Субъективный объективизм (http://www.scrutator.me/)

(https://twitter.com/ixsci) (http://www.scrutator.me/syndication.axd)

← Обзор книги C# 5.0 in a Nutshell: The Definitive Reference (/post/2015/11/24/review_csharp5_in_a_nutshell.aspx)

Обзор книги Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14 → (/post/2015/10/11/effective_modern_cpp.aspx)

Добро пожаловать в параллельный мир. Часть 5: Граница на замке (/post/2015/10/15/parallel_world_p5.aspx)

15. октября 2015 ixSci (http://www.scrutator.me/author/Adminaspx) разработка (/category/разработка.aspx) (0) (/post/2015/10/15/parallel_world_p5.aspx#comment)

Настоящая статья является прямым продолжением предыдущей (/post/2015/08/14/parallel-world-p4.aspx), поэтому, не растекаясь мыслью по древу, предлагаю сразу же перейти к тому, на чём мы остановились в четвёртой части. Если вы не читали предыдущей части, то я всячески рекомендую её прочесть. В процессе написания данной статья я подразумевал, что читатель уже знает всё то, что было написано в предыдущей, кроме того, в этой статье будут неоднократные отсылки к статье предыдущей.

Модель «потребления»

Закончив предыдущую статью на модели захвата-освобождения, настоящую мы начнём с модели *потребления*-освобождения. Данная модель схожа с моделью захвата-освобождения, но является ещё более слабой. Прежде чем мы разберём, что же ещё можно было ослабить в предыдущей

модели, давайте разберём, как мы можем получить модель потребления-освобождения. Для реализации данной модели нужно использовать операции над атомарными объектами со следующими маркерами:

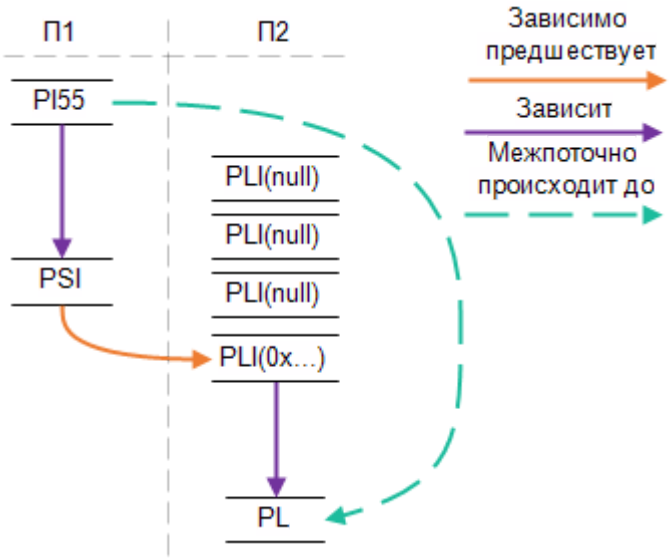
- std::memory_order_consume — этим членом можно маркировать операции, которые загружают значение атомарного объекта. Операции использующие данный маркер являются операциями «потребления».
- std::memory_order_release – этим членом можно маркировать операции записи данных в атомарный объект. Операции использующие данный маркер являются операциями «освобождения».

Как вы можете видеть, описание данной модели отличается лишь в том, что для операций загрузки вместо std::memory_order_acquire используется маркер std::memory_order_consume. Кроме того, в предыдущей модели, операция загрузки носила имя операции «захвата», тогда как в настоящей модели она называется операцией «потребления». Пусть вас не вводит в заблуждение эта терминология, главное понимать, какая разница лежит в сути этих операций. А точнее в сути явлений, которые описываются с помощью этих операций.

Как мы помним из модели захвата-освобождения, пара операций «захвата» и «освобождения» формировала отношение «синхронизируется с». В настоящей модели такое отношение не формируется, зато формируется другое — «зависимо предшествует»(dependency-ordered before). Как вы помните, «зависимо предшествует» как и «синхронизируется с» позволяет реализовать отношение «межпоточно происходит до», что, в свою очередь, позволяет синхронизировать различные точки исполнения программы в разных потоках. Прежде чем мы ударимся в дальнейшие рассуждения, давайте рассмотрим код, в котором мы сформируем отношение ЗП:

```
1  int integer = 0;
2  std::atomic<int*> atomicPtr{nullptr};
3  //...
4
5  void thread1()
6  {
7      integer = 55;
8      atomicPtr.store(&integer, std::memory_order_release);
9  }
10
11 void thread2()
12 {
13     int* ptr = nullptr;
14     while(!ptr)
15         ptr = atomicPtr.load(std::memory_order_consume);
16     assert(*ptr == 55);
17 }
```

Как вы уже догадались, assert гарантировано не сработает и следующая картинка призвана объяснить почему. Давайте условимся, что операция atomicPtr.store(&integer…) будет называется **PSI**, integer = 55; — **SI55**, ptr = atomicPtr.load — **PLI** и *ptr — **PL**.



