Chapter 9

Concurrency revisited

Simultaneously obtaining good parallel performance, correctness despite concurrency, and understandable code is a big challenge in kernel design. Straightforward use of locks is the best path to correctness, but is not always possible. This chapter highlights examples in which xv6 is forced to use locks in an involved way, and examples where xv6 uses lock-like techniques but not locks.

9.1 Locking patterns

Cached items are often a challenge to lock. For example, the filesystem's block cache (kernel/bio.c:26) stores copies of up to `NBUF` disk blocks. It's vital that a given disk block have at most one copy in the cache; otherwise, different processes might make conflicting changes to different copies of what ought to be the same block. Each cached block is stored in a `struct buf` (kernel/buf.h:1). A `struct buf` has a lock field which helps ensure that only one process uses a given disk block at a time. However, that lock is not enough: what if a block is not present in the cache at all, and two processes want to use it at the same time? There is no `struct buf` (since the block isn't yet cached), and thus there is nothing to lock. Xv6 deals with this situation by associating an additional lock (`bcache.lock`) with the set of identities of cached blocks. Code that needs to check if a block is cached (e.g., `bget` (kernel/bio.c:59)), or change the set of cached blocks, must hold `bcache.lock`; after that code has found the block and `struct buf` it needs, it can release `bcache.lock` and lock just the specific block. This is a common pattern: one lock for the set of items, plus one lock per item.

Ordinarily the same function that acquires a lock will release it. But a more precise way to view things is that a lock is acquired at the start of a sequence that must appear atomic, and released when that sequence ends. If the sequence starts and ends in different functions, or different threads, or on different CPUs, then the lock acquire and release must do the same. The function of the lock is to force other uses to wait, not to pin a piece of data to a particular agent. One example is the `acquire` in `yield` (kernel/proc.c:496), which is released in the scheduler thread rather than in the acquiring process. Another example is the `acquiresleep` in `ilock` (kernel/fs.c:289); this code often sleeps while reading the disk; it may wake up on a different CPU, which means the lock may be acquired and released on different CPUs.

Freeing an object that is protected by a lock embedded in the object is a delicate business, since owning the lock is not enough to guarantee that freeing would be correct. The problem case arises when some other thread is waiting in `acquire` to use the object; freeing the object implicitly frees the embedded lock, which will cause the waiting thread to malfunction. One solution is to track how many references to the object exist, so that it is only freed when the last reference disappears. See `pipeclose` (kernel/pipe.c:59) for an example; `pi->readopen` and `pi->writeopen` track whether the pipe has file descriptors referring to it.

9.2 Lock-like patterns

In many places xv6 uses a reference count or a flag in a lock-like way to indicate that an object is allocated and should not be freed or re-used. A process's `p->state` acts in this way, as do the reference counts in `file`, `inode`, and `buf` structures. While in each case a lock protects the flag or reference count, it is the latter that prevents the object from being prematurely freed.

The file system uses `struct inode` reference counts as a kind of shared lock that can be held by multiple processes, in order to avoid deadlocks that would occur if the code used ordinary locks. For example, the loop in `namex` (kernel/fs.c:629) locks the directory named by each pathname component in turn. However, `namex` must release each lock at the end of the loop, since if it held multiple locks it could deadlock with itself if the pathname included a dot (e.g., `a/./b`). It might also deadlock with a concurrent lookup involving the directory and `..`. As Chapter 8 explains, the solution is for the loop to carry the directory inode over to the next iteration with its reference count incremented, but not locked.

Some data items are protected by different mechanisms at different times, and may at times be protected from concurrent access implicitly by the structure of the xv6 code rather than by explicit locks. For example, when a physical page is free, it is protected by `kmem.lock` (kernel/kalloc.c:24). If the page is then allocated as a pipe (kernel/pipe.c:23), it is protected by a different lock (the embedded `pi->lock`). If the page is re-allocated for a new process's user memory, it is not protected by a lock at all. Instead, the fact that the allocator won't give that page to any other process (until it is freed) protects it from concurrent access. The ownership of a new process's memory is complex: first the parent allocates and manipulates it in `fork`, then the child uses it, and (after the child exits) the parent again owns the memory and passes it to `kfree`. There are two lessons here: a data object may be protected from concurrency in different ways at different points in its lifetime, and the protection may take the form of implicit structure rather than explicit locks.

