SE 350: Operating Systems Winter 2023

Lecture 16 — The Readers-Writers Problem

Jeff Zarnett *2022-11-08*

The Readers-Writers Problem

This problem is about concurrent reading and modification of a data structure or record by more than one thread. A writer will modify the data; a reader will read it only without modification. Unlike the producer-consumer problem, some concurrency is allowed:

1. Any number of readers may be in the critical section simultaneously.
2. Only one writer may be in the critical section (and when it is, no readers are allowed).

Or, to sum that up, a writer cannot enter the critical section while any other thread (whether reader or writer) is there. While a writer is in the critical section, neither readers nor writers may enter the critical section [Dow08]. This is very often how file systems work: a file may be read concurrently by any number of threads, but only one thread may write to it at a time (and to prevent reading of inconsistent data, no thread may read during the write).

This is similar to, but distinct from, the general mutual exclusion problem and the producer-consumer problem. In the readers-writers problem, readers do not modify the data (consumers do take things out of the buffer, modifying it). If any thread could read or write the shared data structure, we would have to use the general mutual exclusion solution. Although the general mutual exclusion routine would work in that it would prevent errors, it is a serious performance reduction versus allowing multiple readers concurrently [Sta14]. Thus, this situation is worth examining in its own right.

Let us keep track of the number of readers at any given time with a variable `readers`. We will need a way of protecting this variable from concurrent modifications, so there will have to be a binary semaphore `mutex`. We will also need one further semaphore, `roomEmpty`, as a way of indicating that the room is empty. A writer has to wait for the room to be empty (i.e., `wait` on the `roomEmpty` semaphore) before it can enter. The solution comes from [Dow08]:

Writer

```
1. wait( roomEmpty )
2. [write data]
3. post( roomEmpty )
```

Reader

```
 1. wait( mutex )
 2. readers++
 3. if readers == 1
 4.     wait( roomEmpty )
 5. end if
 6. post( mutex )
 7. [read data]
 8. wait( mutex )
 9. readers--
10. if readers == 0
11.     post( roomEmpty )
12. end if
13. post( mutex )
```

The code for the writer is much simpler than that of the readers. The writer may only enter into the critical section if the room is empty. When it has finished, it indicates that the room is empty. The writer can be certain that when it exits the critical section that there are no other threads in the room, because no thread may enter the room while the writer was there.

The reader code is somewhat more complicated. The first reader that arrives encounters the situation that the room is empty, so it "locks" the room (waiting on the `roomEmpty` semaphore), and that will prevent writers from entering the room. Additional readers do not check if the room is empty; they just proceed to enter. When the last reader leaves the room, it signals that the room is empty ("unlocking it" to allow a writer in). This pattern is sometimes called the *light switch*, as in [HZMG15]: the first one into the room turns on the lights and the last one out turns them off again.

The reader code has that situation that makes us concerned about the possibility of deadlock: a wait on `roomEmpty` inside a critical section controlled by `mutex`. With a bit of reasoning, we can convince ourselves that there is no risk: the only situation in which a thread waits on `roomEmpty` is that a writer is currently in its critical section. No other reader thread can get the `mutex` lock. As long as the write operation takes finite time, eventually the writer will post the `roomEmpty` semaphore and the threads can continue. Deadlock is not a problem.

There is, however, a second problem that we need to be concerned about. Suppose some readers are in the room, and a writer arrives. The writer must wait until all the readers have left the room. When each of the readers is finished, it exits the room. In the meantime, more readers arrive and enter the room. So even though each reader is in the room for only a finite amount of time, there is never a moment when the room has no readers in it. This undesirable situation is not deadlock, because the reader threads are not stuck, but the writer (and any subsequent writers) is (are) going to wait forever. This is a situation called *starvation*: a thread never gets a chance to run.

