Table des matières

Atomicité – Survol de ce document

Rôle de ce volume

Ce document vise à présenter l'atomicité telle qu'elle s'exprime depuis C++ 11 et, dans une moindre mesure, avec C 11. L'essentiel du propos que vous y trouverez est synthétisé de textes et de présentations d'**Anthony Williams** et de **Herb Sutter**, avec des idées prises dans des écrits de **Scott Meyers**, **Hans-J. Boehm**, **Lawrence Crowl**, **Danny Kalev**, **Jeff Phreshing** et **Bartosz Milewski**. Le tout est complété par un ensemble d'expérimentations personnelles.

Merci à **Alex Boulanger**, étudiant à l'Université de Sherbrooke, et à **Adam Galarneau**, ex-étudiant du Collège Lionel-Groulx, pour leur relecture rigoureuse.

Le document que vous lisez présentement n'aurait pas eu raison d'être il y a une quinzaine d'années à peine, du moins au moment d'écrire ces lignes.

En effet, les problèmes adressés par l'atomicité telle que décrite dans la présente n'existent que sur des ordinateurs supportant la multiprogrammation sur le plan matériel, et à plus forte partie quand la multiprogrammation repose sur l'exécution concurrente de *threads* d'un même programme sur plusieurs cœurs ou plusieurs processeurs.

Bonne lecture!

Raisonner sur le code

Supposons le programme (incomplet) proposé à droite.

Il semble simple de prime abord, mais si les fonctions `compterA()` et `compterB()` sont exécutées en parallèle, plusieurs questions viennent à l'esprit.

Parmi celles-ci : que se produira-t-il si `++nAB` est fait essentiellement simultanément par `compterA()` et `compterB()`?

Aussi, si nous évaluons `nAB` une fois `compterA()` et `compterB()` arrêtées, a-t-on alors la certitude que `nAB==(nA+nB)`?

Si ces questions vous semblent avoir une réponse évidente, et si cette réponse est « rien de simultané » dans le premier cas et « bien sûr » dans le deuxième, c'est que votre lecture du code rejoint celle – traditionnelle – des programmes monoprogrammés, et que vous supposez une mémoire qu'on pourrait qualifier de « séquentiellement cohérente » : les événements s'y passent dans l'ordre décrit par le code source. Si votre vision des opérations est que les fonctions `compterA()` et `compterB()` s'exécutent en parallèle, vous supposez probablement que `++` appliqué à un `int` tel que `nA` ou `nB` est une opération indivise, *atomique*.

Ce sont des suppositions raisonnables. Cela dit, elles sont erronées.

```
bool arreter = false;
int nA = 0;
int nB = 0;
int nAB = 0;
int evenementA;
int evenementB;
// ...
void compterA()
{
   while (!arreter)
      if (evenementA > nA)
      {
         ++nA;
         if (evenementB) ++nAB;
      }
}
void compterB()
{
   while (!arreter)
      if (evenementB > nB)
      {
         ++nB;
         if (evenement_) ++nAB;
      }
}
```

Les choses vraiment simples sont rares…

Une opération aussi simple que `++i` sur un `i` de type `int` n'est pas, au sens de la machine, une opération indivise. En pratique, cette opération se décompose en au moins trois étapes :

- charger dans un registre la valeur à l'adresse de `i`;
- incrémenter la valeur contenue dans ce registre;
- écrire à l'adresse de `i` la valeur contenue dans le registre.

Exprimé en pseudo-assembleur, où `Rx` est un registre et où `[i]` est l'adresse de `i`, on peut donc imaginer ce qui suit :

```
LOAD Rx,[i]
INC Rx
STOR [i],Rx
```

Supposons que les *threads* `A` et `B` exécutent en parallèle les fonctions `compterA()` et `compterB()`. Alors, une situation telle que celle visible ci-dessous est possible :

Tic	*Thread* **A**	*Thread* **B**
n	**LOAD Rx,[i]**	**...**
n+1	**INC Rx**	**LOAD Rx,[i]**
n+2	**STOR[i],Rx**	**INC Rx**
n+3	**...**	**STOR [i],Rx**

Dans un tel cas, la valeur de `i` perçue dans le *thread* `A` au moment de son `LOAD` et la valeur de `i` perçue dans le *thread* `B` au moment de son `LOAD` sont identiques, du fait que l'écriture de `Rx` dans `i` du côté `A` n'a pas été effectuée au moment où `B` charge la valeur de `i`.

Ce problème est fondamental et naturel dans un milieu concurrent. Qui plus est, pour maximiser le parallélisme, nous souhaitons faire en sorte que les *threads* soient les plus indépendants les uns des autres en pratique – ici, si `A` opérait sur une variable `i` et si `B` opérait quant à lui sur une autre variable `j`, cette action concurrente serait extrêmement souhaitable. En contrepartie, obtenir ces gains implique de notre part une plus grande prudence dans l'écriture du code.

Raisonner sur le code source

Nous écrivons des programmes qui doivent accomplir des tâches précises. Le texte que nous écrivons est destiné au compilateur, qui le lira et le transformera de manière à générer un programme qui aura, du moins c'est notre souhait, le même comportement manifeste à l'exécution que ce que nous pouvions prédire à partir du code source.

Lorsque nous rencontrons des « surprises » ou des bogues à l'exécution, nous revenons au texte du programme, au code source, pour raisonner sur celui-ci et manière à comprendre ce qui se passe, à faire disparaître les « surprises » et les bogues.

La capacité de raisonner sur le code source est fondamentale à l'acte de programmer, d'analyser un programme, de le déboguer. Nous reviendrons sur la question du raisonnement sur des sources multiprogrammées dans ***Exemples de raisonnement causal***, plus loin.

Prérequis pour raisonner sur le code

Nous écrivons des programmes que nous pensons corrects. Pour ce faire, nous faisons un certain nombre de suppositions, dont une en particulier est importante, implicite et dépend du matériel, soit celle selon laquelle les antémémoires (les *Caches*) de l'ordinateur sont cohérentes, donc qu'il est possible d'avoir un regard cohérent sur le contenu de la mémoire de l'ordinateur.

Du point de vue d'un programme C++, une exigence clé s'ajoute à cette précondition : il ne faut pas avoir de bogue logiciel. Cela signifie ne pas avoir de conditions de course, du moins celles de la famille que le standard de C++ nomme les *Data Races* (nous y reviendrons), et éviter d'utiliser des opérations atomiques qui seraient *relaxed* (nous y reviendrons aussi).

Si ces conditions sont respectées, la présomption de cohérence globale interne demeurera : en l'absence de pannes matérielles et de bogues de compilateurs, les opérations de plusieurs *threads* sur plusieurs cœurs seront intercalées mais, l'ordonnancement observable global sera le même pour chacun d'eux – exprimé autrement, pour tous les *threads*, il semblera y avoir eu une et une seule exécution du programme.

Transformation du code source

On serait tenté de croire qu'une simple écriture prudente des sources suffirait à elle seule à régler des problèmes tels que celui décrit ci-dessus. Malheureusement, le problème ne se limite pas au langage de programmation, et résulte en partie d'opérations souhaitables, réalisées par le compilateur comme par le matériel : les optimisations.

En effet, **le code exécuté par un ordinateur n'est pas, en pratique, le code rédigé par le programmeur**. C'est plutôt un équivalent manifeste, du point de vue des effets observables, du code source, mais amélioré et raffiné par le compilateur, puis amélioré et raffiné par le processeur. Il faut que le code s'exécute *comme si* c'était celui que vous avez écrit.

Dans du code monoprogrammé, le processeur et le compilateur ont énormément de latitude quant aux transformations admissibles sur le code source : garder la valeur d'une variable dans un registre pour optimiser les accès à un compteur dans une boucle; éliminer du code inutile; produire des écritures spéculatives quand, face à une alternative (un « `if` »), il semble hautement probable qu'une des deux branches possibles soit prise et l'autre pas; etc.

