可扩展的被动读写锁

Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks

背景介绍

读者-写者问题描述的是在多线程共享数据的场景下，其中一些线程要对数据进行修改，另一些线程只读取这些数据而不作修改。为了避免数据竞争，常用的机制是对数据加读写锁。读写锁在同一时间内最多允许一个线程进行写操作，或者在没有线程进行写操作时允许多个线程进行读操作。

但是很多读写锁实现中，虽然其扩展性相对于最简单的中心化读写锁有很多提升，但是在这些实现中，要么读者之间仍然存在相互竞争，要么写的延迟非常大。还有一些读写锁虽然可扩展，但无法对一些操作系统的行为做出处理，如线程切换、调度、睡眠等。

该论文实现了一种新的读写锁，称为被动读写锁（passive reader-writer lock, prwlock）。被动读写锁充分结合了操作系统在不同线程之间调度CPU的特点，并利用了核间中断（inter-processor interrupts, IPI）的硬件特性，达到了非常好的读端可扩展性和相对较低的写延迟。在64核AMD机器上表现出了非常好的性能。

已有的读写锁实现

中心化读写锁

早在1971年，P.J.Courtois，F.Heymans和D.L.Parnas [1]在提出读者-写者问题时，便提出了一种基于计数器和互斥锁的实现方式。在读侧获取锁时，将读者计数器加一，若发现自己是当前唯一的读者，则获取写侧互斥锁；读侧获取锁过程需要加另一个互斥锁使之成为临界区，防止第二个读者在第一个读者还没有获得写侧互斥锁时就开始进行读访问；在读侧释放锁时再原子减一，若发现自己是当前唯一的读者，则释放写侧互斥锁；而写侧只需对写侧互斥锁进行操作。由于读侧获取锁时的互斥性，其读性能受到很大的限制。

MCS读写锁

J. Mellor-Crummey和M. Scott.[2]在提出MCS[[1]](#footnote-1)互斥锁后，对它做出一定修改便提出了MCS读写锁。MCS互斥锁用链表构建队列，获取锁时将队列中自己的前驱的next指针指向自己，在释放锁时将锁传给队列中自己的后继。

在MCS读写锁中，链表节点分为了读和写两类。当一个读者成功地获取了锁时，还需要查看其后继的情况；若有后继且也是读者，则将之唤醒。同时，而当写者成功地排队到队头时，可能还要些读者没有释放锁，所有读者计数器仍然是必需的，在读者获取和释放锁时需要原子加减，若自己是最后一个释放锁的读者时，且有下一个写者在等待，则将其唤醒。

大读锁

为了达到读者的可扩展性，需要尽可能避免读者计数器的开销。大读锁（big reader lock, brlock）的实现非常简单，每个线程都有自己的一个互斥锁，在读时只需要获取自己的互斥锁，在写时需要按顺序获取所有线程的互斥锁。

大读锁达到了读锁的可扩展性，但是其写锁的延迟仍然太高。尤其在实际应用中常常出现超线程场景，一些休眠状态的线程也需要写者去获取锁。

C-SNZI与GOLL

F. Ellen，Y. Lev，V. Luchangco，和 M. Moir [3]提出可扩展非零指示器（scalable non-zero indicator, SNZI）。SNZI的功能类似于计数器，有加一（到达，arrive）和减一（离开，depart）的功能，但只能查看它是否非零（query），而不能查看具体值。将线程递归地分组，形成树状结构，每个节点上有一个计数器用于指示非零子节点数。当组内计数器从0加到1或从1减到0时，才递归地更新父节点上的计数器。因此，每个线程对计数器进行加减的时候，只会与组内的其他线程产生冲突，从而增强了可扩展性。

Y. Lev，V. Luchangco，和M. Olszewski [4]在SNZI的基础上提出可关闭的SNZI（closable SNZI, C-SNZI），并基于C-SNZI实现了通用OLL[[2]](#footnote-2)读写锁（General OLL reader-writer lock, GOLL）。C-SNZI额外提供打开（open）和关闭（close）两种操作，在关闭的状态下，到达操作将不会改变计数器值。这样，当写者获取锁时，只需要关闭C-SNZI，之后到达的读者都将进入等待队列，若此时C-SNZI的值仍然非零，该写者也将进入等待队列。而当写者释放锁时，或当读者释放锁并发现自己是最后一个退出的读者时，需要唤醒等待队列中的后继。

