【译】无锁算法导论

孔俊

2023-07-14

Contents

1 译者序

An introduction to lockless algorithms是 Paolo Bonzini 在 LWN.net 上发布的无锁编程系列文章的第一篇，清楚明了地阐述了内存模型中的 acquire/release 语义。

当传统的锁定原语（locking primitives）无法使用或性能不足时，无锁算法就会引起 Linux 内核的兴趣。因此，无锁算法时不时出现在 LWN 上。LWN 最近一次提及无锁算法是在七月，这促使我写下了这一系列文章。更频繁出现的话题是 read-copy-update（RCU——这些2007 年的文章 仍未过时），引用计数，以及种种将无锁原语（lockless primitives）包装成更高级、更理解的 API 的技术。这些文章深入到了无锁编程背后的概念以及如何在内核中运用。内存模型的底层知识通常被认为是连经验丰富的黑客都感到害怕的高级东西，我们的编辑在七月的文章中写道："需要不一样的脑子才能真正理解内存模型。"他说 Linux kernel 内存模型（尤其是 Documentation/memory-barriers.txt）能把小孩吓哭，acquire 和 release 这样的词语可能也同样吓人。与此同时，像 RCU 和 seqlocks 一类的机制在内核中运用的如此广泛，以至于几乎每个开发者早晚会遇到无锁编程接口。因此，你最好能对无锁原语有基本的理解。通过这一系列文章，我会说明 acquire 和 release 语义究竟是什么，并介绍五种相对简单的能够覆盖无锁原语大多数用例的模式。

2 获取，释放和"先发生于"

为了从同步原语（synchronization primitives）舒适区向无锁编程迈一大步，我们首先应该看看为什么锁（locks）能起作用。通常用互斥（mutual exclusion）解释：锁阻止多个线程并发读写同一数据。但是"并发"究竟意味着什么？线程 T 写入数据后，线程 P 访问访问它时究竟会发生什么？

为了回答这些问题，我们需要求助于 Leslie Lamport 在 1978 年的论文 *Time, Clocks and the Ordering of Events in a Distributed System* 中建立的理论框架。根据该论文，分布式系统中的事件（event）根据事件 P 是否先发生于（happen before）另一事件 Q 可以排序为：

- 单一线程中的事件是全序（total）。简单地说，你可以认为同一线程中，任意两个事件一个先发生一个后发生。
- 对于发生在不同线程的两事件，如果事件 P 是消息发送且事件 Q 是与之匹配的消息接收，则事件 P 先发生于（happens before）事件 Q。
- 此关系是可传递的（transitive）。因此，如果事件 P happens before 事件 Q，且事件 Q happens before 事件 R，则事件 P happens before 事件 R。

"先发生于"关系是一个偏序（partial ordering）：可能存在两个事件 P 和 Q 彼此间不存在 happens before 关系。这种情况下，这两个事件是并发的（concurrent）。还记得锁是如何阻止并发访问同一数据结构的吗？这是因为，当你用锁保护某个数据结构时，此数据结构上的所有访问构成一个全序，

看起来就像单线程访问一样。Lamport 的理论框架解释了当线程将锁移交给另一个线程时发生了什么：某种“消息传递”确保线程 T 的解锁（unlock）操作“先发生于”线程 U 的锁定（lock）操作。

上面的例子说明，这不只是一个理论：多核处理器为了确保缓存一致，CPU 间通过总线交换消息，例如 Intel 的 QPI 和 AMD 的 HyperTransport。然而，这一层面上的细节远远超过了我们“导论”的目标。与探究处理器的底层缓存一致原理相反，我们将推广“先发生于”的定义以涵盖所有类型的同步原语。

Lamport 洞察到同步发生于两个线程使用对称操作访问同一数据结构时。在我们推广后的定义中，我们将列出多对同步线程的操作（例如通过同一队列发送和接收消息）。我们还会进一步阐明哪些操作是 release，哪些是 acquire。我们会等价地说他们“具有 release（或 acquire）语义”。