Cela peut surprendre, mais il se trouve que nous *souhaitons* que le processeur et le compilateur aient cette latitude. Le processeur sait mieux que nous l'état de ses *Cache Lines*, et est bien mieux placé que nous pour constater l'impact de certains choix sur l'état actuel de son pipeline de micro instructions. Le compilateur a pour sa part plusieurs responsabilités, incluant celles d'optimiser l'allocation des registres et de séquences les instructions de manière à ce qu'elles tirent au maximum profit du matériel; les programmes sont en moyenne beaucoup plus rapides lorsqu'il est libre d'exercer ses responsabilités[1].

[1] Il est bien sûr possible de faire mieux que le compilateur en procédant manuellement dans des cas particuliers, mais il est improductif d'essayer de faire mieux que le compilateur de manière générale.

Dans du code multiprogrammé, cependant, le compilateur et le processeur doivent être contraints quant à certaines optimisations qui, si elles sont acceptables (souhaitables, même!) dans une situation monoprogrammée, deviennent dangereuse dans du code mutliprogrammé.

Pensons par exemple à quelque chose comme ce qui suit :

```
x = 3;
y = 4;
if (x != f())
   // ...
```

Si `x` et `y` sont des variables locales, il est correct pour un compilateur de réordonnancer les écritures pour que celle sur `x` suive celle sur `y`, ce qui peut lui permettre de conserver la valeur de `x` dans un registre suite à l'affectation pour accélérer le code de comparaison de `x` avec `f()`. Si `x` et `y` sont des variables accessibles ailleurs que dans le contexte local, par contre, et à plus forte partie si `x` et `y` sont des variables accédées concurremment par aux moins deux *threads*, il est possible que la logique d'un autre *thread* dépende de l'ordre de ces deux écritures, et qu'un réordonnancement *ici* brise le code *là*.

Avec cet exemple, il y a une différence fondamentale (pas seulement cosmétique) entre le recours à des variables `x` et `y` qui ne seraient visibles que localement à la fonction, et utiliser des variables `x` et `y` qui seraient globales ou simplement visibles de plus d'un *thread*.

Retenez que le principe de localité [hdPrLoc] n'est pas qu'une question d'hygiène; c'est souvent une optimisation.

Optimisations interdites

Sans surprises, la plupart des transformations qu'apporte un compilateur (ou un processeur, ou…) à du code source se veulent des optimisations.

La règle clé pour les optimisations est de procéder *as if* : si le compilateur peut démontrer que, suite à une transformation, le programme se comportera de la même manière (*au sens des états observables*) que si la transformation n'avait pas été appliquée, alors cette transformation est légale. Cela dit, un compilateur en sait moins sur le programme quand il est multiprogrammé que lorsqu'il est monoprogrammé, ce qui explique que nous devions lui prêter secours.

Parmi les transformations apportées à notre code source par le compilateur, les processeurs, les antémémoires, etc. on trouve :

- des écritures spéculatives, par exemple optimiser une alternative pour le cas `true` ou pour le cas `false` en fonction de ce qui est jugé le plus probable;
- provoquer l'exécution optimiste d'une instruction, quitte à en inverser les effets ultérieurement si cette exécution était… trop optimiste;
- déposer une variable dans un registre pour la durée d'une répétitive, en vue d'éliminer des accès inutiles à la mémoire; etc.

Certaines de ces optimisations sont interdites sur des états visibles d'au moins deux *threads*. Elles ne sont pas interdites en tout temps, bien sûr, et demeurent permises dans la mesure où l'optimisation tient compte de détails comme « est-ce que la variable a changé d'état en cours de route? », utilisant par exemple un *Dirty Bit*.

Évidemment, le compilateur peut s'en donner à cœur joie pour tous les états non-partagés; si le compilateur ne peut pas prouver qu'un état est non-partagé, alors le compilateur doit se limiter à des optimisations qui sont conservatrices.

Répétons-le, parce que ça en vaut la peine : faire en sorte que le code opère essentiellement sur des variables locales n'est pas qu'un détail esthétique; il s'agit d'une optimisation!

Écriture spéculative

La plus évidente est l'écriture spéculative : le compilateur/ le processeur ne doit jamais inventer une écriture sur une variable (partagée) dans laquelle le programme n'aurait pas écrit dans une exécution séquentiellement cohérente, faute de savoir quels mécanismes de synchronisation utiliser pour mener cette opération à bien.

Formellement, une optimisation reposant sur une écriture spéculative est illégale si cette écriture est observable d'un autre *thread* que celui qui la réalise.

Ne pas publier des transitions d'états

Une autre optimisation interdite est d'utiliser un registre sans écrire dans une variable pour un compteur. Cette optimisation est illégale car elle peut introduire un changement d'état hors de la protection d'un outil de synchronisation, à la fin de la répétitive.

Ainsi, réaliser une transformation de ceci :

```
auto somme = vector<T>::value_type{};
for(vector<T>::size_type i= 0; i < v.size(); ++i)
   somme += v[i];
```

à cela :

```
r1 = vector<T>::value_type{};
for(vector<T>::size_type i= 0; i < v.size(); ++i)
   r1 += v[i];
somme = r1
```

où `r1` est un registre, est une optimisation illégale car l'écriture dans `somme` est un ajout qui peut briser la cohérence séquentielle : dans le code original, si `v.empty()`, on n'écrivait pas dans `somme` après la boucle.

Cohérence séquentielle

Ce qu'on cherche à obtenir est une **cohérence séquentielle**, soit *l'impression que nous exécutons le programme que nous avons écrit* – cette définition informelle est inspirée de celle de **Leslie Lamport** dans [LampClock].

Concrètement, lorsque nous raisonnons sur notre code, il nous faut le faire en conceptualisant que le compilateur, à toutes fins pratiques, compile chaque *thread* isolément, sans compréhension globale de l'exécution du système qui résultera de la compilation dans son ensemble. D'une part, le compilateur peut voir des *aliasing* qui nous échappent, donc des relations entre des variables qui nous auraient échappé; d'autre part, le compilateur ne sait pas quels espaces en mémoire sont partagés entre deux *threads* ou plus, et ne sait généralement pas non plus comment nous comptons synchroniser les relations entre les accès à ces espaces-mémoires.

Pour maintenir la relation de causalité dans nos programmes, l'approche vers laquelle ont convergé les divers modèles mémoires des langages de programmation les plus importants aujourd'hui est celle de la cohérence séquentielle. Java a choisi cette voie depuis 2005, et C++ 11 fait de même par défaut : depuis C++ 11, par défaut, un programme est séquentiellement cohérent dans la mesure il ne contient aucune *Data Race*.

Ce qui fait qu'un programme C++ 11 exempt de *Data Race* est séquentiellement cohérent *par défaut* est que les opérations atomiques se font, par défaut, de manière à imposer un ordonnancement séquentiellement cohérent. Il est toutefois possible pour un programme de « relaxer » cette contrainte d'ordonnancement, pour fins d'optimisation – surtout sur des architectures matérielles où la lecture d'une variable atomique est lente.

Briser la cohérence séquentielle d'un programme est périlleux. Ne le faites pas à moins que ce ne soit absolument nécessaire, chiffres à l'appui (et encore, hésitez!).

Il semble que le logiciel ait pris un peu d'avance sur le matériel pour ce qui est des modèles mémoire, mais que le matériel tende lui aussi à converger vers des opérations atomiques qui privilégient la cohérence séquentielle par ordonnancement des lectures et des écritures sur les états partagés. En ce sens, le recours aux opérations « relaxées », en désespoir de cause, peuvent être vues comme des étapes transitoires.

Nous utiliserons l'acronyme SC-DRF pour *Sequentially Consistent if Data Race-Free*.

Modèle mémoire

Le modèle mémoire d'un langage décrit entre autres choses les règles balisant les transformations admissibles pour un compilateur. Java a un modèle mémoire [JavaMemMod] formellement établi depuis 2005, alors que C++ 11 et C 11 introduisent tous deux un modèle mémoire formel propre à ces langages. L'idée d'un modèle mémoire pour un langage donné n'est réellement pertinente que quand ce langage supporte la multiprogrammation de manière portable, ce qui explique que C++ et C n'en aient pas véritablement eu un tant que les détails de multiprogrammation de ces langages dépendaient strictement de la plateforme.

