lock()** et **unlock()** pour déverrouiller.
- La méthode **try_lock()** est très utile pour réduire les contentions
 - `try_lock()` renvoi vrai et appel `lock()` lorsqu'elle peut verrouiller le mutex
 - Autrement, si le verrou est déjà pris, elle renvoie faux au lieu de bloquer
- Un problème peut apparaître si le mutex est verrouillé une seconde fois par le même thread. Dans ce cas on a un auto-blocage du thread !
- La solution consiste à utiliser un mutex récursif disposant d'une forme de compteur de référence comme `std::recursive_mutex`

RAII – Resource Acquisition Is Initialization

- Comme vous vous en doutez, il peut être très facile d'oublier un `unlock()`.
- Comme très souvent en C++, on peut raisonner objet et créer une classe qui `lock()` au moment de la construction, et `unlock()` avec l'appel au destructeur.
- `std::locked_guard<>`, ou `std::scoped_lock<>` depuis le C++11, utilise le RAII dans le but d'éviter ce genre d'écueil.
 - Habituellement on les utilise à l'intérieur d'un scope `{}` pour garantir que le mutex en paramètre de la template est déverrouillé en quittant le scope.
- Une autre alternative est `std::unique_lock<>` qui a comme particularité de pouvoir être `lock/unlock` explicitement, ceci dans le but explicite de transférer la responsabilité (ownership) du mutex à un autre objet.

Cas pratique : singleton

- Quels sont les problèmes posés par ce code ?

```
Singleton* Singleton::getInstance() {  
    if (m_instance == NULL) {  
        m_instance = new Singleton;  
    }  
    return m_instance;  
}
```

Code de mesure (sans lock, en mono thread)

```
std::chrono::duration<double> profileSingleton() {  
    constexpr auto tenMillions = 10000000;  
  
    auto t1 = std::chrono::system_clock::now();  
  
    for (size_t i= 0; i <= tenMillions; ++i){  
        Singleton::getInstance();  
    }  
  
    auto dt = std::chrono::system_clock::now() - t1;  
  
    std::cout << std::chrono::duration<double>(dt).count();  
}
```

Cas pratique : singleton version C++11

- Une solution possible consiste à se passer totalement de new et renvoyer une référence sur variable statique
- Attention, ceci ne marche qu'en C++11 car les standards garantissent l'ordre d'initialisation des variables globales
 - Les variables globales sont toujours initialisées avant d'être utilisées
- ```
Singleton* Singleton::getInstance() {
```
- ```
    static Singleton g_instance;
```
- ```
 return &g_instance;
```
- ```
}
```

Cas pratique : singleton avec lock

- Histoire de comparer les performances en multithreading, utilisons un lock

```
std::mutex g_mtx;  
...  
Singleton* Singleton::getInstance() {  
    std::lock_guard<std::mutex> lock(g_mtx);  
    if (m_instance == NULL) {  
        m_instance = new Singleton;  
    }  
    return m_instance;  
}
```

Cas pratique : double-checked singleton

- Intuitivement on peut supposer que le lock n'est pas gratuit.
- Techniquement un verrou n'est vraiment nécessaire que pour initialiser m_instance
- La technique du double-checked est une solution classique à ce problème ...
 - spoiler: marche pas. Tout le détail ici https://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf

```
std::mutex g_mtx;  
...  
Singleton* Singleton::getInstance() {  
    if (m_instance == NULL) {  
        std::lock_guard<std::mutex> lock(g_mtx);  
        if (m_instance == NULL) {  
            m_instance = new Singleton;  
        }  
    }  
    return m_instance;  
}
```

Code de mesure avec 4 threads

```
auto fut1 = async(launch::async, profileSingleton);  
auto fut2 = async(launch::async, profileSingleton);  
auto fut3 = async(launch::async, profileSingleton);  
auto fut4 = async(launch::async, profileSingleton);  
  
auto total = fut1.get() + fut2.get() + fut3.get() +  
fut4.get();  
std::cout << total.count() << std::endl;
```

