Lecture 16 — The Readers-Writers Problem

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 27, 2022

Reader/Writer Locks

Reads don’t interfere with one another so we can let them run in parallel!

But sometimes writes occur, and nobody can read when this happens.

Image Credit: Understood.org

Reader/Writer Rules

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).

This is very often how file systems work.

With Variation

This is similar to, but distinct from, the general mutual exclusion problem and the producer-consumer 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.

Allowing multiple readers can permit better performance!

And there are many scenarios where updates are rare but reads are common.

Readers-Writers Solution 1

Let us keep track of the number of readers at any given time with `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`.

A writer has to wait for the room to be empty (i.e., `wait` on the `roomEmpty` semaphore) before it can enter.

Readers-Writers Solution 1

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 )
```

Readers-Writers Solution 1 Analysis

The first reader that arrives encounters the situation that the room is empty, so it "locks" the room (waiting on the `roomEmpty` semaphore).

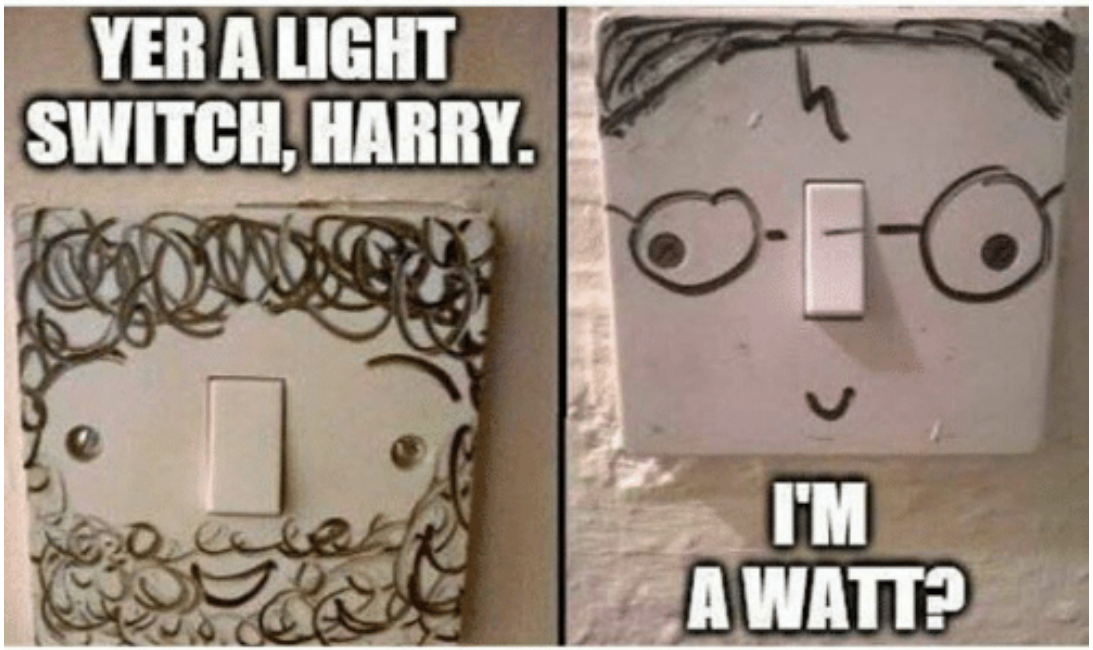
That will prevent writers from entering the room.

Additional readers do not check if the room is empty; they just proceed to enter.

When the last reader leaves the room, it indicates that the room is empty ("unlocking it" to allow a writer in).

Readers-Writers Solution 1 Analysis

This pattern is sometimes called the light switch.



Readers-Writers Solution 1 Analysis

The reader code has that situation that makes us concerned.

A wait on `roomEmpty` inside a critical section controlled by `mutex`.

With a bit of reasoning, we can convince ourselves that there is no deadlock.

A reader waits on `roomEmpty` only if a writer is currently in its critical section.

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.

Readers-Writers Solution 1 Analysis

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.

Readers-Writers Solution 1 Analysis

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.

Starvation

Recall criterion 3 of properties we want in any mutual exclusion solution:

> It must not be possible for a thread 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.

Readers-Writers Solution 2

When a writer arrives, any readers should be permitted to finish their read.
No new readers should be allowed to start reading.

Eventually, all the readers currently in the critical section will finish.

The writer will get a turn, because the room is empty.

When the writer is done, all the readers that arrived after the writer will be able to enter.

Readers-Writers Solution 2

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 )
```

