Operating Systems

Formal Element: Reader-Writer Problem

Lecturer: Dr Raymond Lynch

Student Name: Jack Harding

Student Number: C15441798

Date: 19 December 2018

Contents

Introduction 3
Reader’s Preference 4
Reader 4
Writer 7
Results 8
Writers Preference 9
Reader 9
Writer 10
Results 11
Conclusion 12
References 12

Figure 1 Basic Lock-Unlock Cycle of Thread 3
Figure 2 Reader’s Preference Reader Flowchart 5
Figure 3 Reader’s Preference Writer Flowchart 7
Figure 4 Console Output for Readers 8
Figure 5 Console Output for Writers 8
Figure 6 Writer’s Preference Reader Flowchart 9
Figure 7 Writer’s Preference Writer Flowchart 10
Figure 8 Console Output for Readers 11
Figure 9 Console Output for Writers 12

Introduction

The readers preference describes the solution 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, reading having no effect on the contents of the file should not be blocked. 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.

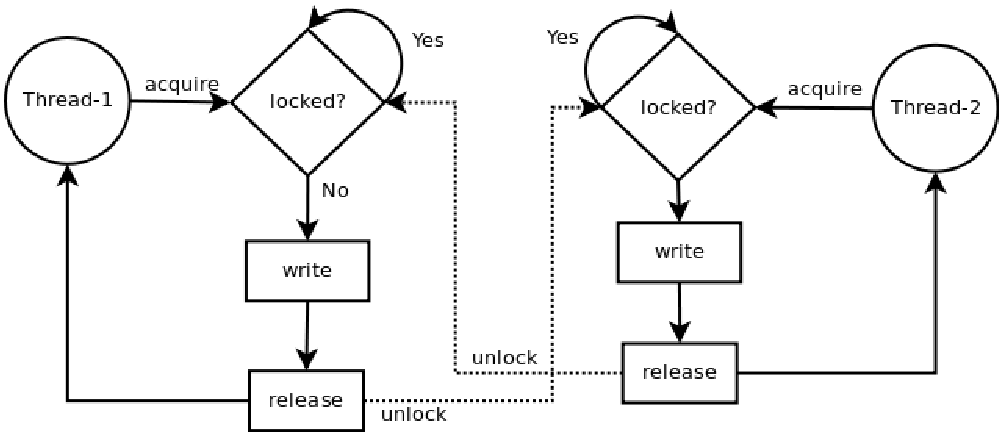


Figure 1 Basic Lock-Unlock Cycle of Thread

The above solutions are implemented using semaphores in the Linux environment, 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.