Recall criterion 3 of the list of properties we want in any mutual exclusion solution: it must not be possible for a thread requiring access to the critical section to be delayed indefinitely. This problem is just as bad as deadlock in that if it is discovered, it eliminates a proposed solution as an acceptable option, even though starvation might only be an unlikely event. We must therefore improve on this solution such that there is no longer the possibility that a writer starves.

Conceptually, the solution that accomplishes the goal looks something like this: when a writer arrives, any readers currently reading should be permitted to finish their read, but no new readers should be allowed to start reading. Thus, eventually, all the readers currently in the critical section will finish, the writer will get a turn, because the room is empty, and when the writer is done, all the readers that arrived after the writer will be able to enter.

A new binary semaphore is needed here, called `turnstile`.

Writer

```
 1. wait( turnstile )
 2. wait( roomEmpty )
 3. [write data]
 4. post( turnstile )
 5. post( roomEmpty )
```

Reader

```
 1. wait( turnstile )
 2. post( turnstile )
 3. wait( mutex )
 4. readers++
 5. if readers == 1
 6.     wait( roomEmpty )
 7. end if
 8. post( mutex )
 9. [read data]
10. wait( mutex )
11. readers--
12. if readers == 0
13.     post( roomEmpty )
14. end if
15. post( mutex )
```

As before, the writer code is simpler, so we will examine it first. When the writer arrives, it will wait on the turnstile. If it is not the first writer, subsequent writers will queue up there, but if proceeds then it will wait for the room to be empty. Because the writer has locked the turnstile, no new readers can enter. There may be an arbitrary number of readers currently in the room, but each is there for only a finite amount of time. These existing-readers will be allowed to finish and leave the room. Then the writer gets a turn. When the writer is done, it posts the turnstile, which might unblock a reader or another writer.

Readers first get to the turnstile, and if they find it is locked, a writer is in its critical section. Thus, readers will queue at the turnstile if necessary, otherwise proceed. After that, the code is the same as we saw before: keep track of the number of readers and post if the room is empty or not empty.

Does this solution satisfy our goals of avoidance of deadlock and starvation? Starvation is fairly easy to assess: the first attempt at the solution had one scenario leading to starvation and this solution addresses it. Problem solved. You should be able to convince yourself that the solution as described cannot starve the writers or readers.

On to deadlock: the reader code is minimally changed from before; we have identified the turnstile code as not being a problem on its own (though its interactions with other threads need to be examined). The more dangerous block of code is on the side of the writer, because it has that pattern: two waits. If the writer is blocked on the `roomEmpty` semaphore, no readers or writers could advance past the turnstile and no writers. If the writer is blocked on that semaphore, it means there are readers in the room, and the readers will individually finish and leave (their progress is not impeded). Given that, the room will eventually become empty and the writer will be unblocked.

Note that this solution does not give writers any particular priority: when a writer exits it posts `turnstile` and that may unblock a reader or a writer. If it unblocks a reader, a whole bunch of readers may enter before the next writer is unblocked and locks the turnstile again. That may or may not be desirable, depending on the application. In any event, it does mean it is possible for readers to proceed even if a writer is queued. If there is a need to give writers priority, there are several techniques for doing so [Dow08].

Let's modify the solution so that writers have priority over readers. Giving writers priority may potentially cause readers to starve, but you may ignore this. We will probably want to break up the `roomEmpty` semaphore into `noReaders` and `noWriters`. A reader in the critical section should hold the `noReaders` semaphore and a writer should hold `noWriters` and `noReaders`.

Writer

```
 1. wait( writeMutex )
 2. writers++
 3. if writers == 1
 4.     wait( noReaders )
 5. end if
 6. post( writeMutex )
 7. wait ( noWriters )
 8. [write data]
 9. post( noWriters )
10. wait( writeMutex )
11. writers--
12. if writers == 0
13.    post( noReaders )
14. end if
15. post( writeMutex )
```

Reader

```
 1. wait( noReaders )
 2. wait( readMutex )
 3. readers++
 4. if readers == 1
 5.    wait( noWriters )
 6. end if
 7. post( readMutex )
 8. post( noReaders )
 9. [read data]
10. wait( readMutex )
11. readers--
12. if readers == 0
13.     post( noWriters )
14. end if
15. post( readMutex )
```

Yikes! The complexity for the writer increased dramatically. The reader is not all that different than it was before, however, and the writer now is to some extent the mirror image of the reader.

