读写自旋锁详解，第 3 部分

基于简单共享变量的读写自旋锁的不足

本系列文章的第 2 部分中给出的实现都基于简单共享变量，简洁实用，但在大规模多核、NUMA 系统上可扩展性较差。我们说某个读写自旋锁的实现是可扩展的，通俗地讲是指在线程访问模式（读者写者数目之比、各自到来的频率及持有锁的时间）不变的前提下增加处理器的个数，线程的吞吐量（单位时间内获得锁的线程数目）也随之大幅增加。如果能接近线性（甚至由于缓存的影响使得超线性）增加，我们说该实现是高可扩展的。

基于简单共享变量的读写自旋锁的可扩展性较差本质上是因为操作共享变量的代价过大：如果系统不能保证缓存的一致性，读写共享变量导致总线的流量激增；即使提供了一致性的缓存，频繁的写操作也会增大处理器间的缓存同步开销。具体而言有 3 点：

1. 所有等待线程均在某个（些）共享变量上自旋。
2. 线程需要通过共享变量来分享一些信息，比如需要一个 nr_readers 的整型变量记录同时持有锁的读者。
3. 这些共享变量频繁被修改。

因此我们可以针对上述 3 点，分别提出相应的改进策略：

1. 所有等待线程均在局部变量上自旋。
2. 尽量减少共享变量的数目。
3. 如果必须使用共享变量，那么考虑使用该变量的可扩展实现。

基于单向链表的公平读写自旋锁

首先我们考虑如何让等待线程只在局部变量上自旋，这涉及四个问题：

1. 线程到来时是否需要自旋？如果锁被写者持有或尚有有等待线程，显然新来的线程只能选择自旋等待；如果锁无人持有，线程无需自旋；如果只有读者持有且新来的线程也是读者，那么为了提高并发性，新来的读者最好不用等待。
2. 谁负责通知某个等待线程结束自旋？这个问题比较容易回答，应该由持有锁的线程负责通知。
3. 何时通知？持有锁的写者必须在释放锁的时候才能通知下一个候选持有者：而读者在持获得锁后应该检查一下下一个候选者是否也是读者，若是，则立即通知。
4. 如何通知？这个问题要求我们能够拥有某种能力可以找到下一个候选持有者，即需要使用特殊的数据结构。

我们很容易想到可以用单向链表将申请线程组织成一个队列，每个线程能够通过指针寻找到自己的所有后继，而且线程申请锁的顺序与线程在队列中的位置保持一致，因此用单向链表可以实现一个公平的读写自旋锁 [3]。

凭空想象算法似乎有些困难，我们还是先在草稿纸上画一个示意图，有了感性认识之后再考虑数据结构和算法的细节。

图 1. 基于单向链表的公平读写自旋锁示意图

点击查看大图

从图上可以看到，为了构建单向链表，每个线程必须有一个自己私有的链表节点数据结构，里面至少包含三个域：自身角色 type、自旋变量 waiting 和指向直接后继的 next 指针。我们还需要一个指向链表尾部的指针 tail，新到的线程通过它可以获得其直接前驱，从而将自己插入链表。这个 tail 指针应当放到锁的数据结构中。

值得注意的是，插入链表这个动作并不是原子的，至少需要 2 个操作：原子地获得 tail 指针并将其指向自己；将直接前驱的 next 指针指向自己。但是线程的直接前驱随时可能释放锁，并继续申请锁而重用其数据结构，因此需要约束线程的行为以保证单向链表的一致性。我们规定：线程释放锁之前必须检查自身是否存在直接后继线程，如果存在则保证在直接后继插入链表之后才能释放锁。这可以通过检查 tail 和 next 指针实现。

连续的读者可以同时持有锁，但是它们并不一定按照申请的顺序释放锁，故而难以在释放锁的时候继续保持这些读者间的链表结构。这是因为我们使用的是单向链表，释放锁的读者很难用简单高效的方法将其节点数据结构从链表中摘下 [4]（即使保存了直接前驱的指针，也可能已经失效），而且该读者也许需要继续申请锁而重用该数据结构，导致其尚未释放锁的直接前驱读者也不能再使用 next 指针遍历后继。单向链表的结构被破坏带来如下 2 个挑战：

1. 从图 2 可以看到，如果 A1– A3都释放锁的时候，应该通知 A4这个写者。但是 A1– A3如何知道自己是最后一个持有者？
2. 如果 A2知道自己最后释放锁，但是 A4并不是它的直接后继，A2如何通知 A4呢？