Reader's Preference

Reader

The includes below allows a programmer to access the required IPC libraries to implement semaphores and shred 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 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).

```
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 integer 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 shred memory with the defined shared memory key with a size of 256 Bytes and with 0777 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.

The readerCount variable must be attached to shared memory so it can be used by all processes using that variable, this us done using shmat which is passed the shmid 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.

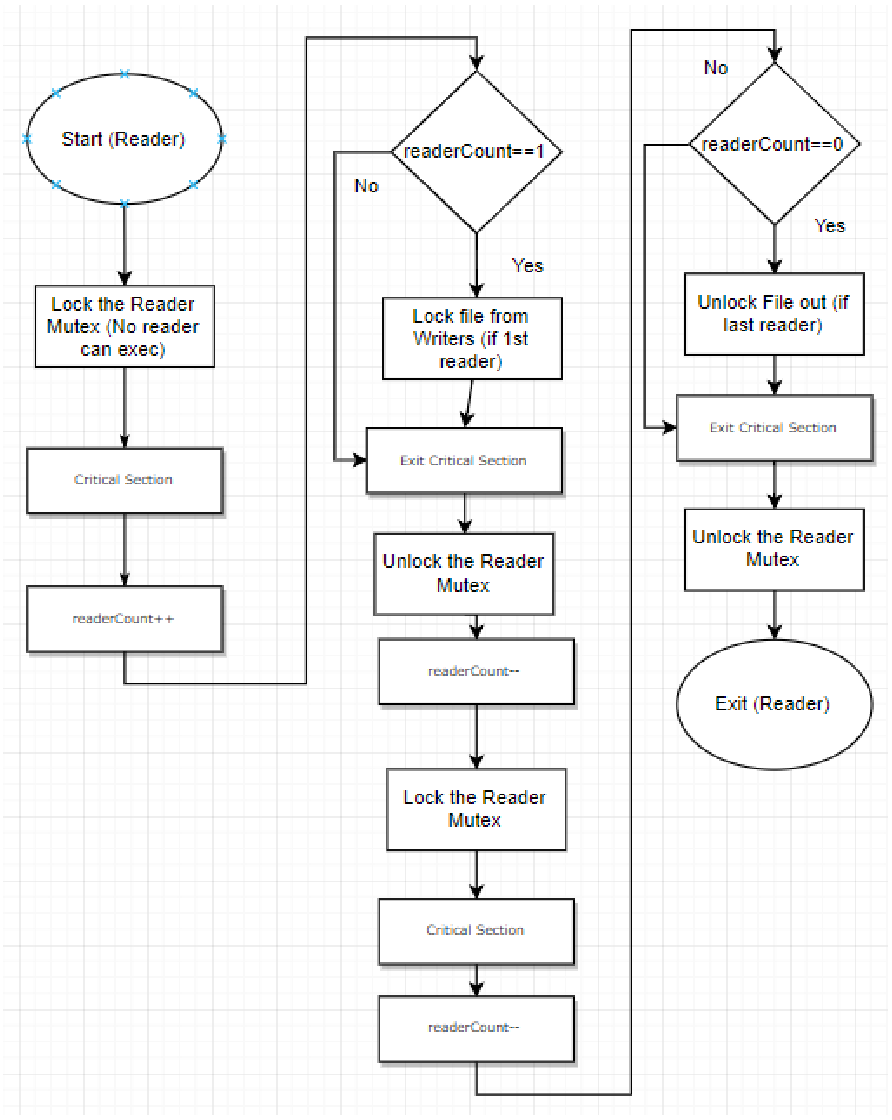


Figure 2 Reader's Preference Reader Flowchart

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 and 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 read for myFile
```

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

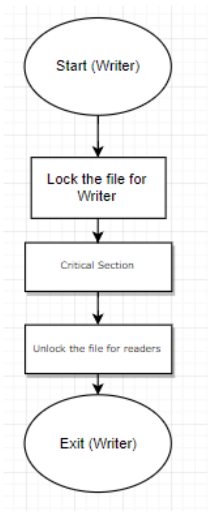


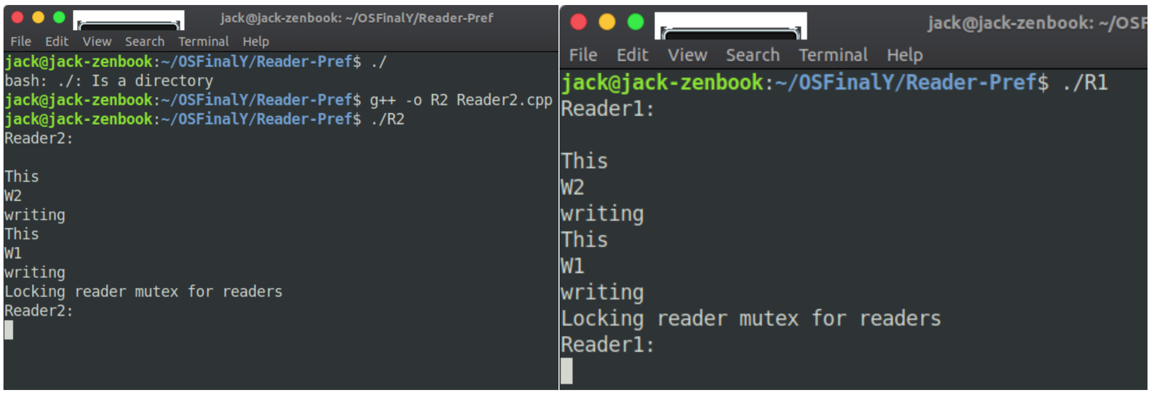
Figure 3 Reader's Preference Writer Flowchart

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.



