链滴

锁？不锁？如何锁？

作者：localvar

原文链接：https://ld246.com/article/1521883993102

来源网站：链滴

加锁、解锁(同步/互斥)是多线程中非常基本的操作，但我却看到不少的代码对它们处理的很不好。简说来有三类问题，一是加锁范围太大，虽然避免了逻辑错误，但锁了不该锁的东西，难免降低程序的率；二是该锁的不锁，导致各种莫名其妙的错误；三是加锁方式不合适，该用临界区的用内核对象等也会降低程序的效率。

要正确的运用锁操作，首先要弄清楚什么时候需要加锁。很多书上都说在可能“同时发生多个写操作或“同时发生读写操作”时，应该加锁。这固然没什么错，但我认为它没有说到问题的根上，更准确表述应该是：如果不加锁会导致不可容忍的数据不一致，那么就应该加锁。据此，我在下表中列出了线程中应该加锁和无需加锁的条件，其中的“简单数据类型”是指cpu可以在一条指令中完成操作的据类型，一般整形和所有比整形小的数据类型都是，除此之外的类型都属于“复杂数据类型”，例如自己定义的结构体等。

	操作的结果与初值无关	**操作的结果与初值相关**
写简单数据类型	不需要加锁①	要加锁②
写复杂数据类型	需要加锁③	需加锁④
读简单数据类型	不需要加锁⑤	需要加锁⑥
读复杂数据类型	需要加锁⑦	需加锁⑧

大家可能注意到，在第1、5、6种情况下，我认为可以不加锁，粗看起来，这与书上的说法有些矛盾其实却不然，因为这些操作可以在一条指令内完成，所以它们具有天然的“原子性”，我们可以认为cu已经给它们加锁了，我们没必要再画蛇添足。如果这个理由还不够的话，你不妨想一下我们再加一锁是否有用，看下面的代码(以第1种情况为例)：

```
Lock(); // ①
n = 10; // ②
Unlock(); // ③
int x = n; // ④
```

看出来了吗？不管语句①③是否存在，这段代码执行完毕后，我们都无法保证x的值是10。也许你会如果把③④两条语句的位置换一下，x就肯定是10了。可是在这个例子中，想让x是10，为什么不把语④直接换成int x = 10;呢？既省了加锁，又减少了键盘的磨损，何乐而不为？！而且，我的这个例子不是刻意构造的，在多线程，这种情况比比皆是。

第2种情况的典型代表是i++;，需要对它加锁是因为它表面上虽然只有一条语句，却要执行至少两个作，一是读出i的初始，二是把加一后的结果写回去，两个操作就没有“原子性”了，所以需要加锁。

另外，上表中判断是否需要加锁的依据是“是否可能造成数据不一致”。实际上，有些情况下数据不致是可以容忍的，如果它发生概率极低、造成的不良后果可以忽略、并能很快自动恢复，那它可能就可以容忍的。对这种数据不一致，我们可以不加锁。不过对它的判定与程序的实际情况关联太大，我在这里就不讨论了。

加锁的方法也可分为三类，临界区、内核对象和互锁函数。相比前两类，互锁函数的知名度要低不少但它却是我用的最多的方法，因为它有一个最大的优点：快！有不少书上比较临界区和内核对象时都临界区的优点是不会进入内核模式，速度快。不过这是不全面的，如果没有冲突（实际发生冲突的概一般很低），临界区确实不会进内核模式，但如果发生了冲突要进行等待，它就要依靠内核对象了。互锁函数则绝不会进内核模式，所以互锁函数是最快的（临界区在没有冲突时的行为是依靠互锁函数现的）。互锁函数的缺点是只能处理相对简单的数据类型（不要和我前面说的“简单数据类型”等价

来），但另一方面，对加锁需求最高的也往往是这些类型的数据。

实际开发中，还有一种锁比较常用，这就是单写多读锁，《windows核心编程》上有一个单写多读锁实现，我的blog上有另一个实现。前者适用于需加锁的对象数量较少（例如如只有一个），访问冲突率相对较高的情况。后者适用于需加锁的对象很多，访问冲突概率很低的情况（对象多了，单个对象访问冲突自然就少了）。两个实现的共同缺点是不支持重入，即同一个线程中，解锁前不能再次加锁临界区在这方面有优势，它支持重入。使用TLS（线程局部存储）技术进行改进应该能让它们支持重，不过这样做了以后我那个实现应该就算不上轻量级了:)。

最后，还有其它的一些不用锁的方法也可以保证多线程中的数据一致性，其中最常用的就是循环。例下面的例子：

```
struct bar
{
    volatile unsigned version;    // 一个额外的版本号字段
    int field1;
    char field2;
    char field3;
    ......
};
bar g_bar = { 0 };
// 写线程
++g_bar.version;    // 加1, version是奇数, 表示正在更新
g_bar.field1 = 10;
......
++g_bar.version;    // 再加1, version是偶数, 表示更新完毕

// 读线程
void ReadGlobalBar( bar* p )
{
    unsigned ver;
    do {
        ver = g_bar.version;
        if( ver % 2 != 0 )    // 正在更新
        {
            Sleep( 0 );    // 等待
            continue;
        }
        p->field1 = g_bar.field1;
        ......
    } while( ver != g_bar.version );
}
```

然而这种方法真的没用锁吗？看你怎么理解了，那个version字段其实就可以看做是锁的。不过它只半个锁，因为它只锁了读操作，而没锁写操作，也就是说写操作可以随时进行而无需等待。如果读操非常多，但写操作较少，并且你不希望写操作经常被打断，那它正好满足你的要求。它的缺点是你要证系统中某个时刻最多有一个“writer”，“writer”一多，它就的无能为力了（这时一般应该用单多读锁）。

2007.10.18：补充一点，关于acquire release semantics

在多处理器平台上，一个处理器的实际的操作顺序，和其它处理器所看到的它的操作顺序可能并不相，例如：

```
a++;
```

b++;

在其他处理器看来，很有可能b++发生在前面，而a++发生在后面。某些情况下，其他处理器看到的序必须和实际的顺序保持一致，所以就需要引入acquire semantics和release semantics了。

说一个操作具有acquire semantics，就表示可以保证其它处理器在看到这一操作的结果前，不会看（该处理器上）后续操作的结果，对该处理器而言，可以理解为它进行此操作前，不会进行后续操作而一个操作具有release semantics，就表示可以保证其它处理器在看到这一操作的结果前，能看到该处理器）上先前所有操作的结果，对该处理器而言，可以理解为在完成所有先前的操作之前，不会行此操作。

vc编译器（其它编译器不一定保证）保证对volatile对象的写操作具有release semantics；对volatile象的读操作具有acquire semantics。基于此点保证，多线程环境中就可以用volatile型对象实现锁操了。

对windows互锁函数的补充

一个轻量级的单写多读锁