在一对操作内，release 操作同步于（synchronizes with）与之匹配的 acquire 操作。“同步”指无论何时一个线程执行 release 操作，另一线程执行与之匹配的 acquire 操作，happens before 关系增加一条从从 release 线程到 acquire 线程的边。“先发生于”关系仍然是一个偏序，但由于传递性，现在它横跨两个甚至更多线程。更正式地说：

- 单一线程内的操作次序是全序。
- 如果具有 release 语义的操作 P 同步于具有 acquire 语义的操作 Q，则操作 P 先发生于操作 Q，即使他们发生在不同线程。
- 和之前一样，此关系是一个可传递的偏序。

如果将 release 语义视作消息发送，将 acquire 语义视作消息接收，那么旧定义仍然成立。消息发送使发送线程同步于接收此消息的线程。我们可以用新定义重新描述我们先前的发现：为了让锁起作用，解锁必须具有释放语义，且必须同步于锁定——锁定相应地必须具有 acquire 语义。不论是否竞争锁，锁带来的“先发生于”边确保锁顺利地从一个线程移交给另一线程。

require 和 release 语义可能看起来像是个抽象概念，但确实为许多常见的多线程编程实践提供了简单的解释。例如，考虑以下两个访问全局变量`s`的应用态线程：

```
1      thread 1                                thread 2
2      --------------------------------        ------------------------
3      s = "hello";
4      pthread_create(&t, NULL, t2, NULL);
5                                              puts(s);
6                                              s = "world";
7      pthread_join(t, NULL);
8      puts(s);
```

这两个线程对变量的访问安全吗？线程 2 可以假设会从`s`读出`"hello"`吗？线程 1 可以假设`s`在`pthread_join()`后值为`"world"`吗？答案是肯定的，我们可以用 acquire 和 release 语义解释：

- `pthread_create()`具有 release 语义且同步于线程 2 的启动（具有 acquire 语义）。因此，线程创建前写入的任何数据都可以从此线程安全地访问。

- 退出线程 2 具有 release 语义且同步于`pthread_join()`（具有 acquire 语义）。因此，线程退出前写入的任何数据都能在`pthread_join()`后安全地访问。

注意，数据在无锁的情况下从一个线程流向另一个线程：恭喜，你已经通过第一个无锁编程示例做到了这一点。总结：

- 如果程序员想让线程 2 “看到”任何先前发生于线程 1 的“效果”，那么这两线程需要彼此同步：通过线程 1 的 release 操作和线程 2 的 acquire 操作实现。
- 知道哪些 API 提供 acquire/release 语义让你能编写依赖于这些 API 提供的次序保证的代码。

理解了 release 和 acquire 如何在高级同步原语中起作用后，我们现在可以在单独的内存访问上下文中思考他们。

3 消息传递模式

上文我们了解了`pthread_create()`和`pthread_join()` 的 acquire/release 语义允许线程创建者与被创建的线程彼此交换信息。现在，我们研究如何以无锁的方式在线程运行时进行这类通信。

如果消息是简单的标量值，例如布尔值，那么可以直接通过内存地址读写。然而如果像下面的例子一样，消息是一个指针，这会发生什么：

```
1     thread 1                              thread 2
2     -------------------------------       ------------------------
3     a.x = 1;
4     message = &a;                         datum = message;
5                                           if (datum != NULL)
6                                             printk("%d\n", datum->x);
```

如果`message`初始化为`NULL`，线程 2 将读到`NULL`或`&a`，我们不确定是哪个。问题在于，即使

```
1     datum = message;
```

读取到`&a`，这一赋值仍然未同步于线程 1 的赋值：

```
1     message = &a;
```

因此，这里不存在连接两线程的 happens before 关系边：

```
1     a.x = 1;                              datum = message;
2        |                                      |
3        |   happens before                     |
4        v                                      v
5     message = &a;                         datum->x
```

因为两线程是未连接的，不能保证从`datum->x`读取到`1`；我们不知道对`a.x`的赋值是否先发生于对它的读。为了让赋值先发生于读，必须为写和读操作分别赋予 release 和 acquire 语义。

