std::memory_order

std:: Библиотека атомарных операций

Определён в заголовочном файле <atomic>

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```
(начиная с C++11)

std::memory_order (упорядочение доступа к памяти) определяет, как обычный, неатомарный доступ к памяти, упорядочивается вокруг атомарных операций. При отсутствии каких-либо ограничений, на многоядерных системах, когда множество потоков одновременно читает и пишет в несколько переменных, один поток может наблюдать изменение значений переменных в порядке, отличающемся от того, в котором другой поток записывает их. На самом деле, видимый порядок изменений может отличаться даже среди нескольких читающих потоков.

Для атомарных операций по умолчанию библиотекой предоставляется *последовательно согласованное упорядочение* (*sequentially consistent ordering*) (см обсуждение ниже). Такое поведение может повредить быстродействию, но атомарным операциям библиотеки может быть передан дополнительный std::memory_order аргумент, чтобы указать точные ограничения, помимо атомарности, которые компилятор и процессор должны обеспечить для этой операции.

Константы

Заголовочный файл <atomic>

Значение	Объяснение
memory_order_relaxed	**Ослабленное**(**Relaxed**) упорядочение: отсутствуют ограничения синхронизации и упорядочения, для данной операции требуется только атомарность.
memory_order_consume	Операция загрузки с этим упорядочением памяти выполняет операцию **поглощения** (**consume**) над задействованной областью памяти: предыдущие записи в зависимую от данных область памяти, сделанные потоком, выполнившим операцию освобождения (release), становятся видимыми для цепочки зависимостей данного потока.
memory_order_acquire	Операция загрузки с этим упорядочением памяти выполняет операцию **захвата** (**acquire**) над задействованной областью памяти: предыдущие записи, сделанные в зависимую от данных область памяти потоком, который выполнил освобождение (release), становятся видимыми в данном потоке.

`memory_order_release`	Операция сохранения с этим упорядочением памяти выполняет операцию **освобождения** (**release**): предыдущие записи в другие области памяти, становятся видимыми для потоков, которые выполняют операцию **поглощения** (**consume**) или **захвата** (**acquire**) над той же областью памяти.
`memory_order_acq_rel`	Операция загрузки с этим упорядочением памяти выполняет операцию **захвата** (**acquire**) над задействованной областью памяти. Операция сохранения с этим упорядочением памяти выполняет операцию **освобождения** (**release**).
`memory_order_seq_cst`	(sequentially-consistent - последовательно согласованное) То же, что и memory_order_acq_rel, плюс существует единый общий порядок, при котором все потоки видят все изменения (см. ниже) в одинаковом порядке.

Формальное описание

Межпоточная синхронизация и упорядочение памяти определяют, как *вычисления* и *побочные эффекты* выражений упорядочиваются между различными потоками выполнения. Отношения определены в следующих правилах:

Расположено-перед

(**Sequenced-before**)

В одном и том же потоке, вычисление A *расположено-перед* вычислением B, если это следует из evaluation order.

Переносит-зависимость-в

(**Carries-dependency-to**)

В одном и том же потоке, вычисление A, которое *расположено-перед* вычислением B, может также переносить-зависимость-в B (то есть B зависит от A), если выполняется любое из следующих утверждений:

1) Значение A используется, как операнд B, кроме случаев
   a) если B, это вызов std::kill_dependency
   b) если A - это левый операнд встроенного оператора `&&`, `||`, `?:`, или `,`.
2) A пишет в скалярный объект M, B читает из M
3) A переносит-зависимость-в другое вычисление X, и X переносит-зависимость-в B

Порядок изменения

Все изменения любой определённой атомарной переменной происходят в общем порядке, который определён для этой атомарной переменной.

Следующие четыре требования гарантированно выполняются для всех атомарных операций:

1) **Запись-запись согласованность**: если вычисление A, которое изменяет некоторую атомарную M (запись) *происходить-раньше* вычисления B, которое изменяем M, тогда A появляется раньше, чем B в *порядке изменения* M.
2) **Чтение-чтение согласованность**: если процесс вычисления значения A некоторой атомарной M (чтение) *происходит-раньше* процесса вычисления значения B этой же M, и если значение A происходит из записи X в M, тогда значение B, это либо значение записанное X, либо значение записанное побочным эффектом Y, воздействующим на M, который возникает позже, чем X, в *порядке изменения* M.
3) **Чтение-запись согласованность**: если процесс вычисления значения A некоторой атомарной M (чтение) *происходит-раньше* операции B применяемой к M (запись), тогда значение A происходит из побочного эффекта (записи) X, который возникает раньше, чем B в *порядке изменения* M.
4) **Запись-чтение согласованность**: если побочный эффект (запись) X воздействующий на атомарный объект M *происходит-раньше* процесса вычисления значения (чтения) B переменной M, тогда вычисление B должно получить своё значение от X или от побочного эффекта Y, который следует за X в *порядке изменения* M.