A final lock-like example is the need to disable interrupts around calls to `mycpu()` (kernel/proc.c:80). Disabling interrupts causes the calling code to be atomic with respect to timer interrupts that could force a context switch, and thus move the process to a different CPU.

9.3 No locks at all

There are a few places where xv6 shares mutable data with no locks at all. One is in the implementation of spinlocks, although one could view the RISC-V atomic instructions as relying on locks im-

plemented in hardware. Another is the `started` variable in `main.c` (kernel/main.c:7), used to prevent other CPUs from running until CPU zero has finished initializing xv6; the `volatile` ensures that the compiler actually generates load and store instructions. A third example is `p->killed`, which is set while holding `p->lock` (kernel/proc.c:579), but checked without a holding lock (kernel/trap.c:56).

Xv6 contains cases in which one CPU or thread writes some data, and another CPU or thread reads the data, but there is no specific lock dedicated to protecting that data. For example, in `fork`, the parent writes the child's user memory pages, and the child (a different thread, perhaps on a different CPU) reads those pages; no lock explicitly protects those pages. This is not strictly a locking problem, since the child doesn't start executing until after the parent has finished writing. It is a potential memory ordering problem (see Chapter 6), since without a memory barrier there's no reason to expect one CPU to see another CPU's writes. However, since the parent releases locks, and the child acquires locks as it starts up, the memory barriers in `acquire` and `release` ensure that the child's CPU sees the parent's writes.

9.4 Parallelism

Locking is primarily about suppressing parallelism in the interests of correctness. Because performance is also important, kernel designers often have to think about how to use locks in a way that both achieves correctness and allows parallelism. While xv6 is not systematically designed for high performance, it's still worth considering which xv6 operations can execute in parallel, and which might conflict on locks.

Pipes in xv6 are an example of fairly good parallelism. Each pipe has its own lock, so that different processes can read and write different pipes in parallel on different CPUs. For a given pipe, however, the writer and reader must wait for each other to release the lock; they can't read/write the same pipe at the same time. It is also the case that a read from an empty pipe (or a write to a full pipe) must block, but this is not due to the locking scheme.

Context switching is a more complex example. Two kernel threads, each executing on its own CPU, can call `yield`, `sched`, and `swtch` at the same time, and the calls will execute in parallel. The threads each hold a lock, but they are different locks, so they don't have to wait for each other. Once in `scheduler`, however, the two CPUs may conflict on locks while searching the table of processes for one that is `RUNNABLE`. That is, xv6 is likely to get a performance benefit from multiple CPUs during context switch, but perhaps not as much as it could.

Another example is concurrent calls to `fork` from different processes on different CPUs. The calls may have to wait for each other for `pid_lock` and `kmem.lock`, and for per-process locks needed to search the process table for an `UNUSED` process. On the other hand, the two forking processes can copy user memory pages and format page-table pages fully in parallel.

The locking scheme in each of the above examples sacrifices parallel performance in certain cases. In each case it's possible to obtain more parallelism using a more elaborate design. Whether it's worthwhile depends on details: how often the relevant operations are invoked, how long the code spends with a contended lock held, how many CPUs might be running conflicting operations at the same time, whether other parts of the code are more restrictive bottlenecks. It can be difficult

to guess whether a given locking scheme might cause performance problems, or whether a new design is significantly better, so measurement on realistic workloads is often required.

9.5 Exercises

1. Modify xv6's pipe implementation to allow a read and a write to the same pipe to proceed in parallel on different cores.

2. Modify xv6's `scheduler()` to reduce lock contention when different cores are looking for runnable processes at the same time.

3. Eliminate some of the serialization in xv6's `fork()`.