深入探索并发编程系列(一)-锁不慢；锁竞争慢

2016-06-20 | High-performance

锁(也叫互斥量)在很长一段时间都被误解了。1986年，在Usenet的有关于多线程的讨论会中，Matthew Dillon说过：大多数人都对锁有个误解，认为锁是慢的。25年后，这种误解似乎在某一时间段又突然出现了。

在某些平台上或者当锁被高度竞争时，锁确实慢。另外，当你在开发一个多线程程序时，由于锁的引入，给性能带来巨大的瓶颈是很常见的。但这并不意味着所有的锁都是缓慢的。我会在这篇文章中解释，有的时候，使用锁的策略反而能带来非常好的性能。

大家对锁的误解可能源自于某个最容易忽视的原因：不是所有的程序员都会意识到轻量级锁和内核锁的区别。我会在下一篇文章中对轻量级锁做专门介绍:总是使用轻量级锁。在这篇文章中，假设你在Windows平台下做C/C++开发，你需要的正是一个Critical Section对象。

有时候，锁是慢的这个结论是由benchmark支撑的。例如，这篇文章在高负载状态下来测试锁的性能：每个线程必须持有锁来完成任何一项任务（高竞争），并且锁都是在极短的时间间隔下被持有（高频率）。这种方式似乎很完美，但在实际应用中，却要避免这种使用锁的方式[注1]。基于这种考虑，我设计了一种benchmark，同时包含对锁使用的最坏情况和最好情况。

由于一些其它的考虑，大家可能不愿意用锁了。存在一系列的技术被称为无锁编程(或者不含锁编程[注2]）。无锁编程是极具挑战性的，但其本身可以在许多实际应用场景下带来高度的性能回报。据我所知，有些程序员会花费许多天甚至几周的时间来设计某种无锁算法，之后再做一系列测试，但在几个月后才发现隐藏的bug. 风险与回报并存对于相当一部分程序员都是有诱惑力的，这当然也包括我，在以后的几篇文章中会提到这些。有了无锁编程的诱惑，大家开始觉得锁使用起来很枯燥，缓慢并且非常糟糕。

但也不能把锁贬的一文不值。在现实软件中，当大家为了保护内存分配器的时候，锁就是一个让人敬仰的东西。Doug Lea的分配器是在游戏开发中非常著名的内存分配器[注3]，但其只支持单线程，这时候我们就必须使用锁机制来进行保护。在游戏运行时，经常会碰到多个线程抢占一个内存分配器，每秒钟抢占次数可达到15000次左右。在加载过程中，每秒钟会达到100000次甚至更多。虽然这并不是个大问题。但你却可以看到，锁能非常出色的来处理这些负载。

锁竞争benchmark

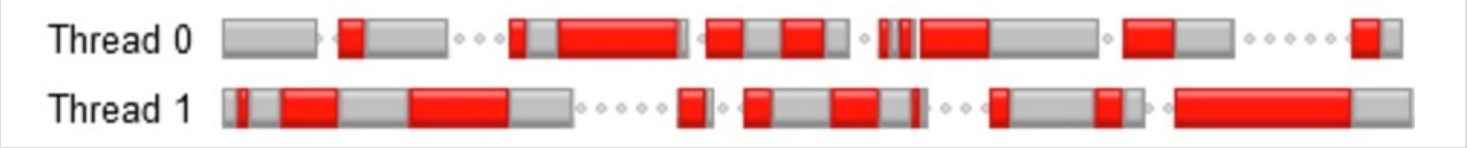
在这次测试中，我们创建一个线程来生成随机数，采用传统的Mersenne Twister生成器来实现。此线程时而获取锁，时而释放锁。获取与释放锁的间隔时间是随机的，但它都很接近我们提前决策出的平均值。举个例子，假设我们要每秒钟获取锁15000次，让持有锁的时间保持在总时间的50%. 下图是部分的timeline。红色说明锁正在被持有，灰色说明锁被释放。

