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 the implementation of a simple linked list:

```
 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` (1301). 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) {
      lk->locked = 1;
      break;
    }
```

```
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` (0501). In one atomic operation, `xchg` swaps a word in memory with the contents of a register. `Acquire` (1373) repeats this `xchg` instruction in a loop; each iteration reads `lk->locked` and atomically sets it to 1 (1382). 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` (1402) 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

The hardest 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, remeber 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. 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 more art than science. Xv6 uses a few coarse data-structure specific locks. Hopefully, the examples of xv6 will help convey a feeling for some of the art.