(/image.axd?picture=dependency-ordered-before.png)

Картинка выше почти идентична той, что мы видели в описании отношения «синхронизируется с», которое реализуется парой операций «захвата»/«освобождения». И это не случайно — замените std::memory_order_consume на std::memory_order_acquire, в вышеприведённом коде, и всё будет работать по прежнему правильно. Это происходит потому, что операция захвата является более «тяжёлой» операцией, которая покрывает все случаи, где может быть использована операция «потребления». Обратное же неверно, операция «потребления» может быть использована только в тех частях программы, где есть зависимые данные. Из предыдущей (/post/2015/08/14/parallel-world-p4.aspx#dependency_ordered) статьи мы уже знаем, что за данные считаются зависимыми. Так, в примере выше, разыменовывание ptr явно зависит от сохранение значения в этот самый ptr, поэтому и формируется нужное нам отношение. Давайте рассмотрим следующий код:

```
1  int integer = 0;
2  char character ='A';
3  std::atomic<int*> atomicPtr{nullptr};
4  //...
5
6  void thread1()
7  {
8      character = 'Z';
9      integer = 55;
10     atomicPtr.store(&integer, std::memory_order_release);
11 }
12
13 void thread2()
14 {
```

```
15      int* ptr = nullptr;
16      while((ptr=atomicPtr.load(std::memory_order_consume)) == nullptr)
17          ;
18      assert(*ptr == 55);
19      assert(character == 'Z');
20  }
```

Так вот, у нас нет никакой гарантии, что assert(character == 'Z'); не сработает, потому что std::memory_order_consume влияет только на зависимые данные, а character никак не зависит от результата операций над atomicPtr. Поэтому будьте бдительны, если захотите использовать std::memory_order_consume.

Но в чём же преимущества std::memory_order_consume над std::memory_order_acquire? Ведь пока видны одни только проблемы, связанные с более сложным сценарием использования. Кто читал мою статью посвящённую барьерам памяти (/post/2015/05/16/memory_barriers.aspx), возможно узнал пример, который был взят из описания барьера для зависимых данных (/post/2015/05/16/memory_barriers.aspx#read_dep_barrier). Кроме того, я там упоминал про процессор DEC Alpha, чья модель, как наименее строгая, учитывалась при разработке многопоточной модели C++. Я не зря упоминаю статью по барьерам здесь, ведь std::memory_order_consume, по сути, является реализацией того. что я назвал «барьером для зависимых данных». Следовательно, к операции «потребления» применимо всё то, что написано для этого барьера(прочтите, если ещё этого не сделали). Т.к. мы пытаемся понять когда и зачем нам использовать std::memory_order_consume, для нас важен только один момент из той статьи: все известные мне процессоры, за исключением DEC Alpha, гарантируют корректность вышеприведённого кода без каких либо дополнительных усилий. А это значит, что операция промаркированная std::memory_order_consume превращается в простую операцию загрузки на абсолютном большинстве, существующих в настоящих момент, процессоров!

Вот вам и цель наличия данной модели в стандарте — в подавляющем большинстве случаев она даёт выигрыш в производительности, т.к. на стороне «потребления» не требует никаких дополнительных усилий. Но если так обстоят дела, то может вообще не использовать этот маркер, а просто выполнять «расслабленное» чтение на атомарном объекте? Большинство из нас пишет на популярных платформах, зачем нам думать над какими-то непонятными «альфами»? Так может рассуждать программист, но в данной логике есть ошибка. Маркеры операций над атомарными объектами превращаются не только в барьеры процессора(там где нужно), но и в барьеры компилятора (/post/2015/05/16/memory_barriers.aspx#compiler). Поэтому, для корректной работы модели, применение std::memory_order_consume является обязательным, даже если вы знаете, что на вашем процессоре эта операция не требует никаких дополнительных усилий. Вообще говоря, если уж начали использовать высокоуровневые конструкции C++, то лучше «забыть» о внутреннем устройстве и полностью подчиняться избранной модели. Так вы гарантируете правильно исполнение и облегчите жизнь тем, кто будет в дальнейшем читать ваш код.

Модель «беззакония»

Наконец мы добрались до последнего члена из перечисления std::memory_order и связанной с ним модели. Разговор в данном параграфе конечно же пойдёт о std::memory_order_relaxed и о том, какие гарантии предоставляются данной моделью — назовём её «ослабленной». А гарантий тут раз-два и обчёлся. Буквально. Гарантия первая: запись и чтение атомарного объекта являются атомарными(кэп?) операциями. И гарантия вторая: если поток **П1** прочитал значение **B**, которое было записано в этой объект, то при следующем чтении этого же объекта, в потоке **П1**, мы гарантировано прочитаем либо это же значение **B**, либо более позднее значение **L**, но никогда не прочитаем значение, которое было записано до **B**. Это, собственно та же самая гарантия, которую мы досконально разобрали в модели захвата-освобождения (/post/2015/08/14/parallel-world-p4.aspx#acq_rel).

Интересно, что обе эти гарантии присущи абсолютно всем операциями над атомарными объектами. И именно поэтому данная модель является «ослабленной» — больше нет никаких гарантий. Что это значит? Это значит, что операции помеченные std::memory_order_relaxed **невозможно использовать для синхронизации**. Такие операции годятся лишь тогда, когда относительный порядок нам совершенно не важен. Нам важна лишь атомарность. Естественно, являясь наименее строгой моделью, настоящая модель, при прочих равных, может давать наибольшую производительность. Так, операции чтения и записи должны быть атомарными, но никаких барьеров памяти они не требуют.

Где использовать такие операции? Там где вам важен лишь факт, что переменная установлена. Пример:

```
1  std::atomic<bool> stopWorking{false};
2  //...
3
4  void thread1()
5  {
6      stopWorking.store(true, std::memory_order_relaxed);
7  }
8
9  void thread2()
10 {
11     while(!stopWorking.load(std::memory_order_relaxed))
12     {
13         // Чего-то делаем
14     }
15 }
```

Здесь мы можем значительно выиграть в чтении в цикле while, т.к. используем наиболее производительную операцию. Но важно помнить, что при таком варианте, данные, которые используются внутри цикла никак не должны зависеть от выставленного флага, т.е. выставленный флаг означает лишь то, что он выставлен. Ни о каких других событиях данный факт не сигнализирует.