这是个泊松分布过程。如果我们知道生成单个随机数的平均时间–在2.66GHz的四核Xeon处理器上需要6.349ns–那么我们用工作单元(work units）而不是秒来衡量时间。可以用我之前的文章中介绍的方法,How to Generate Random Timings for a Poisson Process,算出获取与释放锁的时间间隔有多少个工作单元。下面代码是C++的实现。我省略了一些细节，喜欢的话，可以在 这里 下载完整的源码

```
1  QueryPerformanceCounter(&start);
2  for (;;)
3  {
4      // Do some work without holding the lock
```

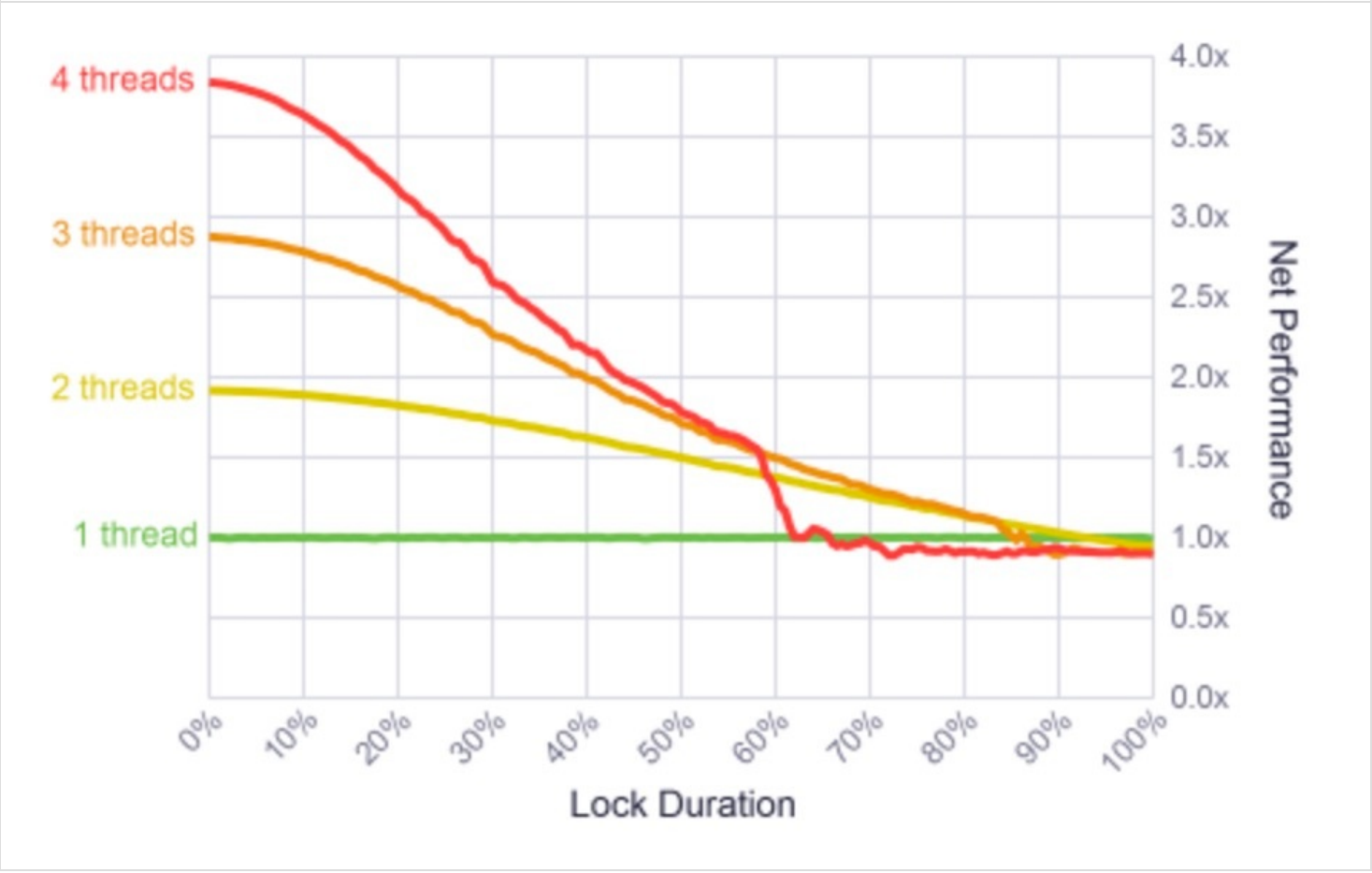
```
5       workunits = (int) (random.poissonInterval(averageUnlockedCount) + 0.5f);
6       for (int i = 1; i < workunits; i++)
7           random.integer();       // Do one work unit
8       workDone += workunits;
9
10      QueryPerformanceCounter(&end);
11      elapsedTime = (end.QuadPart - start.QuadPart) * ooFreq;
12      if (elapsedTime >= timeLimit)
13          break;
14
15      // Do some work while holding the lock
16      EnterCriticalSection(&criticalSection);
17      workunits = (int) (random.poissonInterval(averageLockedCount) + 0.5f);
18      for (int i = 1; i < workunits; i++)
19          random.integer();       // Do one work unit
20      workDone += workunits;
21      LeaveCriticalSection(&criticalSection);
22
23      QueryPerformanceCounter(&end);
24      elapsedTime = (end.QuadPart - start.QuadPart) * ooFreq;
25      if (elapsedTime >= timeLimit)
26          break;
27  }
```