Последовательность освобождения

(**Release sequence**)

После *операции освобождения* (*release*) A, выполненной по отношению к атомарному объекту M, самая длинная непрерывная часть последовательности порядка изменения M, которая состоит из:

1) Записей выполненных тем же потоком, который выполнил A
2) Атомарных чтение-изменение-запись операций, применённых любым потоком к M

называется *последовательностью освобождения во главе с A*.

Предшествует-по-зависимости

(**Dependency-ordered before**)

Межпоточно вычисление A *предшествует-по-зависимости* вычислению B, если выполняется любое из следующих утверждений:

1) A выполняет *операцию освобождения* (*release*) над некоторой атомарной M, и, в другом потоке, B выполняет *операцию поглощения* (*consume*) над той же атомарной M, и B читает значение, записанное любой частью последовательности освобождения во главе с A.

2) A предшестует-по-зависимости X и X переносит-зависимость-в B.

Межпоточно происходит-раньше

(**Inter-thread happens-before**)

Вычисление A *межпоточно происходит-раньше* вычисления B, если выполняется любое из следующих утверждений:

1) A *синхронизируется-с* B
2) A *предшествует-по-зависимости* B
3) A *синхронизируется-с* некоторым вычислением X, и X *расположено-перед* B
4) A *расположено-перед* некоторым вычислением X, и X *межпоточно происходит-раньше* B
5) A *межпоточно происходит-раньше* некоторого вычисления X, и X *межпоточно происходит-раньше* B

Происходит-раньше

(**Happens-before**)

В независимости от потоков, вычисление A *происходит-раньше* вычисления B, если выполняется любое из следующих утверждений:

1) A *расположено-перед* B
2) A *межпоточно происходит-раньше* B

Если одно вычисление модифицирует область памяти, и другое читает или модифицирует эту же область памяти, и если хотя бы одно из вычислений не является атомарной операцией, поведение программы не определено (в программе присутствует гонка за данными) кроме случаев, когда существует отношение *происходит-раньше* между этими двумя вычислениями.

Видимые побочные эффекты

Побочный эффект A, воздействующий на скалярную M (запись) является *видимым* по отношению к процессу вычисления значения B переменной M (чтение), если выполняются оба следующих утверждения:

1) A *происходит-раньше* B
2) Никакой другой побочный эффект X, не действует на M, где A *происходит-раньше* X, и X *происходит-раньше* B

Если побочный эффект A является видимым по отношению в процессу вычисления значения B, тогда самое длинное и непрерывное подмножество побочных эффектов воздействующих на M в *порядке модификации*, где B не *происходит-раньше*, известено, как “видимая последовательность побочных эффектов” (значение M, определённое B, будет значением, сохранённым одним из этих побочных эффектов).

Замечание: межпоточная синхронизация сводится к определения, при каких условиях какие побочные эффекты становятся видимыми.

Операция поглощения

(**Consume operation**)

Атомарная загрузка с упорядочением `memory_order_consume` или более строгим, является операцией поглощения (consume). Учтите, что барьер `std::atomic_thread_fence` не является операцией поглощения.

Операция захвата

(**Acquire operation**)

Атомарная загрузка с упорядочением `memory_order_acquire` или более строгим, является операцией захвата (acquire). Операция lock(), применяемая к Mutex (http://en.cppreference.com/w/cpp/concept/Mutex) , также является операцией захвата. Учтите, что барьер `std::atomic_thread_fence` не является операцией захвата.

Операция освобождения

(**Release operation**)

Атомарное сохранение (запись) с упорядочением `memory_order_release` или более строгим, является операцией освобождения (release). Операция unlock(), применяемая к Mutex (http://en.cppreference.com/w/cpp/concept/Mutex) , также является операцией освобождения. Учтите, что барьер `std::atomic_thread_fence` не является операцией освобождения.