Также данный маркер используется в тех случаях, когда какая-то другая сущность используется для синхронизации. Чтобы не плодить толпы «тяжёлых» операций. Пример:

```
std::atomic<int> first{0};
std::atomic<int> second{0};
std::atomic<int> third{0};
std::atomic<bool> guard{false};
//...

void thread1()
{
    first.store(1, std::memory_order_relaxed);
    second.store(2, std::memory_order_relaxed);
    third.store(3, std::memory_order_relaxed);
    guard.store(true, std::memory_order_release);
}

void thread2()
{
    while(guard.load(std::memory_order_acquire))
        ;
    assert(first.load(std::memory_order_relaxed) == 1);
    assert(second.load(std::memory_order_relaxed) == 2);
    assert(third.load(std::memory_order_relaxed) == 3);
}
```

Ещё один пример, мы увидим в следующем параграфе. А закончить этот параграф хочется ещё одним напоминанием: используйте std::memory_order_relaxed тогда и только тогда, когда синхронизация абсолютно не важна. Если есть сомнения, тогда лучше вообще не используйте данный маркер.

Наконец, подводя итог описанию моделей, хочется привести ссылку (https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html) на замечательный документ, в котором собраны варианты «превращения» атомарных операций C++ в ассемблер наиболее популярных архитектур, для всех моделей доступных в C++. Зная стоимость той или иной инструкции, можно оценить какой прирост может дать смена модели в исполняемом коде.

ЧИЗ

Рассмотрев различные модели упорядочивания операций, пришло время рассмотреть специфичные операции, которые имеют свои, особенные гарантии, которые не зависят от используемой модели. Такими операциями являются операции *чтения-изменения-записи(ЧИЗ(RMW))* и они занимают довольное интересное место в C++. Прежде чем мы рассмотрим, что же такого в них интересного, давайте разберёмся какие операции являются ЧИЗ-операциями. В общем виде, все ЧИЗ-операции явно помечены в стандарте C++, поэтому если есть сомнения, то прочитав их описание все сомнения должны развеяться, искать стоит фразу «read-modify-write». Здесь я приведу группы операций, которые стандарт C++14 считает ЧИЗ-операциями:

- Группа операций exchange
- Группа операций compare_exchange_*
- Группа операций fetch_*
- Группа операций test_and_set(только для atomic_flag)

Вот и всё, в сухом остатке имеем 4 группы операций. Почему я назвал их «группами»? Просто в каждой такой группе есть несколько перегруженных версий, а также версий с разными именами, которые отличаются в поведении. К примеру, есть две операции fetch_add и fetch_sub, первая для сложения, вторая для вычитания. Я считаю, что нет смысла описывать здесь каждую операцию, поэтому просто привёл здесь группы операций.

Итак, как ясно из самого названия операций, они все происходят по следующей схеме:

1. Сначала происходит считывание значения атомарного объекта из памяти.
2. Затем что-то может быть выполнено над этим значением.
3. Наконец новое(оно может не отличаться от старого!) значение записывается в память.

Так вот, все эти три шага гарантировано являются неразделимым, т.е. не может оказаться так, что 1 шаг выполнен, потом вмешалась операция из другого потока, которая что-то новое сохранила в память, и мы в результате записали изменённое устаревшее значение. Нет, такого быть не может. ЧИЗ-операции являются атомарными.

Правда 2 шаг в них вовсе не обязателен, т.е. нет никакой необходимости изменять старое значение, мы может прочитать значение, и его же записать обратно:

```
1 std::atomic_int magicNumber{54};
2 magicNumber.fetch_add(0);
```

В коде выше, мы просто прочитали значение 54(1), прибавили к нему 0(2), а затем записали обратно в память(3). Как вы можете видеть, шаг 2 здесь хоть и присутствует, но абсолютно бесполезен. Давайте разберём другой пример:

```
1 std::atomic_int magicNumber{54};
```

```
2  magicNumber.fetch_add(1);
```

Теперь мы имеем полноценный 2 шаг, т.к. к прочитанному значению 54, на втором шаге мы прибавляем единица и уже полученное значение(55) записываем в память. Но есть случаи, когда второго шага нет вообще, даже бесполезного:

```
1  std::atomic_int magicNumber{54};
2  int value = 54;
3  magicNumber.compare_exchange_strong(value, 100500);
```

В вышеприведённом коде творится следующее: сначала считывается значение magiNumber(54) и сравнивается с value(54) — это шаг 1. Если они равны(а они равны), тогда происходит запись значения 100500 в magicNumber — это шаг 3. Как вы можете видеть, мы не изменяли старое значение, мы просто записали новое, которое не является производным от старого. В этом случае получается, что мы добавили сравнение к шагу один и устранили шаг два совершенно, получив Ч[И]З операцию. Таким образом «И(изменение)» часть ЧИЗ не является обязательной.

Всегда начеку

Разобравшись с тем, что же такое ЧИЗ-операции, пришло время поговорить об их особенностях. Как вы наверное знаете, все ЧИЗ-операции, являясь атомарными операциями, принимают в качестве аргумента маркер из перечисления std::memory_order. Мы уже видели, что в зависимости от переданного маркера, для атомарных операций, может меняться поведение, которое относится не только к синхронизации окружающего кода. Так, мы видели (/post/2015/08/14/parallel-world-p4.aspx#seq_read_order), что для операций чтения, помеченных std::memory_order_seq_cst, всегда будет загружено последнее значение, тогда как для всех других маркеров такой гарантии не существует.

В этом смысле ЧИЗ-операции стоят особняком, т.к. маркеры std::memory_order, переданные им в качестве аргумента, совершенно не влияют на саму операцию, а влияют лишь на окружающий код. Что это значит? Это значит, что ЧИЗ-операция **всегда и при любом маркере**, на первом шаге, загрузит самое последнее значение атомарного объекта. Таким образом, любая ЧИЗ-операция, в части загрузки, ведёт себя точно так же, как простая операция чтения в модели последовательной согласованности, т.е. помеченная маркером std::memory_order_seq_cst. Это уникальное свойство, без которого немыслимы ЧИЗ-операции.