现在假设我们运行两个这样的线程，每个线程运行在不同的CPU核心上。当执行任务时，每个线程有一半的时间是持有锁的，但如果其中一个线程在另一个线程持有锁的情况下试图获取锁，此线程会被强制等待。这就是锁竞争。



在我看来，这是锁在实际程序中应用的非常好的例子。当我们运行上述的场景时，可以发现每个线程会花费25%的时间在等待，75%的时间在执行实际的任务。与单线程相比，两个线程都获得了1.5X的性能提升。

我在2.66GHZ的四核Xeon处理器上做过不同的测试，从一个线程到两个线程，一直到最多四个线程的情况，每个线程都分别运行在不同的CPU核心上。同时，我还改变锁被持有的时间，从锁绝不被持有，到每个线程必须100%的时间持有锁。在所有的case中，锁频率保持一个常数–在执行任务过程中，线程每秒钟获取锁15000次。



结果很有意思。对于短的锁持有时间，比如持有时间占比<10%的情况, 系统可能达到很高的并发性。虽然不是最完美的并发，但很接近。说明锁是非常快的！

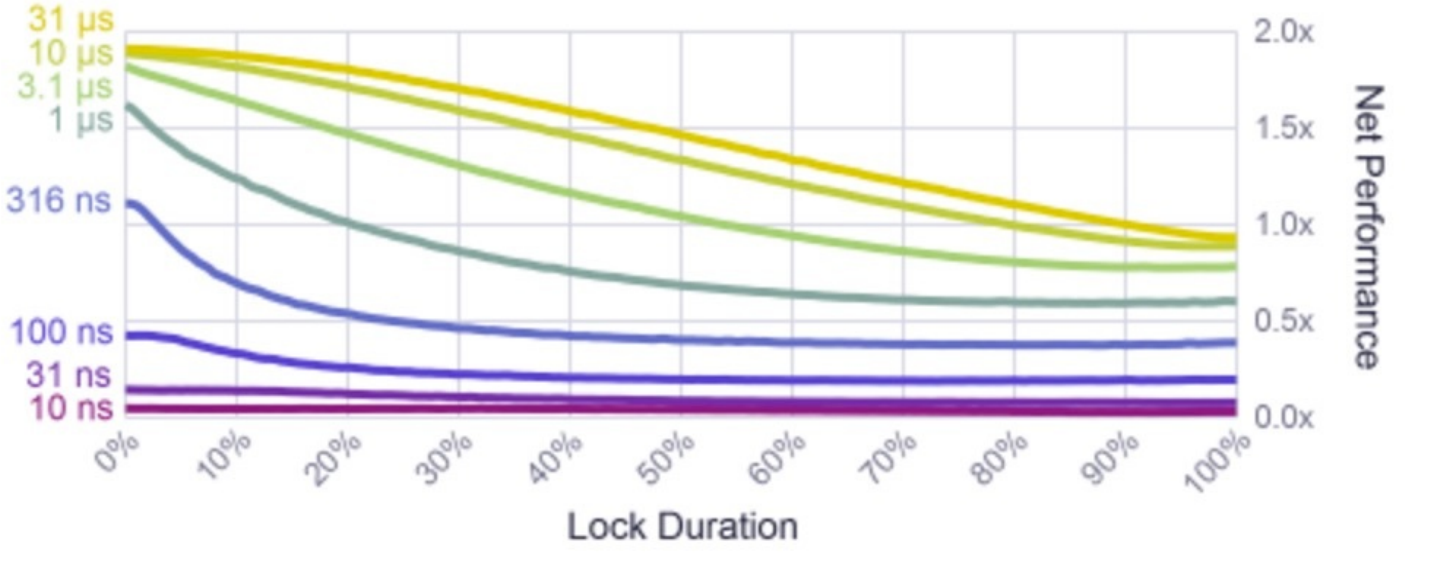
为了把结果解释清楚一些，我用这个分析器分析了多线程游戏引擎中的内存分配锁.在游戏运行时，每秒钟有15000个锁来自三个线程，锁的持有时间在2%左右。正好落在图表中左侧的舒适区(comfort zone).

这些结果都表明一旦锁持有时间超过90%，就没有必要再使用多线程了。这时，单线程会表现的更好。同时，最让人吃惊的是4个线程的性能都急剧下降到60%左右。这看起来像是个异常情况，所以我又重新运行这些测试很多次了，甚至还改变了测试顺序。得到的结果却是一样的。我对此最好的解释就是，测试可能触碰到了Windows分配器的盲区，我没有更进一步的去研究这个问题。