Un modèle mémoire définit ce que constitue une condition de course (*Race Condition*, en particulier les *Data Races* dans le cas de C++), qui sont informellement les endroits dans un programme où les résultats de l'exécution dépendent des circonstances plutôt que du code source, et déterminent les règles quant au comportement des programmes en présence ou en l'absence de ces très vilaines situations.

Avec C++ 11, une *Data Race* se définit comme de la manière indiquée à **Erreur ! Source du renvoi introuvable.** [ISOCppStd], *Erreur ! Source du renvoi introuvable.* : « *The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior* ».

Notons que ceci signifie qu'au moins une des actions concurrentes soit une écriture – si plusieurs lectures concurrentes se produisent sur une même donnée, aucune action conflictuelle n'en résulte. Conséquemment, si une même donnée est accédée concurremment par au moins deux *threads*, et si au moins un des deux accès est en écriture, vous avez une *Data Race* et votre programme tombe dans le fossé du très vilain *Undefined Behavior* [hdUB].

Cela signifie que même un programme comme celui proposé à droite a un comportement indéfini, du fait que la variable `fini` est accédée par deux *threads* (incluant celui exécutant `main()`), au moins l'un d'eux en écriture, et qu'on n'y trouve pas de synchronisation.

Deux solutions simples existent ici : protéger `fini` par un `mutex`, mais c'est fastidieux et il faut protéger chaque accès à la variable, ou simplement faire de fini un `atomic<bool>`. C'est, ici, probablement la meilleure solution.

Il existe plusieurs types de conditions de course qui peuvent compliquer (ou rendre impossible) la déduction des résultats possibles d'exécution d'un programme, mais les *Data Races* sont celles qui entraînent du *Undefined Behavior* avec C++.

```
#include <thread>
#include <iostream>
#include <chrono>
int main()
{
   using namespace std;
   using namespace std::chrono;
   bool fini = false;
   thread th([&fini]() {
      char c;
      cin >> c;
      fini = true;
   });
   while (!fini) {
      cout << '.' << flush;
      this_thread::sleep_for(seconds{1});
   }
   th.join();
}
```

Le modèle mémoire de C++ se veut philosophiquement conforme aux usages de ce langage. L'idée maîtresse derrière ce modèle mémoire est la suivante :

- par défaut, si votre programme C++ n'a pas de *Data Race*, alors le comportement observable du programme, transformations incluses, correspondra à ce que décrit votre code source. C'est ce qu'on nomme `SC-DRF` : *Sequentially Consistent if Data Race-Free*;
- en retour, si votre programme C++ contient des *Data Races*, alors il tombe dans le monde merveilleux du comportement indéterminé (le vilain *Undefined Behavior* [hdUB]) et est inutilisable en pratique, du moins si vous souhaitez un comportement qui soit portable.

Pour être en mesure de raisonner sur le code source, il faut donc en arriver à un programme qui ne contienne pas de conditions de course, en particulier par de *Data Races*.

Un programme `SC-DRF` est tel que les écritures sur les états partagés entre divers *threads* semblent, *a posteriori*, avoir été faits dans le même ordre pour chacun des *thread* les observant. Un programme `SC-DRF` peut *a priori* raconter plusieurs histoires, mais n'en aura raconté qu'une seule *a posteriori*.

Transformations des sources et respect du modèle mémoire

Pour résumer, il importe que le code source soit optimisé et transformé à divers niveaux, mais il importe du même coup que les optimisations soient contraintes, à la fois par les règles de synchronisation imposées par les programmeuses et par les programmeurs dans le code source et par les règles imposées par le modèle mémoire dans l'optique d'assurer une cohérence séquentielle du programme dans son ensemble.

Puisqu'on ne sait pas *a priori* où et quand les transformations sur le code sont faites, le défi commun au processeur, au compilateur et aux programmeuses et programmeurs est de faire en sorte que le programme *semble* séquentiellement cohérent *malgré* ces transformations. En particulier, il importe de faire en sorte que les écritures *semblent* se produire de manière atomique et simultanément pour tous les *threads*, dans l'optique d'assurer une acception cohérente par tous les *threads* d'une exécution du programme – il faut bien sûr retenir qu'il existe plusieurs exécutions séquentiellement cohérentes possibles pour un même programme parallèle.

Garantir la cohérence séquentielle n'est pas une mince affaire. En C++, lorsque l'optimiseur est très agressif sur un programme séquentiel, il est par exemple possible de placer deux entiers sur `16` bits côte à côte en mémoire si les accès à l'un n'empiètent pas sur les accès à l'autre, de manière à opérer sur des blocs de `32` bits en mémoire et à épargner du temps d'exécution.

Une telle manœuvre est illégale sur une variable visible du point de vue d'au moins deux *threads* dans un programme multiprogrammé, car elle implique l'insertion d'écritures silencieuses sur l'une ou l'autre des deux variables, ce qui est susceptible d'introduire une condition de course.

Cohérence à partir des lectures et des écritures

Le raisonnement menant à des transformations respectant la cohérence séquentielle d'un programme peut se faire en forçant des clôtures mémoire à plusieurs endroits, mais cette approche est très coûteuse.

Heureusement, un raisonnement analogue peut se faire à partir des lectures et écritures dans le code source : par exemple, dans un programme donné, si A précède B dans le code source et si A est une écriture dans une variable x alors que B est une lecture de la variable x, il importe qu'une réorganisation du code ne déplace pas B avant A, du moins si l'on souhaite qu'il demeure possible de raisonner sur le programme à partir de son texte. Une transformation ne respectant pas cette contrainte briserait, concrètement, la relation de causalité établie dans le code à l'origine entre les opérations A et B.

Dans certaines architectures matérielles, cette relation de *Data Dependency* est la seule que respecte le processeur lors de ses réordonnancements. Dans de tels cas, assurer la cohérence séquentielle d'un programme requiert beaucoup de travail.

Exemple – l'algorithme de Dekker

Un exemple classique d'algorithme qui serait brisé par une telle compromission de causalité serait l'algorithme de Dekker, qui permet à deux *threads* de s'entendre sur l'accès à une section critique (formulation ci-dessous inspirée du Wiki sur le sujet) :

État initial

```
fanion[0] ← faux
fanion[1] ← faux
tour ← 0
```

L'algorithme original s'exprime en termes de processus partageant des états, ce qui est plus proche de ce que l'on entend par des *threads* aujourd'hui.

Thread 0

```
fanion[0] ← vrai
tant que fanion[1]
   si tour ≠ 0
      fanion[0] ← faux
      tant que tour ≠ 0
         // attente active
      fanion[0] ← vrai
// section critique (code selon l'application)
tour ← 1
fanion[0] ← faux
// code hors de la section critique
```

Thread 1

```
fanion[1] ← vrai
tant que fanion[0]
   si tour ≠ 1
      fanion[1] ← faux
      tant que tour ≠ 1
         // attente active
      fanion[1] ← vrai
// section critique (code selon l'application)
tour ← 0
fanion[1] ← faux
// code hors de la section critique
```

Dans cet algorithme, les écritures aux variables `tour`, `fanion[0]` et `fanion[1]` sont indépendantes les unes des autres au sens du compilateur. De plus, le *thread* `i` écrit dans `fanion[i]` mais n'y lit jamais, ce qui peut laisser entendre au compilateur que cette écriture est redondante et l'inciter à simplifier le code en supprimant ces écritures « inutiles ».

Une option – dispendieuse – pour éviter ces optimisations est, dans les langages C et C++ du moins, de qualifier cette variable `volatile`. Mais une variable `volatile` en C et en C++ ne fait que supprimer certaines optimisations qui seraient invalides dû à des causes externes; ce mécanisme a été pensé pour l'écriture de pilotes, par exemple, donc pour des programmes où des événements physiques, externes au texte du programme, affecteraient les conditions d'exécution. *Ce mécanisme ne règle pas le problème d'une potentielle réorganisation des instructions*.

Exemple – l'algorithme de Peterson

Cela dit, examinons maintenant une variante de l'algorithme de Dekker, vouée à la même tâche que ce dernier, c'est-à-dire l'algorithme de Peterson.