- Pensez à modifier profileSingleton pour le cas multithread en tenant compte du nombre de thread dans les longueurs de boucle (ici on divisera le for par 4)
- On testera d'autres techniques plus tard sur la même base de code

r/w locks

- Aussi appelés **shared mutex**, les mutex de type reader-writer permettent d'autoriser plusieurs accès à une section critique de manière plus souple qu'un simple mutex.
- En l'occurrence, un shared mutex permet à l'accès à la section critique à tout thread en lecture seule mais continue de bloquer l'accès à la section critique pour une écriture.
- On a donc deux niveaux d'accès :
 - Exclusif : lock/unlock typique d'un mutex
 - Partagé : lock_shared/unlock_shared indiquant une autorisation en C++14
- Si le mutex est déjà verrouillé dans un niveau d'accès (lecture ou écriture) l'autre niveau d'accès est bloqué -sauf par le même thread-

r/w locks simplifié en C++11

```
std::shared_timed_mutex m;
```

```
void reader(){  
    std::shared_lock<std::shared_timed_mutex> guard(m);  
    // read-only  
}
```

```
void writer(){  
    std::lock_guard<std::shared_timed_mutex> guard(m);  
    // update  
}
```


Sémaphores

- la primitive semaphore est ici analogue au fonctionnement d'un sémaphore en signalisation ferroviaire.
 - Seul un nombre défini de train est autorisé à circuler sur une section
- C'est une des primitives de base de l'exclusion mutuelle inventée par Dijkstra. https://fr.wikipedia.org/wiki/D%C3%A9rivation_des_philosophes
- Un sémaphore ajoute la notion de compteur qui sert à la fois de limite et d'information quant à la disponibilité des ressources.
- [https://en.wikipedia.org/wiki/Semaphore_\(programming\)](https://en.wikipedia.org/wiki/Semaphore_(programming))
- La particularité est que n'importe quel thread peut acquérir ou restituer (release) une ressource.
- En C++20 on dispose de **counting_semaphore**, **binary_semaphore** est cas particulier pour max=1

Sémaphore

- Dans la nomenclature de Dijkstra on a les fonctionnalités suivantes:
- Init() : **initialise** le compteur (jetons)
- P() : **attribue** un jeton au processus, décrémente le compteur. Le sémaphore bloque le processus –qui est mis dans la file d'attente du sémaphore- lorsque le compteur atteint 0.
- V() : **restitue** un jeton, débloque un processus bloqué si présent dans la file, et lui attribue un jeton.
- Un sémaphore est différent d'un mutex car un sémaphore est avant tout un mécanisme de signal.
 - Un mutex implique l'acquisition d'une ressource par un mécanisme de verrouillage, tandis qu'un sémaphore entraîne une activation, une réponse à une demande d'activation.

Moniteurs (condition variables)

- Un moniteur est une primitive qui permet à la fois l'exclusion mutuelle et l'attente d'une condition (booléenne) sur une variable.
- Généralement, l'usage d'une attente variable conditionnelle implique un déverrouillage temporaire du mutex jusqu'à l'arrivée d'un événement ou notification faisant sortir la variable conditionnellement de l'attente.
- Une condition variable supporte les fonctionnalités suivantes:
- Wait : attend jusqu'à l'arrivée d'une notification (verrouille le mutex)
- Notify once ou broadcast (notify all) : envoi une notification à un thread ou plusieurs (all)
 - Attention, si la condition variable n'est pas en état wait, la notification est perdue !
 - Attention, si la condition variable est en état wait, toute notification lève le mutex!
 - On parle alors de *spurious wake-up*. La solution consiste à checker une variable supplémentaire.
- Comme la condition variable prend l'ownership du mutex, en C++11 on utilise généralement `std::unique_lock` pour implémenter les moniteurs.

Pourquoi ces primitives ?

- Techniquement, si on prend le cas d'un mutex ou semaphore binaire, on pourrait imaginer le programmer sous la forme d'une boucle `while()` testant un booléen.
- C'est ce que l'on appelle faire du **busy-wait** ou **attente active** (le thread tourne (**spin**) sur la condition)
 - c'est-à-dire que l'on va boucler tant que le verrou est pris (booléen à vrai).
- Cela peut être une solution tout à fait acceptable si le délai de la boucle est très courte.
 - Les **spin-lock**, que l'on verra plus tard, sont cependant une meilleure alternative.
- La différence réside dans l'intégration avec l'OS. Les primitives de synchronisation "natives" communiquent avec le CPU et l'OS en utilisant des instructions plus complexes.
 - Ainsi, le scheduler fait mieux son travail en évitant de donner trop de quantum à une tâche bloquée.
- L'attente active force l'exécution du thread sur le CPU sans synchronisation avec l'OS ce qui augmente potentiellement la charge CPU et réduit la marge de manoeuvre du scheduler
 - L'instruction intrinsèque `_mm_pause()` peut aider à transformer notre busy-wait en spin-lock plus optimal

