Lecture 16 — The Readers-Writers Problem

By Jeff Zarnett and Seyed Majid Zahedi
jzarnett@uwaterloo.ca, smzahedi@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

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

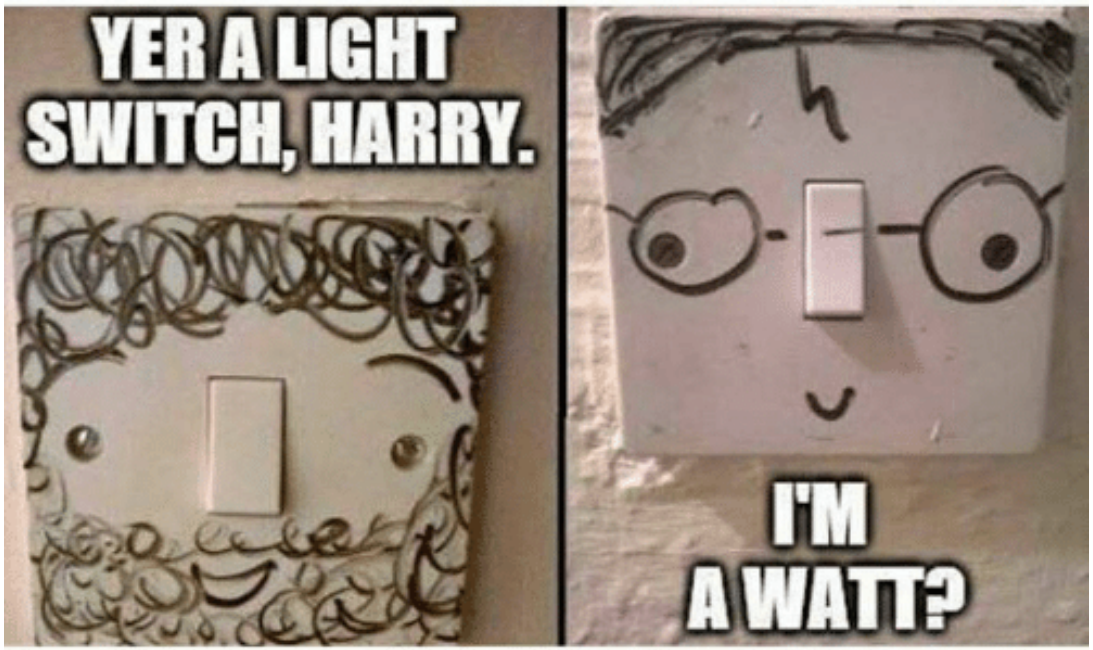
The first reader that arrives encounters the situation that the room is empty, so it "locks" the room (waiting on the `roomEmpty` semaphore).

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).

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 an instance of starvation.

We might want to 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.

Writer-Reader Implementation with Monitor

We will use the mutex for mutual exclusion (duh!).

We will use `ok2write` for writers and `ok2read` for readers to wait on.

Four variables are used to track the state of waiting/active readers and writers:

1. `ww` for # of waiting writers;
2. `wr` for # of waiting readers;
3. `aw` for # of active writers; and
4. `ar` for # of active readers.

Solution 1

Writer

```
 1. lock( mutex )
 2. while aw + ar > 0
 3.     ww++
 4.     cond_wait(ok2write, mutex)
 5.     ww--
 6. end while
 7. aw++
 8. unlock ( mutex )
 9. [write data]
10. lock( mutex )
11. aw--
12. if ww > 0
13.    cond_signal( ok2write )
14. else if wr > 0
15.    cond_broadcast( ok2read )
16. end if
17. unlock ( mutex )
```

Reader

```
 1. lock( mutex )
 2. while aw + ww > 0
 3.    wr++
 4.    cond_wait(ok2read, mutex)
 5.    wr--
 6. end while
 7. ar++
 8. unlock( mutex )
 9. [read data]
10. lock( mutex )
11. ar--
12. if ar == 0 && ww > 0
13.     cond_signal( ok2write )
14. end if
15. unlock( mutex )
```

Analysis of Solution 1

Due to the use of a single mutex, at any given time, we know exactly how many waiting/active reader and writer threads we have.

This pseudocode prioritizes writers over readers for two main reasons.

(1) writers wait if there are any active readers or writers. Readers, on the other hand, wait if there are any active or waiting writers.

(2) on their way out, each writer signals another waiting writer if there are any. The last writer signals all waiting readers if there are any (broadcast). Conversely, the last reader only signals a waiting writer if there are any.

RWLock Syntax

We can write our own writer-reader lock.

```
rwlock_init( rwlock_t * rwlock )
rwlock_destroy( rwlock_t * rwlock )
rwlock_r_lock( rwlock_t * rwlock )
rwlock_w_lock( rwlock_t * rwlock )
rwlock_r_unlock( rwlock_t  *  rwlock )
rwlock_w_unlock( rwlock_t  *  rwlock )
```

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

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

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

Code Example

```
rwlock_t rwlock;

void init( ) {
  rwlock_init( &rwlock );
}

void cleanup( ) {
  rwlock_destroy( &rwlock );
}

void* writer( void* arg ) {
  rwlock_w_lock( &rwlock );
  write_data( arg );
  rwlock_w_unlock( &rwlock );
}

void* reader( void* read ) {
  rwlock_r_lock( &rwlock );
  read_data( arg );
  rwlock_r_unlock( &rwlock );
}
```

Would This Work?

