

**Operating Systems**

**Formal Element: Reader-Writer Problem**

Lecturer: Dr Raymond Lynch

Student Name: Jack Harding

Student Number: C15441798

Date: 20 December 2018

**Contents**

[Introduction 3](#_Toc533072700)

[Reader’s Preference 4](#_Toc533072701)

[Reader 4](#_Toc533072702)

[Writer 7](#_Toc533072703)

[Results 8](#_Toc533072704)

[Writers Preference 9](#_Toc533072705)

[Reader 9](#_Toc533072706)

[Writer 10](#_Toc533072707)

[Results 11](#_Toc533072708)

[Conclusion 12](#_Toc533072709)

[References 12](#_Toc533072710)

[Figure 1 Basic Lock-Unlock Cycle of Thread 3](#_Toc533072711)

[Figure 2 Reader’s Preference Reader Flowchart 5](#_Toc533072712)

[Figure 3 Reader’s Preference Writer Flowchart 7](#_Toc533072713)

[Figure 4 Console Output for Readers 8](#_Toc533072714)

[Figure 5 Console Output for Writers 8](#_Toc533072715)

[Figure 6 Writer’s Preference Reader Flowchart 9](#_Toc533072716)

[Figure 7 Writer’s Preference Writer Flowchart 10](#_Toc533072717)

[Figure 8 Console Output for Readers 11](#_Toc533072718)

[Figure 9 Console Output for Writers 12](#_Toc533072719)

# Introduction

One of the many issues that arise within operating systems are resource deadlocks and starvation. This occurs when a resource, a file in this case, is trying to be accessed by two different threads and one must wait on the other to finish before using it. This approach is suboptimal as it makes little sense to disallow a reader from reading when another is, as it makes no change to the file. The next best option is the reader’s preference.

The reader’s preference presents the solution for the previous approach that disallows two threads to access a resource regardless of their intention, this is done using a mutex (binary semaphore) this prevents one thread from accessing a resource while another is using it. This solution works fine if the threads are writing but is limited in that if two readers are accessing the resource (text file), one is locked out. The first reader-writer problem attempts to solve this.

The solution for above leaves another problem and that is with the order of access for the writers. Suppose a reader has access to a resource and a writer is in the queue for writing to it, according to the readers preference, another reader gets priority over the writer, allowing the second reader to jump ahead of the writer and lock the resource. The next reader-writer solution is to ensure that no writer is made wait longer than necessary. This is solved using a reader-try mutex, when a reader attempts to read the file it must lock and release this mutex. [4]

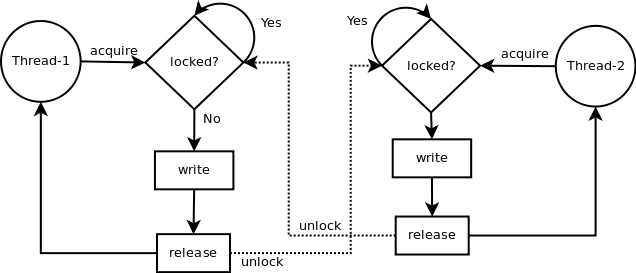


Figure 1 Basic Lock-Unlock Cycle of Thread

The threads are prevented from accessing the resource by using System V semaphores which are used in the Linux and have the necessary functions to use initialise and perform operations on the semaphores. To gain access to a file, a lock must be made on that resource mutex, to release the resource to another reader/writer, an unlock must be made. The diagram in Figure 1 shows an overview of a thread gaining access to a file and releasing it for another to read. [2]

# Reader’s Preference

## Reader

The includes below allows a programmer to access the required IPC libraries to implement semaphores and shared memory. The *fstream* library allows the text file to be loaded in, edited and outputted to the Terminal.

