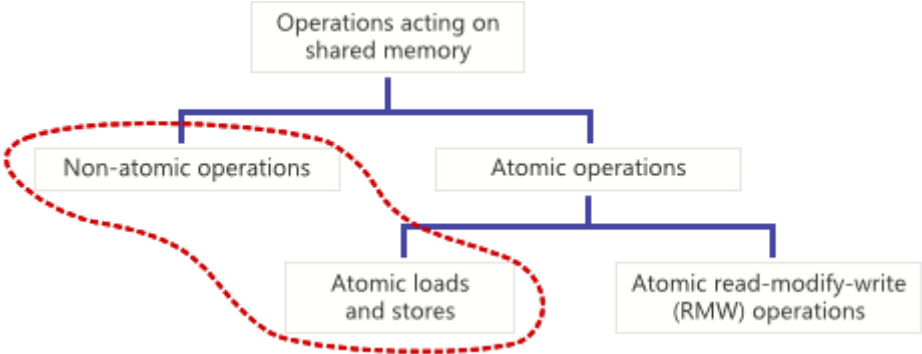


在网上已经有很多有关介绍原子操作的内容，通常都是注重于原子读-修改-写（RMW）操作。然而，这些并不是原子操作的全部，还有同样重要的原子加载和原子存储。在这篇文章中，我将要在处理器级别和C/C++语言级别两个方面来对比原子加载和原子存储与它们相应的非原子部分。沿着这条路，我们将弄清楚C++11中“数据竞争”这个概念。



共享内存中的原子操作是指它是否完成了一个线程相关的单步操作。当一个原子存储作用于一个共享变量时，其他的线程不能监测到这个未完成的修改值。当一个原子加载作用于一个共享变量时，它读取到这个完整的值，就像此时出现了一个单独的时刻，而非原子加载和存储则不能做到这些保证。

如果没有这些保证，无锁编程将不可能实现，因为你不能使不同的线程同时操作一个共享变量。我们可以制定如下规则：

任何时刻两个线程同时操作一个共享变量，当其中一个为写操作时，这两个线程必须使用原子操作。

如果你违反这条规则，并且每个线程都使用非原子操作，你将会看到C++11标准中提到的数据竞争（不要混淆于Java中数据竞争的概念，这个是不同的，或者说是更广义上的竞争情况）。C++11标准并没有告诉你为什么数据竞争是糟糕的，但只要你出现这种情况，就会发生“未定义行为”（1.10.21部分）。这种糟糕的数据竞争的原因是非常简单的：它们导致了读写撕裂。

一个内存操作可以是非原子的，因为它使用非原子的多CPU指令，即使当使用单CPU指令时也是非原子的，因为你不能简单的设想你写出的可移植代码。让我们来看几个例子。

非原子性是由于多CPU指令

假设你有一个64位初始化为0的全局变量。

1	<code>uint64_t sharedValue = 0;</code>
---	--

在某些时刻，你给这个变量赋一个64位的值。

1	<code>void storeValue()</code>
2	<code>{</code>
3	<code> sharedValue = 0x100000002;</code>
4	<code>}</code>

当你在32位的x86环境下使用GCC来编译这个函数时，将会生成如下机器码。

--	--

1	\$ gcc -O2 -S -masm=intel test.c
2	\$ cat test.s
3	...
4	mov DWORD PTR sharedValue, 2
5	mov DWORD PTR sharedValue+4, 1
6	ret
7	...

