WIKIPEDIA

Readers–writers problem

In computer science, the **readers–writers problems** are examples of a common computing problem in concurrency.[1] There are at least three variations of the problems, which deal with situations in which many concurrent threads of execution try to access the same shared resource at one time.

Some threads may read and some may write, with the constraint that no thread may access the shared resource for either reading or writing while another thread is in the act of writing to it. (In particular, we want to prevent more than one thread modifying the shared resource simultaneously and allow for two or more readers to access the shared resource at the same time). A readers–writer lock is a data structure that solves one or more of the readers–writers problems.

The basic reader–writers problem was first formulated and solved by Courtois *et al.*[2][3]

Contents

First readers–writers problem

Suppose we have a shared memory area (critical section) with the basic constraints detailed above. It is possible to protect the shared data behind a mutual exclusion mutex, in which case no two threads can access the data at the same time. However, this solution is sub-optimal, because it is possible that a reader R_1 might have the lock, and then another reader R_2 requests access. It would be foolish for R_2 to wait until R_1 was done before starting its own read operation; instead, R_2 should be allowed to read the resource alongside R_1 because reads don't modify data, so concurrent reads are safe. This is the motivation for the **first readers–writers problem**, in which the constraint is added that *no reader shall be kept waiting if the share is currently opened for reading*. This is also called **readers-preference**, with its solution:

```
semaphore resource=1;
semaphore rmutex=1;
readcount=0;

/*
   resource.P() is equivalent to wait(resource)
   resource.V() is equivalent to signal(resource)
```

```
   rmutex.P() is equivalent to wait(rmutex)
   rmutex.V() is equivalent to signal(rmutex)
*/

writer() {
    resource.P();          //Lock the shared file for a writer

    <CRITICAL Section>
    // Writing is done

    <EXIT Section>
    resource.V();          //Release the shared file for use by other readers. Writers are allowed if there are no readers requesting it.
}

reader() {
    rmutex.P();           //Ensure that no other reader can execute the <Entry> section while you are in it
    <CRITICAL Section>
    readcount++;          //Indicate that you are a reader trying to enter the Critical Section
    if (readcount == 1)   //Checks if you are the first reader trying to enter CS
        resource.P();     //If you are the first reader, lock the resource from writers. Resource stays reserved for subsequent readers
    <EXIT CRITICAL Section>
    rmutex.V();           //Release

    // Do the Reading

    rmutex.P();           //Ensure that no other reader can execute the <Exit> section while you are in it
    <CRITICAL Section>
    readcount--;          //Indicate that you no longer need the shared resource. One fewer reader
    if (readcount == 0)   //Checks if you are the last (only) reader who is reading the shared file
        resource.V();     //If you are last reader, then you can unlock the resource. This makes it available to writers.
    <EXIT CRITICAL Section>
    rmutex.V();           //Release
}
```

In this solution of the readers/writers problem, the first reader must lock the resource (shared file) if such is available. Once the file is locked from writers, it may be used by many subsequent readers without having them to re-lock it again.

Before entering the critical section, every new reader must go through the entry section. However, there may only be a single reader in the entry section at a time. This is done to avoid race conditions on the readers (in this context, a race condition is a condition in which two or more threads are waking up simultaneously and trying to enter the critical section; without further constraint, the behavior is nondeterministic. E.g. two readers increment the readcount at the same time, and both try to lock the resource, causing one reader to block). To accomplish this, every reader which enters the <ENTRY Section> will lock the <ENTRY Section> for themselves until they are done with it. At this point the readers are not locking the resource. They are only locking the entry section so no other reader can enter it while they are in it. Once the reader is done executing the entry section, it will unlock it by signalling the mutex. Signalling it is equivalent to: mutex.V() in the above code. Same is valid for the <EXIT Section>. There can be no more than a single reader in the exit section at a time, therefore, every reader must claim and lock the Exit section for themselves before using it.

Once the first reader is in the entry section, it will lock the resource. Doing this will prevent any writers from accessing it. Subsequent readers can just utilize the locked (from writers) resource. The reader to finish last (indicated by the readcount variable) must unlock the resource, thus making it available to writers.

In this solution, every writer must claim the resource individually. This means that a stream of readers can subsequently lock all potential writers out and starve them. This is so, because after the first reader locks the resource, no writer can lock it, before it gets released. And it will only be released by the last reader. Hence, this solution does not satisfy fairness.