Давайте предположим, что это было бы не так, и мы бы не имели такой гарантии, тогда имея следующий код:

```
1  #include <atomic>
2  #include <thread>
3  #include <cassert>
```

```
4
5  std::atomic<int> counter{0};
6
7  void increment()
8  {
9      counter.fetch_add(1, std::memory_order_relaxed);
10 }
11
12 int main()
13 {
14     std::thread firstThread(&increment);
15     std::thread secondThread(&increment);
16     firstThread.join();
17     secondThread.join();
18     assert(counter.load() == 2);
19     return 0;
20 }
```

Мы бы не могли гарантировать, что assert не сработает. Но тогда невозможно было бы написать элементарный счётчик, который бы корректно работал в многопоточной среде, без использования блокирующих примитивов. Так что мы имеем эту гарантию и она является краеугольным местом многих(если не всех) мало-мальски сложных неблокирующих структур данных. Кстати, в примере выше приведён ещё один пример, когда std::memory_order_relaxed может быть использован. Это такой канонический пример использования данного маркера, т.к. он наиболее показателен — мы имеем задачу увеличения переменной, и у нас нет задачи что-либо синхронизировать, поэтому «ослабленная» модель подходит нам идеально.

Дабы усвоить данный материал ещё лучше, приведу ещё один пример, который, в своё время, не давал мне покоя:

```
1  class SpinLock
2  {
3      std::atomic_flag locked;
4  public:
5      SpinLock() :
6          locked{ATOMIC_FLAG_INIT}
7      {
8      }
9      void lock()
10     {
11         while(locked.test_and_set(std::memory_order_acquire));
12     }
13     void unlock()
14     {
15         locked.clear(std::memory_order_release);
```

```
    }
};
```

Это реализация простенького SpinLock из книги C++ Concurrency in Action: Practical Multithreading (/post/2014/10/03/cpp_concurrency_in_action.aspx). Так вот, имея ЧИЗ-операцию в функции lock() так и подмывает использовать маркер std::memory_order_acq_rel, ведь мы сначала читаем, а потом записываем, поэтому мы должны наше чтение синхронизировать с предыдущей записью, и нашу запись синхронизировать с последующим чтением, правильно? Нет. Это рассуждение ошибочно, т.к. нам не нужно синхронизировать запись из ЧИЗ-операции в функции lock() с чем бы то ни было! Наша задача написать аналог мьютекса, а это значит, что для защиты блока нам нужна базовая модель: «захват»(acquire) в lock() и «освобождение»(release) в unlock(). Поэтому std::memory_order_acq_rel тут избыточен, достаточно пометить операцию как std::memory_order_acquire. Остальное гарантируется тем фактом, что test_and_set() всегда прочитает последнее значение флага и одновременный вход в критическую секцию будет невозможен.

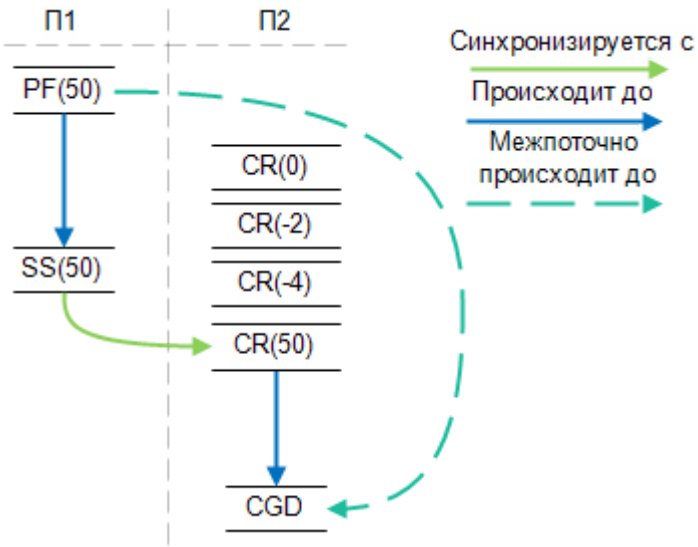
Последовательность освобождения

Прежде чем мы перейдём к описанию того, чем является последовательность освобождения, мы рассмотрим пример, из которого станет ясно, зачем она вообще нужна. Давайте представим, что у нас есть «поставщик», который выполняется в одном потоке и поставляет некоторые данные; так же у нас есть «получатели», которые представлены в нескольких экземплярах, и каждый экземпляр выполняется в своём потоке. Наша задача сделать так, чтобы «получатели» могли узнать когда новые данные поступили и могли изъять эти данные, не мешая друг другу. Разумеется, в результирующем коде у нас не должно быть никаких «гонок». Итак, вот один из вариантов реализации(для простоты используем только 2-х «получателей»):

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

using namespace std;

atomic<int> semaphore{0};
vector<string> preciousData;

void provider()
{
    //...
    // Заполняем данными наш вектор preciousData
    preciousData.push_back("Top Secret String");
    //...
    // Данные готовы, пора сообщить о них
    semaphore.store(preciousData.size(), memory_order_release);
```