Volatile : attention danger

- Dans le cas notable du busy-wait et spin-lock, marquer la variable partagée "**volatile**" est potentiellement important pour garantir la lecture correcte de la valeur.
 - En effet, si la variable est globale, et que le code se contente de lire le contenu, le compilateur peut optimiser le code en évitant de générer la lecture du contenu de la variable.
 - il va directement insérer la valeur, car il suppose que rien ne peut modifier la variable à cet instant, sans supposer qu'un événement externe (hardware, ou un thread) modifie le contenu de la mémoire.
- Dans le lien suivant on trouve un exemple de code qui utilise le mot clé volatile du C++
https://fr.wikipedia.org/wiki/Attente_active
 - L'utilisation est ici correcte car volatile permet de désactiver certaines optimisations du compilateur en forçant la relecture de la variable
- Pour une raison difficilement conciliable avec la raison, certains compilateurs comme Visual C++ forcent l'usage d'une primitive de synchronisation pour les variables déclarées volatile.
 - Cela donne une fausse impression que volatile suffit à rendre le code "thread safe" ce qui est source d'heisenbug sur d'autres plateformes
- Utilisez volatile si besoin, mais pour sa raison d'être, pas pour l'effet de bord Microsoft.

Les primitives à l'épreuve

Cas du producteur-consommateur

Producteur-consommateur

- Cas de gestion des ressources d'une file ou queue (FIFO)
- Les producteurs et consommateurs ne doivent pas traiter le même élément en même temps
- Le producteur ne peut stocker si la file n'est pas pleine
- Le consommateur ne peut rien faire si la file est vide

Producteur-consommateur

- Producteur

- Consommateur

Si place disponible

Produire

Si file non vide

Consommer

- L'implémentation naïve est assez simple mais pas idéale du fait de l'attente active
- Dans les deux fonctions il n'y a ni synchronisation ni attente, les deux thread s'exécute en continue.

Producteur-consommateur : mutex

- Producteur

lock

Si place disponible

Produire

unlock

- Consommateur

lock

Si file non vide

Consommer

unlock

- Avec un mutex, seul un des deux threads peut accéder à la file
- Cependant la contention permet de réduire la charge CPU en évitant l'attente active.

Producteur-consommateur : moniteur

- Producteur

Lock

Si file pleine

Attendre condition_pleine

// place disponible à partir de là

Produire

Notifier condition_vide

- Consommateur

lock

Si file vide

Attendre condition_vide

// element disponible à partir de là

Consommer

Notifier condition_pleine

- Avec un mutex, seul un des deux threads peut accéder à la file
- Cependant la contention permet de réduire la charge CPU en évitant l'attente active.
- Attention ici il manque des vérifications afin d'éviter les réveils inopinés (spurious wake-up)

Thread Pool

- Plutôt que d'allouer constamment des thread il est souvent plus judicieux de les allouer à l'avance.
- Cela peut prendre la forme simple suivante:

```
std::vector<std::thread> threadpool;  
threadpool.emplace_back(task1); etc...
```

- Si l'on veut plus de souplesse, il faudrait pouvoir changer la tâche exécutée par le thread, ce qui peut impliquer la recreation du thread avec la tâche suivante.
- Pour plus de souplesse, on peut considérer que les tâches du thread pool exécutent toutes le même code consistant à récupérer un "job" (fonction + data) sur une queue.

Thread pool et job system

- Le job est donc l'unité de travail et le thread pool le worker
- Une façon théoriquement simple de mettre en place un job system sur la base d'un thread pool conjugué à une forme de producteur-consommateur:
- Pour le boss/scheduler cela consiste à empiler les jobs à effectuer.
- Pour les workers/jobs il suffit de dépiler les tâches (sous formes de fonctions) et de les exécuter (avec éventuellement des paramètres).
- Alternativement, le job system peut être spécialisé sur une tâche comme dans les ECS modernes
 - Plutôt bien adapté au modèle peer-to-peer où il n'y a pas de boss/scheduler mais simplement une liste de tâches

Atomicité

Quelles valeurs sont possibles pour X et Y ?

CPU 0

A = 42

X = B

CPU 1

B = 24

Y = A

- A, B, X et Y sont des variables globales initialisées à zéro

On peut obtenir $X=0$ ET $Y=0$!

