Synchronization, Part 7: The Reader Writer Problem

Edit New Page

Jump to bottom

Bhuvan Venkatesh edited this page on Dec 31, 2016 · 5 revisions

What is the Reader Writer Problem?

Imagine you had a key-value map data structure which is used by many threads. Multiple threads should be able to look up (read) values at the same time provided the data structure is not being written to. The writers are not so gregarious - to avoid data corruption, only one thread at a time may modify (`write`) the data structure (and no readers may be reading at that time).

This is an example of the *Reader Writer Problem*. Namely how can we efficiently synchronize multiple readers and writers such that multiple readers can read together but a writer gets exclusive access?

An incorrect attempt is shown below ("lock" is a shorthand for `pthread_mutex_lock`):

Attempt #1

```
read() {
  lock(&m)
  // do read stuff
  unlock(&m)
}
```

```
write() {
  lock(&m)
  // do write stuff
  unlock(&m)
}
```

At least our first attempt does not suffer from data corruption (readers must wait while a writer is writing and vice versa)! However readers must also wait for other readers. So let's try another implementation..

Attempt #2:

```
read() {
  while(writing) {/*spin*/}
```

```
write() {
  while(reading || writing) {/*spin*/}
```

```
    reading = 1
    // do read stuff
    reading = 0
  }
```

```
    writing = 1
    // do write stuff
    writing = 0
  }
```

Our second attempt suffers from a race condition - imagine if two threads both called `read` and `write` (or both called write) at the same time. Both threads would be able to proceed! Secondly, we can have multiple readers and multiple writers, so lets keep track of the total number of readers or writers. Which brings us to attempt #3,

Attempt #3

Remember that `pthread_cond_wait` performs *Three* actions. Firstly it atomically unlocks the mutex and then sleeps (until it is woken by `pthread_cond_signal` or `pthread_cond_broadcast`). Thirdly the awoken thread must re-acquire the mutex lock before returning. Thus only one thread can actually be running inside the critical section defined by the lock and unlock() methods.

Implementation #3 below ensures that a reader will enter the cond_wait if there are any writers writing.

```
read() {
    lock(&m)
    while (writing)
        cond_wait(&cv, &m)
    reading++;

/* Read here! */

    reading--
    cond_signal(&cv)
    unlock(&m)
}
```

However only one reader a time can read because candidate #3 did not unlock the mutex. A better version unlocks before reading :

```
read() {
    lock(&m);
    while (writing)
        cond_wait(&cv, &m)
    reading++;
    unlock(&m)
```

```
/* Read here! */
    lock(&m)
    reading--
    cond_signal(&cv)
    unlock(&m)
}
```

Does this mean that a writer and read could read and write at the same time? No! First of all, remember cond_wait requires the thread re-acquire the mutex lock before returning. Thus only one thread can be executing code inside the critical section (marked with **) at a time!

```
read() {
    lock(&m);
**  while (writing)
**      cond_wait(&cv, &m)
**  reading++;
    unlock(&m)
/* Read here! */
    lock(&m)
**  reading--
**  cond_signal(&cv)
    unlock(&m)
}
```

Writers must wait for everyone. Mutual exclusion is assured by the lock.

```
write() {
    lock(&m);
**  while (reading || writing)
**      cond_wait(&cv, &m);
**  writing++;
**
** /* Write here! */
**  writing--;
**  cond_signal(&cv);
    unlock(&m);
}
```

Candidate #3 above also uses `pthread_cond_signal` ; this will only wake up one thread. For example, if many readers are waiting for the writer to complete then only one sleeping reader will be awoken from their slumber. The reader and writer should use `cond_broadcast` so that all threads should wake up and check their while-loop condition.

Starving writers

Candidate #3 above suffers from starvation. If readers are constantly arriving then a writer will never be able to proceed (the 'reading' count never reduces to zero). This is known as *starvation* and would be discovered under heavy loads. Our fix is to implement a bounded-wait for the writer. If a writer arrives they will still need to wait for existing readers however future readers must be placed in a "holding pen" and wait for the writer to finish. The "holding pen" can be implemented using a variable and a condition variable (so that we can wake up the threads once the writer has finished).

Our plan is that when a writer arrives, and before waiting for current readers to finish, register our intent to write (by incrementing a counter 'writer'). Sketched below -

```
write() {
    lock()
    writer++

    while (reading || writing)
    cond_wait
    unlock()
  ...
}
```

And incoming readers will not be allowed to continue while writer is nonzero. Notice 'writer' indicates a writer has arrived, while 'reading' and 'writing' counters indicate there is an *active* reader or writer.

```
read() {
    lock()
    // readers that arrive *after* the writer arrived will have to wait here!
    while(writer)
    cond_wait(&cv,&m)

    // readers that arrive while there is an active writer
    // will also wait.
    while (writing)
        cond_wait(&cv,&m)
    reading++
    unlock
  ...
}
```

Attempt #4

Below is our first working solution to the Reader-Writer problem. Note if you continue to read about the "Reader Writer problem" then you will discover that we solved the "Second Reader Writer problem" by giving writers preferential access to the lock. This solution is not optimal. However it satisfies our original problem (N active readers, single active writer, avoids starvation of the writer if there is a constant stream of readers).

Can you identify any improvements? For example, how would you improve the code so that we only woke up readers or one writer?

```
int writers; // Number writer threads that want to enter the critical section (
int writing; // Number of threads that are actually writing inside the C.S. (ca
int reading; // Number of threads that are actually reading inside the C.S.
// if writing !=0 then reading must be zero (and vice versa)

reader() {
    lock(&m)
    while (writers)
        cond_wait(&turn, &m)
    // No need to wait while(writing here) because we can only exit the above l
    // when writing is zero
    reading++
    unlock(&m)

  // perform reading here

    lock(&m)
    reading--
    cond_broadcast(&turn)
    unlock(&m)
}

writer() {
    lock(&m)
    writers++
    while (reading || writing)
        cond_wait(&turn, &m)
    writing++
    unlock(&m)
    // perform writing here
    lock(&m)
    writing--
    writers--
    cond_broadcast(&turn)
    unlock(&m)
}
```

Clone this wiki locally

`https://github.com/angrave/SystemProgramming.wiki.git`