```
18  }
19
20  void client()
21  {
22      int dataIndex = 0;
23      while(dataIndex < 0)
24          dataIndex = semaphore.fetch_sub(1, memory_order_acquire);
25      // Получатель использует полученные данные
26      auto data = preciousData[dataIndex];
27      //...
28  }
29
30  int main()
31  {
32      std::thread firstThread(&provider);
33      std::thread secondThread(&client);
34      std::thread thirdThread(&client);
35      firstThread.join();
36      secondThread.join();
37      thirdThread.join();
38      return 0;
39  }
```

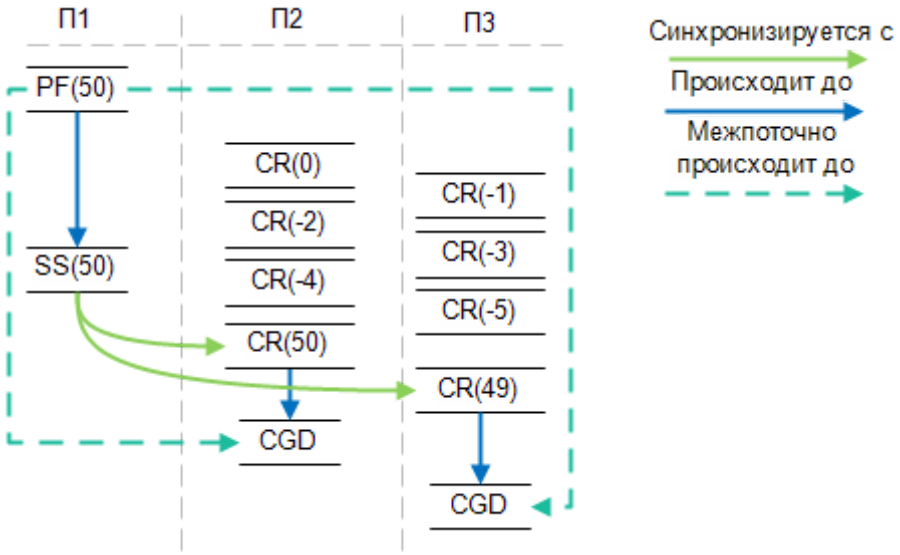
Итак, имея вышеприведённый код, давайте разберём его в терминах отношений (/post/2015/08/14/parallel-world-p4.aspx#relations). Пусть наш «поставщик» поставил **50** строк и пусть поток номер 2(**П2**), успел к кормушке первым, т.е. он первым увидел значение **50**, тогда мы будем иметь следующую картину:



(/image.axd?picture=release_sequence_simple.png)

В картинке выше **PF** – это код заполнения вектора, **SS** это операция освобождения(release) выполненная над semaphore, **CR** это «читательная» часть ЧИЗ-операции fetch_sub и, наконец, **CGD** это получение данных на стороне клиента(preciousData[dataIndex]) . Пока всё в порядке, прям как в

учебнике. Идём дальше, и приходим к тому, что второй поток подоспел к кормушке и увидел значение semaphore, равное **49**-и:



(/image.axd?picture=release_sequence_extended.png)

И снова всё хорошо! Чтение в третьем потоке, получив значение **49**(мы помним, что ЧИЗ-операция всегда видит последнее значение) *синхронизировалась с* записью в потоке первом; всё логично и правильно. За исключением того, что имеющихся у нас гарантий не достаточно, для подобных рассуждений. Вспомните, что означает отношение «синхронизируется с» (/post/2015/08/14/parallel-world-p4.aspx#sync_with) — оно срабатывает тогда, когда операция «захвата» во время исполнения получает результат ранее выполненной операции «освобождения», с которой и происходит синхронизация. В нашем случае операцией «освобождения» мы записали значение **50**, но операцией «захвата» прочитал значение **49**, поэтому никакого отношения здесь наступить не может!

Отношений между двумя «получателями» тоже не происходит, ведь в них мы имеем операции «захвата», которые между собой не синхронизируются. «Так надо просто заменить std::memory_order_acquire на std::memory_order_acq_rel, и тогда установится отношение „синхронизируется с“ между двумя „получателями“» — может заметить внимательный читатель, и будет прав, этот вариант решил бы проблему. Но это так же внесло бы излишнюю синхронизацию между двумя «получателями», в которой нет никакой нужды(не зря же я отношение нарисовал на картинке!).

Так вот, всего один абзац назад я соврал, точнее сказал не всю правду, про то, что отношения «синхронизируется с» не происходит между записью **50** и чтением **49**. В общем случае это действительно так, но наш случай не общий, а вполне конкретный, и он описывается с помощью понятия *последовательности освобождения*(*release sequence*). Пришло время дать определение *последовательности освобождения* — это последовательность операций над атомарным объектом, которая начинается с операции «освобождения» и оканчивается операцией «захвата», при этом, между этими двумя операциями могут находится следующие операции над тем же атомарным объектом:

- Любые(с любым маркером) операции записи, которые выполняются в том же потоке, что и операции «освобождения», но идущие после неё.
- ЧИЗ-операции, которые выполняются в **любых потоках**.

Давайте разберём пример:

П1	П2	П3	...	ПN
ReleaseStore(30)				
	Increment(31)			
RelaxedStore(33)		Decrement(32)		
		Decrement(31)		
RelaxedStore(56)				
	Increment(57)			
	RelaxedStore(60)			Acquire(57) Acquire(60)

(/image.axd?picture=release_sequence_example_1.png)