État initial

```
fanion[0] ← faux
fanion[1] ← faux
tour ← 0
```

L'algorithme original s'exprime en termes de processus partageant des états, ce qui est plus proche de ce que l'on entend par des *threads* aujourd'hui.

Thread 0

```
fanion[0] ← vrai
tour ← 1
tant que fanion[1] et tour = 1
   // attente active
// section critique (code selon l'application)
fanion[0] ← faux
// code hors de la section critique
```

Thread 1

```
fanion[1] ← vrai
tour ← 0
tant que fanion[0] et tour = 0
   // attente active
// section critique (code selon l'application)
fanion[1] ← faux
// code hors de la section critique
```

Dans cet algorithme, l'attente active est fortement à risque lors d'un réordonnancement, et les optimisations que pourraient entraîner les transformations réalisées par le compilateur ou par le processeur sont susceptibles de rendre l'algorithme inopérant ou incorrect.

En fait, le *Store Buffer* du processeur, qui permet de mettre en *Cache* des écritures est suffisant pour causer des problèmes, même si ni le compilateur, ni le processeur ne réordonnancent le code.

Du fait qu'une écriture est plus lente qu'une lecture, les processeurs contemporains disposent d'un *Store Buffer*, sorte de *Cache* de première ligne qui accumule les écritures à court terme pour les réaliser par blocs

Ce qui importe pour que cet algorithme fonctionne est que, dans les transformations appliquées sur le code :

- les écritures aux états partagés ne soient pas supprimées;
- aucune écriture ne soit ajoutée (par le compilateur, typiquement) sur un état partagé; et
- l'ordre de deux écritures sur des états partagés distincts ne soient pas permuté; et
- l'ordre des lectures et les écritures sur un même état partagé ne soient permutés.

Détecter les possibles conditions de course – Exemples concrets

Quelques exemples concrets de programmes pour lesquels il serait légitime de se demander s'ils contiennent une condition ce course.

Exemple 0 – Scalaires distincts

Présumant que `c0` et `c1` soient deux variables distinctes de type `char`, et que `ma` et `mb` soient deux instances de la classe `mutex`. Est-ce que ceci introduit une condition de course?

Thread A	**Thread B**

`{`
`  lock_guard<mutex> _{ma};`
`  c0 = 'a';`
`}`
	`{`
	`  lock_guard<mutex> _{mb};`
	`  c1 = 'a';`
	`}`

Réponse : non, du moins dans la mesure où le compilateur implémente le standard correctement. Les deux variables se trouvent dans des zones mémoire distinctes et le code généré devrait en tenir compte et les considérer comme des objets distincts.

Ici, une condition de course pourrait survenir si le processeur opérait sur des blocs de `32` bits ou de `64` bits à la fois, et si `c0` et `c1` se trouvaient dans suffisamment près l'un de l'autre, mais un compilateur procédant ainsi introduirait une condition de course dans un code source qui n'en contient pas de prime abord, ce qui est interdit par le modèle mémoire de C++ `11`.

Exemple 1 – Membres d'un même agrégat

Reprenons l'exemple ci-dessus, mais supposons que `c0` et `c1` soient deux membres d'un même enregistrement (d'un même `struct`). A-t-on alors introduit une condition de course?

Réponse : non, ici non plus. Les raisons motivant cette réponse sont les mêmes que celles évoquées ci-dessus.

Exemple 2 – Membres d'un même bitfield

Reprenons encore le même exemple, mais supposons maintenant que le `struct` contienne un *bitfield*, p. ex. : `struct X { int c0 : 9; int c1 : 7; };`. A-t-on alors introduit une condition de course?

Réponse : oui, du moins si on utilise des `mutex` différents pour sécuriser les accès à `c0` et à `c1`. Ceci tient à la différence entre une variable dans le langage et le concept d'« objet » au sens de zone mémoire accessible par les instructions matérielles. Les bits de `c0` et de `c1` sont définis dans le programme comme chevauchant des objets distincts, et la faute retombe sur les épaules des programmeuses et des programmeurs.

Le problème ici est qu'il n'est pas possible, sur le plan matériel, d'adresser un bit à la fois (ce serait beaucoup trop lourd); il faut accéder au moins un *byte* à la fois. Conséquemment, deux champs adjacents dans un *bitfield* peuvent être en fait un même « objet », une même zone mémoire, au sens du langage.

Exemple 3 – Tests successifs d'une même condition

L'exemple (en pseudo C++) ci-dessous est-il sécuritaire si `cond` est cohérente, au sens où elle ne change pas en cours de route?

```
if(cond) lock x;
// ...
if(cond) use x;
// ...
if(cond) unlock x;
```

Réponse : oui, en accord avec le modèle mémoire. Cela dit, il se peut que des compilateurs ne génèrent pas du code correct aujourd'hui.

Introduction aux opérations atomiques

Entrons maintenant un peu plus dans le détail des opérations atomiques et de leur fonctionnement, pour comprendre ce qu'elles font et ce qu'elles ne font pas.

Une opération atomique, au minimum, s'exécute sans interférence externe. Contrairement à l'exemple initial de la section ***Raisonner sur le code***, plus haut, remplacer un compteur de type `int` par un compteur `atomic<int>` fait en sorte que si le compteur est autoincrémenté en concurremment par deux *threads* ou plus, chaque autoincrémentation (chaque `++i`) se fasse entièrement, sans que les opérations machines des différents *threads* ne se chevauchent.

Cela ne suffirait pas pour permettre d'atteindre un idéal de cohérence séquentielle. Plus encore, **par défaut, une opération atomique ne peut être réordonnancée par le compilateur ou par le processeur, du moins par d'une manière qui contreviendrait aux règles du modèle mémoire** (voir ***Modèle mémoire***).

C++ étant un langage qui met l'accent sur la « performance », il est possible pour une programmeuse ou pour un programmeur de « relaxer » les règles quant aux réordonnancements des opérations atomiques, du fait que des règles plus souples que les règles par défaut peuvent parfois suffire à assurer le bon fonctionnement d'un programme. Nous examinerons brièvement la question des atomiques relaxées dans la section ***Les atomiques « relaxées »***, plus loin.

Cela dit, à moins d'avoir un besoin clair et s'appuyant sur des métriques, essayez de ne pas trop jouer avec les atomiques « relaxées »; c'est un exercice pour le moins périlleux.

Opérations de sémantique Acquire *et* Release

Le concept clé derrière la démarche qui sous-tend les opérations atomiques est celle de transaction, au sens où un programme, même transformé, doit éviter d'exposer des états incohérents. Ceci s'exprime principalement en termes d'opérations de sémantique *Acquire*, incluant – sans s'y limiter – les lectures (opérations *Load*), et d'opérations de sémantique *Release*, incluant – sans s'y limiter – les écritures (opérations *Store*).

L'ordonnancement et la synchronisation du code peut typiquement se faire à l'aide de clôtures mémoires (voir ***Clôtures mémoires***), de sections critiques, de mutex, d'atomiques ordonnancées ou de mémoire transactionnelle [hdTrMem][2]. Ces outils ne sont pas mutuellement exclusifs; par exemple, il est possible d'écrire le même code avec une atomique et avec un mutex, car une variable atomique permet d'écrire un *Spin Lock*, très efficace si la contention est faible :

```
atomic<int> a_qui_le_tour;
// ...
while (a_qui_le_tour != moi)
   ; // spin lock!
// work
a_qui_le_tour = prochain(a_qui_le_tour);
```

Que ce soit par une atomique ou par un mutex, l'obtention d'un outil de synchronisation est une opération de type *Acquire*, alors que la libération de cet outil est une opération de type *Release*. Il importe qu'aucun réordonnancement ne soit fait autour de ces deux opérations, car cela aurait pour conséquence d'injecter une condition de course dans le programme.

Il est tout autant interdit de permuter les opérations en périphérie de ces deux opérations :

Le compilateur n'a pas le droit de transformer ceci…	…en cela
Opération a	Opération c
ACQUIRE	ACQUIRE
Opération b	Opération b
RELEASE	RELEASE
Opération c	Opération a

Il n'est pas non plus légal de sortir de code d'une région *Acquire*…*Release*. Dans le code ci-dessus, `Opération b` ne peut être déplacée avant `ACQUIRE`, pas plus qu'elle ne peut être déplacée après `RELEASE`.