这个时候你就会看到，编译器会使用两个单独的机器指令来完成这个64位的赋值。第一条指令设置低32位的0x00000002，第二条指令设置高32位的0x00000001。非常明显，这个赋值操作是非原子的。如果共享变量同时被不同的线程存取，就会出现很多错误：

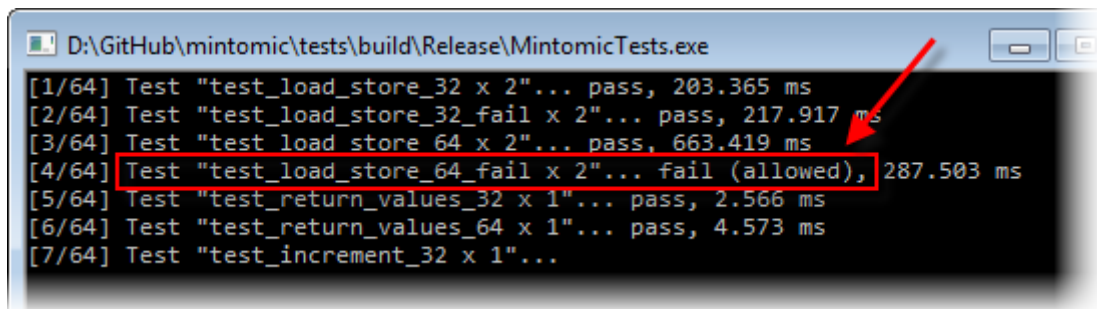
1. 如果一个线程在两个机器指令的间隙先调用存储变量，将会在内存中留下像0x0000000000000002这样的值——这是一个**写撕裂**。在这个时候，如果另一个线程读取共享变量，它将会接收到一个完全伪造的、没有人想要存储的值。
2. 更糟糕的是，如果一个线程在两个机器指令的间隙先占用变量，而另一个线程在第一个线程重新获得这个变量之前修改了sharedValue，那将导致一个永久性的写撕裂：一个线程得到高32位，另一个线程得到低32位。
3. 在多核设备上，并不是只有先行占有其中一个线程来导致一个写撕裂。当一个线程调用storeValue时，任何线程在另一个核上可能同时读取一个明显未修改完的sharedValue。

同时读取sharedValue会带给它一系列的问题：

1	uint64_t loadValue()
2	{
3	return sharedValue;
4	}
5	
6	\$ gcc -O2 -S -masm=intel test.c
7	\$ cat test.s
8	...
9	mov eax, DWORD PTR sharedValue
10	mov edx, DWORD PTR sharedValue+4
11	ret
12	...

这里也一样，编译器会使用两条机器指令来执行这个加载操作：第一条读取低32位到eax，第二条读取高32位到edx。在这种情况下，如果对于sharedValue进行同时存储则会发现，它将导致一个**读撕裂**——即使这个同时存储是原子的。

这个问题并不是理论上的，Mintomic的测试集包含了一个名为test_load_store_64_fail的测试案例，在这个案例中，一个线程使用一个普通的赋值操作，存储了很多64位的值到一个单独的变量，同时另一个线程对这个变量反复地执行一个简单的加载，来确认每一个结果。在一个多核的x86机器上，这个测试像我们想象的一样一直失败。



```
D:\GitHub\mintomic\tests\build\Release\MintomicTests.exe
[1/64] Test "test_load_store_32 x 2"... pass, 203.365 ms
[2/64] Test "test_load_store_32_fail x 2"... pass, 217.917 ms
[3/64] Test "test_load_store_64 x 2"... pass, 663.419 ms
[4/64] Test "test_load_store_64_fail x 2"... fail (allowed), 287.503 ms
[5/64] Test "test_return_values_32 x 1"... pass, 2.566 ms
[6/64] Test "test_return_values_64 x 1"... pass, 4.573 ms
[7/64] Test "test_increment_32 x 1"...
```