#include <fstream>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <sys/shm.h>

using namespace std;

#define SHM\_KEY 9876 // the shared process id

#define SEMKEY 1234

struct sembuf vsembufR, psembufR, vsembufF, psembufF; // Reader Mutex and Resource Mutex

Semaphores are to be shared by processes therefore there must be a way of accessing the same semaphore, this is done by using a semaphore key, as defined above with *SEMKEY*. The value can be any integer provided it’s matched in the other files. The *sembuf* structures on the final line above are required for the basic operations of the semaphores, each of these structures contain *sem\_num* (which semaphore to use), *sem\_op* (operation-down/up), and *sem\_flag* (don’t wait or restore). [3] [4]

union semun { // like struct but all members share the same location

int val;

struct semid\_ds \*buf;

ushort myArray[0];

} arg;

string op;

ifstream myFile; // makes an ifstream object to read from myFile

int shmid = shmget(SHM\_KEY, 256, 0777|IPC\_CREAT);// 0777 is the permission, IPC\_CREAT creates a shared mem block

int semid = semget(SEMKEY, 2, 0777|IPC\_CREAT); // Creates two semaphores

int \*readerCount = (int\*)shmat(shmid, 0, 0); // starting addr of 1, 0 is full R/W

int pause; // used as later for pausing program

\*readerCount = 0;

The *union semun* is required when calling upon semaphores, as it supports the initialisation. The value of the semun union is set using *int val*, the structure of *semid\_bs* is the buffer for *IPC\_STAT* and *IPC\_SET*, used for copying and writing from the kernel, the *unsigned short* array is used for *GETALL* and *SETALL*, which return values of all semaphores and edit all values for semaphores. The string variable is used to move line-by-line through the text file, an *ifstream* object is made to access the resource too. The block of shared memory is generated using the *shmget* function, this allocates a block of shared memory with the defined shared memory key with a size of 256 Bytes and with 0777 (read/write/execute) the permission of the block, the ID of the block is returned and made equal to *shmid*. *semget* creates two semaphores with the *SEMKEY* defined earlier and with the same permissions as the shared block of memory. [3] [1] [4]

The *readerCount* variable must be attached to shared memory so it can be used by all processes using that variable, this is done using *shmat* (attach to shared memory) which is passed the *shmid* (shared memory ID) along with two zeros which indicate that it has full read/write permissions. When calling *shmat* the address of the shared memory segment, this must be type casted to an integer pointer.

psembufR.sem\_num=0; // init reader mutex members

psembufR.sem\_op=-1; // what value to use with semop

psembufR.sem\_flg=SEM\_UNDO; // SEM\_UNDO makes kernel increment for process after decrementing semap

vsembufR.sem\_num=0;

vsembufR.sem\_op=1;

vsembufR.sem\_flg=SEM\_UNDO;

psembufF.sem\_num=1; // resource

psembufF.sem\_op=-1;

psembufF.sem\_flg=SEM\_UNDO;

vsembufF.sem\_num=1;

vsembufF.sem\_op=1;

vsembufF.sem\_flg=SEM\_UNDO;

arg.val = 1; // sets to binary semap

semctl(semid, 0, SETVAL, arg); // initialises semaphore 0. Do this once only

semctl(semid, 1, SETVAL, arg);

The code above sets up the semaphore operations needed to lock and unlock, to define a wait or lock, the *sem\_num* is set to zero, *sem\_op* is -1, and the flag is *SEM\_UNDO*, *SEM\_UNDO* specifies if the process should end before restoring the semaphore count, then it is restored instead by the operating system. The first setup block is used to protect the *readerCount* variable, helping determine who is the first/last reader. The same setup is used for the resource mutex, but this mutex is used to protect the text file that is being accessed. The *arg.val* is set to 1 to initialise the semaphore, the same goes for *semctl* lines, which pass the function the *semun union*, *arg*.