[2] La mémoire transactionnelle n'est pas un mécanisme de C++ en date de C++ 14, mais au moment d'écrire ceci, il semble probable que C++ 17 le supporte.

Sémantique d'une opération de type Release

Une opération de type *Release* doit être vue comme la publication de tout ce qu'un *thread* donné a fait jusqu'à ce point. Pour cette raison :

- un réordonnancement qui bougerait une instruction avant une opération *Acquire* serait illégal;
- un réordonnancement qui bougerait une instruction après une opération *Release* serait illégal;
- on ne peut pas permuter une opération *Acquire* avec une opération *Release*, même si une opération *Release* suit immédiatement une opération *Acquire* – une telle transformation pourrait provoquer un interblocage.

Il est possible, à titre de transformation, d'insérer du code dans une région *Acquire*…*Release*. Ceci peut ralentir le code, mais ne peut pas causer de bogues dans la mesure où on ne parle pas d'écriture sur des variables partagées – prudence toutefois dans les systèmes en temps réel! Ceci permet entre autres d'insérer des instructions spéciales pour forcer le matériel à collaborer avec les mécanismes de synchronisation de niveau « langage ».

Une opération *Store-Release* d'une donnée publie les accès préalables sur cette donnée à un *thread* qui réalise un *Load-Acquire* sur la même donnée. Ces opérations vont en quelque sorte de pair, pour assurer la cohérence séquentielle du programme. Il devient donc possible d'exprimer les relations entre *threads* sur une variable donnée en termes de clôtures partielles, *Acquire* ou *Release*, plutôt qu'en termes de clôtures pleines et entières (beaucoup plus lourdes), outre peut-être quelques cas pathologiques comme un *Atomic Swap*.

Aidez-vous et évitez des variables qui seraient mal alignées en mémoire, ou qui chevaucheraient deux *Cache Lines*, pour éviter des mises à jour partielles de vos données.

Exemples de raisonnement causal sur des atomiques

Deux exemples suivent, tous deux fortement inspirés de [SuttAtom1]. L'un porte que la transitivité, le raisonnement causal et les atomiques, alors que l'autre porte sur l'ordonnancement total des écritures dans un programme séquentiellement cohérent.

Transitivité et causalité

Soit le programme à trois *threads* suivant.

État initial

```
g = 0;
x = 0;
y = 0;
```

Pour cet exemple, il importe de considérer `g`, `x` et `y` comme des variables atomiques (des `std::atomic<int>` en notation C++ `11`), dans leur acception par défaut, soit celle offrant la garantie `SC-DRF`.

***Thread* A**	***Thread* B**	***Thread* C**
`g = 1;` `x = 1;`	`if(x == 1)` `  y = 1;`	`if(y == 1)` `  assert(g == 1);`

Ici, dans le *thread* `C`, l'assertion sera nécessairement vérifiée car si `y` vaut `1`, c'est que *thread* `B` y a écrit; si *thread* `B` a déposé `1` dans `y`, c'est que `x` valait `1` selon lui. Enfin, si `x` vaut `1`, c'est que *thread* `A` avait d'ores et déjà déposé `1` dans `g`.

Cette relation causale entre *threads* est rendue possible par les règles du modèle mémoire (voir ***Modèle mémoire***), qui permettent en quelque sorte de raconter une histoire cohérente de l'exécution du programme. Notez que plusieurs histoires sont possibles ici : par exemple, si *thread* `B` s'est exécuté avant *thread* `A`, alors la condition `x==1` ne se sera pas avérée, et `y` ne sera jamais devenue `1`; d'autres exécutions séquentiellement cohérentes sont aussi possibles pour le même programme.

La cohérence séquentielle garantit que tous les *threads* constateront une histoire d'exécution cohérente, mais ne dit généralement pas *a priori* quelle sera cette histoire.

Ordonnancement total des écritures

Soit le programme à quatre *threads* suivant.

État initial

```
x = 0;
y = 0;
```

Pour cet exemple, il importe de considérer `g`, `x` et `y` comme des variables atomiques (des `std::atomic<int>` en notation C++ `11`), dans leur acception par défaut, soit celle offrant la garantie `SC-DRF`.

***Thread* A**	***Thread* B**	***Thread* C**	***Thread* D**

`x = 1;`	`y = 1;`	`if(x == 1 && y == 0)`
		`  cout << "x d'abord"`
		`  << endl;`
			`if(x == 0 && y == 1)`
			`  cout << "y d'abord"`
			`  << endl;`

La cohérence séquentielle fait en sorte que, dans un cas comme celui-ci, il est possible de voir s'afficher `"x d'abord"` comme il est possible de voir s'afficher `"y d'abord"`. Il est aussi possible de ne voir s'afficher aucun des deux messages, dans le cas où les *threads* `A` et `B` ont tous deux agi entièrement avant (ou après) les *threads* `C` et `D`.

Il est toutefois impossible de voir afficher les deux messages, car cela impliquerait une histoire incohérente de l'exécution du logiciel, à savoir que le *thread* `C` aurait constaté l'écriture faite par le *thread* `A` avant celle faite par le *thread* `B` alors que le *thread* `D` aurait constaté l'écriture faite par le *thread* `B` avant celle faite par le *thread* `A`. Cette histoire systémique est incohérente, et fait fi de la causalité.

Il devrait être clair à ce stade que la cohérence séquentielle, qui détermine le respect global de la relation de causalité sur les changements apportés aux états partagés d'un même programme, permet de raisonner sur le code et d'écrire du code correct.

Décrite de manière succincte, la cohérence séquentielle est un prérequis du raisonnement.

Ordonnancement et `mutex`

Il est typiquement possible d'atteindre la cohérence séquentielle avec des `mutex`. Dans le programme précédent, il est possible d'avoir recours à un `mutex` pour accéder à `x` et un d'utiliser un autre `mutex` pour accéder à `y`.

Notez toutefois que, dans les *threads* `C` et `D`, les variables `x` et `y` sont toutes deux accédées alors il importe que ces deux *threads* prennent les deux `mutex` toujours dans le même ordre, pour éviter un interblocage. Cet ordonnancement manuel est périlleux, et le soin qui lui est apporté doit l'être à chaque point d'utilisation de ces deux variables.

Par exemple, si `mx` est un `mutex` protégeant `x` et si `my` est un `mutex` protégeant `y`, le code ci-dessous assure la cohérence séquentielle du système :

***Thread* A**	***Thread* B**	***Thread* C**	***Thread* D**
`mx.lock();`	`my.lock()`	**`mx.lock();`**	**`mx.lock();`**
`x = 1;`	`y = 1;`	**`my.lock();`**	**`my.lock();`**
`mx.unlock();`	`my.unlock();`	`if(x == 1 && y == 0)`	`if(x == 0 && y == 1)`
		`  cout << "x d'abord"`	`  cout << "y d'abord"`
		`  << endl;`	`  << endl;`
		`my.unlock();`	`my.unlock();`
		`mx.unlock();`	`mx.unlock();`

En retour, le code ci-dessous est sujet aux interblocages :

***Thread* A**	***Thread* B**	***Thread* C**	***Thread* D**
`mx.lock();`	`my.lock()`	**`mx.lock();`**	**`my.lock();`**
`x = 1;`	`y = 1;`	**`my.lock();`**	**`mx.lock();`**
`mx.unlock();`	`my.unlock();`	`if(x == 1 && y == 0)`	`if(x == 0 && y == 1)`
		`  cout << "x d'abord"`	`  cout << "y d'abord"`
		`  << endl;`	`  << endl;`
		`my.unlock();`	`my.unlock();`
		`mx.unlock();`	`mx.unlock();`

En effet, dans ce dernier cas, il y a un risque bien réel que le *thread* `C` prenne `mx` et bloque en attente de `my` et que le *thread* `D` prenne `my` et bloque en attente de `mx`, un interblocage classique. Il serait possible de remplacer les appels à `lock()` par des verrous testables non-bloquants (des *Try-Locks*) mais dans un cas comme celui-ci, cette tactique pourrait mener à des *Livelocks* (j'essaie, ça ne fonctionne pas, je recommence, *ad vitam aeternam*).