非原子的CPU指令

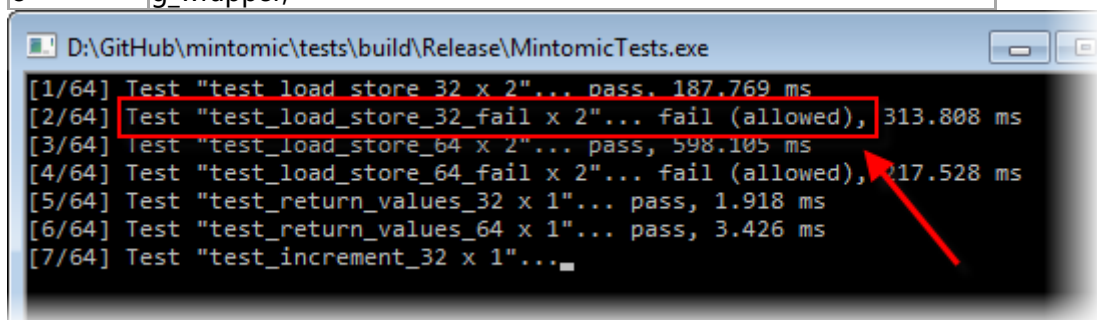
一个内存操作可以是非原子的，甚至是当由一个单CPU指令来执行的时候。例如，ARMv7指令设置包含了将两个由32位源寄存器的内容存储到内存中的一个64位值的strd指令。

1	strd r0, r1, [r2]
---	-------------------

在一些ARMv7处理器中，这条指令是非原子的。当这个处理器遇到这条指令时，它实际上在底层执行两个单独的32位存储（A3.5.3部分）。再来一次，另一个线程在一个单独的核上运行，有可能观察到一个写撕裂。有趣的是，写撕裂更可能出现在一个单核的设备上：例如，一个预定线程的上下文切换的系统中断，确实可以执行在两个内部的32位存储之间！在这种情况下，当这个线程从这个中断恢复时，它将再一次重新调用这个strd指令。

再看另一个例子，众所周知，在x86环境下，如果内存操作数是自然对齐的，那么一个32位的mov指令就是原子的，但如果不是自然对齐，那么将是非原子的。换句话说，原子性的保证仅仅是当一个32位整数的地址正好是4的倍数的时候。Mintomic提出另一个证实这个保证的测试案例，test_load_store_32_fail。就像写的那样，这个测试在x86总是成功的，但是如果你修改这个测试，强制将sharedInt置于一个未对齐的地址，它将失败。在我的Core 2 Quad Q6600上，这个测试失败了，因为sharedInt在一个寄存器中越界了。

1	// Force sharedInt to cross a cache line boundary:
2	#pragma pack(2)
3	MINT_DECL_ALIGNED(static struct, 64)
4	{
5	char padding[62];
6	mint_atomic32_t sharedInt;
7	}
8	g_wrapper;



```
D:\GitHub\mintomic\tests\build\Release\MintomicTests.exe
[1/64] Test "test_load_store_32 x 2"... pass, 187.769 ms
[2/64] Test "test_load_store_32_fail x 2"... fail (allowed), 313.808 ms
[3/64] Test "test_load_store_64 x 2"... pass, 598.105 ms
[4/64] Test "test_load_store_64_fail x 2"... fail (allowed), 117.528 ms
[5/64] Test "test_return_values_32 x 1"... pass, 1.918 ms
[6/64] Test "test_return_values_64 x 1"... pass, 3.426 ms
[7/64] Test "test_increment_32 x 1"...
```

现在已经有很多特定于处理器的细节，让我们再来看看C/C++语言级别的原子性。

所有的C/C++操作被认定为非原子的

在C和C++中，所有操作被认定是非原子的，甚至是普通的32位整数赋值，除非被别的编译器或者硬件供应商指定。

```
1      uint32_t foo = 0;
2
3      void storeFoo()
4      {
5          foo = 0x80286;
6      }
```

这个语言标准并没有提到任何有关于这种情况下的原子性。也许整型赋值是原子的，也许不是。因为非原子操作没有做任何保证，在C定义中，普通整型赋值是非原子的。

实际上，我们对我们的目标平台了解的更多。例如，大家都知道在现在的x86、x64、Itanium、SPARC、ARM和PowerPC处理机上，只要目标变量是自然对齐的，那么普通32位整型赋值就是原子的，你可以通过查询你的处理机手册或者编译器文档来证实。在游戏行业，我可以告诉你很多关于32位整型赋值依赖这个特殊保证的例子。

