如何理解 C++11 的六种 memory order?

2022/10/24更新：

这篇回答只讨论了重排，但是基本没有讨论在多核心情况下，通过缓存一致协议同步缓存过程当中造成的数据变更顺序在各个核心看起来可能不同的问题。

比如核心1更新了x，核心2更新了y，在核心3看来到底是x先更新，还是y先更新，在核心4看来又如何，等等一系列问题。

虽然下文在大的方向和要阐述的总的思想上不因此发生变化，但是还是请以相关标准和正式定义为准。

（或者，下面这篇回答更为严谨，推荐

https://www.zhihu.com/answer/83422523）

看了一下，觉得没有触及实质的回答，所以补充一下。

这个问题的难点在于，很多人以为它们是限制多线程之间的执行顺序（包括我写这篇回答时的最高赞回答看起来也是这么认为的），**然而其实不是。**

事实上，Sequentially-consistent ordering是目前绝大多数编译器的缺省设置。如果按照高赞回答的意思，那么多线程如果使用了atom操作，貌似就几乎变成了单线程（或者回合制）？真的吗?

既然是多线程，那么线程之间的执行顺序就一定不是确定性的（你只能在某些点同步它们，但是在任何其它的地方是没有办法保证执行顺序的）。C++11所规定的这6种模式，其实并不是限制（或者规定）两个线程该怎样同步执行，**而是在规定一个线程内的指令该怎样执行。**

是的，我知道这部分的文档（规定）以及给出的例子里面，满屏都是多线程。但是其实讨论的是单线程的问题。**更为准确地说，是在讨论单线程内的指令执行顺序对于多线程的影响的问题。**

首先，什么是原子操作？原子操作就是对一个内存上变量（或者叫左值）的读取-变更-存储（load-add-store）作为一个整体一次完成。

让我们考察一下普通的非原子操作：

```
x++
```

这个表达式如果编译成汇编，对应的是3条指令：

那么在多线程环境下，就存在这样的可能：当线程A刚刚执行完第二条指令的时候，线程B开始执行第一条指令。那么就会导致线程B没有看到线程A执行的结果。如果这个变量初始值是0，那么线程A和线程B的结果都是1。

如果我们想要避免这种情况，就可以使用原子操作。使用了原子操作之后，你可以认为这3条指令变成了一个整体，从而别的线程无法在其执行的期间当中访问x。也就是起到了锁的作用。

所以，atom本身就是一种锁。它自己就已经完成了线程间同步的问题。这里并没有那6个memory order的什么事情。

问题在于以这个原子操作为中心，其前后的代码。这些代码并不一定需要是原子操作，只是普通的代码就行。

什么问题？比如还有另外一个变量y，在我们这个原子操作代码的附近，有一个

```
y++
```

那么现在的问题是，这个y++到底会在我们的x++之前执行，还是之后？

注意这完全是单线程当中指令的执行顺序问题，与多线程风马牛不相及。但是，这个问题会导致多线程执行结果的不同。理解了这个，就理解了那6种memory order。

为啥？因为我们对x进行原子操作的地方，锁定了线程间的关系，是一个同步点。那么，以这个点为基准，我们就可以得出两个线程当中其它指令执行先后顺序关系。

比如，A线程先对x进行了自增操作。因为对x的访问是原子的，所以B线程执行该行代码（假设代码当中对x的访问只有这一处）的时间点必然在A完成之后。

那么，如果在A线程当中，y++是在x++之前执行的，那么我们就可以肯定，对于B线程来说，在x++（同步点）之后所有对y的参照，必定能看到A线程执行了y++之后的值（注意对y的访问并非原子）

但是有个问题。如果在程序当中y++紧靠x++，那么其实它到底是会先于x++执行（完毕），还是晚于x++执行（完毕），这个是没准儿的。

为啥呢？首先编译器对代码可能进行指令重排。也就是说，编译器编译之后（特别是开了优化之后）的代码执行顺序，是不一定严格按照你写代码的顺序的。

但是如果仅仅如此，也只是二进制（机器码）的顺序与源代码不同，还不至于导致A和B当中的指令执行顺序不同（因为A和B执行的是相同的机器码程序）。但是实际上，在非常微观的层面上，A和B也是可能不同的，甚至于，A每次执行到这里顺序都不见得一样。

啥？...还真的是这样。原因在于当代CPU内部也有指令重排。也就是说，CPU执行指令的顺序，也不见得是完全严格按照机器码的顺序。特别是，当代CPU的IPC（每时钟执行指令数）一般都远大于1，也就是所谓的多发射，很多命令都是同时执行的。比如，当代CPU当中（一个核心）一般会有2套以上的整数ALU（加法器），2套以上的浮点ALU（加法器），往往还有独立的乘法器，以及，独立的Load和Store执行器。Load和Store模块往往还有8个以上的队列，也就是可以同时进行8个以上内存地址（cache line）的读写交换。

是不是有些晕？简单来说，你可以理解当代CPU不仅是多核心，而且每个核心还是多任务（多指令）并行的。计算机课本上的那种一个时钟一条指令的，早就是老黄历了 （当然，宏观来看基本原理并没有改变）

看到这里还没有晕的话，那么恭喜你，你快要理解什么是memory order了。所谓的memory order，其实就是限制编译器以及CPU对单线程当中的指令执行顺序进行重排的程度（此外还包括对cache的控制方法）。这种限制，决定了以atom操作为基准点（边界），对其之前的内存访问命令，以及之后的内存访问命令，能够在多大的范围内自由重排（或者反过来，需要施加多大的保序限制）。从而形成了**6种模式。它本身与多线程无关，是限制的单一线程当中指令执行顺序。**但是（合理的）指令执行顺序的重排在单线程环境下不会造成逻辑错误而在多线程环境下会，所以这个问题的**目的是为了解决多线程环境下出现的问题。**