



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



(<https://twitter.com/ixsci>)



(<http://www.scrutator.me/syndication.axd>)

← 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)

Обзор книги Modern Operating Systems → (/post/2015/08/12/modern_operating_systems.aspx)

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

📅 14. августа 2015 👤 ixSci (<http://www.scrutator.me/author/Admin.aspx>) 📁 разработка (</category/разработка.aspx>)

💬 (0) (</post/2015/08/14/parallel-world-p4.aspx#comment>)

Не прошло и 3-х лет с последней (</post/2012/08/28/parallel-world-p3.aspx>) статьи в цикле, как я решился на написание обещанной четвёртой. Все мы знаем, что обещанного ждут три года, поэтому я, безусловно, выдержал марку. В настоящей статье речь пойдёт об атомарных объектах и операциях над ними. Какой-то материал будет частично повторять уже сказанное в 3-й части, но это не потому, что мне писать не о чем, а потому, что эти важные сведения должны, по моему мнению, быть сведены в одном месте. Более того, я надеюсь, что в этот раз я смогу донести все моменты более понятно, чем сделал это 3 года назад. Ну да хватит предисловий – давайте ещё раз окунёмся в обновлённую межпоточную модель исполнения C++.

Порядок и спокойствие

Порядок изменения

Итак, разговор о модели исполнения мы начнём с описания *порядка изменения(modification order)*. Порядок изменения(ПИ) присущ каждому атомарному объекту представленному в C++, т.е. объектам типов `std::atomic<T>` и `std::atomic_flag`. Важно понимать, что ПИ присущ только атомарным объектам; никакие другие объекты не имеют детерминированного ПИ, тогда как для любого атомарного объекта можно получить онный. Что же из себя представляет ПИ? Давайте представим, что у нас есть атомарный объект, который изменяется в разных частях программы(может быть в разных потоках, мы этого не знаем):

?

```
1  std::atomic_int variable{0};
2  //...
3  variable = 7;
4  //...
5  variable = 15;
6  //...
7  variable = 42;
8  //...
9  variable = 777;
10 //...
11 variable = 88;
12 //...
13 variable = 356;
14 //...
15 variable = 11;
```

Что мы можем сказать о ПИ атомарного объекта `variable`? Мы совершенно точно можем утверждать, что первоначальным значением оно является 0, а также мы можем утверждать, что в определенные моменты времени данный объект принимал одно из следующих значений: 7, 15, 42, 777, 88, 356, 11. Теперь, исходя из нашего условия, что мы не знаем ничего о том, при каких условиях наш объект был изменён, мы можем считать, что ПИ будет обязательно состоять из вышеприведенных цифр, но порядок этих цифр не определён. Другими словами, для `variable` мы имеем столько возможных ПИ, сколько всевозможных перестановок может быть сделано для вышеприведённых цифр, т.е. для нашего случая мы имеем 8! возможных ПИ. Разумеется при одном исполнении программы только один ПИ может существовать для одного объекта. К примеру, для `variable` может иметь место следующий порядок, при одном исполнении программы:

42	15	7	356	11	777	88
----	----	---	-----	----	-----	----

Порядок формируется слева-направо, где каждая ячейка таблицы говорит нам какое значение `variable` имел в определенный момент исполнения программы. Слева находятся более старые значения, справа, соответственно, более новые. Нужно понимать, что при последующем запуске программы ПИ может быть совершенно другим, т.к. мы не наложили никаких ограничений. В реальной жизни всё будет немного по другому, мы будем больше знать о том, когда и при каких

условиях изменятся атомарный объект, а значит количество возможных ПИ будет уменьшено. Но в общем случае каждый атомарный объект может иметь множество порядков изменения, которые варьируются от запуска к запуску.

Основной и очень важной гарантией, которую даёт нам C++ является тот факт, что каждое изменение в ПИ является неделимым и самодостаточным, т.е. 2 изменения не могут быть сделаны одновременно. Всегда одно изменение предшествует другому и поэтому всегда есть порядок – никогда не может быть ситуации, при которой 2 значения могут претендовать на одну ячейку в таблице(если говорить языком данной статьи).

Влияние на порядок изменения

Рассмотрев, что такое порядок изменения, приступим к рассмотрению того, как мы, программисты, можем повлиять на него. Т.е. как мы можем ограничить количество существующих ПИ для атомарного объекта. Зачем это нужно? Чем больше мы можем судить о том, в каком порядке изменяется наш объект, тем больше мы можем сказать о том как наша программа выполняется, и тем больше мы имеем власти над событиями в программе.

Начнём мы, пожалуй, с понятия, которое привязано не только к атомарным объектам. Стандарт C++ определяет отношение *предшествования*(*sequenced before*). Это отношение применимо только в рамках одного потока исполнения, и означает буквально следующее: имея два полных выражения **A** и **B**, если **B** находится после **A** в исходном коде, тогда выражение **A** *предшествует* выражению **B**, т.е. начинает и заканчивает своё исполнение раньше, чем **B** начинает своё.

Я не буду вдаваться в подробности, что такое «полное выражение», в большинстве случаев это выражение, которое заканчивается «;»(точкой с запятой). Остальные случаи, хотя и интересны, но прямого отношения к статье не имеют. Кроме того, далее в статье под «выражением» будет подразумеваться «полное выражение», если не указано явно обратное.

Или кодом:

```
1 nonAtomic = 6; // A
2 variable = 7; // B
```



Как вы можете видеть, изменение простого объекта `nonAtomic` *предшествует* изменению атомарного объекта `variable`. Всё это логично и интуитивно понятно. Как же это влияет на порядок изменения? Да очень просто. Добавим немного определённости в наш изначальный код:

```
1 std::atomic_int variable{0};
2 int nonAtomic = 0;
3 //...
```



```

4  void thread1()
5  {
6      nonAtomic = 6;
7      variable = 7;
8      //...
9      variable = 15;
10 }
11
12 //...
13 variable = 42;
14 //...
15 variable = 777;
16 //...
17 variable = 88;
18 //...
19 variable = 356;
20 //...
21 variable = 11;

```

Мы поместили два изменения нашего объекта в один поток. Это, разумеется, не изменило состав нашего ПИ:

42	7	11	356	15	777	88
----	---	----	-----	----	-----	----

Но появилось одно очень важное ограничение: 15 **ни при каких условиях**, не может появиться в ПИ раньше чем 7. Это и составляет суть отношения *предшествует*.

