Chapter 4

Locking

Xv6 runs on multiprocessors, computers with multiple CPUs executing code independently. These multiple CPUs operate on a single physical address space and share data structures; xv6 must introduce a coordination mechanism to keep them from interfering with each other. Even on a uniprocessor, xv6 must use some mechanism to keep interrupt handlers from interfering with non-interrupt code. Xv6 uses the same low-level concept for both: locks. Locks provide mutual exclusion, ensuring that only one CPU at a time can hold a lock. If xv6 only accesses a data structure while holding a particular lock, then xv6 can be sure that only one CPU at a time is accessing the data structure. In this situation, we say that the lock protects the data structure.

As an example, consider several processors sharing a single disk, such as the IDE disk in xv6. The disk driver maintains a linked list of the outstanding disk requests (3720) and processors may add new requests to the list concurrently (3854). If there were no concurrent requests, you might implement the linked list as follows:

```
 1    struct list {
 2      int data;
 3      struct list *next;
 4    };
 5
 6    struct list *list = 0;
 7
 8    void
 9    insert(int data)
10    {
11      struct list *l;
12
13      l = malloc(sizeof *l);
14      l->data = data;
15      l->next = list;
16      list = l;
17    }
```

Proving this implementation correct is a typical exercise in a data structures and algorithms class. Even though this implementation can be proved correct, it isn't, at least not on a multiprocessor. If two different CPUs execute `insert` at the same time, it could happen that both execute line 15 before either executes 16. If this happens, there will now be two list nodes with `next` set to the former value of `list`. When the two assignments to `list` happen at line 16, the second one will overwrite the first; the node involved in the first assignment will be lost. This kind of problem is called a race condition. The problem with races is that they depend on the exact timing of the two CPUs involved and are consequently difficult to reproduce. For example, adding print statements while debugging `insert` might change the timing of the execution enough to make the race disappear.

The typical way to avoid races is to use a lock. Locks ensure mutual exclusion, so that only one CPU can execute `insert` at a time; this makes the scenario above impossible. The correctly locked version of the above code adds just a few lines (not numbered):

```
 6    struct list *list = 0;
      struct lock listlock;
 7
 8    void
 9    insert(int data)
10    {
11      struct list *l;
12
        acquire(&listlock);
13      l = malloc(sizeof *l);
14      l->data = data;
15      l->next = list;
16      list = l;
        release(&listlock);
17    }
```

When we say that a lock protects data, we really mean that the lock protects some collection of invariants that apply to the data. Invariants are properties of data structures that are maintained across operations. Typically, an operation's correct behavior depends on the invariants being true when the operation begins. The operation may temporarily violate the invariants but must reestablish them before finishing. For example, in the linked list case, the invariant is that `list` points at the first node in the list and that each node's `next` field points at the next node. The implementation of `insert` vioilates this invariant temporarily: line X creates a new list element `l` with the intent that `l` be the first node in the list, but `l`'s next pointer does not point at the next node in the list yet (reestablished at line 15) and `list` does not point at `l` yet (reestablished at line 16). The race condition we examined above happened because a second CPU executed code that depended on the list invariants while they were (temporarily) violated. Proper use of a lock ensures that only one CPU at a time can operate on the data structure, so that no CPU will execute a data structure operation when the data structure's invariants do not hold.

Code: Locks

Xv6's represents a lock as a `struct spinlock` (1451). The critical field in the structure is `locked`, a word that is zero when the lock is available and non-zero when it is held. Logically, xv6 should acquire a lock by executing code like

```
void
acquire(struct spinlock *lk)
{
  for(;;) {
    if(!lk->locked) {
```

```
26          lk->locked = 1;
27          break;
28        }
29      }
30    }
```

Unfortunately, this implementation does not guarantee mutual exclusion on a modern multiprocessor. It could happen that two (or more) CPUs simultaneously reach line 25, see that `lk->locked` is zero, and then both grab the lock by executing lines 26 and 27. At this point, two different CPUs hold the lock, which violates the mutual exclusion property. Rather than helping us avoid race conditions, this implementation of `acquire` has its own race condition. The problem here is that lines 25 and 26 executed as separate actions. In order for the routine above to be correct, lines 25 and 26 must execute in one atomic step.