至此，我们获得了 store-release 和 lead-acquire 操作。store-release 操作 P 不仅写入数据到内存位置，还在 load-acquire 操作 Q 读取 P 写入的值时同步于 load-acquire 操作 Q。这是用 Linux 的`smp_store_release()`和`smp_load_acquire()`修复后的版本：

```
    thread 1                                thread 2
    --------------------------------        ------------------------
    a.x = 1;
    smp_store_release(&message, &a);        datum = smp_load_acquire
        (&message);
                                            if (datum != NULL)
                                              printk("%x\n", datum->x
                                                  );
```

这一修改使得，`datum`值为`&a`时，我们可以肯定写先发生于读。（为了简单起见，假设只有一个线程可以写`&a`到`message`。说”线程 2 读取到线程 1 写入的值“不是指写入到内存中的比特，而是指线程 1 的写是线程 2 最后一个可观测到的结果）。现在两线程关系看起来就像这样：

```
    a.x = 1;
        |
        v
    smp_store_release(&message, &a);  -----> datum = smp_load_acquire
        (&message);
                                                    |
                                                    v
                                            datum->x
```

一切正常。由于传递性，不论何时线程 2 读取到线程 1 写入的值，线程 1 在 store-release 前做的所有操作都在 load-acquire 后对线程 2 可见。上面的示例图只对线程 2 读取了线程 1 写入值时有效。

在 Linux 内核中，上述代码常书写为一种稍微不同的方式：

```
    thread 1                                thread 2
    --------------------------------        ------------------------
    a.x = 1;
    smp_wmb();
    WRITE_ONCE(message, &a);                datum = READ_ONCE(message);
                                            smp_rmb();
                                            if (datum != NULL)
                                              printk("%x\n", datum->x);
```

这个例子中，release 和 acquire 语义由内存屏障（memory barriers）`smp_wmb()`和`smp_rmb()`提供。内存屏障也具有 acquire 和 release 语义，但他们比简单的读写操作更难以推断。我们在讨论 seqlock 时会重新回顾它们。

不论使用 load-acquire/store-release 和`smp_rmb()`/`smp_wmb()`与否，这都是一个我们必须深刻理解的常见模式。我们发现了以下用例：

- 各种回环缓冲区（ring buffer）。回环缓冲区的条目经常指向其他数据；通常回环缓冲区还存在一个包含下标的头/尾位置。生产者一侧使用 store-release 操作，同步于消费者的 load-acquire

操作。

- RCU。对编译器而言，`rcu_dereferences()`和`rcu_assign_pointer()`系列 API 类似 load-acquire 和 store-release 操作。由于某些假设对除 Alpha 以外的所有处理器均成立，`rcu_derefence()`可以编译成普通的 load 操作；尽管如此，`rcu_assign_pointer()`仍同步于`rcu_dereference()`，就像是一个 load-acquire 操作一样。
- 发布指针到数组中。在这段（修改后的）KVM 代码中，如果`kvm_get_vcpu()`看到递增后的`kvm->online_vcpus`，那么数组中的对应条目一定是可靠的（译者注：指能读取到新写入的值）。

```
1     kvm_vm_ioctl_create_vcpu()                    kvm_get_vcpu()
2     ----------------------------------------
         ----------------------------------------------
3     kvm->vcpus[kvm->online_vcpus] = vcpu;         if (idx <
         smp_load_acquire(&kvm->online_vcpus))
4     smp_store_release(&kvm->online_vcpus,           return kvm->vcpus[
         idx];
5                       kvm->online_vcpus + 1);     return NULL;
```

除了 load-acquire/store-release 操作，还存在别的值得思考的消息传递模式：单一生产者算法。如果存在多个写者，必须使用其他手段（如互斥锁）避免它们互相影响。无锁算法不能凭空存在。他们只是并发编程工具箱的一部分，和其他更传统的工具搭配使用效果最佳。

这只是无锁算法系列的开始，下一部分将研究原子内存操作如何排序，并探究内存屏障如何成为 *seqcounts* 机制和 Linux 调度器的核心。