Разрешите с вами синхронизироваться

Отношение *предшествует* безусловно очень интересно и знать о нём нужно, но оно ничего не говорит и не знает о многопоточной модели, поэтому мы переходим к первому и очень важному отношению, которое позволяет связать 2 операции сквозь потоки. Это отношение называется *синхронизируется с* (*synchronize with*). Как же формируется это отношение? Данное отношение формируется лишь между *операциями захвата* (*acquire*) и *освобождения* (*release*), т.е. операция захвата *синхронизируется с* операцией освобождения. Важно понимать, что отношение формируется не между операциями *загрузки* (*load*) и *записи* (*store*), а именно между операциями захвата и освобождения. Мы ещё поговорим об этом, но запомнить нужно сразу: операция загрузки не всегда является операцией захвата, а операция записи не всегда является операцией освобождения.

Давайте добавим в наш пример отношение *синхронизируется с*:

```

1  std::atomic_int variable{0};
2  int nonAtomic = 0;
3  //...

```

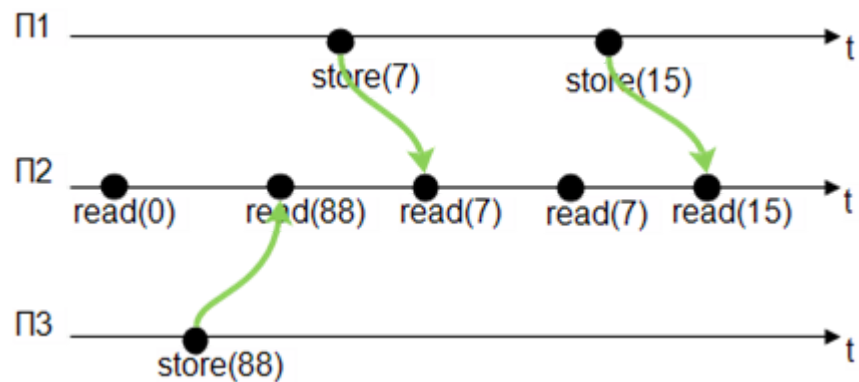
```

4  void thread1()
5  {
6      nonAtomic = 6;
7      variable = 7;
8      //...
9      variable.store(15);
10 }
11
12 void thread2()
13 {
14     while(variable.load() != 15)
15         ;
16     variable = 42;
17 }
18
19 //...
20 variable = 777;
21 //...
22 variable = 88;
23 //...
24 variable = 356;
25 //...
26 variable = 11;

```

Как вы можете видеть, в примере выше, мы заменили `=` на store и вместо простой проверки while(variable != 15) используем load. В данном примере, в этом нет никакой необходимости, я просто привёл такую запись для того, чтобы намерения были выражены более явно. Для стандартного объекта std::atomic, если методы load/store **используются с аргументами по умолчанию**, то эти операции являются операциями захвата/освобождения. А это значит, что variable.store(15) синхронизируется с variable.load()!

Ключевым моментом здесь является следующее: все эти отношения, о которых мы уже сказали, и о которых нам ещё предстоит поговорить, являются отношениями времени исполнения. В статике, т.е. до того как код начал исполнение на процессоре, никакого отношения не существует. Более того, до того момента, как в variable было сохранено число 15, а в другом потоке(thread2) это же число было прочитано, **никакого отношения между двумя этими операциями не происходит**.



(/image.axd?picture=sync_with_2.png)

На рисунке зелёными стрелками изображены отношения *синхронизации*, и как вы можете видеть, thread2() читал разные значения variable, но когда он прочитал 15, тогда наступило действие отношения и variable.store(15) *синхронизировался* с variable.load() из thread2() (мы видим и другие отношения синхронизации на рисунке, но они нам не так важны). Что это нам даёт? Скоро узнаём, но прежде рассмотрим ещё одно отношение, которое предлагает нам стандарт C++

Упорядочиваем отношения

Итак, следующим отношением является *межпоточно происходит до* (МПД (*inter-thread happens before*)). Если **А** *межпоточно происходит до* **Б**, то мы имеем строгие гарантии того, что операция **А**, выполняющаяся в потоке **Т1**, происходит до операции **Б**, которая выполняется в потоке **Т2**. Другими словами, при исполнении операции **Б**, результат операции **А** гарантированно виден. Но когда данное отношение вступает в силу? Тут всё довольно просто и интуитивно понятно. **А** *межпоточно происходит до* **Б**, если:

- **А** *синхронизируется* с **Б**, или
- **А** *предшествует* **Д** и **Д** *синхронизируется* с **Б**.