一种直观简单的解决方案是用一个计数器 nr_readers 记录当前同时持有锁的数目，读者获得锁前原子递增 nr_readers；释放锁的时候原子递减，如果递减后 nr_readers 为 0，表示自己此刻是最后一个持有锁的读者。此外还需要一个记录等待线程中的第一个写者的指针 next_writer。这两个变量也应当放到锁的数据结构中。

读者并发还带来一个棘手的问题。还是以图 2 为例，当读者 A1获得锁时，如果此时知道 A2已经到来了（通过检查自己的 next 指针），那么应该由 A1通知 A2。但是可能 A2到来的时候 A1已经在访问共享资源或者正在释放锁，那么 A2应该自行获得锁。因为获得锁涉及到原子递增 nr_readers，所以 A1和 A2必须达成一致。我们的方案是：读者到来时先检查其直接前驱读者是否还在自旋，若是，则原子地在其前驱的节点数据结构中标记一个特殊值。如果标记成功，表明前驱尚未结束自旋，那么由前驱负责通知；如果标记失败，说明前驱已经退出自旋而获得锁，那么读者直接获得锁。标记值可以选得有意义一些，这儿我们选择线程的角色，于是线程很容易知道直接后继的类型，而不用通过 next 指针进一步查看。我们在节点数据结构中增加变量 successor_type。

下面我们详细阐述读写自旋锁的算法。假设申请线程为 A。

A 是读者，申请锁：

1. A 使用原子交换操作将读写自旋锁的 tail 指针指向自己的 qnode 结构以确定在链表中的位置，并返回原来的 tail 值作为自己的直接前驱 pred。即使多个线程同时申请锁，由于交换操作的原子性，每个执行线程的申请顺序将会被唯一确定，不会出现不一致的现象。
2. 如果 pred 等于 NULL，说明锁无人持有或只有读者持有（A 的直接前驱是读者，且已经释放锁），那么 A 可以立即持有锁，跳到第 4 步。
3. 此时 pred 不等于 NULL。如果 pred 是写者，A 只能依赖 pred 在释放锁的时候通知自己，所以 A 所要做的只是将 pred->next 指向自己的，然后自旋等待。如果 pred 是读者，那么需要按照前面描述的那样，向 pred 标记 A 的读者角色。如果标记成功，则取得锁，跳到第 4 步；否则自旋等待。
4. A 准备结束函数调用，但还需要检查一下自己的 successor_type 变量，如果直接后继是读者，那么先等它将链表构建完整，然后通知其结束自旋。

A 是读者，释放锁：

1. A 先检查是否存在直接后继。这可以用原子比较 tail 指针是否还是指向自己并更新为 NULL 的操作实现。如果成功地将 tail 指针更新为 NULL，说明没有直接后继，那么跳到第 3 步。
2. A 有直接后继 B，先需要等待 B 将链表构建完整，然后检查 B 是不是写者，若是，则将锁结构中的 next_writer 指针指向 B。
3. A 判断自己是否是最后一个持有者，若是，且 next_writer 指针不为 NULL，则应该通知 next_writer。具体做法是：A 原子递减 nr_readers，如果 nr_readers 新值大于 0，说明还有读者持有锁，于是 A 直接拍拍屁股走人。如果 nr_readers 新值等于 0，这并不能说明 A 在后面通知 next_writer 时还是最后一个持有者，因为在 A 执行后续操作的间隙中可能依次来了新的读者 C 和 D。假定读者 D 以极快的速度获得并执行释放锁，将 tail

指针置为 NULL。此刻又来了写者 W，因为 W 发现 tail 为 NULL，那么 W 必须将 next_writer 置为自己，否则 A 或 D 无法通知 W。现在轮到 A 继续执行了，A 发现 next_writer 不为 NULL，于是试图通知 W 结束自旋，但是 C 仍然持有锁，导致错误。这个例子说明通知 next_writer 前还需要再检查一下 nr_readers 的值，如果 nr_reader 仍然等于 0，才能说明是最后一个持有锁的读者。如果 C 也迅速释放锁，那么 A 和 C 可能同时试图通知 W。通知一个已经退出自旋的线程是危险的，因为该线程可能重新申请锁，而导致提前退出自旋。因此 A 通知 W 时，需要检测 next_writer 指针是否没有变化并原子地将其置为 NULL，这样 C 会发现已经有人通知过 W 了。