To execute those two lines atomically, xv6 relies on a special 386 hardware instruction, `xchg` (0529). In one atomic operation, `xchg` swaps a word in memory with the contents of a register. `Acquire` (1523) repeats this `xchg` instruction in a loop; each iteration reads `lk->locked` and atomically sets it to 1 (1532). If the lock is held, `lk->locked` will already be 1, so the `xchg` returns 1 and the loop continues. If the `xchg` returns 0, however, `acquire` has successfully acquired the lock—`locked` was 0 and is now 1—so the loop can stop. Once the lock is acquired, `acquire` records, for debugging, the CPU and stack trace that acquired the lock. When a process acquires a lock and forget to release it, this information can help to identify the culprit. These debugging fields are protected by the lock and must only be edited while holding the lock.

`Release` (1552) is the opposite of `acquire`: it clears the debugging fields and then releases the lock.

Modularity and recursive locks

System design strives for clean, modular abstractions: it is best when a caller does not need to know how a callee implements particular functionality. Locks interfere with this modularity. For example, if a CPU holds a particular lock, it cannot call any function f that will try to reacquire that lock: since the caller can't release the lock until f returns, if f tries to acquire the same lock, it will spin forever, or deadlock.

There are no transparent solutions that allow the caller and callee to hide which locks they use. One common, transparent, but unsatisfactory solution is "recursive locks," which allow a callee to reacquire a lock already held by its caller. The problem with this solution is that recursive locks can't be used to protect invariants. After `insert` called `acquire(&listlock)` above, it can assume that no other function holds the lock, that no other function is in the middle of a list operation, and most importantly that all the list invariants hold. In a system with recursive locks, `insert` can assume nothing after it calls `acquire`: perhaps `acquire` succeeded only because one of `insert`'s caller already held the lock and was in the middle of editing the list data structure. Maybe the invariants hold or maybe they don't. The list no longer protects them. Locks are just as important for protecting callers and callees from each other as they are for protecting different CPUs from each other; recursive locks give up that property.

Since there is no ideal transparent solution, we must consider locks part of the function's specification. The programmer must arrange that function doesn't invoke a function f while holding a lock that f needs. Locks force themselves into our abstractions.

Code: Using locks

A hard part about using locks is deciding how many locks to use and which data and invariants each lock protects. There are a few basic principles. First, any time a variable can be written by one CPU at the same time that another CPU can read or write it, a lock should be introduced to keep the two operations from overlapping. Second, remember that locks protect invariants: if an invariant involves multiple data structures, typically all of the structures need to be protected by a single lock to ensure the invariant is maintained.

The rules above say when locks are necessary but say nothing about when locks are unnecessary, and it is important for efficiency not to lock too much. If efficiency wasn't important, then one could use a uniprocessor computer and no worry at all about locks. For protecting kernel data structures, it would suffice to create a single lock that must be acquired on entering the kernel and released on exiting the kernel. Many uniprocessor operating systems have been converted to run on multiprocessors using this approach, sometimes called a "giant kernel lock," but the approach sacrifices true concurrency: only one CPU can execute in the kernel at a time. If the kernel does any heavy computation, it would be more efficient to use a larger set of more fine-grained locks, so that the kernel could execute on multiple CPUs simultaneously.

Ultimately, the choice of lock granularity is an exercise in parallel programming. Xv6 uses a few coarse data-structure specific locks; for example, xv6 uses a single lock protecting the process table and its invariants, which are described in Chapter 5. A more fine-grained approach would be to have a lock per entry in the process table so that threads working on different entries in the process table can proceed in parallel. However, it complicates operations that have invariants over the whole process table, since they might have to take out several locks. Hopefully, the examples of xv6 will help convey how to use locks.

Lock ordering

If a code path through the kernel must take out several locks, it is important that all code paths acquire the locks in the same order. If they don't, there is a risk of deadlock. Let's say two code paths in xv6 needs locks A and B, but code path 1 acquires locks in the order A and B, and the other code acquires them in the order B and A. This situation can result in a deadlock, because code path 1 might acquire lock A and before it acquires lock B, code path 2 might acquire lock B. Now neither code path can proceed, because code path 1 needs lock B, which code path 2 holds, and code path 2 needs lock A, which code path 1 holds. To avoid such deadlocks, all code paths must acquire locks in the same order. Deadlock avoidance is another example illustrating why locks must be part of a function's specification: the caller must invoke functions in a consistent order so that the functions acquire locks in the same order.