Readers Writers Example

Using the pseudocode as above, we can implement the readers-writers behaviour in a given program using only semaphores (optionally using a mutex to replace a semaphore). But in the pthread library there is support for readers-writers lock types, meaning we don't have to reinvent the wheel. We can probably well imagine the actual complexity of the previous solution, so it's our hope that the readers-writers lock would make our code simpler to read and harder to make mistakes.

The type for the lock is `pthread_rwlock_t`. It is analogous, obviously, to the mutex type `pthread_mutex_t`. Let's consider the functions that we have:

```
pthread_rwlock_init( pthread_rwlock_t * rwlock, pthread_rwlockattr_t * attr )
pthread_rwlock_rdlock( pthread_rwlock_t * rwlock )
pthread_rwlock_tryrdlock( pthread_rwlock_t * rwlock )
pthread_rwlock_wrlock( pthread_rwlock_t * rwlock )
pthread_rwlock_trywrlock( pthread_rwlock_t  *  rwlock )
pthread_rwlock_unlock( pthread_rwlock_t * rwlock )
pthread_rwlock_destroy( pthread_rwlock_t * rwlock )
```

In general our syntax very much resembles that of the mutex (attribute initialization and destruction not shown but they do exist). There are some small noteworthy differences, other than obviously the different type of the structure passed. Whereas before we had functions for lock and trylock, we now have those split into readlock and writelock (each of which has its own trylock function). As before, we will return to the subject of how trylock works soon.

In theory, the same thread may lock the same rwlock n times; just remember to unlock it n times as well.

And speaking of unlock, there's no specifying whether you are releasing a read or write lock. This is because it is unnecessary; the implementation unlocks whatever type the calling thread was holding. Much like `close()`, if we can figure out what we're closing we don't need the caller of the function to specify what to do.

As for whether readers or writers get priority, the specification says this is implementation defined. If possible, for threads of equal priority, a writer takes precedence over a reader. But your system may vary.

Consider the following example of the simple readers-writers (without writer priority and with risk of starvation) using the "old" way:

```
int readers;
pthread_mutex_t mutex;
sem_t roomEmpty;

void init( ) {
  readers = 0;
  pthread_mutex_init( &mutex, NULL );
  sem_init( &roomEmpty, 0, 1 );
}

void cleanup( ) {
  pthread_mutex_destroy( &mutex );
  sem_destroy( &roomEmpty );
}
```

```
void* writer( void* arg ) {
  sem_wait( &roomEmpty );
  write_data( arg );
  sem_post( &roomEmpty );
}

void* reader( void* read ) {
  pthread_mutex_lock( &mutex );
  readers++;
  if ( readers == 1 ) {
    sem_wait( &roomEmpty );
  }
  pthread_mutex_unlock( &mutex );
  read_data( arg );
  pthread_mutex_lock( &mutex );
  readers--;
  if ( readers == 0 ) {
    sem_post( &roomEmpty );
  }
  pthread_mutex_unlock( &mutex );
}
```

Now see it as it would be with the use of a rwlock!

```
pthread_rwlock_t rwlock;

void init( ) {
  pthread_rwlock_init( &rwlock, NULL );
}

void cleanup( ) {
  pthread_rwlock_destroy( &rwlock );
}
```

```
void* writer( void* arg ) {
  pthread_rwlock_wrlock( &rwlock );
  write_data( arg );
  pthread_rwlock_unlock( &rwlock );
}

void* reader( void* read ) {
  pthread_rwlock_rdlock( &rwlock );
  read_data( arg );
```

```
  pthread_rwlock_unlock( &rwlock );                    }
```

Seek and Destroy: the Search-Insert-Delete Problem

This is an extension of the readers-writers problem called the search-insert-delete problem. Instead of two types of thread, reader and writer, there are three types of thread: searchers, inserters, deleters. They operate on a shared linked list of data.