На картинке выше мы можем видеть, что мы используем «ослабленную» модель, для записи значений **{33, 56}** в **П1**, а также для записи значения **{60}** в **П2**. Также мы используем ЧИЗ-операции инкремента в **П2**, посредством которых получаем значения **{31, 57}**. В **П3** у нас находятся операции декремента, которые дают нам значения **{32, 31}**. Таким образом, в **ПN**, операция «освобождения» может видеть одно из следующих значений: **{30, 33, 56, 60, 31, 57, 32, 31}**. Теперь внимание: отношение «синхронизируется с» наступит при чтении любого значения, из предыдущего списка, кроме числа **60**. Это происходит потому, что «ослабленная» операция записи, находящаяся в потоке, отличном от операции «освобождения» не участвует в последовательности освобождения! Как вы видите на картинке выше, Acquire(60) выделено цветом, отличным от всех других операций, т.к. эта операция не входит ни в какую последовательности, равно как и StoreRelaxed(60) не входит ни в какую последовательность и не имеет никакого *отношения* с Acquire(60). Графически, мы имеем операции 3-х цветов, которые определяют принадлежность к «группе».

Таким образом, отношение «синхронизируется с» формируется если операция «захвата» встречает **любое** значение из последовательности освобождения. В сущности, когда мы имеем всего две операции, пару «освобождения»-«захвата», то мы имеем вырожденный случай последовательности освобождения, в котором нет промежуточных операций.

Исходя из всего вышенаписанного, теперь должно быть понятно почему я нарисовал отношении «синхронизируется с», при том, что мы получили значение **49**, а не **50** — **49** является результатом промежуточной ЧИЗ-операции, которая входит в последовательность освобождения. Последовательность освобождения может быть сколь-угодно длиной.

Данное понятие может показаться несколько сложным, но, на самом деле, оно интуитивно понятно, когда к нему привыкнешь. Главное помнить из чего состоит последовательность и какие гарантии она даёт. Это позволит быть более уверенным, при анализе неблокирующего алгоритма. На этом мы завершаем разговор о ЧИЗ-операциях и переходим к

Изгородь

Вплоть до настоящего момента, мы говорили только об атомарных объектах и операциях над ними, но в C++ существует ещё одна группа операций, которая позволяет синхронизировать доступ. Эти операции получили название «*заборов*»(*fences*).

> «Забор» звучит, безусловно, забавно, но называть их «барьерами» у меня не поворачивается рука, т.к. это уже настолько перегруженный термин, что дальше некуда. Поэтому будет «забор», тем более, что это позволяет легко отличать его от «барьера».

Вышеозначенная «группа» операций состоит из одной функции std::atomic_thread_fence(std::memory_order), которая превращается в «заборы» различного типа, в зависимости от маркера, переданного в качестве аргумента. В отличии от всех атомарных операций, рассмотренных ранее, настоящая операция не имеет значения по умолчанию и требует явного указания одного из значений std::memory_order. Давайте посмотрим, во что превращается наш «забор», в зависимости от переданного маркера:

- std::memory_order_relaxed – превращает операцию в «пустое место», т.е. не имеет никакого эффекта.
- std::memory_order_acquire – превращает операцию в «забор захвата»
- std::memory_order_consume – превращает операцию в «забор захвата»
- std::memory_order_release – превращает операцию в «забор освобождения»
- std::memory_order_acq_rel – превращает операцию в «забор захвата-освобождения»
- std::memory_order_seq_cst – превращает операцию в «забор захвата-освобождения». Дополнительно прилагаются гарантии, которые присущи модели последовательной согласованности.

Как вы можете видеть, всё вышенаписанное ничем не отличается от того, что мы рассматривали касательно операций над атомарными объектами. Действительно, давайте рассмотрим как можно переписать код с атомарными операциями, на операции с «забором». Начнём с уже знакомых нам операций:

```
1  std::atomic_int atomicInteger{0};
2  int simpleInteger{0};
3  int oneMoreInteger{0};
```

```
4  std::atomic_bool flag{false};
5
6  void thread1()
7  {
8      simpleInteger = 55;
9      atomicInteger.store(66, std::memory_order_relaxed);
10     flag.store(true, std::memory_order_release);
11     oneMoreInteger = 77;
12 }
13
14 void thread2()
15 {
16     while(flag.load(std::memory_order_acquire) != true)
17         ;
18     assert(atomicInteger.load(std::memory_order_relaxed) == 66);
19     assert(simpleInteger == 55);
20 }
```

В этом коде вам должно быть всё понятно, поэтому я не буду его комментировать. Давайте перепишем его с применением «заборов»:

```
1  std::atomic_int atomicInteger{0};
2  int simpleInteger{0};
3  int oneMoreInteger{0};
4  std::atomic_bool flag{false};
5
6  void thread1()
7  {
8      simpleInteger = 55;
9      atomicInteger.store(66, std::memory_order_relaxed);
10     flag.store(true, std::memory_order_relaxed);
11     std::atomic_thread_fence(std::memory_order_release);
12     oneMoreInteger = 77;
13 }
14
15 void thread2()
16 {
17     while(flag.load(std::memory_order_relaxed) != true)
18         ;
19     std::atomic_thread_fence(std::memory_order_acquire);
20     assert(atomicInteger.load(std::memory_order_relaxed) == 66);
21     assert(simpleInteger == 55);
22 }
```

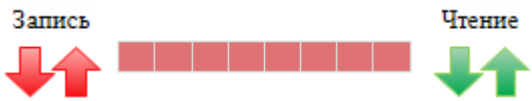
Итак, мы заменили операции «освобождения» и «захвата» flag на «ослабленную» запись и чтение. Кроме того, мы добавили два «забора». Таким нехитрым образом мы переписали наш код. Есть ли разница, между этими двумя подходами? Безусловно. Давайте разберём каждое отличие.

Глобальность

Как мы уже много раз говорили операции над атомарными объектами синхронизируются только между операциями над одним и тем же объектом. Но «заборы» не имеют никаких ассоциированных с ними объектов, поэтому между ними происходя «глобальная отношения», в этом плане «заборы» очень похожи на «барьеры» (/post/2015/05/16/memory_barriers.aspx)(похожи, но ими не являются!). В вышеприведённом коде мы имеем «ослабленную» запись между flag в потоке **1** и «ослабленное»-же чтение в потоке **2**. Т.е. сами по себе эти две операции не формируют отношения «синхронизируюсь с», для этого там есть «заборы». Таким образом, отношение **формируется симбиозом атомарной операции и «забора»**, по отдельности ни те, ни другие никакого отношения не формируют. «Забор захвата» находится после чтения флага, а запись флага находится до «забора освобождения», поэтому при чтении true во flag, «забор освобождения» синхронизируется с «забором захвата», перед которым это самое true и было записано во flag. Т.е. мы как бы разделили flag.store(true, std::memory_order_release); на атомарную операцию записи(«ослабленную») и на указание синхронизации(«забор»). Но подобное преобразование возможно лишь в том случае, если «заборы» не будут пропускать операции сквозь себя.

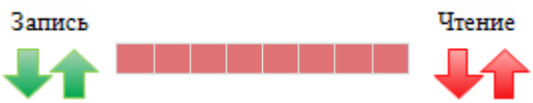
Заборонено!

Чтобы быть полезными, заборы должны предотвращать «просачивание» операций чтения и записи сквозь них, ведь если бы заборы не имели этого свойства, тогда ни о какой синхронизации с их помощью, не могло бы быть и речи. Но, разумеется, «заборы» обладают такими свойствами. Давайте рассмотрим, какие ограничения вводит тот или иной забор на окружающие его операции. Имея «забор освобождения» ни одна операция записи не может быть перенесена через забор в любую сторону, или графически:



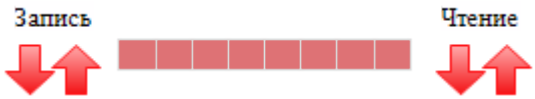
(/image.axd?picture=release_fence.png)

Если вы читали мою статью по барьерам, то вам уже очевидно, что данный «забор» по поведению идентичен «барьеру записи» (/post/2015/05/16/memory_barriers.aspx#write_barrier). Давайте теперь рассмотрим «забор захвата» — он не пропускает сквозь себя операции чтения:



(/image.axd?picture=acquire_fence.png)

Очевидно, что его поведение идентично «барьеру чтения» (/post/2015/05/16/memory_barriers.aspx#read_barrier). Для случая, когда используется «забор захвата-освобождения», картинки совмещаются и получается, что операции чтения находящиеся до «забора» не могут быть перемешаны с операциями чтения после оного. Тоже самое с операциями записи. Но(!) нет никакого запрета на перемешивание операций чтения с операциями записи. Картинка:



(/image.axd?picture=acq_rel_fence.png)

Наконец, при использовании std::memory_order_seq_cst с «забором» мы получаем «полный барьер» (/post/2015/05/16/memory_barriers.aspx#full_barrier) — никакие операции до «забора», не могут быть перемешаны с операциями после него. Правда здесь стоит указать, всё-же, что стандарт C++ оперирует «заборами», а не «барьерами» и всё, что касается описания работы «заборов» приводится в контексте одного атомарного объекта. К примеру, если мы имеем следующий код:

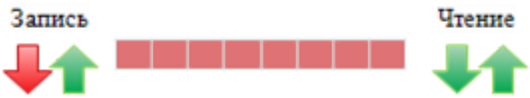
```
1  std::atomic_int first;
2  std::atomic_int second;
3
4  void thread1()
5  {
6      first.store(55, std::memory_order_relaxed);
7      std::atomic_thread_fence(std::memory_order_seq_cst);
8      second.store(11, std::memory_order_relaxed);
9  }
```

Исходя из текста стандарта, на вышеозначенный код не накладывается никаких ограничений, т.к. стандарт оперирует операциями над *одним* атомарным объектом, когда говорит о «заборах», с другой стороны, если мы имеем вместо этого «забора» полноценный «полный барьер», тогда мы имеем ограничения на перемещение операций сквозь него, вне зависимости от того одни ли объекты находятся по разные стороны «барьера» или нет. Но тут нужно понимать, что компилятор не может(теоретически это возможно, но практически в этом нет никакого смысла) проводить такой глубокий анализ, поэтому наиболее вероятно, что при использовании вышеозначенных «заборов» он будет вставлять в результирующий код соответствующие «барьеры», безотносительно того, что в некоторых случаях они как-бы не требуются. Подтверждением вышеозначенной логики может служить следующая ссылка (https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html), пройдя по которой вы можете увидеть во что, наиболее вероятно, превратятся те или иные «заборы» на разных архитектурах.

Ещё раз: хотя вышеозначенное соответствие «заборов» и «барьеров» может иметь место, не стоит забывать, что понятие «барьера» в стандарте отсутствует, и C++ код стоит анализировать с точки зрения гарантий стандарта, а не предположений о том, какой «барьер» будет использован. Благо модель C++ является довольно строгой и понятной, когда к ней привыкнешь.