A 是写者，申请锁：

1. A 使用原子交换操作将读写自旋锁的 tail 指针指向自己的 qnode 结构以确定在链表中的位置，并返回原来的 tail 值作为自己的直接前驱 pred。
2. 如果 pred 等于 NULL，说明锁无人持有或仍有读者持有。A 先将 next_writer 指向自己，然后检查 nr_readers，如果 nr_readers 大于 0，说明尚有读者持有锁，那么自旋等待。如果 nr_readers 等于 0，也有可能某个（些）读者正在释放锁，因此 A 需要检测 next_writer 指针是否仍然指向自己并原子地将其置为 NULL。如果成功置为 NULL，获得锁并返回；否则等待前面的读者通知自己。
3. 如果 pred 不等于 NULL，那么 A 标记自己的角色并将链表构建好，然后自旋等待。

A 是写者，释放锁：

1. A 先检查是否存在直接后继。这可以用原子比较 tail 指针是否还是指向自己并更新为 NULL 的操作实现。如果成功地将 tail 指针更新为 NULL，说明没有直接后继，那么直接返回。
2. A 有直接后继 B，先需要等待 B 将链表构建完整，然后通过 next 指针通知 B 结束自旋。如果 B 是读者，还需要先原子递增 nr_readers。

具体代码如清单 5 所示，有兴趣的读者朋友可以尝试证明一下正确性。

清单 1. 基于单向链表的公平读写自旋锁的实现

```
#define NONE 0 ``#define READER 1 ``#define WRITER 2 typedef struct _qnode{ `` ``int type; `` ``union { `` ``volatile int state; //(1) `` ``struct { `` ``volatile short waiting; `` ``volatile short successor_type; `` ``}; `` ``}; `` ``_qnode *volatile next; ``} __cacheline_aligned_in_smp qnode; typedef struct { `` ``qnode *volatile tail; `` ``qnode *volatile next_writer; `` ``atomic_t nr_readers; ``} rwlock_t void init_lock(rwlock_t *lock) ``{ `` ``lock->tail = NULL; `` ``lock->next_writer = NULL; `` ``lock->nr_readers = ATOMIC_INIT(0); ``} void reader_lock(rwlock_t *lock) ``{ `` ``qnode *me = get_myself(); //(2) `` ``qnode *pred = me; ``me->type = READER; `` ``me->next = NULL; `` ``me->state = 1;// successor_type == NONE && waiting == 1 ``xchg(&lock->tail, pred); `` ``if (pred == NULL) { `` ``atomic_inc(&lock->nr_readers); `` ``me->waiting = 0; `` ``} else { `` ``If ((pred->type == WRITER) `` ``|| (cmpxchg(&pred->state, 1, 0x00010001) == 1)} { //(3) `` ``pred->next = me; `` ``while (me->waiting) `` ``cpu_relax(); `` ``} else { `` ``atomic_inc(&lock->nr_readers); `` ``pred->next = me; `` ``me->waiting = 0; `` ``} `` ``} ``if (me->successor_type == READER) { `` ``while (me->next == NULL) `` ``cpu_relax(); `` ``atomic_inc(&lock->nr_readers); `` ``me->next->waiting = 0; `` ``} ``} void reader_unlock(rwlock_t *lock) ``{ `` ``qnode *w; `` ``qnode *me = get_myself(); ``if ((me->next != NULL) ``|| (cmpxchg(&lock->tail, me, NULL) != me)) { `` ``while (me->next == NULL) `` ``cpu_relax(); ``if (me->successor_type == WRITER) `` ``lock->next_writer = me->next; `` ``} ``if ((atomic_dec_return(&lock->nr_readers) == 1) `` ``&& ((w = lock->next_writer) != NULL) ``&& (atomic_read(lock->nr_readers) == 0) ``&& (cmpxchg(&lock->next_writer, w, NULL) == w)) `` ``w->waiting = 0; ``} void writer_lock(rwlock_t *lock) ``{ `` ``qnode *me = get_myself(); ``qnode *pred = me; ``me->type = WRITER; `` ``me->next = NULL; `` ``me->state = 1; ``xchg(&lock->tail, pred); `` ``if (pred == NULL) { `` ``lock->next_writer = me; `` ``if ((atomic_read(lock->nr_readers) == 0) `` ``&& (cmpxchg(&lock->next_writer, me, NULL) == me)) `` ``me->waiting = 0; `` ``} else { ``pred-
```

