Lecture 8: Reader/Writer Locks

Goal: walk through an example synchronization problem, found in many operating system kernels, that can illustrate the various aspects of locks and condition variables.

Illustrate method for writing correct synchronization code. And if you have some code, how can you tell if it works? (Stare at it for a few hours?)

Problem statement:

Shared data, accessed by multiple threads. Very common in databases (e.g., at Amazon, many more queries about books than purchases of books, so data for how many books are left could be protected by a reader /writer lock). But also found in operating systems: linux is converting to use RCU locks in the kernel, which are a kind of reader/writer lock.

Two classes of threads:
- Readers -- never modify shared data
- Writers -- read and modify shared data

Using a single lock on the data would be overly restrictive. Want:
- many readers at same time
- only one writer at same time

Constraints:

0. At most one writer can access data at same time – safety

1. Readers can access data when no writers (Condition okToRead) – progress

2. Writers can access data when no readers or writers (Condition okToWrite) – progress

3. Only one thread manipulates state variables at a time. – safety

Basic structure of solution

Reader
  wait until no writers
  access database
  check out -- wake up waiting writer

Writer
  wait until no readers or writers
  access database
  check out -- wake up waiting readers or writer

State variables:
  # of active readers -- AR = 0
  # of active writers -- AW = 0
  # of waiting readers -- WR = 0
  # of waiting writers -- WW = 0

  Condition okToRead = NIL
  Condition okToWrite = NIL
  Lock lock = FREE

Recall: **Condition variable**: a queue of threads waiting for something **inside** a critical section

Condition variables support three operations:

Wait() -- release lock, go to sleep, re-acquire lock
  Releasing lock and going to sleep is atomic

Signal() -- wake up a waiter, if any

Broadcast() -- wake up all waiters

// implements writer priority, but not writer FIFO
// readers wait if there are any writers waiting
Code:
  Reader() {

```
  // first check self into system
  lock.Acquire();
  while ((AW + WW) > 0) {   // check if safe to read
                                          // if any writers, wait
      WR++;
      okToRead.Wait(&lock);
      WR--;
   }
  AR++;
  lock.Release();

  Access DB

  // check self  out of system
  lock.Acquire();
  AR--;
  if (AR == 0 && WW > 0)//if no other readers still
                                          // active, wake up writer
      okToWrite.Signal(&lock);
  lock.Release();
}

Writer() {   //  symmetrical
  // check in
  lock.Acquire();
  while ((AW + AR) > 0) { // check if safe to write
                                  // if any readers or writers, wait
      WW++;
      okToWrite.Wait(&lock);
      WW--;
   }
  AW++;
  lock.Release();

  Access DB
```

```
    // check out
    lock.Acquire();
    AW--;
    if (WW > 0)   // give priority to other writers
        okToWrite.Signal(&lock);
    else if (WR > 0)
        okToRead.Broadcast(&lock);
    lock.Release();
}
```

Illustrate: show state of ready list and each lock and CV as we execute.

One reader enters and leaves. Back to the beginning.
One writer enters. One reader blocks. Writer continues, wakes up reader. Third thread slips in and starts the write. Reader wakes up and goes back to sleep.

Questions:

1. Does the code work for all cases? How would you know? Well, you could try to prove the constraints from the code. Or you could try model checking – run every possible interleavings. How many interleavings are there? Infinite?

2. Say there are a fixed number of readers and writers? Then not infinite! Because there are no race conditions, we can prove by exhaustive search – we don't need to worry about any time slice while the lock is held.

3. Why does checkRead need a while?

4. Can readers starve? Yes, the example shows that a writer can slip in after the reader wakes up the writer.
5. Can writers starve? This one is tricky! A reader can't slip in: we check to see if there's a WW, and if so, the new reader will wait. But what about a new writer? The new writer can slip in ahead of a waiting writer.

5.Do we need to keep track of WW and WR? Start drawing lines through the code

---

6. How might we modify the solution to ensure that neither readers nor writers will starve? Simple solution would be to use a regular lock! So we also want to allow multiple readers when it is possible to do so.

First, let's ask: can we ensure that a writer never waits for a later arriving writer. What if we add WW into the writer wait loop? That won't work! The waking writer will see that the WW is non-zero, and so go back to sleep.

```
Writer() {   //  symmetrical
  // check in
  lock.Acquire();
  while ((AW + AR + WW) > 0) {   // doesn't work!
      WW++;
      okToWrite.Wait(&lock);
      WW--;
   }
  AW++;
  lock.Release();
```

OK, so what if we kept a counter of all writers?

```
Writer() {   //  symmetrical
  // check in
  lock.Acquire();
  myPos = numWriters++;
  while ((AW + AR > 0 && myPos > nextToGo) {
      WW++;
      okToWrite.Wait(&lock);
      WW--;
   }
  AW++;
  lock.Release();

  Access DB

  // check out
  lock.Acquire();
  AW--;
```

```
nextToGo++;
 if (WW > 0) {
     okToWrite.Signal(&lock);
 } else if (WR > 0)
     okToRead.Broadcast(&lock);
 lock.Release();
}
```

Still doesn't work! What if there's a spurious wakeup? Then the waiter will wake up, put themselves back to sleep, and might end up at the back of the line. The signal wakes up a waiter – not necessarily the first one to have ever waited.

Could fix this by using broadcast, but that's a pain. Instead: fix it by creating a CV per waiter.

```
Writer() {   //  symmetrical
 // check in
 lock.Acquire();
 myPos = numWriters++;
 myCV = new CV;
 writeHeap.Insert(myPos, myCV); // sorted by myPos
 while ((AW + AR > 0 && myPos > nextToGo) {
     WW++;
     myCV.Wait(&lock);
     WW--;
  }
 AW++;
 delete myCV;
 lock.Release();

 Access DB

 // check out
 lock.Acquire();
 AW--;
nextToGo++;
```

```
    if (WW > 0) {
        (pos, cv) = writeHeap.RemoveFront();
        cv.Signal(&lock);
    } else if (WR > 0)
        okToRead.Broadcast(&lock);
    lock.Release();
}
```

How would you ensure neither readers nor writers starved? Would need a CV for every writer and every group of readers that arrived between two writers.

Not enough time for this:

Digression: serializability. Suppose you have a file system, and you have multiple processes accessing the file system. If each is touching different parts of the file system, no problem, just lock the different parts as you read or write those data structures. But what if they are accessing the same part of the file system?

E.g., what if one thread does: move subdirA/file subdirB While the other thread is doing a virus scan? You'd want the scan to work – to find the item in the old or the new place, not in both or neither.

Serializability: operations appear to occur in some order

One way to do that would be one giant lock on the whole file system, but yuck – then only one thread can use the file system at the same time, even when they are operating on different parts. When is it safe?

File system is implemented as a set of directories and files. A lock per directory/file?

Looks like it would work: lock both directories, release when done, so virus scanner will not see an intermediate state.

But what if we become more complex, as in:
One thread does move "/user/tom/subdirA/file" to "/user/sunjayc/subdirB/"

Another thread does move “/user/tom/subdirA” to “/user/sunjayc/subdirB/”

To prevent this, you’d need to lock /user/tom – but the first thread didn’t check the lock on user/tom. So grab every lock on every directory down from root? No concurrency!

Sufficient to lock the top level directories for read, and the ones I’m modifying for write.

You can get more sophisticated: which types of operations can be done at the same time without violating serializability. E.g., I can lock a particular entry in the directory while others can be updated.

End digression.