尽管如此，但在写真正的可移植的C和C++代码时，有一个历史悠久的传统，就是我们所知道的仅仅是语言标准告诉我们的。可移植的C和C++代码的设计是为了可以运行在任何可能的计算设备上，过去的、现在的以及虚拟的。就我自己而言，我想设计一种机器，它的内存仅仅可以通过先到先得来改变：



在这样的机器上，你绝对不会想要在执行一个并发的读操作的同时执行一个普通的赋值，你可能会最终读取到一个完全随机的值。

在C++11中，有一个最终的方案来执行实际的可移植原子加载和存储——C++11原子库。通过使用C++11原子库来执行原子加载和存储，甚至可以运行在虚拟的计算机上，即使这意味着C++11原子库必须默默地加一个互斥量来确保每一个操作都是原子的。这里还有我上个月发布的Mintomic库，它并不支持这么多平台，但是可以运行在很多以前的编译器上，它是优化过的，并且保证是**无锁**的。

宽松的原子操作

让我们回到前面那个sharedValue例子最开始的地方，我们将用Mintomic重写它，这样所有的操作就可以原子地执行在任何Mintomic支持的平台上了。首先，我们必须声明sharedValue为Mintomic原子数据类型中的一个。

```
1      #include "mintomic/mintomic.h"
2
3      mint_atomic64_t sharedValue = { 0 };
```

mint_atomic64_t类型保证了在所有平台上原子存取的正确内存对齐。这是非常重要的，因为，例如ARM的GCC4.2编译器附带的Xcode3.2.5并不保证普通的uint64_t以8字节对齐。

对于storeValue，通过执行一个普通的、非原子的赋值来替代，我们必须调用。

```
1 void storeValue()
2 {
3     mint_store_64_relaxed(&sharedValue, 0x100000002);
4 }
```

相似的，在loadValue中，我们调用mint_load_64_relaxed。

```
1 uint64_t loadValue()
2 {
3     return mint_load_64_relaxed(&sharedValue);
4 }
```

使用C++11的术语，这些函数现在不存在数据竞争。当并发操作执行时，无论代码运行在ARMv6/ARMv7（Thumb或者ARM模式）、x86、x64 或者PowerPC上，绝对不可能出现读写撕裂。

你是否好奇mint_load_64_relaxed和mint_store_64_relaxed是如何工作的，这两个函数在x86上都是扩展到一个内联的cmpxchg8b指令上，对于其他平台，请查询Mintomic的[实现](#)。

在C++11中明确的写出了类似的代码：

```
1 #include <atomic>
2
3 std::atomic<uint64_t> sharedValue(0);
4
5 void storeValue()
6 {
7     sharedValue.store(0x100000002, std::memory_order_relaxed);
8 }
9
10 uint64_t loadValue()
11 {
12     return sharedValue.load(std::memory_order_relaxed);
13 }
```

你会注意到，在Mintomic和C++11的例子中都使用了宽松的原子性，由_relaxed后缀的多个标识符来证明。_relaxed后缀暗示了，就像普通的加载和存储一样，没有内存访问排序的保证。

一个宽松的原子加载（或存储）和一个非原子加载（或存储）之间的唯一区别就是，宽松的原子操作保证了原子性，没有其他区别来保证。

特别的，在程序指令中，一个宽松的原子操作，被它前面或者后面的指令由于处理机本身任何一个因为编译器重新排序或者内存重新排序所产生的影响，对内存来说依然是合法的。编译器甚至可以在冗余的宽松原子操作上执行优化，就像非原子操作一样。就一切情况而言，这些操作仍然是原子的。

当并发操作同时共享内存时，我认为，一直使用Mintomic或者C++11原子库函数是非常好的练习，甚至当你知道在你的目标平台上，一个普通的加载或者存储已经是原子的情况下。一个原子库函数就像提示这个变量是并发数据存储的目标。

[转自:]<http://blog.jobbole.com/54345/>