Есть ещё один вариант, но мы пока его не будем рассматривать. Как вы можете видеть *МПД* состоит из двух рассмотренных ранее отношений, но что действительно важно в этом, так это то, что «*синхронизируется с*» — это отношение между двумя синхронизирующими операциями над атомарными объектами, тогда как «*предшествует*» — это отношение между двумя **любыми операциями**, над **любыми объектами**. Если кто ещё не понял, почему это очень важно, тогда я поясню на примере:

```

1  std::atomic_int variable{0};
2  int nonAtomic = 0;
3  //...
4  void thread1()
5  {
6      nonAtomic = 6;
7      variable = 7;
8      //...
9      variable.store(15);

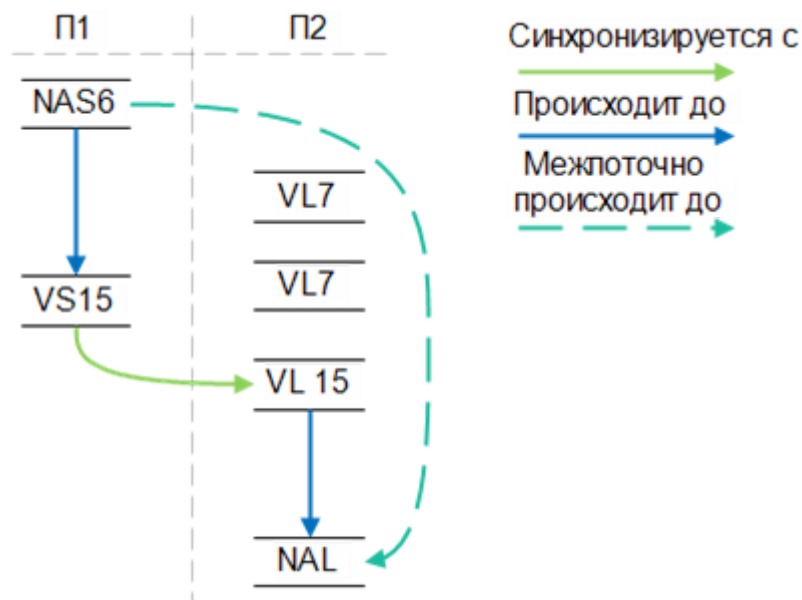
```

```

10 }
11
12 void thread2()
13 {
14     while(variable.load() != 15)
15         ;
16     variable = 42;
17     assert(nonAtomic == 6)
18 }
19
20 //...
21 variable = 777;
22 //...
23 variable = 88;
24 //...
25 variable = 356;
26 //...
27 variable = 11;

```

Как вы можете видеть в коде выше, мы добавили `assert`, который гарантировано не сработает и вот почему. Пусть `nonAtomic = 6`; будет выражением **NAS6**, `variable.store(15)`; – **VS15**, `variable.load()` загрузившая 15 – **VL15**, а `variable.load()` загрузившая 7 — **VL7** и `assert(nonAtomic == 6)` – **NAL**. Теперь построим граф отношений из этих выражений:



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

Как вы можете видеть из рисунка выше, **NAS6** межпоточно происходит до **NAL**, а это значит, что значение записанное в **NAS6** гарантировано будет загружено в **NAL**. Таким образом, отношение МПД является основой основ и краеугольным камнем синхронизации в C++. Ведь с помощью него мы можем с математической точностью выстраивать отношения между операциями так, чтобы убрать из программы любую “вредную” неопределённость.

Под вредной неопределённостью я имею ввиду такую неопределённость, которая не позволяет судить о том, как программа выполняется. К примеру, если мы имеем два инкремента глобальной переменной, в двух потоках, то неопределённость того, какой поток первым инкрементирует нельзя считать вредной, т.к. в результате счётчик будет увеличен на 2 при любом раскладе. Но если увеличение счётчика в одном потоке, не синхронизируется с увеличением в другом потоке, то мы имеем вредную неопределённость – мы больше не можем судить о результатах выполнения программы сколь бы то ни было точно.

Наконец, чтобы завершить картину мы введём ещё одно отношение: *происходит до*. Тут уже не будет никаких прорывов: операция **A** *происходит до* операции **B**, если:

- **A** предшествует **B**, или
- **A** межпоточно происходит до **B**

Вот и всё – всё просто. Это как-бы верхушка айсберга, мы рассмотрели базовые блоки из которых строится отношение *происходит до*, которое, по сути, и является тем самым отношением, которое вы будете использовать в построении модели исполнения вашей программы. Ведь самое важно это знать, что происходит до чего, а уж детали того, как это происходит не так важны.

Отношения зависимости

Итак, мы рассмотрели основные отношения, которых достаточно для спекуляции о практически любой программе использующий атомарные операции. Тем не менее, картина остаётся не полной, т.к. мы поговорили не обо всех отношениях. Я намеренно не стал вносить отношения, о которых речь пойдёт далее, в предыдущий текст, т.к. их использование является более редким случаем, и я не хотел усложнять предыдущие параграфы. В сторону лирику – приступим к делу. Первым отношением является отношение *зависимости* (*carries a dependency*). Это строго однопоточное отношение, которое говорит о следующем: имея два выражения **A** и **B**, **B** *зависит* от **A**, если **B** использует результат **A**. Всё это довольно просто и интуитивно понятно:



```
1  int a = 5;
2  int b = 6;
3  int c = a + b; //A
4  int d = c*b; //B
5
6  int* p = new int(6); //A
7  *p = 77; //B
```


Как вы можете видеть выше, выражения **Б** явно зависят от выражений **А**. В целом, выражения выше подчиняются и отношению *предшествует*, которое является надмножеством отношения *зависит*. Правда есть случаи, когда можно избавиться от отношения *зависимости*, но нет таких случаев, когда можно было бы избавиться от отношения *предшествования*. Давайте рассмотрим примеры, как можно избавиться от зависимости. В примерах далее выражение **Б** будет независимым от **А**:

?

```
1  int a = 5;
2  int b = 6;
3  int c = a + b; //А
4  c > 10 ? 5 : 0; //Б
5  std::kill_dependency(c); //Б
6  c || a; //Б
7  c && b; //Б
8  c, a, b; //Б
```

Таковыми вот нехитрыми манипуляциями разрушается отношение зависимости. Зачем вообще нужно это отношение? Оно является базовым блоком для другого отношения, которое уже является важным. Это отношение называется *зависимо предшествует* (ЗП(*dependency-ordered before*)) и это уже полноценное, межпоточное отношение. Суть его точно такая же как у отношения *синхронизируется с*, но вместо операции *захвата* используется операция *потребления* (*consume*). Отношение ЗП является частью МПД и теперь мы можем представить отношение *межпоточно происходит до* полностью:

А межпоточно происходит до Б, если:

- **А синхронизируется с Б**, или
- **А предшествует Д** и **Д синхронизируется с Б**, или
- **А зависимо предшествует Б**

Теперь мы имеем полную картину отношений, которые используются в C++. Если у вас уже голова идёт кругом от этих отношений – не беда, всё станет яснее, когда мы перейдём к практике. Но перед этим необходимо упомянуть последнее, но очень важное свойство отношений – они транзитивны. Таким образом, если **А происходит до Б**, а **Б происходит до В**, тогда **А происходит до В**.