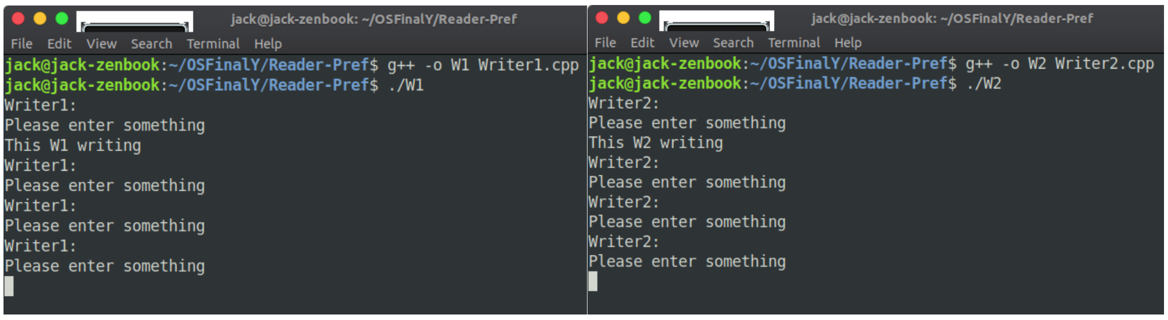
jack@jack-zenbook: ~/OSFinalY/Reader-Pref
File Edit View Search Terminal Help
jack@jack-zenbook: ~/OSF

```
jack@jack-zenbook:~/OSFinalY/Reader-Pref$ ./
bash: ./: Is a directory
jack@jack-zenbook:~/OSFinalY/Reader-Pref$ g++ -o R2 Reader2.cpp
jack@jack-zenbook:~/OSFinalY/Reader-Pref$ ./R2
Reader2:

This
W2
writing
This
W1
writing
Locking reader mutex for readers
Reader2:
```

```
jack@jack-zenbook:~/OSFinalY/Reader-Pref$ ./R1
Reader1:

This
W2
writing
This
W1
writing
Locking reader mutex for readers
Reader1:
```

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.



jack@jack-zenbook: ~/OSFinalY/Reader-Pref
File Edit View Search Terminal Help

```
jack@jack-zenbook:~/OSFinalY/Reader-Pref$ g++ -o W1 Writer1.cpp
jack@jack-zenbook:~/OSFinalY/Reader-Pref$ ./W1
Writer1:
Please enter something
This W1 writing
Writer1:
Please enter something
Writer1:
Please enter something
Writer1:
Please enter something
```

```
jack@jack-zenbook:~/OSFinalY/Reader-Pref$ g++ -o W2 Writer2.cpp
jack@jack-zenbook:~/OSFinalY/Reader-Pref$ ./W2
Writer2:
Please enter something
This W2 writing
Writer2:
Please enter something
Writer2:
Please enter something
Writer2:
Please enter something
```

Figure 5 Console Output for 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, is 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.

