

# Synchronization, Part 9: The Reader Writer Problem (part 2)

Alex Kizer edited this page on Apr 6 · 17 revisions

## The introduction

The reader-writer problem appears in many different contexts. For example a web server cache needs to quickly serve the same static page for many thousands of requests. Occasionally however an author may decide to update the page.

We will examine one specific form of the reader-writer problem where there are many readers and some occasional writers and we need to ensure that a writer gets exclusive access. For performance however readers should be able to perform the read without waiting for another reader.

See [Synchronization,-Part-7:-The-Reader-Writer-Problem](#) for part 1

## Candidate solution #3

Candidate solutions 1 and 2 are discussed in [part 1](#).

In the code below for clarity `lock` and `cond_wait` are shortened versions `pthread_mutex_lock` and `pthread_cond_wait` etc. respectively

Also 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)
}
```

▼ Pages 51

[Home](#)

[#Example Markdown](#)

[#Informal Glossary](#)

[#Piazza: When And How to Ask For Help](#)

[C Programming, Part 1: Introduction](#)

[C Programming, Part 2: Text Input And Output](#)

[C Programming, Part 3: Common Gotchas](#)

[C Programming, Part 4: Debugging](#)

[Deadlock, Part 1: Resource Allocation Graph](#)

[Deadlock, Part 2: Deadlock Conditions](#)

[File System, Part 1: Introduction](#)

[File System, Part 2: Files are inodes \(everything else is just data...\)](#)


[File System, Part 3: Permissions](#)


[File System, Part 4: Working with directories](#)


[File System, Part 5: Virtual file systems](#)


Show 36 more pages...


Clone this wiki locally


 Clone in Desktop











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
    ...
}
```

## Candidate solution #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 (so
int writing; // Number of threads that are actually writing inside the C.S. (can
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 loop
    // 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)
}
```

Legal and Licensing information: Unless otherwise specified, submitted content to the wiki must be original work (including text, java code, and media) and you provide this material under a [Creative Commons License](#). If you are not the copyright holder, please give proper attribution and credit to existing content and ensure that you have license to include the materials.