- Plusieurs raisons à cela :
- Une lecture (dans le cache d'un coeur/CPU) peut arriver avant que l'écriture d'une variable (dans le cache d'un autre coeur/CPU) ne soit **visible**.
 - Avant que les caches d'un coeur/CPU puissent être mis à jour, il faut que les modifications des caches d'un autre coeur/CPU soient répercutées en RAM puis dans le cache.
 - Le mécanisme d'écriture est appelé "**write-back**" et participe de la **cohérence** des caches
- Sur les CPU "*out-of-order*", les instructions peuvent être **réordonnées**
- Les instructions sont en fait découpées en **plusieurs opérations** (lecture, modification, écriture...)

Accès atomique

- En C++11, `std::atomic` nous permet d'effectuer des lectures et écritures sans avoir besoin de verrou.
 - Note: il faut toujours initialiser manuellement une variable atomique (même globale)
- Les types simples comme `std::atomic_int` etc... sont bas-niveau sur chaque plateforme (potentiellement plus rapide, car moins de contraintes)
 - En x86_64 cela se traduit par le préfixe `LOCK` pour les opérations et `XCHG` pour les écritures
 - En x86_64 les `MOV` sont toujours consistants car le modèle mémoire est fort
- De plus on a la garantie que les lectures et écritures ne sont pas réordonnancées
 - Par défaut, les variables atomiques en C++11 ont un ordre total, c'est-à-dire que l'on a la garantie d'une consistance des lectures et des écritures dans l'ordre d'exécution du programme.
 - Cette ordre mémoire peut-être modifié comme on le verra ultérieurement.

Ordre mémoire ?

- Par défaut les accès atomiques sont fortement synchronisés pour limiter les réordonnancements.
 - `_seq_cst = "sequentially consistent ordering"`
- Il est possible de customiser le fonctionnement des lectures et/ou écritures atomiques afin d'optimiser les accès mémoires (plus de détail prochainement).

```
enum memory_order {  
    memory_order_relaxed,  
    memory_order_consume,  
    memory_order_acquire,  
    memory_order_release,  
    memory_order_acq_rel,  
    memory_order_seq_cst    // Par défaut, le plus coûteux aussi  
};
```

Volatile, le retour

- J'appuie encore le fait que volatile est très différent d'atomic
 - Ne vous laissez pas bernier par les programmes et exemples Win32 qui utilisent volatile comme méthode de synchronisation
- La seule garantie offerte par volatile (hormis sous Visual C++) est que la lecture de la variable ne sera pas supprimée par l'optimisation.
- Rien n'empêche le compilateur de réordonnancer l'ordre des opérations sur des volatile, et on n'a aucune garantie d'atomicité.
 - Visual C++ utilise des fences (et peut être même des accesseurs atomiques) pour les volatile ce qui contredit le fonctionnement standard de ce mot clé.
- Quoiqu'il en soit, le rôle de volatile est de permettre l'accès à des zones mémoires spéciales mais n'a pas de rôle à jouer en programmation concurrente, contrairement aux atomics (et vice-versa)

variable atomique != opération atomique

- Prenons un cas simple

```
std::atomic<int> valeur{0};
```

```
valeur++;           // opération atomique
```

```
valeur += 42;       // atomique également
```

```
valeur.store(18);    // pareil
```

```
valeur = valeur + 24; // PAS atomique, mais atomique par partie
```

- La raison est ici que l'on n'a pas une seule opération mais plusieurs
 - Valeur.load() suivi de l'addition suivi de l'opérateur= qui appelle valeur.store()
- Du fait de l'ordre mémoire par défaut des load et store, la seule garantie qu'on a est que l'ordre des opérations load et store sera préservé.

$X=0$ ET $Y=0$ est maintenant impossible

CPU 0

atomic<int> A(0);

atomic<int> X(0);

A.**store**(42);

X = B.**load**();

CPU 1

atomic<int> B(0);

atomic<int> Y(0);

B.**store**(24);

Y = A.**load**();

Cas de plusieurs threads

- CPU0: A.store(42)
 - CPU1: B.store(24)
 - CPU2: X=A.load() Y=B.load()
 - CPU3: Z=B.load() W=A.load()
-
- L'ordre total (car consistant, par défaut) nous garanti ceci:
 - Le CPU2 lit forcément A en premier, de même que le CPU3 lit B d'abord
 - Si le CPU0 et/ou le CPU1 s'exécute(nt) en premier(s) les load verront toujours les stores.
 - Dès lors plusieurs cas sont possibles ici en fonction de l'ordre d'exécution de CPU0 et CPU1
 - Si le CPU2 s'exécute avant ou en même temps que le CPU3, mais après le CPU0
 - X et W ont forcément la valeur de A = 42
 - Y et Z ont forcément la valeur précédente de B = 0