Second readers–writers problem

The first solution is suboptimal, because it is possible that a reader R_1 might have the lock, a writer W be waiting for the lock, and then a reader R_2 requests access. It would be unfair for R_2 to jump in immediately, ahead of W; if that happened often enough, W would starve. Instead, W should start as soon as possible. This is the motivation for the **second readers–writers problem**, in which the constraint is added that *no writer, once added to the queue, shall be kept waiting longer than absolutely necessary*. This is also called **writers-preference**.

A solution to the writers-preference scenario is:[2]

```
int readcount, writecount;                  //(initial value = 0)
semaphore rmutex, wmutex, readTry, resource; //(initial value = 1)

//READER
reader() {
<ENTRY Section>
  readTry.P();                  //Indicate a reader is trying to enter
  rmutex.P();                   //lock entry section to avoid race condition with other readers
  readcount++;                  //report yourself as a reader
  if (readcount == 1)           //checks if you are first reader
    resource.P();               //if you are first reader, lock  the resource
  rmutex.V();                   //release entry section for other readers
  readTry.V();                  //indicate you are done trying to access the resource

<CRITICAL Section>
//reading is performed

<EXIT Section>
  rmutex.P();                   //reserve exit section - avoids race condition with readers
  readcount--;                  //indicate you're leaving
  if (readcount == 0)           //checks if you are last reader leaving
    resource.V();               //if last, you must release the locked resource
  rmutex.V();                   //release exit section for other readers
}

//WRITER
writer() {
<ENTRY Section>
  wmutex.P();                   //reserve entry section for writers - avoids race conditions
  writecount++;                 //report yourself as a writer entering
  if (writecount == 1)          //checks if you're first writer
    readTry.P();                //if you're first, then you must lock the readers out. Prevent them
from trying to enter CS
  wmutex.V();                   //release entry section
  resource.P();                 //reserve the resource for yourself - prevents other writers from
simultaneously editing the shared resource
<CRITICAL Section>
  //writing is performed
  resource.V();                 //release file

<EXIT Section>
  wmutex.P();                   //reserve exit section
  writecount--;                 //indicate you're leaving
  if (writecount == 0)          //checks if you're the last writer
    readTry.V();                //if you're last writer, you must unlock the readers. Allows them to
try enter CS for reading
  wmutex.V();                   //release exit section
}
```

In this solution, preference is given to the writers. This is accomplished by forcing every reader to lock and release the readtry semaphore individually. The writers on the other hand don't need to lock it individually. Only the first writer will lock the readtry and then all subsequent writers can simply use the resource as it gets freed by the previous writer. The very last writer must release the readtry semaphore, thus opening the gate for readers to try reading.

No reader can engage in the entry section if the readtry semaphore has been set by a writer previously. The reader must wait for the last writer to unlock the resource and readtry semaphores. On the other hand, if a particular reader has locked the readtry semaphore, this will indicate to any potential concurrent writer that there is a reader in the entry section. So the writer will wait for the reader to release the readtry and then the writer will immediately lock it for itself and all subsequent writers. However, the writer will not be able to access the resource until the current reader has released the resource, which only occurs after the reader is finished with the resource in the critical section.

The resource semaphore can be locked by both the writer and the reader in their entry section. They are only able to do so after first locking the readtry semaphore, which can only be done by one of them at a time.

If there are no writers wishing to get to the resource, as indicated to the reader by the status of the readtry semaphore, then the readers will not lock the resource. This is done to allow a writer to immediately take control over the resource as soon as the current reader is finished reading. Otherwise, the writer would need to wait for a queue of readers to be done before the last one can unlock the readtry semaphore. As soon as a writer shows up, it will try to set the readtry and hang up there waiting for the current reader to release the readtry. It will then take control over the resource as soon as the current reader is done reading and lock all future readers out. All subsequent readers will hang up at the readtry semaphore waiting for the writers to be finished with the resource and to open the gate by releasing readtry.

The rmutex and wmutex are used in exactly the same way as in the first solution. Their sole purpose is to avoid race conditions on the readers and writers while they are in their entry or exit sections.

