implemented 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 are some uses of `p->parent` in `proc.c` (kernel/proc.c:398) (kernel/proc.c:306) where proper locking could deadlock, but it seems clear that no other process could be simultaneously modifying `p->parent`. A fourth example is `p->killed`, which is set while holding `p->lock` (kernel/proc.c:611), 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 achieves both correctness and good 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()`.

Chapter 10

Summary

This text introduced the main ideas in operating systems by studying one operating system, xv6, line by line. Some code lines embody the essence of the main ideas (e.g., context switching, user/kernel boundary, locks, etc.) and each line is important; other code lines provide an illustration of how to implement a particular operating system idea and could easily be done in different ways (e.g., a better algorithm for scheduling, better on-disk data structures to represent files, better logging to allow for concurrent transactions, etc.). All the ideas were illustrated in the context of one particular, very successful system call interface, the Unix interface, but those ideas carry over to the design of other operating systems.

Bibliography

[1] The RISC-V instruction set manual: privileged architecture. `https://riscv.org/specifications/privileged-isa/`, 2019.

[2] The RISC-V instruction set manual: user-level ISA. `https://riscv.org/specifications/isa-spec-pdf/`, 2019.

[3] Hans-J Boehm. Threads cannot be implemented as a library. *ACM PLDI Conference*, 2005.

[4] Edsger Dijkstra. Cooperating sequential processes. `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html`, 1965.

[5] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. 2012.

[6] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[7] Donald Knuth. *Fundamental Algorithms. The Art of Computer Programming. (Second ed.)*, volume 1. 1997.

[8] L Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 1974.

[9] John Lions. *Commentary on UNIX 6th Edition*. Peer to Peer Communications, 2000.

[10] Paul E. Mckenney, Silas Boyd-wickizer, and Jonathan Walpole. RCU usage in the linux kernel: One decade later, 2013.

[11] Martin Michael and Daniel Durich. The NS16550A: UART design and application considerations. `http://bitsavers.trailing-edge.com/components/national/_appNotes/AN-0491.pdf`, 1987.

[12] David Patterson and Andrew Waterman. *The RISC-V Reader: an open architecture Atlas*. Strawberry Canyon, 2017.

[13] Dave Presotto, Rob Pike, Ken Thompson, and Howard Trickey. Plan 9, a distributed system. In *In Proceedings of the Spring 1991 EurOpen Conference*, pages 43–50, 1991.

[14] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

Index