Отношение *происходит до* не всегда будет транзитивным, есть вариант, при котором транзитивность будет нарушена. Я не буду его расписывать, т.к. он основан на использовании отношения *зависимо предшествует*, которое, по моему мнению, не слишком часто используется. Тем не менее, если вам интересно, вы можете прочитать замечание(note) в 1.10/13 стандарта C++14. Оно короткое и довольно простое.

Последовательная согласованность

Рассмотрев целый блок теории, мы, наконец, переходим к практике(которая тоже будет содержать кучу теории). Начнём мы с наиболее строгой и простой для понимания модели – модели *последовательно согласованности(sequential consistency)*. Как вы наверняка знаете, любая операция над атомарным объектом в C++, явно или неявно, одним из параметров принимает одно из значений перечисления `std::memory_order`. Так вот, для реализации модели последовательной согласованности в C++, все операции должны использовать `std::memory_order_seq_cst`, в качестве значения этого аргумента. По счастливой «случайности», данное значение является значением по умолчанию, а это значит, что если для операции явно не указано иное, такая операция описывается в рамках модели последовательной согласованности. А вот что это значит мы сейчас и рассмотрим.

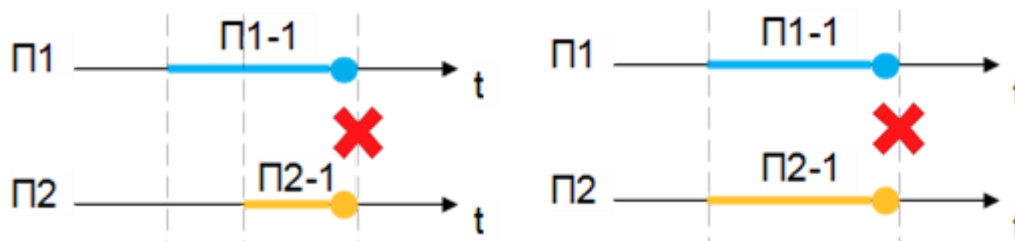
Ранее мы разбирали порядок изменения, который присущ всем атомарным объектам в отдельности. Разумеется, все операции в рамках последовательной согласованности(*ПС-операции*) тоже участвуют в ПИ. Но помимо этого порядка, все ПС-операции также формируют некий *глобальный порядок событий*, а именно: все операции, помеченные `std::memory_order_seq_cst`, исполняются так, как будто есть один единственный поток исполнения, где отношения между двумя операциями легко прогнозируются с помощью отношения *предшествования*. Это чрезвычайно важное свойство, которое нужно пояснить.

Пусть у нас есть следующий код:



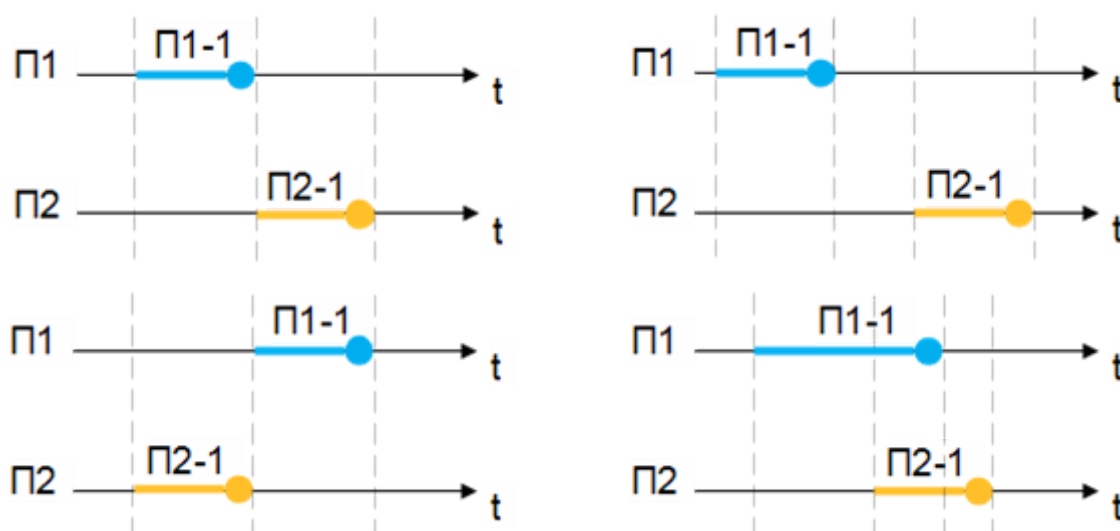
```
1  std::atomic_int first{0};
2  std::atomic_int second{0};
3  std::atomic_int third{0};
4
5  void thread1()
6  {
7      first.store(5); //П1-1
8      second.load(); //П1-2
9      third.store(6); //П1-3
10 }
11
12 void thread2()
13 {
14     second.store(12); //П2-1
15     first.store(13); //П2-2
16     third.load(); //П2-3
17 }
18
19 void thread3()
20 {
21     third.store(33); //П3-1
22     second.load(); //П3-2
23     first.load(); //П3-3
24 }
```

Как вы можете видеть, мы используем операции с аргументом по умолчанию. Это значит, что все наши операции являются ПС-операциями. Теперь давайте рассмотрим две операции в потоках **П1** и **П2** – **П1-1** и **П2-1**. Т.к. **П1-1** и **П2-1** являются ПС-операциями, то события не могут происходить так, как это изображено на следующих графиках:



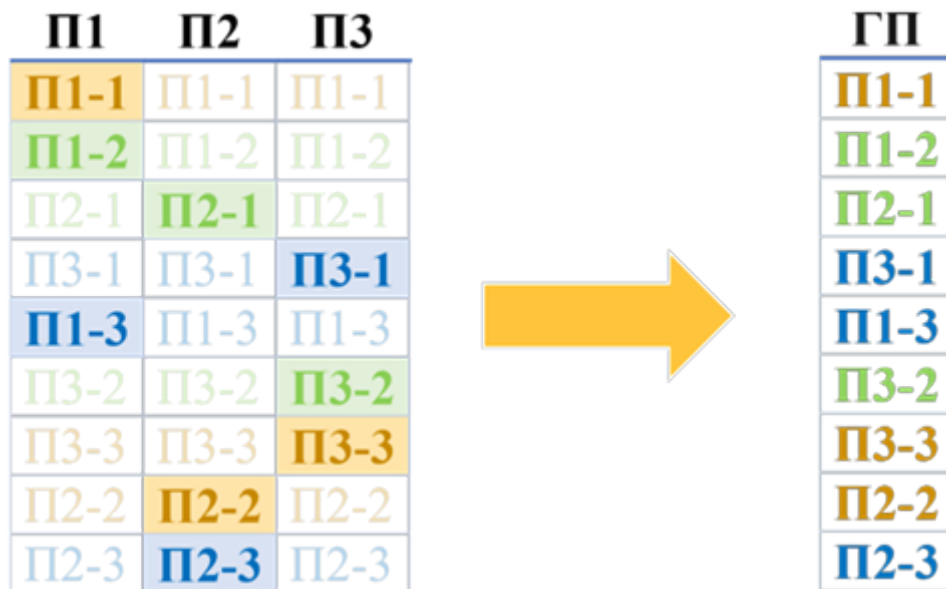
(/image.axd?picture=seq_con_forbidden.png)

А вот следующие графики изображают возможное развитие событий:



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

Как вы можете видеть, допустимо иметь любое расположение операций друг относительно друга, за исключением ситуации, когда обе операции завершаются в одно и то же время. Другими словами, все операции должны быть выстроены в очередь, и ни одно место в очереди не может быть занято более чем одной операцией. Давайте представим вышеприведённый код в виде следующего графика:



(/image.axd?picture=global_thread_1.png)

На рисунке выше, ГП – означает *главный поток* и это несуществующая, абсолютно эфемерная сущность. Хотя никаких ГП не существует, данное представление очень удобно для понимания последовательной согласованности. Все операции выполняются так, как будто бы существует единый поток исполнения и все операция выстроены в этом потоке относительно друг друга, как и в обычном потоке исполнения с помощью отношения *предшествования*. Как вы можете видеть, каждая строка вышеприведённой таблицы содержит в одном из столбцов операцию, которая выражена ярким цветом. Остальные колонки содержат «дух» это операции, изображённый тусклым цветом. Это означает, что никакая другая операция не может занять место, которое занимает «дух» операции.

Безусловно, вышеприведённый график является лишь одним из вариантов, как вышеозначенный код мог бы быть исполнен. На код выше накладываются только два ограничения: все операции принадлежащие одному потоку должны быть выполнены согласно отношению *предшествования*, и никакие две операции не могут быть выполнены одновременно. К примеру, следующий вариант так же мог бы иметь место:



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

Как и любой другой из ещё (9! – 2) вариантов.

Порядок чтения

Ещё одним важным свойством ПС-операций является тот факт, что при использовании операции чтения на атомарном объекте, такая операция гарантировано вернёт последнее записанное значение. Поясню, что это значит. Пусть у нас имеется атомарный объект **A** и его порядок изменения выглядит следующим образом(возьмём из предыдущего параграфа):

42	7	11	356	15	777	88
----	---	----	-----	----	-----	----

Так вот, как можно догадаться из цветовой гаммы вышеприведённой таблицы, единственным значением, которое может получить текущая ПС-операция чтения, является **88**. Но ведь это же очевидно! Может кто-то воскликнуть. Нет, это не всегда очевидно и мы увидим это далее. Нужно понимать, что понятия «очевидности» и «интуитивности» это как раз про последовательную согласованность. Эта модель как раз для таких случаев и придумана, всё, что с ней связано, является очевидным и интуитивным. Кто-то может спросить: «Так чего ты тут тогда распинаешься, если всё и так интуитивно понятно?». Всё потому, что за рамками модели последовательной согласованности всё становится совершенно не очевидно, и интуицию можно отправить к чёрту. Именно поэтому очень важно понимать модель последовательной согласованности и иметь представление о том, какие гарантии она с собой несёт.

Кстати, в абзаце выше я не упомянул ничего про то, какая операция записи должна предшествовать ПС-операции чтения, чтобы мы имели подобную гарантию. Не упомянул я это просто потому, что это совершенно не важно. ПС-операция чтения гарантировано получает последнее значение, записанное в атомарный объект, **любой операцией записи**. Т.е. операция записи не обязательно должна быть ПС-операцией, она может быть ослаблена настолько, насколько это нужно. Гарантия является неотъемлемой частью ПС-операции чтения и не нуждается в каких либо других гарантиях. Это, безусловно, очень важное свойство.

Завершая разговор о последовательной модели исполнения, предлагаю рассмотреть код, который гарантированно отработает правильно с моделью ПС:



```
1  #include <atomic>
2  #include <thread>
3  #include <cassert>
4
5  std::atomic_int integer{0};
6  std::atomic_bool flagA{false};
7  std::atomic_bool flagB{false};
8
9  void thread1()
10 {
11     flagB.store(true);
12     if(flagA.load())
13         ++integer;
14 }
15
16 void thread2()
17 {
18     flagA.store(true);
19     if(flagB.load())
20         ++integer;
21 }
22
```

```

23  int main()
24  {
25      std::thread firstThread(&thread1);
26      std::thread secondThread(&thread2);
27      firstThread.join();
28      secondThread.join();
29      assert(integer > 0);
30      return 0;
31  }

```

Вооружившись знаниями, полученными в данном параграфе легко проверить, что `assert` никогда не сработает. Я не буду пояснить почему это так, вам это уже должно быть понятно. Если нет, тогда отпишите в комментариях.

