Raft Locking Advice

If you are wondering how to use locks in the 6.824 Raft labs, here are some rules and ways of thinking that might be helpful.

Rule 1: Whenever you have data that more than one goroutine uses, and at least one goroutine might modify the data, the goroutines should use locks to prevent simultaneous use of the data. The Go race detector is pretty good at detecting violations of this rule (though it won't help with any of the rules below).

Rule 2: Whenever code makes a sequence of modifications to shared data, and other goroutines might malfunction if they looked at the data midway through the sequence, you should use a lock around the whole sequence.

An example:

```
  rf.mu.Lock()
  rf.currentTerm += 1
  rf.state = Candidate
  rf.mu.Unlock()
```

It would be a mistake for another goroutine to see either of these updates alone (i.e. the old state with the new term, or the new term with the old state). So we need to hold the lock continuously over the whole sequence of updates. All other code that uses rf.currentTerm or rf.state must also hold the lock, in order to ensure exclusive access for all uses.

The code between Lock() and Unlock() is often called a "critical section." The locking rules a programmer chooses (e.g. "a goroutine must hold rf.mu when using rf.currentTerm or rf.state") are often called a "locking protocol".

Rule 3: Whenever code does a sequence of reads of shared data (or reads and writes), and would malfunction if another goroutine modified the data midway through the sequence, you should use a lock around the whole sequence.

An example that could occur in a Raft RPC handler:

```
  rf.mu.Lock()
  if args.Term > rf.currentTerm {
   rf.currentTerm = args.Term
  }
  rf.mu.Unlock()
```

This code needs to hold the lock continuously for the whole sequence. Raft requires that currentTerm only increases, and never decreases. Another RPC handler could be executing in a separate goroutine; if it

Another RPC handler could be executing in a separate goroutine; if it were allowed to modify rf.currentTerm between the if statement and the update to rf.currentTerm, this code might end up decreasing rf.currentTerm. Hence the lock must be held continuously over the whole sequence. In addition, every other use of currentTerm must hold the lock, to ensure that no other goroutine modifies currentTerm during our critical section.

Real Raft code would need to use longer critical sections than these examples; for example, a Raft RPC handler should probably hold the lock for the entire handler.

Rule 4: It's usually a bad idea to hold a lock while doing anything that might wait: reading a Go channel, sending on a channel, waiting for a timer, calling time.Sleep(), or sending an RPC (and waiting for the reply). One reason is that you probably want other goroutines to make progress during the wait. Another reason is deadlock avoidance. Imagine two peers sending each other RPCs while holding locks; both RPC handlers need the receiving peer's lock; neither RPC handler can ever complete because it needs the lock held by the waiting RPC call.

Code that waits should first release locks. If that's not convenient, sometimes it's useful to create a separate goroutine to do the wait.

Rule 5: Be careful about assumptions across a drop and re-acquire of a lock. One place this can arise is when avoiding waiting with locks held. For example, this code to send vote RPCs is incorrect:

```
  rf.mu.Lock()
  rf.currentTerm += 1
  rf.state = Candidate
  for <each peer> {
    go func() {
      rf.mu.Lock()
      args.Term = rf.currentTerm
      rf.mu.Unlock()
      Call("Raft.RequestVote", &args, ...)
      // handle the reply...
    } ()
  }
  rf.mu.Unlock()
```

The code sends each RPC in a separate goroutine. It's incorrect because args.Term may not be the same as the rf.currentTerm at which the surrounding code decided to become a Candidate. Lots of time may pass between when the surrounding code creates the goroutine and when the goroutine reads rf.currentTerm; for example, multiple terms may come and go, and the peer may no longer be a candidate. One way to fix this is for the created goroutine to use a copy of rf.currentTerm made while the outer code holds the lock. Similarly, reply-handling code after the Call() must re-check all relevant assumptions after

after the call(), must re-check all relevant assumptions after
re-acquiring the lock; for example, it should check that rf.currentTerm hasn't changed since the decision to become a candidate.

It can be difficult to interpret and apply these rules. Perhaps most puzzling is the notion in Rules 2 and 3 of code sequences that shouldn't be interleaved with other goroutines' reads or writes. How can one recognize such sequences? How should one decide where a sequence ought to start and end?

One approach is to start with code that has no locks, and think carefully about where one needs to add locks to attain correctness. This approach can be difficult since it requires reasoning about the correctness of concurrent code.

A more pragmatic approach starts with the observation that if there were no concurrency (no simultaneously executing goroutines), you would not need locks at all. But you have concurrency forced on you when the RPC system creates goroutines to execute RPC handlers, and because you need to send RPCs in separate goroutines to avoid waiting.
You can effectively eliminate this concurrency by identifying all places where goroutines start (RPC handlers, background goroutines you create in Make(), &c), acquiring the lock at the very start of each goroutine, and only releasing the lock when that goroutine has completely finished and returns. This locking protocol ensures that nothing significant ever executes in parallel; the locks ensure that each goroutine executes to completion before any other goroutine is allowed to start. With no parallel execution, it's hard to violate Rules 1, 2, 3, or 5. If each goroutine's code is correct in isolation (when executed alone, with no concurrent goroutines), it's likely to still be correct when you use locks to suppress concurrency. So you can avoid explicit reasoning about correctness, or explicitly identifying critical sections.

However, Rule 4 is likely to be a problem. So the next step is to find places where the code waits, and to add lock releases and re-acquires (and/or goroutine creation) as needed, being careful to re-establish assumptions after each re-acquire. You may find this process easier to get right than directly identifying sequences that must be locked for correctness.

(As an aside, what this approach sacrifices is any opportunity for better performance via parallel execution on multiple cores: your code is likely to hold locks when it doesn't need to, and may thus unnecessarily prohibit parallel execution of goroutines. On the other hand, there is not much opportunity for CPU parallelism within a single Raft peer.)