Third readers–writers problem

In fact, the solutions implied by both problem statements can result in starvation — the first one may starve writers in the queue, and the second one may starve readers. Therefore, the **third readers–writers problem** is sometimes proposed, which adds the constraint that *no thread shall be allowed to starve*; that is, the operation of obtaining a lock on the shared data will always terminate in a bounded amount of time. A solution with fairness for both readers and writers might be as follows:

```
int readcount;                // init to 0; number of readers currently accessing resource

// all semaphores initialised to 1
semaphore resource;           // controls access (read/write) to the resource
semaphore rmutex;             // for syncing changes to shared variable readcount
semaphore serviceQueue;       // FAIRNESS: preserves ordering of requests (signaling must be FIFO)

//READER
reader() {
<ENTRY Section>
  serviceQueue.P();           // wait in line to be serviced
  rmutex.P();                 // request exclusive access to readcount
  readcount++;                // update count of active readers
  if (readcount == 1)         // if I am the first reader
    resource.P();             // request resource access for readers (writers blocked)
  serviceQueue.V();           // let next in line be serviced
  rmutex.V();                 // release access to readcount

<CRITICAL Section>
//reading is performed

<EXIT Section>
  rmutex.P();                 // request exclusive access to readcount
  readcount--;                // update count of active readers
  if (readcount == 0)         // if there are no readers left
```

```
    resource.V();             // release resource access for all
  rmutex.V();                 // release access to readcount
}

//WRITER
writer() {
<ENTRY Section>
  serviceQueue.P();           // wait in line to be serviced
  resource.P();               // request exclusive access to resource
  serviceQueue.V();           // let next in line be serviced

<CRITICAL Section>
// writing is performed

<EXIT Section>
  resource.V();               // release resource access for next reader/writer
}
```

This solution can only satisfy the condition that "no thread shall be allowed to starve" if and only if semaphores preserve first-in first-out ordering when blocking and releasing threads. Otherwise, a blocked writer, for example, may remain blocked indefinitely with a cycle of other writers decrementing the semaphore before it can.

Simplest reader writer problem

The simplest reader writer problem which uses only two semaphores and doesn't need an array of readers to read the data in buffer.

Please notice that this solution gets simpler than the general case because it is made equivalent to the Bounded buffer problem, and therefore only *N* readers are allowed to enter in parallel, *N* being the size of the buffer.

Reader

```
do {
    wait(read)
    ............
    reading data
    ............
    signal(write)
} while (TRUE);
```

Writer

```
do {
    wait(write)
    ............
    writing data
    ............
    signal(read)
} while (TRUE);
```

Algorithm

1. Reader will run after Writer because of read semaphore.
2. Writer will stop writing when the write semaphore has reached 0.
3. Reader will stop reading when the read semaphore has reached 0.

In writer, the value of write semaphore is given to read semaphore and in reader, the value of read is given to write on completion of the loop.

See also

- ABA problem
- Producers-consumers problem
- Dining philosophers problem
- Cigarette smokers problem
- Sleeping barber problem
- Readers–writer lock
- seqlock
- read-copy-update

References

1. Tanenbaum, Andrew S. (2006), *Operating Systems - Design and Implementation, 3rd edition [Chapter: 2.3.2 The Readers and Writers Problem]*, Pearson Education, Inc.
2. Courtois, P. J.; Heymans, F.; Parnas, D. L. (1971). "Concurrent Control with "Readers" and "Writers" " (http://cs.nyu.edu/~lerner/spring10/MCP-S10-Read04-ReadersWriters.pdf) (PDF). *Communications of the ACM*. **14** (10): 667–668. doi:10.1145/362759.362813 (https://doi.org/10.1145%2F362759.362813). S2CID 7540747 (https://api.semanticscholar.org/CorpusID:7540747).
3. Taubenfeld, Gadi (2006). *Synchronization Algorithms and Concurrent Programming*. Pearson Education. p. 301.

- Morris JM (1979). A starvation-free solution to the mutual exclusion problem. Inf Process Lett 8:76–80
- Fair Solution to the Reader-Writer-Problem with Semaphores only. H. Ballhausen, 2003 arXiv:cs/0303005 (https://arxiv.org/abs/cs/0303005)
- Faster Fair Solution for the Reader–Writer Problem. V. Popov, O. Mazonka 2013 arXiv:1309.4507 (https://arxiv.org/abs/1309.4507)

External links

- Algorithmic description of the third readers–writers problem (http://www.rfc1149.net/blog/2011/01/07/the-third-readers-writers-problem/)