锁频率benchmark

一个轻量级锁也会带来开销。正如我的下一篇文章中说的,对Windows Critical Section的lock/unlock成对操作会花费23.5ns（基于上述测试的CPU). 因此可以说明，每秒钟有15000个锁已经足够少了，锁的开销并不会在很大程度上影响整个结果。但如果我们提高锁频率，又会发生什么呢？

算法中，严格控制锁与锁之间执行的任务数，因此我做了一系列新的测试：锁与锁的间隔时间从10ns到最高31ns(相对应每秒钟大约32000次锁).每次测试都使用两个线程。



正如你想的那样，锁频率很高的话，锁本身的开销就已经高于所执行的任务本身了. 在网上找到的一些benchmarks,包括前面提到的那个分析器， 都落在图表中的右下角。在这些高频率下，和一些CPU指令的规模一样，锁的间隔比较小。好消息是，当锁与锁之间的任务比较简单的时候，无锁编程更可行。

与此同时，结果表明当锁频率达到每秒钟32000次时（锁间隔是3.1us)也是可以接受的。在游戏开发中，内存分配器就可能会在加载过程中达到这个频率。如果锁间隔比较短暂，你仍然可以得到1.5X的并发度。

我们已经了解了一系列锁性能的例子：有性能表现的很好的时候，也有性能慢的跟爬虫似的时候。我已经证明了游戏引擎中的内存分配器一直都能保持非常好的性能。把这个例子运用到实际场景中，不能说锁是慢的。不得不承认，锁很容易被滥用，但你不必太害怕–只要经过仔细的分析，任何情况下都能找出导致性能瓶颈的因素。当你正在考虑锁有多可靠，并去理解锁的相关优化方法时(相比无锁编程)，锁有时候表现的真的非常出色。