Объяснение

Ослабленное упорядочение

Атомарные операции, отмеченные как **std::memory_order_relaxed**, не являются синхронизирующими операциями, они не упорядочивают память. Они гарантируют только атомарность и согласованность порядка модификации.

Например, при x и y изначально равных нулю, (до C++14)

```
// Thread 1:
r1 = y.load(memory_order_relaxed); // A
x.store(r1, memory_order_relaxed); // B
// Thread 2:
r2 = x.load(memory_order_relaxed); // C
y.store(42, memory_order_relaxed); // D
```

допускается, чтобы r1 == r2 == 42 потому, что хотя A *расположено-раньше* B и C *расположено-раньше* D, ничего не мешает D появиться раньше A в порядке изменения y, и B появиться раньше C в порядке изменения x.

Даже при ослабленной модели памяти, произвольным значениям не разрешено циклически зависеть от вычисления самих себя, например, при x и y изначально равных нулю,

```
// Thread 1:
r1 = y.load(memory_order_relaxed); // A
if (r1 == 42) x.store(r1, memory_order_relaxed); // B
// Thread 2:
r2 = x.load(memory_order_relaxed); // C
if (r2 == 42) y.store(42, memory_order_relaxed); // D
```

(начиная с C++14)

не допускается, чтобы r1 == r2 == 42, так как запись 42 в y возможна только, если запись в x записывает 42, и в свою очередь циклически зависит от записи в y, записывающей 42.

Типичное использование ослабленного упорядочения памяти - это обновление счётчиков, таких как счётчики ссылок в std::shared_ptr, так как оно требуют только атомарности, но не упорядочения или синхронизации.

Упорядочение Освобождение-Захват

Если атомарное сохранение в потоке A отмечено упорядочением **std::memory_order_release** и атомарная загрузка в потоке B из этой же переменной отмечена упорядочением **std::memory_order_acquire**, все записи памяти (не атомарные и с ослабленным упорядочением), которые *происходят-раньше* атомарной записи с точки зрения потока A, становятся *видимыми побочными эффектами* в потоке B, то есть, после того, как атомарная загрузка завершена, поток B гарантированно увидит всё, что поток A записал в память.

Синхронизация устанавливается только между *освобождающим* и *захватывающим* одну и ту же атомарную переменную потоками. Другие потоки могут видеть другой порядок доступа к памяти, чем один или оба синхронизируемых потока.

На системах со строгим упорядочением (x86, SPARC TSO, IBM) упорядочение освобождение-захват используется автоматически для большинства операций. Для организации данного режима синхронизации не требуется дополнительных инструкций процессора, только некоторые оптимизации компилятора могут оказывать влияние (на порядок инструкций) (например, компилятору запрещено перемещать не атомарные операции записи после (по порядку) атомарных операций записи-освобождения или выполнять не атомарные операции загрузок до атомарных операций загрузки-захвата). На системах с ослабленным упорядочением (ARM, Itanium, PowerPC), должны использоваться специальные инструкции процессора для загрузки (load) или для задания барьеров памяти.

Взаимоисключающие блокировки (такие как std::mutex или atomic spinlock) являются примерами синхронизации вида освобождение-захват: когда блокировка освобождается потоком A и захватывается потоком B, всё, что происходит в критической секции (перед операцией освобождения) в контексте потока A, становится видимым потоку B (после операции освобождения), который выполняет ту же критическую секцию.

Запустить этот код

```
#include <thread>
#include <atomic>
#include <cassert>
#include <string>

std::atomic<std::string*> ptr;
int data;

void producer()
{
    std::string* p  = new std::string("Hello");
    data = 42;
    ptr.store(p, std::memory_order_release);
}

void consumer()
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_acquire)))
        ;
    assert(*p2 == "Hello"); // равенство выполняется всегда
    assert(data == 42); // равенство выполняется всегда
}

int main()
{
```

```
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join(); t2.join();
}
```

Следующий пример демонстрирует транзитивность упорядочения освобождение-захват между тремя потоками

Запустить этот код

```
#include <thread>
#include <atomic>
#include <cassert>
#include <vector>

std::vector<int> data;
std::atomic<int> flag = {0};

void thread_1()
{
    data.push_back(42);
    flag.store(1, std::memory_order_release);
}

void thread_2()
{
    int expected=1;
    while (!flag.compare_exchange_strong(expected, 2, std::memory_order_acq_rel)) {
        expected = 1;
    }
}

void thread_3()
{
    while (flag.load(std::memory_order_acquire) < 2)
        ;
    assert(data.at(0) == 42); // равенство будет выполняться всегда
}

int main()
```

```
{
    std::thread a(thread_1);
    std::thread b(thread_2);
    std::thread c(thread_3);
    a.join(); b.join(); c.join();
}
```