Т.к. мы рассматриваем отличие «заборов» от простых атомарных операций, то нельзя не упомянуть, что для атомарных операций нужно меньше гарантий, а значит они налагают на окружающий код меньше ограничений. Это, чисто теоретически, может вылиться в более эффективный код при использовании атомарных операций без «заборов». Действительно, мы уже видели какие ограничения налагают «заборы» на окружающий код, давайте рассмотрим чем отличаются атомарные операции:

Имея атомарную операцию «освобождения» следующие ограничения налагаются на окружающий код: ни одна операция записи, предшествующая операции «освобождения» не должна быть перемещена за(после) неё. С другой стороны, операции записи, которые находятся после операции «освобождения», могут быть перенесены за(до) неё. Графически:



(/image.axd?picture=release_operation.png)

Как вы можете видеть операция «освобождения» налагает вполовину меньше ограничений на окружающий код, чем это делает «забор освобождения». По другим типам операций и «заборов» может быть проведён тот же анализ и результат будет схож. Я не буду этого делать, т.к. текста уже слишком много, а смысла в повторении одного и того же не очень много. Таким образом, как мы увидели, использование атомарных операций может дать определённый выигрыш перед использованием «заборов».

volatile

Чтобы понять какое отношение volatile имеет к рассматриваемой нами теме, предлагаю провести сравнение между volatile и атомарным объектом, над которым выполняются операции в «ослабленной» модели, с применением std::memory_order_relaxed. Это сравнение будет показательным, ведь «ослабленная» модель представляет наименьшее количество гарантий, а следовательно мы сравниваем volatile с сущностью, которая практически ничего не гарантирует.

Итак, давайте вспомним, что за гарантии мы имеем для атомарного объекта в «ослабленной» модели:

1. Любая операция является атомарной, ни один поток не может наблюдать частично записанный атомарный объект.
2. Операция чтения не может прочитать значение объекта, старее чем то, что уже было

прочитано. Никаких гарантий того, что будет загружено последнее записанное значение не существует.
3. Операции над атомарным объектом не влияют на операции, которые окружают этот объект. Т.е. данный объект не может быть использован для синхронизации с другими операциями.

Теперь давайте рассмотрим, что из вышеназванного гарантируется volatile. Первое, атомарность, — volatile объект не является атомарным и, следовательно, различные потоки могут наблюдать частично записанный объект. Второе, компилятору запрещена любая оптимизация чтения или записи volatile объекта, а это значит, что при чтении всегда будет прочитано последнее записанное значение. Ну и наконец третье — операции над volatile объектом, как и в случае выше, не влияют на сторонние операции, а значит volatile является «вещью в себе» и не может никаким образом быть использован для синхронизации.

Таким образом, вы можете видеть, что volatile никоим образом не является механизмом схожим с атомарными объектами, **он вообще не предназначен для межпоточного взаимодействия**. О volatile можно и нужно рассуждать лишь как о регистре некоторого устройства. Т.е. таком участке памяти, которое может быть обновлено без участия процессора, а значит минуя все механизмы узнавания об этом событии. В этих случаях используется volatile. Никогда не используйте volatile для межпоточного взаимодействия, даже если ваш компилятор добавляет некоторые гарантии, которых нет в стандарте, для этого есть std::atomic.

Итог

Данной статьёй я завершаю цикл описания возможностей C++ для межпоточной разработки. Я надеюсь, что смог донести до вас все те нюансы, которые могут быть непонятно при первом знакомстве с новыми возможностями C++. Т.к. тема затронутая в цикле весьма сложна и обширна, возможно я уделил каким-то моментам слишком мало внимания, где-то плохо объяснил и т.п. В любом случае, если это имело место или же у вас просто имеется идея по тому, что ещё стоит раскрыть в смежных темах — пишите в комментариях.

Остальные статьи цикла:

Часть 1: Мир многопоточный (/post/2012/04/04/parallel-world-p1.aspx)

Часть 2: Мир асинхронный (/post/2012/06/03/parallel-world-p2.aspx)

Часть 3: Единый и Неделимый (/post/2012/08/28/parallel-world-p3.aspx)

Часть 4: Порядки и отношения (/post/2015/08/14/parallel-world-p4.aspx)

Метки : C++ (http://www.scrutator.me/?tag=C%2b%2b), multithreading (http://www.scrutator.me/?

tag=multithreading), atomic (http://www.scrutator.me/?tag=atomic), fences (http://www.scrutator.me/?tag=fences), memory barriers (http://www.scrutator.me/?tag=memory+barriers), memory model (http://www.scrutator.me/?tag=memory+model)

Текущий рейтинг: 5.0 (2 голосов)

Похожие записи

Добро пожаловать в параллельный мир. Часть 5: Граница на замке (/post/2015/10/15/parallel_world_p5.aspx)

Настоящая статья является прямым продолжением предыдущей, поэтому, не растекаясь мыслью по древу, пр...

Вся правда об указателях. Часть 1: Вводная (/post/2015/11/26/pointers_demystified_p1.aspx)

С самого начала моей карьеры, как программиста, я постоянно встречаю людей(лично или в сети), которы...

Добро пожаловать в параллельный мир. Часть 4: Порядки и отношения (/post/2015/08/14/parallel-world-p4.aspx)

Не прошло и 3-х лет с последней статьи в цикле, как я решился на написание обещанной четвёртой. Все ...

Последние записи

Вся правда об указателях. Часть 3: Завершающая (/post/2016/03/30/pointers_demystified_p3.aspx)

Рейтинг: 3.5 / 4

Обзор книги Dependency Injection in .NET (/post/2016/03/19/review_dep_injection.aspx)

Нет оценок

Вся правда об указателях. Часть 2: Памятная (/post/2015/12/30/pointers_demystified_p2.aspx)

Рейтинг: 5 / 2

Обзор книги Applied Cryptography (/post/2015/12/25/review_applied_cryptography.aspx)
Рейтинг: 5 / 1

Вся правда об указателях. Часть 1: Вводная (/post/2015/11/26/pointers_demystified_p1.aspx)
Рейтинг: 5 / 3

Обзор книги C# 5.0 in a Nutshell: The Definitive Reference (/post/2015/11/24/review_csharp5_in_a_nutshell.aspx)
Нет оценок

Добро пожаловать в параллельный мир. Часть 5: Граница на замке (/post/2015/10/15/parallel_world_p5.aspx)
Рейтинг: 5 / 2

Список категорий

(/category/feed/книги.aspx) книги (15) (/category/книги.aspx)

(/category/feed/разработка.aspx) разработка (43) (/category/разработка.aspx)

Список записей по годам/месяцам

2011

2012

2013

2014

2015

2016

Март (/2016/03/default.aspx) (2)