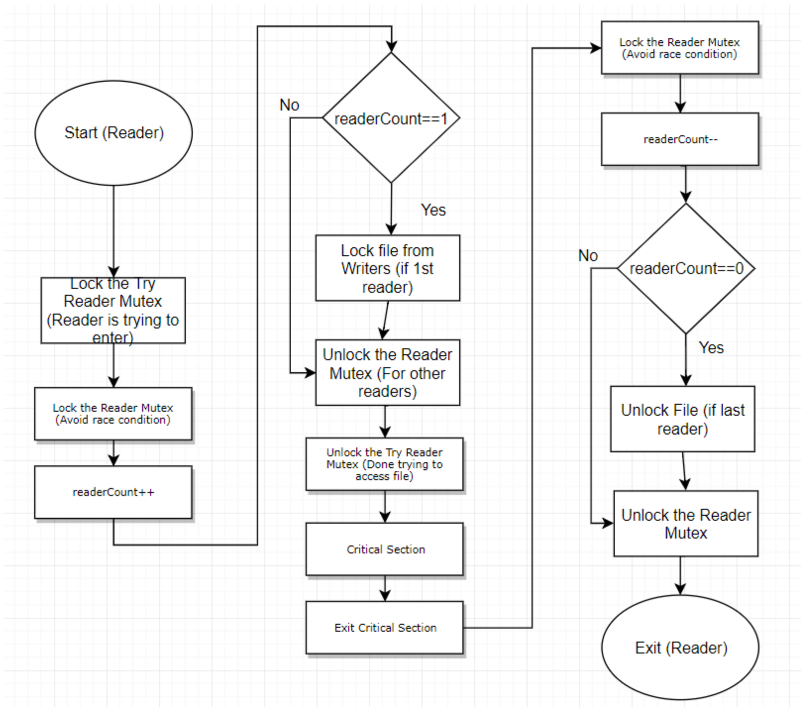

Start (Reader)
Lock the Try Reader Mutex (Reader is trying to enter)
Lock the Reader Mutex (Avoid race condition)
readerCount++
No
readerCount==1
Yes
Lock file from Writers (if 1st reader)
Unlock the Reader Mutex (For other readers)
Unlock the Try Reader Mutex (Done trying to access file)
Critical Section
Exit Critical Section
Lock the Reader Mutex (Avoid race condition)
readerCount--
No
readerCount==0
Yes
Unlock File (if last reader)
Unlock the Reader Mutex
Exit (Reader)

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 ariter 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.

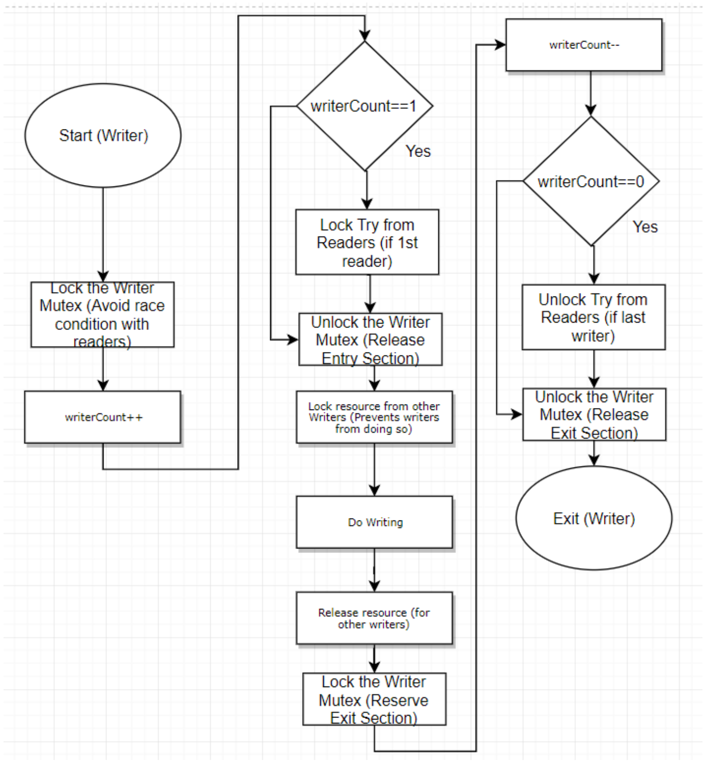

Start (Writer)
Lock the Writer Mutex (Avoid race condition with readers)
writerCount++
writerCount==1
Yes
Lock Try from Readers (if 1st reader)
Unlock the Writer Mutex (Release Entry Section)
Lock resource from other Writers (Prevents writers from doing so)
Do Writing
Release resource (for other writers)
Lock the Writer Mutex (Reserve Exit Section)
writerCount--
writerCount==0
Yes
Unlock Try from Readers (if last writer)
Unlock the Writer Mutex (Release Exit Section)
Exit (Writer)