Avec C++ 11, une fonction `std::lock()` prend en paramètre autant de `mutex` que souhaité et les verrouille toujours selon un ordre prédéterminé (p. ex. : en ordre croissant d'identifiant), ce qui permet d'éviter ce problème.

Ordonnancement et atomiques

Les `mutex` sont des verrous placés manuellement dans le code source. Ils facilitent le raisonnement (parfois, du moins) mais demandent du doigté.

Il est aussi possible d'atteindre une cohérence séquentielle à l'aide d'atomiques, mais seulement si ces variables sont bien utilisées. Notons tout de suite qu'il ne s'agit pas de quelque chose de simple – chaque fois que quelqu'un réalise un nouvel algorithme synchronisé sans verrous, on parle d'un résultat susceptible d'être publié!

Avec une atomique, la programmeuse ou le programmeur étiquette le type d'une variable, pas chacun de ses points d'utilisation comme dans le cas d'un mutex. En ce sens, on obtient alors une sorte d'économie d'efforts. Sur une variable atomique :

- chaque lecture est atomique (indivise);
- chaque écriture est atomique (indivise)
- pour un *thread* donné, les lectures et les écritures sur cette atomique se font dans l'ordre annoncé par le code source;
- certaines opérations spéciales sont possibles et supportées directement par le substrat matériel (des opérations de type *exhange* ou *Copy and Swap*, qu'on écrit typiquement `CAS`).

Opérations atomiques fondamentales

Sous C++ `11`, les opérations `CAS` viennent en deux saveurs, soit le `CAS` fort (*Strong*) et le `CAS` faible (*Weak*).

Un **`CAS` fort** sur une variable atomique `val` s'utilise de la manière suivante :

Code	Signification
`if (val.compare_exchange_strong(cur,voulu))` `  // ...`	Suis-je celui qui a changé la valeur de `cur` à `voulu`?

Le sens de cette expression peut être exprimé en langage naturel comme « suis-je celui qui a changé la valeur de `cur` à `voulu`? ». Conséquemment, cette expression se loge typiquement dans une alternative (un `if`).

Un **`CAS` faible** sur une variable atomique `val` s'utilise de la manière suivante :

Code	Signification
`while (!val.compare_exchange_weak(cur,voulu))` `  // ...`	Tant que je ne suis pas celui qui a changé la valeur de `cur` à `voulu`...

Le sens de cette expression peut être exprimé en langage naturel comme « tant que je ne suis pas celui qui a changé la valeur de `cur` à `voulu` ». Conséquemment, cette expression se loge typiquement dans une répétitive (un `while`).

En pratique, le `CAS` faible fait le même travail que `CAS` fort, mais est plus rapide tout en étant assujetti à des défectuosités occasionnelles, ou *Spurious Failures*, ce qui force à revérifier le résultat par la suite.

Clôtures mémoires

Une autre approche possible pour pallier des risques de réordonnancement des instructions est d'insérer des clôtures (ou des barrières) dans le code. Ces clôtures portent divers noms (*Memory Fences*, *Memory Barriers*) et sont représentées par des instructions à même le code machine, ce qui est nécessaire du fait que pour bloquer des optimisations dans le processeur, il faut communiquer directement avec le matériel.

Si, par exemple, il importe qu'une opération `++` sur un entier `i` soit traitée comme s'il s'agissait d'une opération indivise, il serait possible d'insérer une clôture mémoire avant et après cette opération, comme dans le cas à droite (j'utilise ici la notation `mfence` pour fins illustratives seulement; les opérations varient en nom et en détail selon les architectures matérielles).

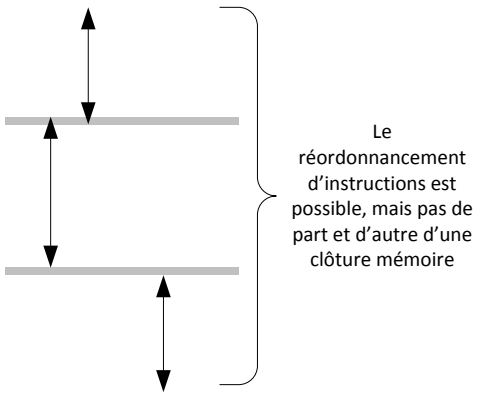
```
mfence;
++i;
mfence;
```

Le comportement décrit ici est essentiellement celui des variables qualifiées `volatile` en C# et en Java, mais pas celui des variables qualifiées `volatile` en C ou en C++. Avec Microsoft Windows, `MemoryBarrier()` est une macro [WinMB].

Une clôture mémoire signifie informellement « tout ce qui se passe avant dans le code source se passe avant en pratique, et tout ce qui se passe après dans le code source se passe après en pratique ».

Agir ainsi est toutefois très dispendieux, car pour réaliser son travail, une clôture mémoire doit bloquer le bus de données de l'ordinateur, ce qui gèle essentiellement l'exécution de tous les cœurs autres que celui qui exécute cette instruction.

Le parallélisme est donc perdu, même au niveau matériel, et un ordinateur à plusieurs cœurs se transforme temporairement en ordinateur à un seul cœur un peu plus lent.



Il est parfois possible d'opérer avec des restrictions moins contraignantes; par exemple, il peut arriver qu'une clôture mémoire avant soit nécessaire mais qu'on puisse omettre la clôture mémoire après. Cela dit, les clôtures mémoire, bien que fonctionnelles, sont coûteuses et difficiles à bien utiliser, devant être placées manuellement en périphérie de *tous* les accès à certaines variables. Le risque d'une erreur est élevé.

Si elles sont utilisées avec rigueur, les clôtures mémoire permettent d'éliminer les *Data Races*, mais entraînent des coûts déraisonnables, du moins pour un programme C ou C++ puisque ces langages sont typiquement choisis pour réduire toute forme de coût à l'exécution (taille occupée en mémoire, vitesse d'exécution, temps de synchronisation, etc.).

C'est pourquoi C++ supporte les clôtures mémoires [CppMB], mais aussi pourquoi la plupart des programmeuses et des programmeurs dans ce langage privilégient des outils permettant un contrôle plus fin sur les exigences de synchronisation.

Ordonnancement et pleines clôtures

Essentiellement toutes les plateformes contemporaines proposent une instruction machine représentant une **pleine clôture** (*Full Fence*), bloquant l'accès au bus de données et transformant à toutes fins pratiques un ordinateur parallèle en ordinateur séquentiel. Ces instructions sont foncièrement non-portables, et dépendent à la fois du processeur et du système d'exploitation du fait qu'elles ont une conséquence directe sur le matériel.

Le rôle d'une pleine clôture est d'insérer un marqueur dans le code machine tel qu'il devient impossible pour un processeur (et pour un compilateur, par convention) de réordonnancer des instructions autour de ce point. Conséquemment, une instruction apparaissant avant une telle clôture ne pourra être déplacée après, et une instruction se trouvant après cette clôture ne pourra être déplacée avant.

Les deux plus grands désavantages des pleines clôtures sont (a) qu'elles doivent être écrites correctement à chaque point d'utilisation, et (b) qu'elles sont coûteuses, étant par nature pessimiste et ne distinguant pas entre les sémantiques *Acquire* et les *Release*. Les pleines clôtures sont dénuées de sémantique, du fait qu'elles ne sont pas porteuses des contraintes qui motivent leur utilisation ou leur relation avec le reste du code – typiquement, les clôtures réduisent la capacité des programmes d'être vraiment concurrents.

Clôtures en C++

Les clôtures de C++ `11`, représentées par des `std::atomic_thread_fence`, sont des pleines clôtures par défaut, mais sont aussi paramétrables sur la base des sémantiques de synchronisation telles qu'*Acquire* et *Release*.

Comme dans le cas des atomiques, la clôture par défaut permet de raisonner plus facilement sur la base du code source, mais il est parfois possible de relaxer les contraintes sémantiques de synchronisation pour alléger le poids de la synchronisation sur l'exécution du programme.