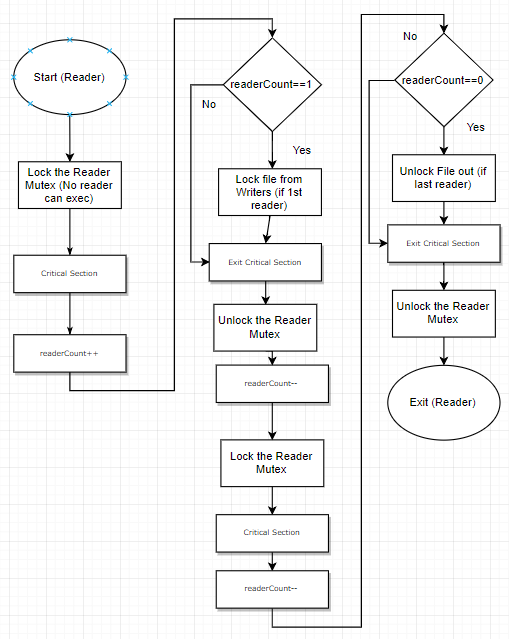


Figure 2 Reader’s Preference Reader Flowchart

The flowchart in Figure 2 shows the basic steps towards checking the *readerCount*, accessing the resource and finally releasing both the semaphores.

while(1){

cout << "Reader1:\n";

pause = getchar(); // pause prevents error

semop(semid, &psembufR, 1); // lock reader mutex

\*readerCount++;

if(\*readerCount == 1){ // is this first reader

cout << "Locking reader from writers\n";

semop(semid, &psembufF, 1); // lock resource from writers if 1st reader

}

semop(semid, &vsembufR, 1); // unlock reader mutex

// Critical Section

myFile.open ("myFile.txt", ios::out | ios::app); // ::app appends the myFile (new line)

if(myFile.is\_open()){

while(getline(myFile, op)){

cout << op << endl;

}

myFile.close();

}

semop(semid, &psembufR, 1); // lock reader mutex, can use readerCount

\*readerCount--;

if(\*readerCount == 0) { // is this the last reader

semop(semid, &vsembufF, 1); // unlock resource mutex for writers

cout << "Locking reader mutex for readers\n";

}

semop(semid, &vsembufR, 1); // unlock reader mutex

semop(semid, &vsembufF, 1); // unlock resource mutex

}

The infinite *while* loop above is where the actual semaphore operations are carried out. To start the process is printed to the console, this is followed by a delay to prevent errors. To access the *readerCount* variable it must first be locked using the *semop* function which is passed the *semid* (must match the one used earlier), reference to the *psembufR* (reader sembuf structure declared), and *nsops* which is the number of sumbuf structures in the array (one in this case). *readerCount* is incremented, the following *if* statement checks “is this the first reader” (*readerCount* is initialised to 0) if so, the resource is locked using the *semop* function. The reader mutex is then unlocked and the critical section is entered.

First, the file is opened using *ios∷out* and *app*, this appends all input operations to the end of the file, out specifies that the file is open for output. Then a check is made to see if the file is open (should be if first reader), a while loop then cycles through the entire file and prints all lines out, the file is then closed.

To determine if the current process is the last reader, the *readerCount* is decremented and an *if* statement checks if equal to zero, if so, the resource is unlocked, as is the reader. The issue with this solution is that all readers must be done with the resource for any writers to use it, this can cause starvation and significantly slow the system.

## Writer

The writer follows the same setup and initialisation as the reader but with an ofstream object for input.

string ip;

ofstream myFile; // makes an ofstream object to write for myFile

The flowchart in Figure 3 shows the writer code for the Reader’s Preference. There has very little intelligence to the code with the writer writing without any conditions, in an infinite loop.