Searchers merely examine the list; hence they can execute concurrently with each other. Searcher threads must call `void search( void* target )` where the argument to the searcher thread is the element to be found. These most closely resemble readers in the readers-writers problem.

Inserters add new items to the end of the list; only one insertion may take place at a time. However, one insert can proceed in parallel with any number of searches. Inserter threads call `node* find_insert_loc()` to find where to do the insertion; then `void insert( void* to_insert, node* after )` where the arguments are the location and the element to be inserted. Assume `insert` is written so the insertion can be done in parallel with the searches. Inserters resemble readers with an additional rule that only one of them can manipulate the list at a time.

Deleters remove items from anywhere in the list. At most one deleter process can access the list at a time, and when the deleter is accessing the list, no inserters and no searchers may be accessing the list. Deleter threads call `void delete( void* to_delete )` where the argument to the deleter thread is the element to be deleted. These most closely resemble writers in the readers-writers problem.

It turns out we don't need to modify things too much to allow for this third kind of thread. We need to keep track of when there are "no inserters" and "no searchers" (some hints for our semaphores) and another mutex to go around the actual insertion... See the code implementation below.

```
pthread_mutex_t searcher_mutex;
pthread_mutex_t inserter_mutex;
pthread_mutex_t perform_insert;
sem_t no_searchers;
sem_t no_inserters;
int searchers;
int inserters;


void init( ) {
  pthread_mutex_init( &searcher_mutex, NULL );
  pthread_mutex_init( &inserter_mutex, NULL );
  pthread_mutex_init( &perform_insert, NULL );
  sem_init( &no_inserters, 0, 1 );
  sem_init( &no_searchers, 0, 1 );
  searchers = 0;
  inserters = 0;
}

void* searcher_thread( void *target ) {
  pthread_mutex_lock( &searcher_mutex );
  searchers++;
  if ( searchers == 1 ) {
    sem_wait( &no_searchers );
  }
  pthread_mutex_unlock( &searcher_mutex );

  search( target );

  pthread_mutex_lock( &searcher_mutex );
  searchers--;
  if ( searchers == 0 ) {
    sem_post( &no_searchers );
  }
  pthread_mutex_unlock( &searcher_mutex );
}
```

```
void* deleter_thread( void* to_delete ) {
  sem_wait( &no_searchers );
  sem_wait( &no_inserters );

  delete( to_delete );

  sem_post( &no_inserters );
  sem_post( &no_searchers );
}


void* inserter_thread( void *to_insert ) {
  pthread_mutex_lock( &inserter_mutex );
  inserters++;
  if ( inserters == 1 ) {
    sem_wait( &no_inserters );
  }
  pthread_mutex_unlock( &inserter_mutex );

  node * insert_after = find_insert_location( );
  pthread_mutex_lock( &perform_insert );
  insert( to_insert, insert_after ); /* Can update its
      position if needed */
  pthread_mutex_unlock( &perform_insert );

  pthread_mutex_lock( &inserter_mutex );
  inserters--;
  if ( inserters == 0 ) {
    sem_post( &no_inserters );
  }
  pthread_mutex_unlock( &inserter_mutex );
}
```

Question: could you implement this with a `pthread_rwlock_t` despite there being three kinds of thread?

References

[Dow08] Allen B. Downey. *The Little Book of Semaphores (2nd Edition)*. Green Tea Press, 2008.

[HZMG15] Douglas Wilhelm Harder, Jeff Zarnett, Vajih Montaghami, and Allyson Giannikouris. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2015. Online; version 0.15.08.17.

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.