第九章：并发

要同时获得良好的的性能、并发的正确性和易于理解的代码是内核设计的一大挑战。直接使用锁是保证正确性的最佳途径，但不总是可行的。本章重点介绍了xv6不得不使用锁的例子，以及使用类似锁（但不是锁）的例子。

9.1 Locking patterns

缓存项通常是锁的一个挑战。例如，文件系统的块缓存(kernel/bio.c:26)存储了**NBUF**个磁盘块的副本。一个给定的磁盘块在缓存中最多只有一个副本，这一点非常重要；否则，不同的进程对同一磁盘块的不同副本进行修改时可能会发生冲突。每一个缓存的磁盘块都被存储在一个**buf**结构中(kernel/buf.h:1)。**buf**结构有一个锁字段，它确保每次只有一个进程使用一个给定的磁盘块。然而，这个锁是不够的：如果一个块根本不存在于缓存中，而两个进程想同时使用它怎么办？没有 **buf** (因为该块还没有被缓存)，因此没有能加锁的东西。Xv6对所有块的标识符关联一个额外的锁来处理这种情况。判断块是否被缓存的代码（e.g. bget(kernel/bio.c:59)），或改变缓存块集合的代码，必须持有**bcache.lock**。当代码找到它所需要的块和**buf**结构后，它就可以释放**bcache.lock**，然后锁定特定的块，这是一种通用模式：一组一把锁，外加每个项一把锁。

通常情况下，获取锁的函数会释放那把锁。但更准确的说法是，当一个序列需要保证原子性时，会在该序列开始时获取锁，而在序列结束时释放锁。如果序列的开始和结束在不同的函数中，或者不同的线程中，或者在不同的CPU上，那么锁的获取和释放也必须是一样的。锁的功能是强制其他的使用者等待，而不是将一段数据绑定给特定的代理。一个例子是yield中的**acquire**(kernel/proc.c:515)，它是在调度线程中释放的，而不是在获取锁的进程中释放的。另一个例子是**ilock**(kernel/fs.c:289)中的**acquiresleep**；这段代码经常在读取磁盘时睡眠；它可能在不同的CPU上被唤醒，这意味着锁可能在不同的CPU上获取和释放。

释放一个被锁保护的且锁内嵌在其中的对象是一件很棘手的事情，因为拥有锁并不足以保证释放对象的正确性。当有其他线程在**acquire**中等待时，问题就会出现；释放这个对象就意味着释放内嵌的锁，而释放这个锁会导致等待线程出错。一种方式是追踪该对象有多少个引用，只有在最后一个引用消失时才会释放对象。**pipeclose (kernel/pipe.c:59)**就是这样的例子。**pi->readopen**和**pi->writeopen**跟踪是否有文件描述符引用该管道。

9.2 Lock-like patterns

在许多地方，xv6使用引用计数或标志位作为一种软锁（soft lock），以表明一个对象已被分配，不应该被释放或重用。进程的**p->state**依此工作，**file**、**inode**和**buf**结构中的引用计数也是如此。虽然在每种情况下，锁都会保护标志位或引用计数，但正是标志位或引用计数防止了对象被过早释放。

文件系统使用结构体**inode**的引用计数作为一种共享锁，可以由多个进程持有，以避免代码使用普通锁时出现的死锁。例如，**namex**(kernel/fs.c:626)中的循环依次锁定路径上的每个目录。然而，**namex**必须在循环末尾释放每一个锁，因为如果它持有多个锁，那么如果路径名中包含.(即当前目录，例如，a/./b)，它可能会与自己发生死锁。它也可能因为涉及目录和..的并发查找而死锁。正如第8章所解释的那样，解决方案是让循环将目录的inode带入下一次迭代，并增加其引用计数，但不锁定。

有些数据项在不同的时候会受到不同机制的保护。它有时可能会被xv6代码的结构隐式保护，而不是通过显式的锁来防止并发访问。例如，当一个物理页是空闲的时候，它被**kmem.lock（kernel/kalloc.c:24）**保护。如果页面被分配作为管道(kernel/pipe.c:23)，它将被一个不同的锁(内嵌的**pi->lock**)保护。如果该页被重新分配给一个新进程的用户内存，它就不会受到锁的保护。相反，分配器不会将该页交给任何其他进程（直到它被释放）的事实保护了它不被并发访问。一个新进程的内存的所有权是很复杂的：首先父进程在**fork**中分配和操作它，然后子进程使用它，（在子进程退出后）父进程再次拥有内存，并将其传递给**kfree**。这里有两个需要注意的地方：第一，一个数据对象在其生命周期中的不同时刻可以用不同的方式来保护它不被并发访问；第二，保护的形式可能是隐式结构而不是显式锁。