Модель захвата-освобождения

Рассмотрев уютный мирок модели последовательной согласованности мы окунаемся в совсем другой мир, где больше не работает интуиция. Но в сторону страшилки, давайте уже начнём знакомится с более слабыми моделями исполнения.

Модели являются «более слабыми» в том смысле, что они накладывают меньше ограничений на исполняемый код и, следовательно, дают программисту меньше гарантий.

Начнём мы с модели *захвата-освобождения* (*acquire-release*). В C++ данная модель присуща операциями, которые промаркированы следующими членами `std::memory_order`:

- `std::memory_order_acquire` – этим членом можно маркировать операции, которые загружают значение атомарного объекта. Операции использующие данный маркер являются операциями захвата.
- `std::memory_order_release` – этим членом можно маркировать операции записи данных в атомарный объект. Операции использующие данный маркер являются операциями освобождения.
- `std::memory_order_acq_rel` – этим членом можно маркировать операции *чтения-изменения-записи* (ЧИЗ (*read-modify-write* (RMW))). Операции использующие этот маркер являются операциями захвата для предыдущих операций и операциями освобождения для последующих операций.

Отложим на время в сторону операции ЧИЗ, и рассмотрим пару *захвата-освобождения*. Как мы говорили в первом параграфе, посвящённом отношениям, пара `std::atomic::store/load`, с параметрами по умолчанию, формирует отношение *синхронизируется с* между собой. Что позволяет в дальнейшем сформировать отношение *происходит до* для не атомарных операций. Там же я

упоминал, что это происходит в виду того факта, что аргументы по умолчанию делают эти операции, операциями захвата/освобождения. Но параметры по умолчанию(`std::memory_order_seq_cst`) не являются единственным вариантом превращения `std::atomic::store/load` в операции захвата освобождения. Как мы уже увидели из перечисления выше, ещё 3 маркера позволяют добиться того же самого результата.

Давайте рассмотрим следующий код:

```
1  int simpleInt{0};
2  std::atomic_bool flagA{false};
3  std::atomic_bool flagB{false};
4
5  void thread1()
6  {
7      simpleInt = 911;
8      flagB.store(true, std::memory_order_release);
9  }
10
11 void thread2()
12 {
13     while(flagB.load(std::memory_order_acquire))
14         ;
15     assert(simpleInt == 911);
16 }
```

Как вы наверное догадались, `assert` никогда не сработает и я полагаю, что пояснить почему, нет никакого смысла[подсказка: отношение *происходит до*]. Итак мы видим, что даже используя ослабленную модель мы можем с лёгкостью реализовать отношение *происходит до*. Но это буквально всё, что нам даёт данная модель. Больше нет **никаких** гарантий.

А какие гарантии нам ещё нужны? В модели ПС мы имели гипотетический главный поток – забудьте, нечего подобного здесь нет. Возьмём код из предыдущего параграфа и попробуем порассуждать над ним:

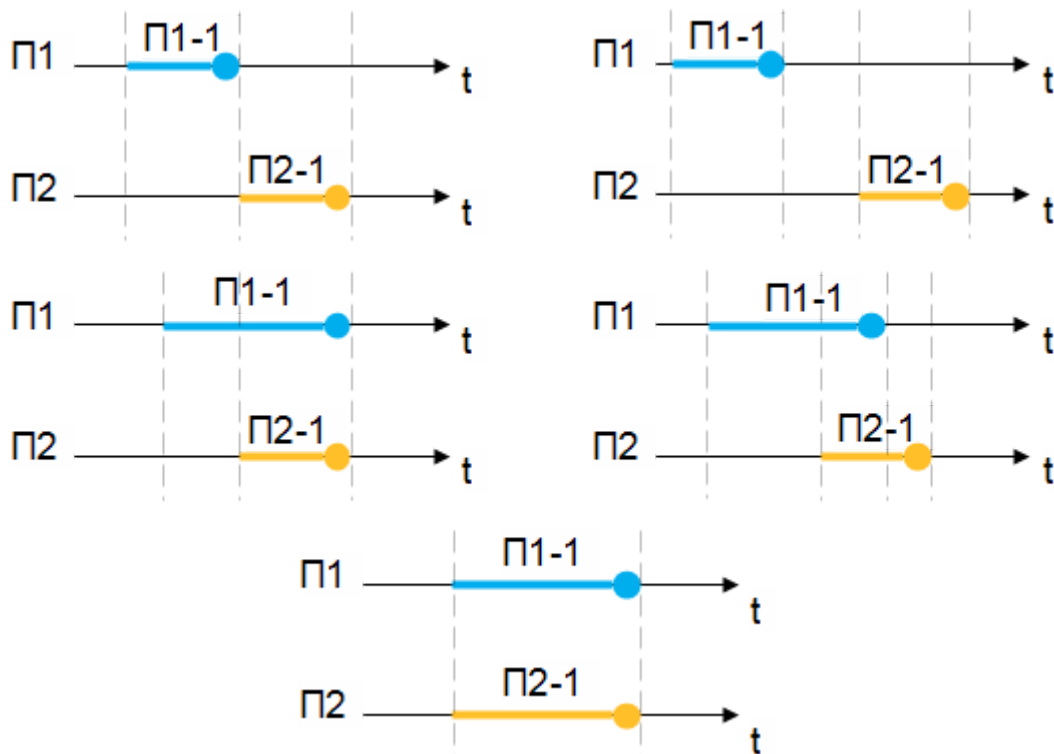
```
1  std::atomic_int first{0};
2  std::atomic_int second{0};
3  std::atomic_int third{0};
4
5  void thread1()
6  {
7      first.store(5, std::memory_order_release); //№П1-1
8      second.load(std::memory_order_acquire); //№П1-2
9      third.store(6, std::memory_order_release); //№П1-3
10 }
```

```

11
12 void thread2()
13 {
14     second.store(12, std::memory_order_release); //№П2-1
15     first.store(13, std::memory_order_release); //№П2-2
16     third.load(std::memory_order_acquire); //№П2-3
17 }
18
19 void thread3()
20 {
21     third.store(33, std::memory_order_release); //№П3-1
22     second.load(std::memory_order_acquire); //№П3-2
23     first.load(std::memory_order_acquire); //№П3-3
24 }