写这篇文章的目的是为了让锁得到应有的尊敬–欢迎批评指正。锁在工业应用程序中有广泛的应用，至于锁的性能，并不总是能达到一个很好的均衡。如果你在自己的经验中发现类似这样的例子，非常乐意看到你的评论[注4]。

注释

注释1:一种避免或者降低锁冲突的科学思想是partition，避免资源集中。例如，对于hashtable，可以由之前的一个hashtable对应一把锁，改为每个bucket配置一把锁，这样冲突将大大降低。再例如，计数程序，如果大家都对同一个全局变量进行读写而加一把锁，那么冲突严重，可以适当的选择多个计数器，不同的线程累加对应的计数器，一个线程负责将这些计数器的值求和。等等等等。

注释2: 这里的无锁编程，原文为lock free。不含锁编程，原文为lockless。但是需要注意的是，lock free并不是无锁的意思，它的本质是说一组线程，总有（至少）一个线程能make progress，和有没有锁没有本质联系。lock free目前一般都翻译为无锁（有些地方也翻译为锁无关），因此本文也采用这种译法，但是读者需要特别注意。另外lockless就是真正的无锁、不包含锁的编程。

注释3: Doug Lea是并发编程的大牛，《Java并发编程实战》的作者之一，非常乐意分享。他写的这个分配器非常出名，glibc所采用的内存分配器实现就是基于他设计的算法。

注释4: 本文的描述和试验可能让人有点迷糊，这里提供一下Paul E. McKenney大叔在他的著作《Is Parallel Programming Hard, And, If So, What Can You Do About It?》中第4章中的例子来解释，让读者更好的理解：

```
1  pthread_rwlock_t rwl = PTHREAD_RWLOCK_INITIALIZER;
2  int holdtime = 0;
3  int thinktime = 0;
4  long long *readcounts;
5  int nreadersrunning = 0;
6  #define GOFLAG_INIT 0
```

```
#define GOFLAG_RUN  1
#define GOFLAG_STOP 2
char goflag = GOFLAG_INIT;
void *reader(void *arg)
{
   int i;
   long long loopcnt = 0;
   long me = (long)arg;
   __sync_fetch_and_add(&nreadersrunning, 1);
   while (ACCESS_ONCE(goflag) == GOFLAG_INIT) {
     continue;
   }
   while (ACCESS_ONCE(goflag) == GOFLAG_RUN) {
     if (pthread_rwlock_rdlock(&rwl) != 0) {
       perror("pthread_rwlock_rdlock");
       exit(-1);
     }
     for(i=1;i<holdtime;i++){
       barrier();
     }
     if (pthread_rwlock_unlock(&rwl) != 0) {
       perror("pthread_rwlock_unlock");
       exit(-1);
     }
     for (i=1;i<thinktime;i++) {
       barrier();
     }
     loopcnt++;
   }
   readcounts[me] = loopcnt;
   return NULL;
}
```

其中16-18行等待测试开始的信号；19行开始测试；holdtime控制临界区的长短，thinktime用来控制两次申请锁之间的间隔。测试的时候有三个变量：holdtime、thinktime、线程数（1个、2个、4个、直到核数的两倍）。试试看。

Acknowledgement

本译文由 @Diting0x 与 @睡眼惺忪的小叶先森 共同完成，在原文的基础上添加了许多详细注释，帮助大家理解。

感谢好友@skyline09_ 与@小伙伴-小伙伴儿 阅读了初稿，并给出意见。

原文： http://preshing.com/20111118/locks-arent-slow-lock-contention-is/

Source Insight for Linux

1条评论

qiangliu

~

9小时前 回复 顶 转发

社交帐号登录: 微信 微博 QQ 人人 更多»

说点什么吧...

发布

多说强力驱动