最后一个类似于锁的例子是在调用**mycpu()**(kernel/proc.c:68)时需要禁用中断。禁用中断会导致调用代码对定时器中断是原子性的，而定时器中断可能会强制上下文切换，从而将进程移到不同的CPU上。

9.3 No locks at all

xv6有几个地方是在完全没有锁的情况下共享可变数据的。一个是在**spinlocks**的实现中，尽管你可以把RISC-V原子指令看作是依靠硬件实现的锁。另一个是**main.c** (kernel/main.c:7)中的**started**变量，用来防止其他CPU运行，直到CPU 0完成xv6的初始化；**volatile**确保编译器真正生成加载和存储指令。第三个例子是proc.c(kernel.proc.c:398)(kernel/proc.c:306)中的p->parent。它的一些用法会导致死锁，但是明显不会有其他进程能够同时修改p->parent。第四个例子是p->killed。它在持有p->lock时被设置，但在检查时却并不需要锁。

Xv6包含这样的情况：一个CPU或线程写一些数据，另一个CPU或线程读数据，但没有专门的锁来保护这些数据。例如，在**fork**中，父进程写入子进程的用户内存页，子进程(可能在不同的CPU上)读取这些页。这些页没有锁来显式地保护。严格来说，这不是锁的问题，因为子进程在父进程写完后才开始执行。这是一个潜在的内存操作的顺序问题（见第6章），因为没有内存屏障，没有理由期望一个CPU看到另一个CPU的写入。然而，由于父进程CPU释放锁，而子进程CPU在启动时获取锁，所以在**acquire**和**release**中的内存屏障保证了子进程CPU能看到父进程CPU的写入。

9.4 Parallelism

锁主要是为了正确性而抑制并行性。但是性能也很重要，所以内核设计者经常要考虑如何使用锁，来同时保证正确性和良好的并行性。虽然xv6并未对高性能进行系统地设计，但仍然值得考虑哪些xv6操作可以并行执行，哪些操作可能在锁上发生冲突。

xv6中的管道是一个并行性相当好的例子。每个管道都有自己的锁，因此不同的进程可以在不同的CPU上并行读写不同的管道。然而，对于一个给定的管道，writer和reader必须等待对方释放锁，他们不能同时读/写同一个管道。还有一种情况是，从一个空管道读（或向一个满管道写）必须阻塞，但这不是锁的方案导致的问题。

上下文切换是一个更复杂的例子。两个在各自CPU上执行的内核线程，可以同时调用**yield**、**sched**和**swtch**，并且这些调用能并行执行。这两个线程各自持有一个锁，但是不同的锁，所以它们不必等待对方。一旦进入**scheduler**，两个CPU在遍历进程表寻找一个RUNNABLE的进程的时候，却可能会发生锁冲突。也就是说，xv6在上下文切换的过程中，很可能会从多个CPU中获得性能上的好处，但可能没有那么多。

另一个例子是在不同的CPU上从不同的进程并发调用**fork**。这些调用可能需要互相等待**pid_lock**和**kmem.lock**，以及在进程表中搜索一个**UNUSED**进程所需的进程锁。另一方面，两个正在fork的进程可以完全并行地复制用户内存页和格式化页表页。

上述每个例子中的锁方案在某些情况下都牺牲了并行性能。在每一种情况下，都有可能通过更复杂的设计获得更多的并行性。这是否值得取决于细节：相关操作被调用的频率、代码在锁竞争的情况下所花费的时间、有多少CPU可能同时运行冲突的操作、是否代码的其他部分才是性能瓶颈。很难猜测一个给定的锁方案是否会导致性能问题，或者一个新的设计是否有明显的改进，所以往往需要在现实的工作负载上进行测量。

9.5 Exercises

1. 修改xv6管道的实现，允许对同一管道的读和写在不同内核上并行进行。
2. 修改xv6 **scheduler()**，以减少不同内核同时寻找可运行进程时的锁竞争。
3. 消除**fork**中一些串行执行的代码。