Writer

```
 1. lock( mutex )
 2. while aw + ar > 0
 3.     ww++
 4.     cond_wait(ok2write, mutex)
 5.     ww--
 6. end while
 7. aw++
 8. unlock ( mutex )
 9. [write data]
10. lock( mutex )
11. aw--
12. if ww > 0
13.    cond_signal( ok2write )
14. else if wr > 0
15.    cond_broadcast( ok2read )
16. end if
17. unlock ( mutex )
```

Reader

```
 1. lock( mutex )
 2. while aw + ww > 0
 3.    wr++
 4.    cond_wait(ok2read, mutex)
 5.    wr--
 6. end while
 7. ar++
 8. unlock( mutex )
 9. [read data]
10. lock( mutex )
11. ar--
12. if ar == 0 && ww > 0
13.     cond_signal( ok2write )
14. end if
15. unlock( mutex )
```

How About This?

Writer

```
 1. lock( mutex )
 2. while aw + ar > 0
 3.     ww++
 4.     cond_wait(ok2go, mutex)
 5.     ww--
 6. end while
 7. aw++
 8. unlock ( mutex )
 9. [write data]
10. lock( mutex )
11. aw--
12. if ww > 0
13.    cond_signal( ok2go )
14. else if wr > 0
15.    cond_broadcast( ok2go )
16. end if
17. unlock ( mutex )
```

Reader

```
 1. lock( mutex )
 2. while aw + ww > 0
 3.    wr++
 4.    cond_wait(ok2go, mutex)
 5.    wr--
 6. end while
 7. ar++
 8. unlock( mutex )
 9. [read data]
10. lock( mutex )
11. ar--
12. if ar == 0 && ww > 0
13.     cond_signal( ok2go )
14. end if
15. unlock( mutex )
```

How About This?

Writer

```
 1. lock( mutex )
 2. while aw + ar > 0
 3.     ww++
 4.     cond_wait(ok2go, mutex)
 5.     ww--
 6. end while
 7. aw++
 8. unlock ( mutex )
 9. [write data]
10. lock( mutex )
11. aw--
12. if ww > 0
13.    cond_broadcast( ok2go )
14. else if wr > 0
15.    cond_broadcast( ok2go )
16. end if
17. unlock ( mutex )
```

Reader

```
 1. lock( mutex )
 2. while aw + ww > 0
 3.    wr++
 4.    cond_wait(ok2go, mutex)
 5.    wr--
 6. end while
 7. ar++
 8. unlock( mutex )
 9. [read data]
10. lock( mutex )
11. ar--
12. if ar == 0 && ww > 0
13.     cond_broadcast( ok2go )
14. end if
15. unlock( mutex )
```

What About Starvation?

Readers could starve because writers have higher priority.

But this starvation is ‘OK’ because it is allowed by design.

However, the code suffers from another form of starvation for writer threads.

This starvation has the same nature as we discussed for the producer-consumer implementation (the ‘where is my ice cream?’ problem).

What About Starvation?

In particular, a writer could indefinitely wait for other writer threads.

As we discussed in the previous lecture, the probability of such an event happening is extremely low.

So, we might want to leave this as is and move on.

However, like before, let's try to make our solution 100% starvation-free!

Solution 2: Does It Work?

Writer

```
 1. lock( mutex )
 2. while aw + ar + ww > 0
 3.     ww++
 4.     cond_wait(ok2write, mutex)
 5.     ww--
 6. end while
 7. aw++
 8. unlock ( mutex )
 9. [write data]
10. lock( mutex )
11. aw--
12. if ww > 0
13.    cond_signal( ok2write )
14. else if wr > 0
15.    cond_broadcast( ok2read )
16. end if
17. unlock ( mutex )
```

Reader

```
 1. lock( mutex )
 2. while aw + ww > 0
 3.    wr++
 4.    cond_wait(ok2read, mutex)
 5.    wr--
 6. end while
 7. ar++
 8. unlock( mutex )
 9. [read data]
10. lock( mutex )
11. ar--
12. if ar == 0 && ww > 0
13.     cond_signal( ok2write )
14. end if
15. unlock( mutex )
```

Analysis of Solution 2

It could cause a deadlock (why?).

Instead, to properly address the issue, we should take the same approach we took with the producer-consumer implementation.

Namely, we use a take-a-number system and make writers wait in a FIFO line.

Solution 3

Writer

```
 1. lock( mutex )
 2. my_turn = w_turn++
 3. fifo_push(w_fifo, my_ok2write)
 4. while aw + ar > 0 ||
          next_w_turn < my_turn
 5.     ww++
 6.     cond_wait(my_ok2write, mutex)
 7.     ww--
 8. end while
 9. aw++
10. fifo_pop( w_fifo )
11. unlock ( mutex )
12. [write data]
13. lock( mutex )
14. aw--
15. next_w_turn++
16. if ww > 0
17.    cond_signal(fifo_head(w_fifo))
18. else if wr > 0
19.    cond_broadcast( ok2read )
20. end if
21. unlock ( mutex )
```

Reader

```
 1. lock( mutex )
 2. while aw + ww > 0
 3.    wr++
 4.    cond_wait(ok2read, mutex)
 5.    wr--
 6. end while
 7. ar++
 8. unlock( mutex )
 9. [read data]
10. lock( mutex )
11. ar--
12. if ar == 0 && ww > 0
13.    cond_signal(fifo_head(w_fifo))
14. end if
15. unlock( mutex )
```

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 );
}
```

```
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 `rwlock_t` despite there being three kinds of thread?