```

Как и в прошлый раз выделим две операции П1-1 и П2-1 и посмотрим, что мы можем сказать о них. В отличие от модели ПС, любой из ранее рассмотренных вариантов развития событий возможен:



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

Как вы можете видеть из рисунка выше, настоящая модель позволяет двум независимым операциями завершаться одновременно. Применяя аналогию, введенную ранее – используя модель захвата-освобождения, и имея очередь операций, в каждой ячейке очереди может находиться сколько угодно много операций одновременно. Напомню, что в модели ПС, в каждой ячейке очереди могла находиться лишь одна операция. Давайте изобразим один из возможных вариантов исполнения предыдущего кода графически:

П1	П2	П3
	П2-1	
П1-1		П3-1
	П2-2	
П1-2	П2-3	П3-2
		П3-3
П1-3		

(/image.axd?picture=acq_rel_threads_1.png)

Что мы можем вынести из графика выше? Во-первых, в отличие от аналогичного графика для модели ПС, здесь мы не видим «духа» операции, а это значит, что любую незанятую колонку в одной строке, может занять любая другая операция. Второе проистекает из первого – так как «духа» нет, то операции из разных потоков могут выстраиваться параллельно на одной строке. Ну и в-третьих, в результате всего этого таблица может содержать меньше строк, чем это было с моделью ПС(что мы и видим). Действительно, если вспомнить вариант с моделью ПС, то для вышеприведённого кода могло быть использовано ровно 9 строк таблицы – ни больше, ни меньше. Для текущей же модели это не так, таблица может состоять из любого количества строк в промежутке от 3-х до 9-и. В этом-то и кроется преимущество данной модели.

Если забыть аналогии и уйти от графического представления, то таблица это ничто иное как модель исполнения. Таким образом, используя модель ПС мы фактически убираем параллелизм, т.к. ни одна операция не может соревноваться с другой, тогда как с моделью захвата-освобождения операции могут исполняться одновременно, и мы получаем настоящее распараллеливание. Всё вышесказанное означает, что с точки зрения исполнения модель захвата-освобождения может дать преимущество в скорости исполнения. Она за тем и придумана, чтобы убрав несколько гарантий – ускорить производительность.

Порядок чтения

Итак, мы уже показали, что модель захвата-освобождения может не быть настолько же понятной, насколько модель ПС, но остался ещё один момент(едва ли не самый главный). Мы говорили, что с моделью ПС любая операция чтения загружает последнее сохранённое значение в атомарном объекте. Так вот, с моделью захвата-освобождения это не так. Единственным ограничением, которое накладывает данная модель является следующее: если порядок изменений атомарного объекта **A** состоит из значений ($i_1, i_2, i_3, \dots, i_n$, где i_L происходит позже i_B , если $L > B$), тогда если поток **П1** загрузил значение i_L , то ни при каких обстоятельствах при следующей загрузке он не может получить значение i_B , где $B < L$.

Для примера давайте возьмём всё тот же ПИ из предыдущего параграфа. Положим, что последним значением загруженным в **П1** было **11**, таким образом при следующей загрузке могут быть следующие варианты:

42	7	11	356	15	777	88
----	---	----	-----	----	-----	----

Как вы можете видеть, мы можем загрузить как **11**, снова, так и любое другое последующее значение. При этом, для **П2**, который только первый раз загружает значение данного атомарного объекта следующие варианты возможны:

42	7	11	356	15	777	88
----	---	----	-----	----	-----	----

А для **П3**, увидевшего **356**, следующие:

42	7	11	356	15	777	88
----	---	----	-----	----	-----	----

Ну вы поняли. И это всё, что мы можем сказать о том, какое значение будет получено при следующей операции чтения в рамках модели захвата-освобождения. Три разных потока, могут загрузить совершенно разные значения одного и того же объекта, в одно и тоже время! Интересно, не правда ли? Что это будет означать для нас на практике? Давайте возьмём код из предыдущего параграфа и изменим модель исполнения:



```
1  std::atomic_int integer{0};
2  std::atomic_bool flagA{false};
3  std::atomic_bool flagB{false};
4
5  void thread1()
6  {
7      flagB.store(true, std::memory_order_release); //П1-1
8      if(flagA.load(std::memory_order_acquire)) //П2-2
9          ++integer; //П2-3
10 }
11
12 void thread2()
13 {
14     flagA.store(true, std::memory_order_release); //П2-1
15     if(flagB.load(std::memory_order_acquire)) //П2-2
16         ++integer; //П2-3
17 }
18
19 int main()
20 {
21     std::thread firstThread(&thread1);
22     std::thread secondThread(&thread2);
23     firstThread.join();
24     secondThread.join();
25     assert(integer > 0);
26     return 0;
27 }
```

Что можно сказать о коде выше? У нас нет никакой гарантии, что `assert` не сработает, т.к. следующий вариант исполнения вполне может иметь место:

П1	П2
B.store(true)	A.store(true)
	B.load(false)
A.load(false)	

([/image.axd?picture=acq_rel_runtime_example_1.png](#))

Вот и приехали, хотя в одном потоке значение уже было изменено, другой поток всё ещё может об этом не знать. Что же делать? Использовать циклы. Изменим предыдущий код так, чтобы `assert` гарантировано никогда не сработал. Для этого можно изменить либо обе процедуры, либо одну из них, изменим одну:



```
1 void thread1()
2 {
3     flagB.store(true, std::memory_order_release);
4     while(flagA.load(std::memory_order_acquire) == false)
5         ;
6     ++integer;
7 }
```

Теперь `integer` никогда не будет равняться 0 при проверке в `assert`. Если бы мы добавили аналогичный код в `thread2()`, тогда мы бы гарантировали, что `integer == 2` в `assert`, сейчас же мы имеем гарантию аналогичную той, что мы имели при ❖❖ использовании модели ПС.