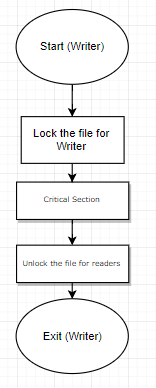


Figure 3 Reader’s Preference Writer Flowchart

The implementation of the flowchart is shown below. The resource is locked so as to access the file, the file is opened, the user is prompted to enter a string value to be inserted into the file, after inputting, the file is closed, and the resource released.

while(1){

cout << "\nWriter1: \n";

semop(semid, &psembufF, 1); // lock file for writer

myFile.open ("myFile.txt", ios::out | ios::app); // ::app appends the myFile (new line)

if(myFile.is\_open()){

cout << "Please enter something\n";

cin >> ip;

myFile << ip << endl;

myFile.close();

}

semop(semid, &vsembufF, 1); // unlock file for writers if no readers request

}

## Results

Figure 4 shows the output of the reader processes; the readers have no issues in the code above as they’re given priority. Reader1 and Reader2 have identical code, both ran in different windows, the contents of the text file is printed to the console.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Figure 4 Console Output for Readers

As before the Writer1 and Writer2 have identical code. Figure 5 shows the writers’ attempts at writing to the file. The user input is appended to the file as mentioned before.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Figure 5 Console Output for Writers

The console above shows as many readers as possible reading from the resource, as they are given preference over the writers.

# Writers Preference

The writer’s preference attempts to improve where the reader’s preference fell short with the potential starvation of writers. This solution will require two additional semaphores, one to indicate that the reader is trying to read and another for the *writerCount*, which like the *readerCount* earlier, in a shared memory block. The try semaphore indicates the reader is trying to read and helps give the writer priority over the queue of readers.

## Reader

The initialisation of the reader in the reader’s preference is very similar with the only difference being the number of semaphores, additional setup is shown below.

struct sembuf vsembufR, psembufR, vsembufW, psembufW; // reader and writer semaphores

struct sembuf vsembufF, psembufF, vsembufT, psembufT; // resource (file) and read try semaphores

psembufW.sem\_num=0; // writer

psembufW.sem\_op=-1;

psembufW.sem\_flg=SEM\_UNDO;

vsembufW.sem\_num=0;

vsembufW.sem\_op=1;

vsembufW.sem\_flg=SEM\_UNDO;

psembufT.sem\_num=1; // try

psembufT.sem\_op=-1;

psembufT.sem\_flg=SEM\_UNDO;

vsembufT.sem\_num=1;

vsembufT.sem\_op=1;

vsembufT.sem\_flg=SEM\_UNDO;

In Figure 6 the flowchart for the reader is shown, as mentioned, the try mutex is used and must be locked to indicate desire to enter, this is followed by the reader being locked as to allow for use of the *readerCount* variable. The series of events is the same as the previous reader, when the protected variable needs to be used (in comparison or increment/decrement) it is locked and unlocked when not needed. The main distinction lies in the try semaphore which is locked before accessing the resource, then unlocked after the *if* statement, this indicates to the writer that the readers are trying to access the resource.

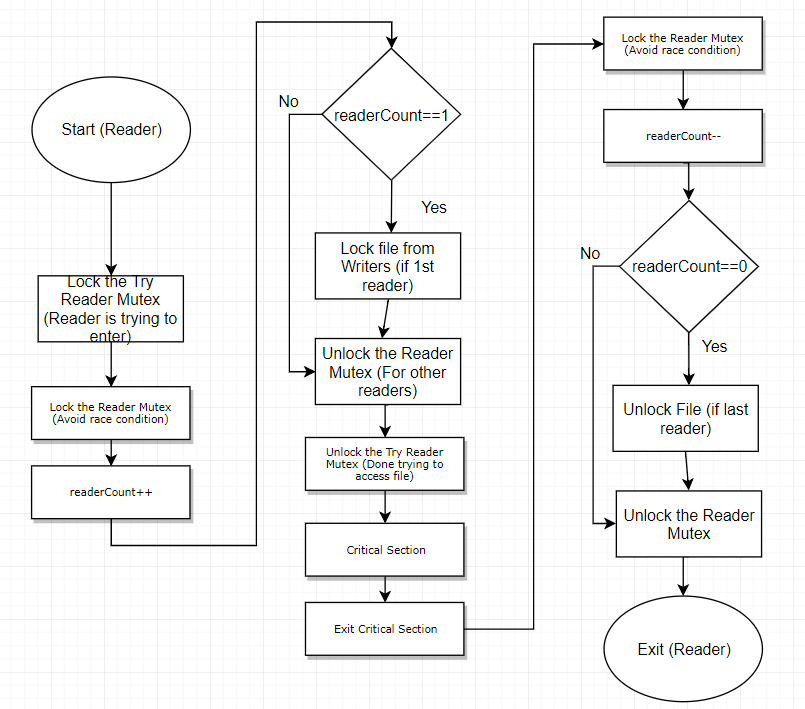


Figure 6 Writer’s Preference Reader Flowchart

Below is the implementation of the flowchart in Figure 6.