Readers-Writers Solution 2 Analysis

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.

You should be able to convince yourself that the solution as described cannot starve the writers or readers.

Readers-Writers Solution 2 Analysis

On to deadlock: the reader code is minimally changed from before

The writer has that dangerous 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, there are readers in the room.

The readers will individually finish and leave (their progress is not impeded).

So the room will eventually become empty; the writer will be unblocked.

Readers-Writers Solution 2 Analysis

Note that this solution does not give writers any particular priority: when a writer exits it posts on `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 techniques for doing so.

Business Class Passengers Board in Zone 1

Image Credit: Tag Along Travel

Business Class Passengers Board in Zone 1

Let's modify the solution so that writers have priority over readers.

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 Priority

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 )
```

Analysis: Writer Priority

Yikes! The complexity for the writer increased dramatically.

The reader is not all that different than it was before.

The writer now is to some extent the mirror image of the reader.

Too Much Typing?

Using the pseudocode as above, we can implement the readers-writers behaviour in a given program using only semaphore and mutex constructs.

But in the pthread library there is support for readers-writers lock types, meaning we don't have to reinvent the wheel.

The type for the lock is `pthread_rwlock_t`.

RW Lock Syntax: Init, Destroy

```
pthread_rwlock_init( pthread_rwlock_t * rwlock, pthread_rwlockattr_t * attr )
```

`rwlock`: the lock to initialize.

`attr`: the attributes (NULL for defaults is fine.)

```
pthread_rwlock_destroy( pthread_rwlock_t * rwlock )
```

`rwlock`: the lock to destroy.

Both of these are exactly like the regular mutex.

RW Lock Syntax: Locking

```
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 )
```

We can acquire a read lock, or a write lock.

Pretty self-explanatory, which is which and when you use what.

For each of those there is an associated trylock function.

All four functions take one argument: `rwlock`, the lock.

RW Lock Syntax: Unlock

```
pthread_rwlock_unlock( pthread_rwlock_t * rwlock )
```

Unlock works just like the regular mutex unlock.

We do not need to specify what kind of luck we are releasing.

Writer Priority?

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.

Mine, Mine, Mine!

In theory, the same thread may lock the same rwlock *n* times.

Just remember to unlock it *n* times as well.

Readers get priority? Implementation defined.

Without RW Locks

```
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 );
}
```

With RW Locks

```
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 );
}
```

Conclusion: don’t reinvent the wheel!

Seek and Destroy

An extension of the readers-writers problem: the search-insert-delete problem.

Three kinds of thread: searcher, inserter, deleter.

Searcher Rules

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.

Inserter Rules

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.

Inserters resemble readers, with restrictions.

Deleter Rules

Deleters remove items from anywhere in the list. At most one deleter process can access the list at a time.

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 )`.

These resemble writers.

Let’s Get It Done

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”.

Plus another mutex to go around the actual insertion…

Search-Insert-Delete

```
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;
}
```

Search-Insert-Delete

```
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 );
}
```

Search-Insert-Delete

```
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 );
}
```

Search-Insert-Delete

```
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 );
  pthread_mutex_unlock( &perform_insert );

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

Make It Work

Could you implement Search-Insert-Delete with a `pthread_rwlock_t` despite there being three kinds of thread?