Хотя может показаться, что мы отделались малой кровью, и нам пришлось внести лишь незначительные изменения для корректного исполнения кода, я хочу предостеречь вас от поспешных выводов. Лёгкость, с какой мы исправили код, это заслуга элементарности использованного примера. В реальных «боевых» условиях всё будет гораздо сложнее, и рассуждать о корректности кода, который использует модель захвата-освобождения, так просто уже не получится. Реальный код мало-мальски сложного алгоритма, использующего атомарные объекты с семантикой ослабленной модели, крайне сложен для анализа «простыми смертными».

Как мы уже упоминали, весь смысл подобной ослабленной модели, заключается в том, что при определенных обстоятельствах, использование оной *может* дать прирост в производительности. Почему «может»? Потому что это зависит от архитектуры процессора и других факторов. Тут как и с другими случаями, касающимися производительности – нужно всё измерять. Если вспомнить барьеры памяти ([/post/2015/05/16/memory_barriers.aspx](#)), о которых мы говорили ранее, то становится очевидно, что разные модели будут использовать разные барьеры памяти. За счёт этого и может

быть получена разница в производительности. Главное не гнаться за производительностью, ради призрачного выигрыша, который не измерен, путём ослабления модели и, как следствие, уменьшения возможности проверки корректности кода алгоритма «простым смертным».

Вместо завершения

Я честно планировал сделать эту статью последней в цикле, но уже ~4000 слов, а сказать ещё нужно о многом. Поэтому я прерываю повествование и переношу оставшийся материал в следующую статью. Торжественно обещаю, что следующей статьи не придётся ждать 3 года. В следующей статье вы увидите: завершение разговора о моделях исполнения C++(memory_order_consume, memory_order_relaxed), описания операций ЧИЗ и что в них такого интересного, описание барьеров(*fence*) в C++ и как они соотносятся с барьерами(*barriers*) о которых мы говорил ранее, также мы, возможно, поговорим о volatile и завершим цикл. «Оставайтесь с нами, не переключайтесь».

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 5: Граница на замке (/post/2015/10/15/parallel_world_p5.aspx)

Метки : multithreading (<http://www.scrutator.me/?tag=multithreading>), C++ (<http://www.scrutator.me/?tag=C%2b%2b>), C++11 (<http://www.scrutator.me/?tag=C%2b%2b11>), atomic (<http://www.scrutator.me/?tag=atomic>), memory model (<http://www.scrutator.me/?tag=memory+model>)

Текущий рейтинг: 5.0 (1 голосов)

Похожие записи

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 ...

Добро пожаловать в параллельный мир. Часть 2: Мир асинхронный (/post/2012/06/03/parallel-world-p2.aspx)

Продолжая тему начатую в части 1, перейдём от простейшего способа использования потоков, к следующему

Добро пожаловать в параллельный мир. Часть 1: Мир многопоточный (/post/2012/04/04/parallel-world-p1.aspx)

Ну вот мы и дождались, C++, наконец, перестал игнорировать ситуацию за окном и признал, что "исполня

2 Комментариев

scrutator.me

1 Войти ▾

♥ Рекомендовать 2

🔗 Поделиться

Старое в начале ▾



Присоединиться к обсуждению...



Александр Крахмаль • год назад

<<

Модель захвата-освобождения

Порядок чтения

...если порядок изменений атомарного объекта A состоит из значений ($i_1, i_2, i_3, \dots, i_n$, где i_k происходит позже i_m , если $k > m$), тогда если поток П1 загрузил значение i_m , то ни при каких обстоятельствах при следующей загрузке он не может получить значение i_k , где $m < k$

>>

а далее, в примере для П1, показываете противоположную ситуацию, что как раз для следующей загрузки на выбор есть значения от i_m и до i_n , при $m < k < n$.

либо утверждение не верно и должно быть $k < m$, либо пример не то кажет, либо я "не догоняю" (не переключился для восприятия материала о данной модели)

^ | ▾ • Ответить • Поделиться ›



Evgeniy Shcherbina Автор ➔ Александр Крахмаль • год назад

Нет, Вы всё поняли правильно, а вот в тексте ошибка. Исправил ошибку и использовал другие буквы, чтобы стало ещё понятнее :)(Before-Later)

^ | ▾ • Ответить • Поделиться ›

ТАКЖЕ НА SCRUTATOR.ME

**Субъективный объективизм |
Конструирование в C++11. Часть 1: ...**

1 комментарий • год назад*

clang — огромное спасибо

**Обзор книги Effective Modern C++: 42
Specific Ways to Improve Your Use of ...**

3 комментария • год назад*

flop ws — книга Скотта Мейерса

Эффективный и современный C++: 42

рекомендации по использованию C++11 и

Добро пожаловать в параллельный мир. Часть 5: Граница на замке

2 комментария • год назад*

Evgeniy Shcherbina — Вы знаете, смешивать различные модели можно, но что про это написать, я пока не знаю. Т.е.

Субъективный объективизм | Добро пожаловать в параллельный мир. ...

2 комментария • год назад*

Evgeniy Shcherbina — Спасибо! Починил ссылку.

 Подписаться  Добавь Disqus на свой сайт [Добавить Disqus](#) [Добавить](#)  Конфиденциальность **DISQUS**

Введите текст для поиска...

Поиск

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)