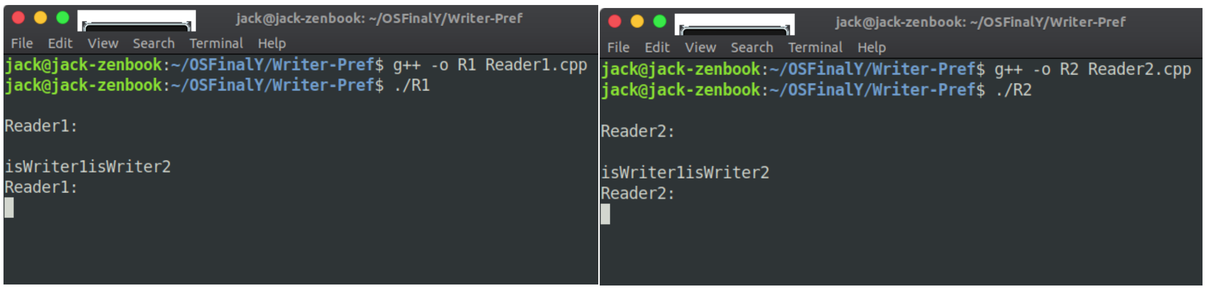
Figure 7 Writer's Preference Writer Flowchart

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 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

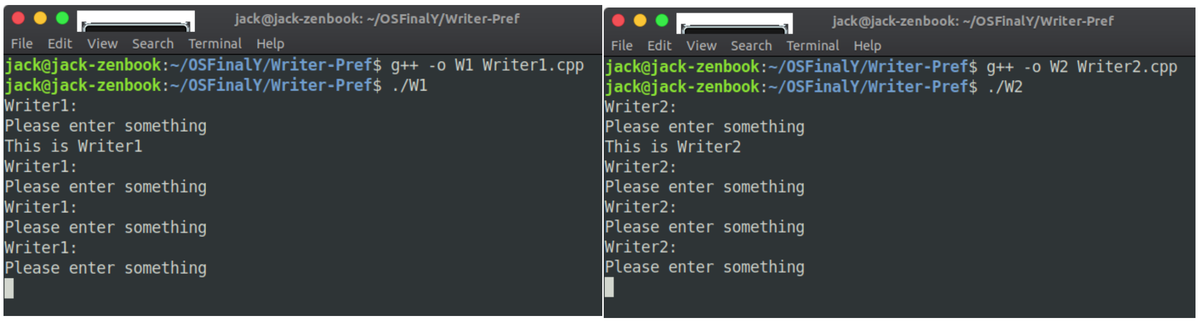

jack@jack-zenbook: ~/OSFinalY/Writer-Pref
File Edit View Search Terminal Help
jack@jack-zenbook: ~/OSFinalY/Writer-Pref
File Edit View Search Terminal Help

```
jack@jack-zenbook:~/OSFinalY/Writer-Pref$ g++ -o R1 Reader1.cpp
jack@jack-zenbook:~/OSFinalY/Writer-Pref$ ./R1

Reader1:

isWriter1isWriter2
Reader1:
```

```
jack@jack-zenbook:~/OSFinalY/Writer-Pref$ g++ -o R2 Reader2.cpp
jack@jack-zenbook:~/OSFinalY/Writer-Pref$ ./R2

Reader2:

isWriter1isWriter2
Reader2:
```

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.


jack@jack-zenbook: ~/OSFinalY/Writer-Pref
File Edit View Search Terminal Help
jack@jack-zenbook: ~/OSFinalY/Writer-Pref
File Edit View Search Terminal Help

```
jack@jack-zenbook:~/OSFinalY/Writer-Pref$ g++ -o W1 Writer1.cpp
jack@jack-zenbook:~/OSFinalY/Writer-Pref$ ./W1
Writer1:
Please enter something
This is Writer1
Writer1:
Please enter something
Writer1:
Please enter something
Writer1:
Please enter something
```

```
jack@jack-zenbook:~/OSFinalY/Writer-Pref$ g++ -o W2 Writer2.cpp
jack@jack-zenbook:~/OSFinalY/Writer-Pref$ ./W2
Writer2:
Please enter something
This is Writer2
Writer2:
Please enter something
Writer2:
Please enter something
Writer2:
Please enter something
```

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

References

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

[2] L. Luce, "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].