while(1){

cout << "\nReader1: \n";

pause = getchar(); // prevent errors

semop(semid, &psembufT, 1); // lock reader try mutex (trying to enter)

semop(semid, &psembufR, 1); // lock reader mutex (avoid race condition)

\*readerCount++;

if(\*readerCount == 1) // is this first reader

semop(semid, &psembufF, 1); // lock resource from writers if 1st reader

semop(semid, &vsembufR, 1); // unlock reader mutex (for other readers)

semop(semid, &vsembufT, 1); // unlock try mutex (done accessing file)

// Critical Section

myFile.open ("myFile.txt", ios::out | ios::app); // ::app appends the myFile (new line)

if(myFile.is\_open()){

while(getline(myFile, op)){

cout << op; // reads

}

myFile.close();

}

semop(semid, &psembufR, 1); // lock reader mutex (avoid race)

\*readerCount--;

if(\*readerCount == 0) // is this the last reader

semop(semid, &vsembufF, 1); // unlock resource

semop(semid, &vsembufR, 1); // unlock reader mutex

}

## Writer

The code below shows the difference between the reader and writer initialisation code, instead the *writerCount* is in the shared block of memory so that the other writer may access it.

int \*writerCount = (int\*)shmat(shmid, 0, 0); // starting addr of 1, 0 is full R/W

\*writerCount = 0;

Figure 7 is the flowchart for the writer implementation, it is far more complex than the previous one which only locked and released the semaphore when it needed to write.

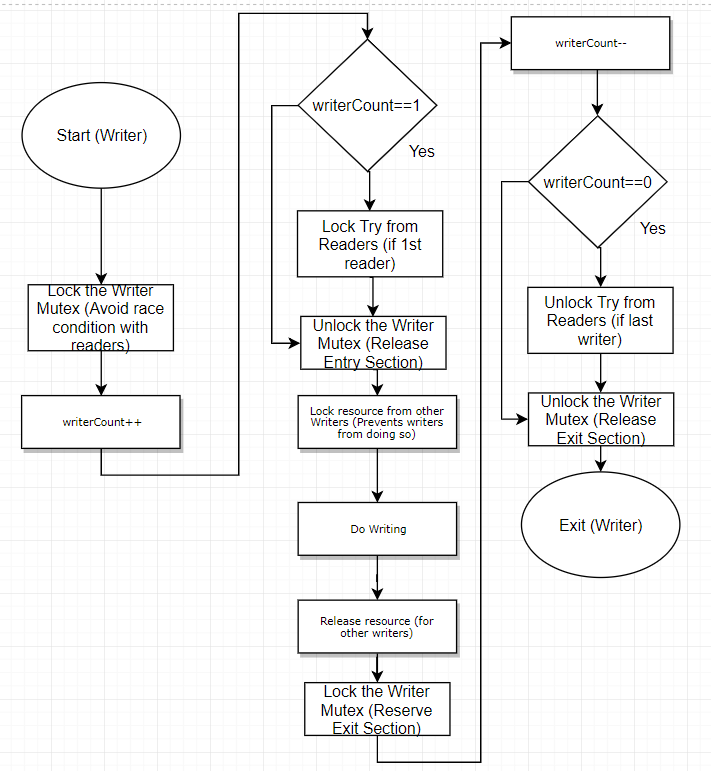


Figure 7 Writer’s Preference Writer Flowchart

In the code, three semaphores are used to handle the resource, reader try and the writer semaphores. To access the *writerCount* variable, the writer must first be locked and then counter incremented to tell the program the number of writers has increased. If this incrementation results in being a count of 1, the try mutex is locked from the readers and the writer is unlocked. To write to the file the resource is locked, written to and released. To check if this is the last writer, the writer is locked, decremented and if *writerCount* is zero (last writer) the try mutex is unlocked as to allow the readers to access the file.