La synchronisation combat activement les optimisations contemporaines, matérielles et logicielles. Elle transforme nos programmes et nos ordinateurs parallèles en programmes et ordinateurs séquentiels, quand bien même ce ne serait que pour de brèves périodes. Nous souhaitons donc le moins possible de synchronisation dans nos programmes.

Nous souhaitons éviter le recours aux clôtures mémoire, et nous souhaitons surtout éviter d'écrire de telles instructions manuellement dans nos programmes. Pour cette raison, nous privilégions les `mutex` et les variables atomiques, qui peuvent être vues comme équivalentes à plusieurs points de vue. Un `mutex`, sur plateforme `IA64`, s'obtient par un *Lock Acquire* (itération sur `ld.acq` en assembleur) et se libère par un *Lock Release* (en assembleur, un `st.rel`); ce sont les mêmes opérations que pour des accès sur une atomique au niveau de la machine.

Relations dans un système multiprogrammé

Pour comprendre les règles qui permettent à un programme multiprogrammé d'en arriver à un comportement respectant la cohérence séquentielle, du moins en C++, il faut comprendre quelques relations décrites par le standard, leur rôle et comment les réifier.

On détermine la cohérence de l'exécution d'un programme multiprogrammé à partir de l'ordre dans lequel en sont modifiés les états partagés entre au moins deux *threads*. Si un objet *M* est modifié par un événement *A* puis par un événement *B*, on dira que *A* précède *B* dans l'ordre de modification de *M*. Ceci détermine, dans un programme séquentiellement cohérent, une **relation *Happens Before***, et cette relation sera respectée dans l'ordonnancement total du programme s'il est `SC-DRF`.

Relation Sequenced-Before

La relation la plus simple est ***Sequenced-Before***. Concrètement, *A* est *Sequenced-Before B* si *A* et *B* sont exécutés par un même *thread* et si l'exécution de *A* précède celle de *B*. Cette relation est transitive.

Deux événements *A* et *B* peuvent ne pas être séquencés s'il n'existe pas de relation *Sequenced-Before* entre eux; dans un tel cas, les deux exécutions peuvent se chevaucher.

Relation Synchronizes-With

Certaines opérations provoquent une **relation *Synchronizes With***. Par exemple, un *Store-Release* sur une variable atomique se synchronisera avec un *Load-Acquire* sur la même variable.

À titre d'exemple, le verrouillage d'un mutex (opération de sémantique *Acquire*) « lira » le résultat de la plus récente opération de déverrouillage de ce mutex (opération de sémantique *Release*); conséquemment, cette paire d'opérations respectera la relation *Synchronizes With*.

Relation Carries-Dependency

Outre quelques cas pathologiques[3], une **relation *Carries-Dependency*** entre *A* et *B* naît du fait que la valeur de *A* sert d'opérande à *B*.

Le standard précise que cette relation est strictement interne à un seul et même *thread*, et est un sous-ensemble de la relation *Sequenced-Before*.

[3] Appel à `std::kill_depencency()`, évaluation en tant qu'opérande de gauche de l'opérateur « `,` » ou d'une expression logique, etc.

Relation Dependency-Ordered Before

Une **relation *Dependency-Ordered Before*** entre *A* et *B* signifie que :

- *A* réalise une opération de sémantique *Release* sur un objet atomique alors que, dans un autre *thread*, *B* réalise une opération de sémantique *Consume* sur le même objet et lit une valeur résultant de la séquence d'opérations de sémantique *Release* débutant à *A*, ou encore
- pour une évaluation *X* quelconque, *A* est *Dependency-Ordered Before X*, et *X Carries-Dependency* sur *B*.

Le standard précise que *Dependency-Ordered Before* est une relation analogue à *Synchronizes With*, mais là où *Synchronizes With* repose sur une paire *Release-Acquire*, *Dependency-Ordered Before* repose sur une paire *Release-Consume*.

Relation Inter-Thread Happens Before

La **relation *Inter-Thread Happens Before*** décrit une suite de concaténations de relations de sémantique *Sequenced Before*, *Synchronizes With* et *Dependency-Ordered Before*.

Il y a quelques exceptions :

- une telle concaténation ne peut se terminer par un *Dependency-Ordered Before* suivi par un *Sequenced Before*, l'opération de sémantique *Consume* ne pouvant porter la dépendance de la première partie vers la seconde;
- une telle concaténation ne peut se limiter strictement à des *Sequenced Before*, pour assurer que la relation *Inter-Thread Happens Before* soit transitivement fermée.

Relation Happens Before

Enfin, formellement, la **relation *Happens Before*** entre *A* et *B* survient si :

- *A* est *Sequenced-Before B*, ou
- *A Inter-Thread Happens Before B*.

C'est la relation *Happens-Before* qui permet de déterminer une causalité systémique dans une exécution d'un programme.

Les atomiques « relaxées »

Une atomique de C++ 11 offre par défaut des opérations séquentiellement cohérentes. Cependant, il est possible de réduire la cohérence opératoire d'une atomique en ayant recours à des opérations dites « relaxées », au sens où les contraintes d'ordonnancement de ces opérations sont moins strictes que celles imposées sur les atomiques séquentiellement cohérentes.

Une opération atomique demeure atomique, qu'elle soit séquentiellement cohérente ou non : elle a lieu en entier, de manière indivise, comme une transaction. L'impact de la contrainte stricte ou « relaxée » est d'influencer le réordonnancement des opérations en périphérie de l'opération atomique en question.

Il se trouve que les opérations atomiques simples, incluant les lectures (les *Load*) qui devraient être presque gratuites par principe, peuvent être dispendieuses à exécuter sur certaines architectures n'offrant pas un support à une sémantique opératoire *Acquire*/ *Release* et forçant un recours à une pleine clôture mémoire pour implémenter les atomiques.

Pour cette raison pragmatique, il est possible de contourner ces coûts architecturaux et de réduire les contraintes d'ordonnancement sur une atomique en « relaxant » temporairement les exigences de cohérence séquentielle.

C'est ce qu'implique le recours à une atomique « relaxée » : accepter que des opérations ne soient pas pleinement respectueuses de la cohérence séquentielle, et réduire par le fait-même volontairement notre capacité de raisonner sur le programme à partir du code source. Par défaut, rappelons-le, une atomique de C++ offre des opérations séquentiellement cohérentes.

La première règle à appliquer dans le cas des atomiques relaxées est de ne pas les utiliser du tout. Elles sont dangereuses! Mais bon, elles existent et peuvent être utilisées quand le modèle séquentiellement cohérent est trop fort pour les réels besoins applicatifs.

Les modèles de cohérence d'un programme multiprogrammé, si on les place en ordre décroissant de « force », sont :

- un modèle SC complet, un peu comme un gros programme séquentiel. Mieux veut oublier cela en pratique;
- un modèle SC-DRF, avec C++ 11, C 11, Java 05, etc. En pratique, ce modèle est la réalité des programmes concurrents contemporains; et
- tout ce qui est « relaxé ». À partir d'ici, le modèle mémoire soutenant l'exécution du programme est *considérablement* plus faible.

Utiliser des atomiques « relaxées », c'est accepter de perdre (en partie) les relations de causalité. Un impact direct et que les threads s'exécutant concurremment peuvent ne plus voir une histoire cohérente de l'exécution du programme, donc peuvent ne plus voir les écritures à des états partagés comme se produisant dans un ordre unique. Dans un programme avec atomiques « relaxées », il est possible qu'un *thread* voit A avant B alors qu'un autre voit B avant A.

Comme l'indiquait **Hans Boehm** : « *we've taken great care that* ***without relaxed atomics****, 'simultaneously' really means what you thought it did* ». Une fois que vous faites le choix délibéré de relaxer les contraintes sur vos atomiques, vous êtes responsables des conséquences.

