Reading Assignment 5 - Starve Free Readers Writers Problem

Riteesh Reddy Sykam
Enroll. No. 19114084

March 15, 2021

Abstract

In computer science, the readers–writers problems are examples of a common computing problem in concurrency. 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 a 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. In this report I shall be discussing about the various types of readers-writers problems and the pseudo codes of solving them.

Contents of the report:

---

1 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 R1 might have the lock, and then another reader R2 requests access. It would be foolish for R2 to wait until R1 was done before starting its own read operation; instead, R2 should be allowed to read the resource alongside R1 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.

```
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 are no longer needing 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.

---

2 Second readers–writers problem

The first solution is suboptimal, because it is possible that a reader R1 might have the lock, a writer W be waiting for the lock, and then a reader R2 requests access. It would be unfair for R2 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:

```
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.

---

3 Starve-free 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 readc=0;  //counts the number of readers accessing the resource at current time.
semaphore resource=1;  // controls access to the resource
semaphore rmutex=1;
semaphore requestqueue=1;   //Keeps order of requests in FIFO mode

wait(Semaphore S)
{
   while S<=0
     ; //no operation   //if S<=0 block the process i.e, wait on S
   S--;
}

signal(S)
{
   S++;   //Start one of the waiting process on S
}

reader() {
<ENTRY Section>
  wait(requestqueue);  //wait in the request order to be serviced
  wait(rmutex);        // request access to readc
  readc++;         //increase the reader count by 1
  if (readc==1){
    wait(resource);   //if he is the first reader request resource access for readers.
}
  signal(requestqueue);    // let next in order be serviced
  signal(rmutex);       // release access to readc

<CRITICAL Section>
//reading is performed

<EXIT Section>
  wait(rmutex);         // request access to readc
  readc--;
  if (readc==0){
    signal(resource);   //if he is the last reader release resourse access
}
  signal(rmutex);       // release access to readcount
}
```

```
writer() {
<ENTRY Section>
  wait(requestqueue);  // wait in the request order to be serviced
  wait(resource);      // request access to resource
  signal(requestqueue);          // let next in line be serviced

<CRITICAL Section>
//writing is performed

<EXIT Section>
  signal(resource);    // release resource access for next request
}
```

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.

We will build the solution using C code. We will use semaphores for mutual exclusion (mutex). Those semaphores, being used as locks, are all initialized in the released state (1 available place); those initializations will not appear in code snippets below but will be shown in comments.

First of all, we said earlier that we want fair-queuing between readers and writers in order to prevent starvation. To achieve that, we will use a semaphore named orderMutex that will materialize the order of arrival. This semaphore will be taken by any entity that requests access to the resource, and released as soon as this entity gains access to the resource:

```
semaphore orderMutex;      // Initialized to 1

void reader()
{
  P(orderMutex);           // Remember our order of arrival
  ...
  V(orderMutex);           // Released when the reader can access the resource
  ...
}

void writer()
{
  P(orderMutex);           // Remember our order of arrival
  ...
  V(orderMutex);           // Released when the writer can access the resource
}
```

Now, we can write the writer code as it is the most straightforward of both. The writer wants an exclusive access to the resource. We will create a new semaphore named accessMutex that the writer will request before modifying the resource:

```
semaphore accessMutex;     // Initialized to 1
semaphore orderMutex;      // Initialized to 1

void reader()
{
  P(orderMutex);           // Remember our order of arrival
  ...
  V(orderMutex);           // Released when the reader can access the resource
  ...
}

void writer()
{
  P(orderMutex);           // Remember our order of arrival
  P(accessMutex);          // Request exclusive access to the resource
  V(orderMutex);           // Release order of arrival semaphore (we have been served)

  WriteResource();         // Here the writer can modify the resource at will

  V(accessMutex);          // Release exclusive access to the resource
}
```

The reader code is a bit more complicated as multiple readers can simultaneously access the resource. We want the first reader to get access to the resource to lock it so that no writer can access it at the same time. Similarly, when a reader is done with the resource, it needs to release the lock on the resource if there are no more readers currently accessing it.

This is similar to a light switch in a dark room: the first person entering the room will turn the lights on, while the last person leaving the room will turn the lights off. However, it is much more simple if the room has only one door and only one person can enter or leave the room at the same time, to prevent someone from switching the lights off when someone enters at the same time using another door. This is why we will use a counter named readers representing the number of readers currently accessing the resource, as well as a semaphore named readersMutex to protect the counter against conflicting accesses.

```
semaphore accessMutex;     // Initialized to 1
semaphore readersMutex;    // Initialized to 1
semaphore orderMutex;      // Initialized to 1

unsigned int readers = 0;  // Number of readers accessing the resource

void reader()
{
  P(orderMutex);           // Remember our order of arrival

  P(readersMutex);         // We will manipulate the readers counter
  if (readers == 0)        // If there are currently no readers (we came first)...
    P(accessMutex);        // ...requests exclusive access to the resource for readers
  readers++;               // Note that there is now one more reader
  V(orderMutex);           // Release order of arrival semaphore (we have been served)
  V(readersMutex);          // We are done accessing the number of readers for now

  ReadResource();           // Here the reader can read the resource at will

  P(readersMutex);          // We will manipulate the readers counter
  readers--;                // We are leaving, there is one less reader
  if (readers == 0)         // If there are no more readers currently reading...
    V(accessMutex);         // ...release exclusive access to the resource
  V(readersMutex);          // We are done accessing the number of readers for now
}

void writer()
{
  P(orderMutex);            // Remember our order of arrival
  P(accessMutex);           // Request exclusive access to the resource
  V(orderMutex);            // Release order of arrival semaphore (we have been served)

  WriteResource();          // Here the writer can modify the resource at will

  V(accessMutex);           // Release exclusive access to the resource
}
```

Checking the solution for the required properties
We can look back the this solution and check if it looks like it fits our requirements. It can be formally proved correct using for example Petri nets.

- orderMutex cannot be part of a deadlock since at the time it is requested no other semaphore is ever held by the calling process, so we can forget about it in our deadlock analysis (this property holds if every process only calls reader() or writer() once, but it works in the general case).
- readers is always strictly greater than 0 if any reader is currently executing lines 15 to 20 (inclusive): the variable is unconditionally incremented at line 14 and unconditionally decremented at line 21 without any other modification, and no modification ever occurs without an exclusive lock granted by readersMutex so the readers integrity is guaranteed.
- accessMutex is taken at line 13 just before readers goes from 0 to 1. It is released at line 23 just after readers goes back to 0. Detection of those raising or falling edges is guaranteed by the use of the readersMutex lock. As seen above, it means that no reader will be executing lines 14 to 22 (inclusive) without accessMutex being taken by a reader (possibly gone now), especially line 18 which represents resource access.
- accessMutex can be requested when readersMutex is held (on line 13), and readersMutex can be requested when accessMutex is held (on line 20). However, the first situation (line 13) only happens when readers is equal to 0 (first reader to get access to the resource), which can never be the case if another reader is currently trying to execute line 20 as seen above, so those two potentially deadlocking reservations can never occur at the same time. Thus those two semaphores cannot interact and cause a deadlock.

- The resource is never accessed by a writer without accessMutex being taken in an unconditional and exclusive way. Since we have seen above that accessMutex is always collectively taken by readers before they access the resource, the resource is properly protected.
- Fair access is guaranteed through the orderMutex which is taken upon arrival and released only when access to the resource has been granted.