被动读写锁

算法

该论文作者提出了被动读写锁，使得读者和写者都达到了较高的性能。在大读锁中，每个线程都有一个状态（空闲/已锁），在写者获取锁的过程中，需要检查所有线程的状态。在超线程场景下，这将使得写者获取锁的开销过大。在被动读写锁中，写者只需要访问每个处理器核的状态（空闲/被动）。当一个核获取读锁时，状态由空闲变为被动；若所运行的线程在释放锁之前被调度走，把该线程的状态称为主动状态，处理器核的状态变为空闲，且该线程再次被运行的时候也不会回到被动状态；释放读锁时，状态变回空闲。同时全局共享的计数器记录了当前主动的读者数，每个线程在被动状态被调度走、或在空闲状态释放锁时需要对该计数器进行加减。

读者获取锁的过程只有两步，先等待写侧互斥锁变为空闲，随后将自己的状态变为被动。

而写者首先要获取写侧互斥锁，这个锁只有写者会持有，如果在获取写侧互斥锁的时候发现等待队列中有前驱，则获取到写侧互斥锁后可以直接开始写操作。否则的话应该检查读者的状态。注意到这个时候不会再有新的线程变为被动了，但还会有被动线程变为主动，所有应该先检查每个处理器核，等待其状态都不再为被动，随后再等待主动读者计数器归零。

读者在被动状态下拥有较高的性能，而在主动状态下，由于本身被调度走，一次计数器加一的操作的开销可以接受。

该算法的实现需要对操作系统的调度器加一些挂钩，使得在线程切换时能及时更新处理器核的状态和主动读者计数器。

基于版本的写同步[[3]](#footnote-3)

在获取写锁时需要做的一个操作是检查每个处理器核，等待其状态都不再为被动。这要求每次状态的改变之后，写者能看到这项更新；在某些硬件平台上，这就要求读者在状态改变之后加内存屏障（memory barrier）。在这种情况下，很多内存屏障其实是不必要的，有可能在状态改变的时候根本没有写者。为了消除这层内存屏障，写者检查的读者状态不再是空闲/被动，而是一个新的状态版本（version）。每当写者获取了写侧互斥锁之后，会将版本号加一；同时每个处理器核会维护一个局部版本号，在获取读锁前、释放读锁后和由被动状态变为主动状态后，都会更新其版本号。写者在将版本号加一后应该等待所有处理器的版本号更新至最新，但有些处理器核上可能会较长时间根本不访问该读写锁；这时需要用IPI来通知这些核更新。当一个核收到IPI时，只要此时状态为空闲，则将版本号更新至最新。

唤醒机制

为了不占用过多的CPU时间，在所有等待期间可以使当前线程进入睡眠状态。当写者释放锁的时候，可能需要唤醒多个读者。此时若使用传统的全局唤醒队列，则效率较低。为了加速该过程，将此全局唤醒队列改为逐核唤醒队列。每个核维护自己的唤醒队列，当在一个核上运行的线程进入睡眠状态时，会被加入到该核的唤醒队列中。当一个核释放写锁时，会通过mwait/monitor机制来通知其他核唤醒各种的队列中需要唤醒的线程。

用户空间

由于被动读写锁用到了很多需要操作系统权限的机制，在用户空间使用该读写锁需要系统调用开销。为了减少读侧的性能开销，希望将读锁在用户空间内完成。为此需要作出修改，将每个核的空闲/被动状态变为每个线程的空闲/被动/主动状态，在被调度走的时候将被动状态变为主动状态，在释放锁时若是主动状态则需要将主动读者计数器减一。

实验

将Linux内核中虚拟内存管理部分的读写锁替换为prwlock，在64核的机器上运行Histogram、Metis和Psearchy三个测试程序，分别获得了185%、55%和20%的加速。而在用户空间，通过将pthread\_rwlock替换为prwlock，使得Kyoto Carbinet数据库的性能变为原来的7.37倍。

结论

该论文提出了被动读写锁，提供了具备很好的扩展性的读性能，以及相对较好的写性能。被动读写锁可以在内核空间和用户空间使用。在64核的机器上的测试确认了其高性能与高可扩展性，在内核的虚拟内存管理模块和在数据库中使用都能显著地提升程序的性能。

参考文献（原文除外）

[1] P. Courtois, F. Heymans, and D. Parnas. Concurrent control with readers and writers. Comm. ACM, 14(10), 1971.

[2] J. Mellor-Crummey and M. Scott. Synchronization without contention. In ASPLOS, pages 269-278. ACM, 1991.

[3] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable nonzero indicators. In PODC, pages 13–22. ACM, 2007.

[4] Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. In SPAA, 2009.

1. Mellor-Crummey和Scott的缩写 [↑](#footnote-ref-1)
2. Olszewski、Lev 和Luchangco的缩写 [↑](#footnote-ref-2)
3. 目前没有完全理解该机制所带来的好处，比如在x86等架构上内存屏障不构成性能问题，相比该机制的开销来说得不偿失 [↑](#footnote-ref-3)