Ordonnancements mémoire possibles

Les ordonnancements mémoire pour des atomiques de C++ 11 sont les suivants. Ils peuvent être appliqués sur une atomique au sens large, tout comme ils peuvent être appliqués sur une atomique une opération à la fois :

- l'ordonnancement `memory_order_relaxed`, où l'opération est indivise mais où il ne reste aucune garantie quant à l'ordonnancement des opérations;
- l'ordonnancement `memory_order_acquire`;
- l'ordonnancement `memory_order_release`;
- l'ordonnancement `memory_order_acq_rel`, qui assure le respect à la fois des ordonnancements `memory_order_acquire` et `memory_order_release` mais ne garantit pas de cohérence séquentielle; et
- l'ordonnancement `memory_order_seq_cst`, soit le comportement par défaut.

Le standard dit des opérations `memory_order_relaxed` que « *no operation orders memory* ». Sur les opérations de sémantique *Acquire*, le standard indique « *a load operation performs an acquire operation on the affected memory location* », et sur les opérations de sémantique *Release*, le standard indique « *a store operation performs a release operation on the affected memory location* ».

Il existe aussi un ordonnancement pointu nommé `memory_order_consume`, qui ressemble à `memory_order_acq_rel` mais avec lequel il faut parfois avoir recours à une annotation telle que `[[carries_dependency]]` comme à des fonctions spécialisées telles que `kill_dependency()`. Le standard indique « *a load operation performs a consume operation on the affected memory location* ».

En pratique, on parle de quelque chose de très difficile à utiliser, et dont la spécification dans le standard C++ 14 est telle qu'il est difficile d'écrire un programme qui en profite de manière portable. L'essence de cet ordonnancement est que, sur certaines architectures, les processeurs respectent implicitement des enchaînements d'accès en lecture et en écriture sur des données (des *Data Dependencies*); `memory_order_consume` permettrait à un compilateur suffisamment agressif de profiter de ce respect implicite et de réduire les contraintes explicites (et coûteuses) dans le code généré.

Tiré du standard, §29.3, on trouve ceci (remarquez la notation C, qui résulte du fait que C 11 et C++ 11 ont adopté des atomiques mutuellement compatibles) :

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

Le standard ajoute que ces garanties ne sont des garanties que si l'objet atomique est en soi indivise (qu'il ne chevauche pas deux zones mémoire). Cela signifie en particulier que, pour bénéficier des atomiques, mieux vaut éviter les *bitfields*.

Applications des atomiques « relaxées »

Toute architecture confondue, il existe de réels cas d'utilisation pour les atomiques « relaxées », donc d'opérations qui ne garantissent pas `SC-DRF`, mais (a) ils sont rares, et doivent être encadrés, ou (b) sont des solutions *temporaires* à des problèmes *matériels* (des palliatifs pour des architectures trop strictes ou trop dispendieuses). Des applications classiques sont les compteurs d'événements, les *Dirty Flags* et le comptage de références.

Mieux vaut encadrer ces opérations de manière à ce que le code client ne les voit pas.

Exemple – Compteur d'événements

Supposons par exemple un système à *n threads* (outre le *thread* principal), et où chaque *thread* auxiliaire fait ce qui suit. Dans cet exemple, supposez que `count` soit atomique et ait initialement la valeur zéro :

```
while (...)
{
   // ...
   if (...)
      ++count; // pourrait être count.fetch_add(1, memory_order_relaxed)
   // ...
}
```

Le *thread* principal, quant à lui, ferait ceci :

```
int main()
{
   // lancer les threads
   // ...
   // join() sur tous les threads
   cout << count << endl;
   // pourrait être cout << count.load(memory_order_relaxed) << endl;
}
```

Les opérations `join()`, tout comme les fins des `async` ou des `std::thread`, provoquent des relations *happens-before*, alors il n'est pas nécessaire de à se préoccuper d'ordonnancement avant/ après pour ces opérations.

Pour le reste, ici, si *rien* d'autre n'est fait avec `count`, il devient correct d'utiliser des opérations « relaxées ». Prudence en particulier avec les conditions des `if`!

Ici, bien que le code proposé fonctionne, il serait nettement mieux de faire de `count` un attribut d'instance dans une classe, et de cacher les opérations « relaxées » dans des méthodes pour ne pas les exposer au grand public. Cela réduirait le risque qu'un tiers n'accède à `count` de manière inappropriée.

Exemple – Fanion

Un autre cas d'application serait un fanion où des *threads* font une opération telle que `val = true;` et où le *thread* principal teste `val` après l'appel aux `join()` des *threads*.

Supposons par exemple un système à `n` *threads* (outre le *thread* principal), et où chaque *thread* auxiliaire fait ce qui suit. Dans cet exemple, supposez que `dirty` soit atomique relaxée et ait initialement la valeur `false` :

```
// La condition ci-dessous pourrait s'exprimer stop.load(memory_order_relaxed) car
// elle consomme sans publier. Un peu comme si un thread était plus lent que les autres
while (!stop)
{
   // ...
   if (...)
      dirty = true; // pourrait être dirty.store(true, memory_order_release)
   // ...
}
```

Le *thread* principal, quant à lui, ferait ceci :

```
int main()
{
   // lancer les threads
   // ...
   stop = true; // cette opération n'est pas relaxed
   // join sur tous les threads
   if (dirty) // dirty.load(memory_order_acquire)
      nettoyage();
}
```

Exemple – Compteur de clients d'un `shared_ptr`

Pour implémenter un pointeur intelligent tel qu'un `shared_ptr`, où le constructeur incrémente un compteur et le destructeur décrémente un compteur, il est *possible* que l'incrémentation initiale soit « relaxée » : elle ne publie rien, donc passer de `0` à `1` est gratuit.

De même, il est *probable* que la décrémentation soit `acq_rel` : elle pourrait mener à une action posée par un tiers, mais descendre à zéro est gratuit puisque seul le *thread* actif est susceptible de le constater.

Pour un type susceptible d'être fortement utilisé, comme c'est le cas de `shared_ptr`, considérer le recours (circonscrit) à des atomiques relaxées peut rapporter… Cela dit, votre implémentation commerciale du standard fait probablement les choses correctement.

Individus

Les individus suivants sont mentionnés dans le présent document. Vous trouverez, en suivant les liens proposés à droite du nom de chacun, des compléments d'information à leur sujet et des suggestions de lectures complémentaires. Avis aux curieuses et aux curieux!

Hans Boehm	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#hans_boehm
Alex Boulanger	Un de mes anciens étudiants au Collège Lionel-Groulx et à l'Université de Sherbrooke
Lawrence Crowl	Contributeur de plusieurs propositions au comité de standardisation `ISO` de C++
Adam Galarneau	Un de mes anciens étudiants au Collège Lionel-Groulx
Danny Kalev	Contributeur au comité de standardisation `ISO` de C++ et spécialiste de l'analyse statique de code
Leslie Lamport	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#leslie_lamport
Scott Meyers	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#scott_meyers
Bartosz Milewski	Contributeur au comité de standardisation `ISO` de C++, spécialiste de multiprogrammation et de programmation fonctionnelle
Jeff Phreshing	Blogueur et spécialiste de programmation à bas niveau
Herb Sutter	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#herb_sutter
Anthony Williams	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#anthony_williams

Références

Les références qui suivent respectent un format quelque peu informel. Elles vous mèneront soit à des notes de cours de votre humble serviteur, soit à des documents pour lesquels mes remarques sont proposées de manière électronique et à partir desquels vous pourrez accéder aux textes d'origine ou à des compléments d'information.

[CppMB] http://en.cppreference.com/w/cpp/atomic/atomic_thread_fence

[hdPrLoc] http://h-deb.clg.qc.ca/Sujets/Developpement/Pratique-programmation.html#principes

[hdTrMem] http://h-deb.clg.qc.ca/Liens/Multiprogrammation--Liens.html#memoire_transactionnelle

[hdUB] http://h-deb.clg.qc.ca/Liens/Langages-programmation--Liens.html#undefined_behavior

[JavaMemMod] https://jcp.org/en/jsr/detail?id=133

[LampClock] http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf

[ISOCppStd] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3485.pdf

[SuttAtom1] http://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2

[SuttAtom2] http://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-2-of-2

[WinMB] http://msdn.microsoft.com/en-us/library/windows/desktop/ms684208%28v=vs.85%29.aspx