Упорядочение Освобождение-Поглощение

Если атомарное сохранение в потоке A отмечено упорядочением **std::memory_order_release** и атомарная загрузка в потоке B из той же переменной отмечена упорядочением **std::memory_order_consume**, все записи в память (не атомарные и с ослабленным упорядочением), которые *предшествуют-по-зависимостям* атомарному сохранению с точки зрения потока A, становятся *видимыми побочными эффектами* в рамках этих операций в потоке B, в котором операция загрузки *переносит-зависимость-в*. То есть, когда атомарная операция загрузки завершена, те операторы и функции в потоке B, которые используют значение, полученное посредством загрузки, гарантированно увидят то, что поток A записал в память.

Синхронизация устанавливается только между *освобождающим* и *поглощающим* одну и ту же атомарную переменную потоками. Другие потоки могут видеть другой порядок доступа к памяти, чем один или оба синхронизируемых потока.

Типичными случаями использования данного упорядочения являются: организация одновременного доступа на чтение к редко записываемым структурам данных (таблицам маршрутизации, конфигурациям, политикам безопасности, правилам брандмауэра, и т.д.) и случай издатель-подписчик с публикацией данных опосредованно через указатель. То есть, когда производитель публикует указатель, через который потребитель может получить доступ к информации, не требуется делать видимым для потребителя ничего, кроме того, что производитель записал в память (что может быть дорогой операцией в архитектурах с ослабленным упорядочением). Примером подобного сценария является rcu_dereference .

См. также std::kill_dependency и [[carries_dependency]] для детального контроля цепочки зависимостей.

Этот пример демонстрирует синхронизацию по принципу *предшествует-по-зависимости* для опосредованной через указатель публикации данных: целочисленные данные не относятся к указателю на строку через отношение зависит-от-данных, поэтому их значение не определено в потребителе (потоке выполняющем функцию consumer()).

Запустить этот код

```
#include <thread>
#include <atomic>
#include <cassert>
#include <string>
```

```
std::atomic<std::string*> ptr;
int data;

void producer()
{
    std::string* p  = new std::string("Hello");
    data = 42;
    ptr.store(p, std::memory_order_release);
}

void consumer()
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_consume)))
        ;

    // равенство выполняется всегда: загрузка ptr переносит-зависимость-в *p2
    assert(*p2 == "Hello");
    // равенство может выполняться, но может и не выполняться, т.к. загрузка из
    // ptr не переносит-зависимость-в data
    assert(data == 42);
}

int main()
{
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join(); t2.join();
}
```

Последовательно-согласованное упорядочение

Атомарные операции отмеченные признаком **std::memory_order_seq_cst** не только упорядочивают память таким же образом как и упорядочение захват/освобождение (всё, что происходит-до сохранения в одном потоке, становится видимым побочным эффектом в потоке, который выполнил загрузку), но и также устанавливают единый общий порядок изменения всех атомарных операций, которые отмечены тем же признаком упорядочения.

С формальной точки зрения,

Каждая операция B, выполняющая загрузку из атомарной переменной M, отмеченная упорядочением `memory_order_seq_cst`, наблюдает одно из следующих состояний:

- результат последней операции A, которая изменяет M, которая появляется перед B

в едином общем порядке изменения переменных.

- ИЛИ, если A имеет такое же упорядочение как и B, B может наблюдать результат

некоторых изменений переменной M, которые не упорядочены посредством `memory_order_seq_cst` и не *происходят-раньше* A

- ИЛИ, если A имеет упорядочение отличное от B, B может наблюдать результат некоторых

независимых изменений переменной M которые упорядочены не при помощи `memory_order_seq_cst`.

Если операция X, упорядоченная посредством `memory_order_seq_cst` с использование барьера std::atomic_thread_fence *расположена-перед* B, тогда B наблюдает одно из следующих состояний:

- последнее изменение M, упорядоченное посредством `memory_order_seq_cst`, которое появляется перед X в едином общем порядке изменения переменных.
- некоторые независимые изменения переменной M, которые появляются позже в порядке изменения переменной M.