Because xv6 uses coarse-grained locks and xv6 is simple, xv6 has few lock-order chains. The longest chain is only two deep. For example, `ideintr` holds the ide lock while calling `wakeup`, which acquires the ptable lock. There are a number of other examples involving sleep and wakeup . These orderings come about because sleep and wakeup have a complicated invariant, as discussed in Chapter 5. In the file system there are a number of examples of chains of two because the file system must, for example, acquire a lock on a directory and the lock on a file in that directory to unlink a file from its parent directory correctly. xv6 always acquires the locks in the order first parent directory and then the file.

Interrupt handlers

Xv6 uses locks to protect interrupt handlers running on one CPU from non-interrupt code accessing the same data on another CPU. For example, the timer interrupt handler (3014) increments `ticks` but another CPU might be in `sys_sleep` at the same time, using the variable (3373). The lock `tickslock` synchronizes access by the two CPUs to the single variable.

Locks are useful not just for synchronizing multiple CPUs but also for synchronizing interrupt and non-interrupt code on the *same* CPU. The `ticks` variable is used by the interrupt handler and also by the non-interrupt function `sys_sleep`, as we just saw. If the non-interrupt code is manipulating a shared data structure, it may not be safe for the CPU to interrupt that code and start running an interrupt handler that will use the data structure. Xv6's disables interrupts on a CPU when that CPU holds a lock; this ensures proper data access and also avoids deadlocks: an interrupt handler can never acquire a lock already held by the code it interrupted. One way to think about this is that locks provide atomicity between code running on different processors and turning off interrupts provides atomicity between code running on the same processor.

Before attempting to acquire a lock, `acquire` calls `pushcli` (1525) to disable interrupts. `Release` calls `popcli` (1571) to allow them to be enabled. (The underlying x86 instruction to disable interrupts is named `cli`.) `Pushcli` (1605) and `popcli` (1616) are more than just wrappers around `cli` and `sti`: they are counted, so that it takes two calls to `popcli` to undo two calls to `pushcli`; this way, if code acquires two different locks, interrupts will not be reenabled until both locks have been released.

The interaction between interrupt handlers and non-interrupt code provides a nice example why recursive locks are problematic. If xv6 used recursive locks (a second acquire on a CPU is allowed if the first acquire happened on that CPU too), then interrupt handlers could run while non-interrupt code is in a critical section. This could create havoc, since when the interrupt handler runs, invariants that the handler relies on might be temporarily violated. For example, `ideintr` (3802) assumes that the linked list with outstanding requests is well-formed. If xv6 would have used recursive locks, then `ideintr` might run while `iderw` is in the middle of manipulating the linked list, and the linked list will end up in an incorrect state.

Memory ordering

This chapter has assumed that processors start and complete instructions in the order in which they appear in the program. Many processors, however, execute instructions out of order to achieve higher performance. If an instruction takes many cycles to complete, a processor may want to issue the instruction early so that it can overlap with other instructions and avoid processor stalls. For example, a processor may notice that in a serial sequence of instruction A and B are not dependent on each other and start instruction B before A so that it will be completed when the processor completes A. Concurrency, however, may expose this reordering to software, which lead to incorrect behavior.

For example, one might wonder what happens if `release` just assigned 0 to `lk->locked`, instead of using `xchg`. The answer to this question is unclear, because different generations of x86 processors make different guarantees about memory ordering. If `lk->locked=0`, were allowed to be re-ordered say after `popcli`, than `acquire` might break, because to another thread interrupts would be enabled before a lock is released. To avoid relying on unclear processor specifications about memory ordering, xv6 takes no risk and uses `xchg`, which processors must guarantee not to reorder.

Real world

locking is hard and not well understood.

approaches to synchronization still an active topic of research.

best to use locks as the base for higher-level constructs like synchronized queues, although xv6 does not do this.

user space locks too; xv6 doesn't let processes share memory so no need.

semaphores.

no need for atomicity really; lamport's algorithm.

lock-free algorithms.

Exercises

1. get rid off the xchg in acquire. explain what happens when you run xv6?

2. move the acquire in iderw to before sleep. is there a race? why don't you observe it when booting xv6 and run slamfs? increase critical section with a dummy loop; what do you see now? explain.

3. do posted homework question.