`>successor_type = WRITER; ``smp_wmb(); //(4) `` ``pred->next = me; ``}` `while (me->waiting) `` ``cpu_relax(); ``}` `void writer_unlock(rwlock_t *lock) ``{ `` ``qnode *me = get_myself();` `` if ((me->next != NULL) ``|| (cmpxchg(&lock->tail, me, NULL) != me)) { ``while (me->next == NULL) `` ``cpu_relax(); ``` `` if (me->next->type == READER) `` ``atomic_inc(&lock->nr_readers);` `me->next->waiting = 0; `` ``} ``}`

1. 使用 union 结构是为了方便后面将 successor_type 和 waiting 合在一起做原子操作。
2. 如果事先知道线程的数目，例如代码用于中断上下文，qnode 可以包装在 lock 数据结构中，每个线程一个；否则，可以使用线程本地存储区（Thread Local Storage，使用 gcc 关键字 __thread）分配 qnode。我们不关心 qnode 的分配细节，假定有个 get_myself() 函数可以获得当前线程的 qnode。
3. cmpxchg 原子操作将 [successor_type，waiting] 原子地从 [NONE，TRUE] 改变为 [READER，TRUE]。
4. 此处的“Write Memory Barrier”目的是确保对 successor_type 的赋值先完成。因为这两个赋值没有相关性，如果处理器乱序执行写指令，且对 next 的赋值先完成，那么链表就已构建完毕，前驱可能随时释放锁导致 pred 指针失效。

进一步提高读写自旋锁的可扩展性

基于单向链表的读写自旋锁并不完美，因为线程还是得操作额外的共享变量 nr_readers 和 next_writer，这是由于读者释放锁的时候无法继续保持单向链表的结构。一种改进想法是使用双向链表 [5]，因为双向链表的插入和删除操作能够做得相对高效。于是读者释放锁的时候可以将自己的节点结构从双向链表中删除，从而继续保持链表的结构。读者通过判断前驱指针是否为 NULL 就知道自己是不是最后一个持有者，而且也再不需要 next_writer 指针。但是该算法中对双向链表的操作使用了节点级别的细粒度普通自旋锁，在连续读者较多且几乎同时释放读写自旋锁的情况下，同步开销依然巨大。

第二种想法是为每个线程分配一个局部的普通自旋锁，读者只需获得自己的自旋锁即可，而写者必须将包括自己在内的所有人的自旋锁都获得才行 [6]。这个算法显然是个读者优先的实现，在写者较少的情况下可扩展性相当理想。不足之处有两点：一是写者得知道其它线程的自旋锁在哪儿，因此比较适用于固定线程的场景；其次是读者数目越多，对写者也就越不公平。

我们看到：一般而言，在一段可观测的时间内，读者数量远远大于写者，很多时候甚至没有写者。因此在基于单向链表的实现中，只有共享变量 nr_readers 才是一个明显的瓶颈。进一步分析可知，我们其实并不需要知道 nr_readers 的具体值，只是想了解 nr_readers 是否大于 0 还是等于 0，于是 Sun 的研究人员提出使用一种称为可扩展非零指示器（Scalable Non-Zero Indicator）的数据结构，大大降低了线程间的同步开销 [7]。

结束语

本系列文章详细论述读写自旋锁的原理和实现，本文是其中的第三部分，针对大规模多核系统讨论如何设计和实现可扩展的读写自旋锁。

相关主题

- 大家可以从 kernel.org 下载 Linux Kernel 2.6.39源代码。
- 林昊翔、秦君，《 Linux 内核的排队自旋锁 (FIFO Ticket Spinlock) 》。这篇文章详细介绍了 Linux 内核排队自旋锁的设计与实现。
- John M. Mellor-Crummey, Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. 清单 6 的代码源于这篇文章。
- Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming.
- Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A Fair Fast Scalable Reader-Writer Lock.
- W. C. Hsieh and W. E. Weihl. Scalable reader-writer locks for parallel systems.

- Yossi Lev, Victor Luchangco and Marek Olszewski. Scalable Reader-Writer Locks.
- 在 developerWorks Linux 专区 寻找为 Linux 开发人员（包括 Linux 新手入门）准备的更多参考资料，查阅我们 最受欢迎的文章和教程。
- 在 developerWorks 上查阅所有 Linux 技巧 和 Linux 教程。
- 随时关注 developerWorks 技术活动和网络广播。