Если для пары атомарных операций применяемых к M, именуемых A и B, где A пишет, а B читает значение M, мы имеем два барьера std::atomic_thread_fence X и Y с упорядочением `memory_order_seq_cst`, и если A *расположена-перед* X, Y *расположен-перед* B, и X появляется перед Y в Едином Общем Порядке, тогда B наблюдаяет либо:

- побочный эффект создаваемый A
- некоторые независимые изменения переменной M, которые появляются после A в порядке

изменения M

Для пары атомарных изменений переменной M, именуемых A и B, B происходит после A в порядке изменения переменной M, если

- имеется барьер std::atomic_thread_fence X, с упорядочением `memory_order_seq_cst`, расположенный так, что A *расположенно-перед* X и X появляется перед B в Едином Общем Порядке.
- или, если имеется барьер std::atomic_thread_fence Y, с упорядочением `memory_order_seq_cst`, расположенный так, что Y *расположен-перед* B и A появляется перед Y в Едином Общем Порядке.
- или, если имеются барьеры std::atomic_thread_fence X и Y, с упорядочением `memory_order_seq_cst`, расположенные так, что A *расположено-перед* X, Y *расположен-перед* B, и X появляется перед Y в

Едином Общем Порядке.

Учтите, что это означает следующее:

1) как только в общей картине появляются атомарные операция упорядоченные не при помощи `memory_order_seq_cst`, последовательная согласованность теряется.

2) последовательно-согласованные барьеры устанавливают общее упорядочение только для самих себя, не для атомарных операций в общем случае (*расположено-перед* не является межпоточным отношением, в отличие от *происходит-раньше*)

Последовательное упорядочение может быть необходимым для случая "несколько производителей - несколько потребителей", где все потребители должны наблюдать действия всех производителей которые происходят в одном и том же порядке.

Общее последовательное упорядочевание требует наличия полных барьеров памяти на уровне инструкций процессора на всех многоядерных системах. Это может стать узким местом производительности, так как потребуется обеспечить доступ к задействованной памяти для любого ядра.

Этот пример демонстрирует ситуацию, когда последовательное упорядочение необходимо. Любое другое упорядочение может справоцировать срабатывание assert'a, т.к. потоки c и d могли бы наблюдать изменения атомарных x и y в противоположном порядке.

Запустить этот код

```
#include <thread>
#include <atomic>
#include <cassert>

std::atomic<bool> x = {false};
std::atomic<bool> y = {false};
std::atomic<int> z = {0};

void write_x()
{
    x.store(true, std::memory_order_seq_cst);
}

void write_y()
{
    y.store(true, std::memory_order_seq_cst);
}

void read_x_then_y()
{
```

```
    while (!x.load(std::memory_order_seq_cst))
        ;
    if (y.load(std::memory_order_seq_cst)) {
        ++z;
    }
}

void read_y_then_x()
{
    while (!y.load(std::memory_order_seq_cst))
        ;
    if (x.load(std::memory_order_seq_cst)) {
        ++z;
    }
}

int main()
{
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join(); b.join(); c.join(); d.join();
    assert(z.load() != 0);  // неравенство будет выполняться всегда
}
```

Отношения с `volatile`

Внутри потока исполнения, операции доступа ко всем объектам с модификатором `volatile` (чтение и запись) гарантированно не будут переупорядочены относительно друг друга, но не гарантируется, что этот порядок будет видимым для других потоков, так как доступ к volatile переменным не устанавливает отношений межпоточной синхронизации.

В дополнение, операции доступа volatile не являются атомарными (параллельные чтение и запись являются data race) и не упорядочивают память (не-volatile обращения к памяти могут быть свободно переупорядочены вокруг volatile операций доступа).

Одно существенное исключение - это Visual Studio, где, с настройками по умолчанию, любая volatile запись имеет семанику освобождения (release) и любое volatile чтение имеет семанку захвата (acquire) (MSDN (http://msdn.microsoft.com/en-us/library/12a04hfd(v=vs.100).aspx)), и поэтому операции с volatile могут быть использованы для межпоточной синхронизации. Стандартная же семантика volatile не подходит для многопоточного программирования, хотя её и достаточно, например, для связи с обработчиком сигнала (см. также `std::atomic_signal_fence`).

Источник — «https://ru.cppreference.com/mwiki/index.php?title=cpp/atomic/memory_order&oldid=43295»