Double-Checked locking pattern : la revanche

- Le problème réside essentiellement autour des tests de `m_instance`, qu'il suffit de rendre atomiques !

```
std::mutex g_mtx;
// on suppose std::atomic<Singleton*> m_instance;
Singleton* Singleton::getInstance() {
    Singleton* instance = m_instance.load();
    if (instance == NULL) {
        std::lock_guard<std::mutex> lock(g_mtx);
        instance = m_instance.load(); // force le reload !
        if (instance == NULL) {
            instance = new Singleton;
            m_instance.store(instance);
        }
    }
    return instance;
}
```

Généraliser l'application des accès atomiques

- Comme on peut le voir dans l'exemple précédent, il a été nécessaire de déclarer une variable locale pour stocker et modifier le pointeur vers l'instance avant de repercuter (commit) le changement.
- C'est une formule générique pour garantir l'atomicité des types complexes.
 - On effectue tous les calculs de manière privée (locale ici)
 - Puis on répercute les changements de façon atomique (store)

Exemple: job system

- Il est souvent utile de connaître le nombre de jobs ayant été traité par l'ensemble des workers / consommateurs
- Conceptuellement, un job system a moins de contrainte qu'un producteur-consommateur dans le sens où, pour le producteur (le boss ici), il n'est pas forcément nécessaire pour lui de gérer des notifications
 - Ce qui signifie que l'on peut facilement utiliser un modèle peer-to-peer où il n'y a que des workers thread
- C'est d'autant plus vrai qu'en théorie, push et pop sont des actions indépendantes travaillant sur des variables indépendantes
 - Le seul cas de figure embêtant est celui où il n'y a qu'une place disponible et que les thread sont tous en train de lire en même temps que l'écriture.
 - On a utilisé un verrou pour ce cas.
- On peut remplacer la notification par un simple compteur, ou plutôt deux compteurs:
 - Un simple compteur qui maintient le nombre total de jobs (pas besoin d'être reset)
 - Un compteur qui est incrémenté après chaque exécution d'un job – ce compteur **doit être atomique**
 - La différence nous indique la place disponible dans la queue, et donc aussi que le job system travaille (*is busy*)

Exemple: job system

- Comme le boss/scheduler ne produit pas les tâches mais les dispatch, on remplace l'attente de la notification par un polling direct

```
void poll() {  
    cv.notify_one();  
    this_thread::yield(); // évite la surcharge/blocage du système...ou alors faire autre chose  
}  
  
bool isBusy() { return finishedJobs.load() < currentJobs; } // lecture atomique de finishedJobs (worker fait finishedJobs++)  
  
void wait() { if (isBusy()) poll(); }  
  
void pushJob(Job job) {  
    while (queue.isFull()) { poll(); }  
    queue.pushJob(job);           // un mutex est nécessaire ici, sauf si vous utilisez une file FIFO atomique  
    cv.notify_one();  
    currentJobs++;  
}
```

Références

- <http://www.cplusplusconcurrencyinaction.com/>
- <https://docs.microsoft.com/en-us/archive/msdn-magazine/2005/august/concurrency-what-every-dev-must-know-about-multithreaded-apps>
- La notion de "section critique" ne doit pas être confondue avec le type CRITICAL_SECTION de l'API Win32.
 - Note: sous Windows les std::mutex sont plus probablement des CRITICAL_SECTION car les mutex Win32 sont assez lent
 - <https://docs.microsoft.com/en-us/archive/msdn-magazine/2003/december/break-free-of-code-deadlocks-in-critical-sections-under-windows>
- "Race condition vs data race" <https://blog.regehr.org/archives/490>
- <https://www.justsoftwaresolutions.co.uk/threading/locks-mutexes-semaphores.html>
- <https://stackoverflow.com/questions/12033188/how-would-you-implement-your-own-reader-writer-lock-in-c11> (voir la toute dernière réponse et le lien wikipedia)
- <https://preshing.com/20150316/semaphores-are-surprisingly-versatile/>
- Le diner des philosophes version sushis vaut le détour https://perso.ens-lyon.fr/michael.rao/ASR2/cours_slides_8.pdf
- Un cours de programmation système bien sympathique (en complément) <https://sites.uclouvain.be/SystInfo/distrib/SINF1252-Theorie.pdf> (les chapitres 4, mais aussi 3 et 5 sont les plus pertinents pour nous)
- <https://bartoszmilewski.com/2008/08/04/multicores-and-publication-safety/>