while(1){

cout << "Writer1:\n";

semop(semid, &psembufW, 1); // lock writer to prevent race conditions

\*writerCount++;

if(\*writerCount == 1){ // if 1st writer

semop(semid, &psembufT, 1); // lock the try semap (from 1st reader)

}

semop(semid, &vsembufW, 1); // unlock writer to release entry section

semop(semid, &psembufF, 1); // lock resource from other writers

// Critical Section

myFile.open ("myFile.txt", ios::out | ios::app);// ::app appends the myFile (new line)

if(myFile.is\_open()){

cout << "Please enter something\n";

cin >> ip;

myFile << ip << endl;

myFile.close();

}

semop(semid, &vsembufF, 1); // unlock file for other writers

semop(semid, &psembufW, 1); // lock writer to reserve exit section

writerCount--;

if(writerCount == 0){ // if last writer

semop(semid, &vsembufT, 1); // unlock the try semap, if last writer

}

semop(semid, &vsembufW, 1); // unlock writer mutex, release exit section

}

The code above works well to prevent the starvation of writers but can result in the starvation of readers as result, an improvement to the above could be made by making it so no reader/writer should be made wait longer than a certain time limit.

## Results

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Figure 8 Console Output for Readers

Reader1 and Reader2 in Figure 8 have the same code but ran in different windows, as in the reader’s preference. This version does not allow the readers free reign like before and requires the writer to release the try semaphore to allow the readers to lock the resource.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Figure 9 Console Output for Writers

Figure 9 is where the writers have preference and can write to the file until the writer count is zero, the last writer. This allows the writers to starve reader until the writers are finished.

# Conclusion

The semaphores were setup, the shared block of memory was created to protect the reader/writer counter, and the semaphores were initialised to know what value to change to when a *semop* is called. After this the reader’s preference was implemented it was shown in console windows in the solution results in the readers locking the resource until all readers are finished, causing the writers to starve.

The solution to this is to use a reader try semaphore, this indicates to the writer that a reader is trying to read. The increased intelligence allows the writer to only release the resource when the last writer is finished writing, of course this causes another with reader starvation. This allowed writers to complete all tasks required and then hand priority to the readers, this in turn will cause reader starvation.

The optimal solution would be a weighted system, where waiting time could be taken into account and have influence on the conditions within the code above.

# References

[1]CyberiaPC, "Blocking: Semaphores", CyberiaPC, N/A. [Online]. Available: <http://www.cyberiapc.com/os/blocking-semaphores.ht>m [Accessed: 10- Dec- 2018].

[2] L. Luce, "[Python threads synchronization: Locks, RLocks, Semaphores, Conditions, Events and Queues](https://www.laurentluce.com/posts/python-threads-synchronization-locks-rlocks-semaphores-conditions-events-and-queues/)", *LaurentLuce*, 2011. [Online]. Available: <https://www.laurentluce.com/posts/python-threads-synchronization-locks-rlocks-semaphores-conditions-events-and-queues/>. [Accessed: 15- Dec- 2018].

[3] M. Kerrisk, "semctl man page", *man7.org*, 2018. [Online]. Available: <http://man7.org/linux/man-pages/man2/semctl.2.html> . [Accessed: 16- Dec- 2018].

[4]"Synchronization: Semaphores", *UCHI*, 2018. [Online]. Available: <https://www.classes.cs.uchicago.edu/archive/2017/winter/51081-1/LabFAQ/lab7/Semaphores.html> [Accessed: 16- Dec- 2018].