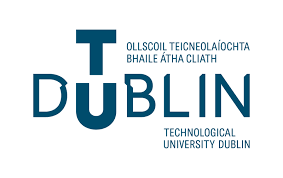
******

**SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING**

**Bachelor of Engineering (Hons) BE in Electrical & Electronic / Comp & Comm Eng**

**Programme Code: (DT021A)**

**<YEAR 4>**

**Name of Module (OPERATING SYSTEMS,** **COMP 3602)**

**Name of Lecturer (Lynch, Raymond)**

Formal Assignment

**Student Name** \_Sajjad Ullah, C17344483\_\_\_\_\_\_\_\_\_\_\_\_\_\_**\_\_\_\_\_\_\_\_\_\_\_**

**Title \_**Readers-Writers Assignment \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_

**DECLARATION**

I hereby certify that the material, which is submitted in this assignment, is entirely my own work and has not been submitted for any academic assessment other than as part fulfilment of the assessment procedures.

Signature of student: ……Sajjad ullah……………….

Date: ……21/12/21…………………………

Contents

[Introduction 4](#_Toc91001885)

[Problem Statement 4](#_Toc91001886)

[Readers-Preference Explained 5](#_Toc91001887)

[Writers-Preference Explained 5](#_Toc91001888)

[Problem Encountered 6](#_Toc91001889)

[Programming Readers-Preference 7](#_Toc91001890)

[Reader Code 7](#_Toc91001891)

[Writer Code 9](#_Toc91001892)

[Flowcharts for reader preference 11](#_Toc91001893)

[Programming Writers-Preference 12](#_Toc91001894)

[Writer Code 12](#_Toc91001895)

[Reader Code 16](#_Toc91001896)

[Flowchart for Writer-Preference 19](#_Toc91001897)

[Testing Readers Preference 20](#_Toc91001898)

[Testing Writers Preference 24](#_Toc91001899)

[Final changes to both solutions 29](#_Toc91001900)

[Conclusion 30](#_Toc91001901)

[Bibliography 31](#_Toc91001902)

# Introduction

This formal assignment is about the readers-writers problem, which is an example of a common concurrency problem that deals with a situation in which many concurrent threads of execution try to access the same shared resource at one time.

# Problem Statement

The readers and writers’ problem is used to model access to a shared object such as a database or file. Several competing processes wish to access it either to read from it (reader) or to write to it (writer). It is acceptable to have multiple processes reading the shared object at the same time, but a writer process must have exclusive access to the shared object. No other processes readers or writers may access the shared object, whilst a writer process is updating it.

**Task:** To design and code programs that will demonstrate 2 solutions to the readers and writers’ problem, readers-preference, and writers-preference.

You will make use of the system calls to create and use **semaphores**.

You will have four programs (and hence four processes) **two Writers** and the other **two Readers**.

**Objectives**

* These programs will be run in separate terminal windows.
* The reader/writer sections will be in an infinite while loop.
* The output from the processes should demonstrate that solution solves the problem.
* The Writer process should **open a text file** and **append** some **text** to the file.
* The text will be some input from the keyboard.
* When the Writer is finished writing to the file the file should be closed.
* The Reader should type out the text in the file line by line, in response to a request from the user.
* When the Reader is finished reading the file should be closed.

# Readers-Preference Explained

It is possible to protect shared data behind a mutual exclusion mutex where no two threads can access the data at the shared time.

However, this solution would be sub-optimal because Reader R1 might have the lock, then another reader R2 would have to wait for R1 to finish.

It would be foolish for R2 to wait for R1 since they both only read and do not change/modify the shared resource thus R2 should be allowed to read the resource alongside R1.

The first solution to the readers-writer problem known as readers-preference makes concurrent reads safe in which *no reader shall be kept waiting if the share is currently opened for reading.*

# Writers-Preference Explained

A problem that can occur in readers preference is where Reader R1 has the lock, a writer W1 is waiting for the lock but then another Reader R2 requests access which is given priority over W1 due to readers preference. If this happens long enough to W1 then it never get a chance to write/ update the shared resource.

The second solution to the readers-writer problem known as writers-preference where writers are given priority to update the resource, *no writer, once added to the queue, shall be kept waiting longer than absolutely necessary.*

# Problem Encountered

When programming commenced for the formal element *semget* was used to create two semaphores at once for the readers preference code as shown below:

#define SEMKEY 151

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

semctl(semid, 0, SETVAL,1); //initialise the semaphores

semctl(semid, 1, SETVAL,1);

struct sembuf Wmutex, Smutex, Wresource , Sresource;

Wmutex.sem\_num = 0;

Wmutex.sem\_op = -1;

Wmutex.sem\_flg = SEM\_UNDO;

Smutex.sem\_num = 0;

Smutex.sem\_op = 1;

Smutex.sem\_flg = SEM\_UNDO;

Wresource.sem\_num = 1;

Wresource.sem\_op = -1;

Wresource.sem\_flg = SEM\_UNDO;

Sresource.sem\_num = 1;

Sresource.sem\_op = 1;

Sresource.sem\_flg = SEM\_UNDO;

However, during execution, both semaphores were not working due to some unknown issue that could not be resolved even with assistance from lecturer during lab.

Therefore, it was decided to create the semaphores independently with their own *semid* and *SEMKEY* as this was working when we use *semget* to **create 1 semaphore at a time**.

This was non-ideal but still served the same purpose during the limited time to complete the assignment.

# Programming Readers-Preference

To start we will need 2 semaphores, one for mutex and one for resource for these we declared 2 SEMKEY and 1 SHM\_KEY for the shared memory of the read count.

All initialisation is done in reader 1 (r1) file.

## Reader Code

#define SEMKEY1 151

#define SEMKEY2 152

#define SHM\_KEY 9876

int main()

{

printf("\n--------main is Starting for writer --------------------");

int shmid,semid1,semid2;

char \*ptr\_read\_count;

/\*semid1 will be the mutex \*/

semid1 = semget(SEMKEY1, 1 , 0777|IPC\_CREAT);

semctl(semid1,0,SETVAL,1);

/\*semid2 will be the resource \*/

semid2 = semget(SEMKEY2, 1 , 0777|IPC\_CREAT);

semctl(semid2,0,SETVAL,1);

shmid = shmget(SHM\_KEY, 256 , 0777|IPC\_CREAT);

Next attach a process to the shared memory as this will be the read counter, use this pointer to initialise the counter to zero.

ptr\_read\_count = (int\*)shmat(shmid, 0, 0);

\*ptr\_read\_count = 0;

Define a number of variables of type *sembuf* and initialise them.

struct sembuf Wmutex, Smutex,Wresource, Sresource;

Wmutex.sem\_num = 0;

Wmutex.sem\_op = -1;

Wmutex.sem\_flg = SEM\_UNDO;

Smutex.sem\_num = 0;

Smutex.sem\_op = 1;

Smutex.sem\_flg = SEM\_UNDO;

Wresource.sem\_num = 0;

Wresource.sem\_op = -1;

Wresource.sem\_flg = SEM\_UNDO;

Sresource.sem\_num = 0;

Sresource.sem\_op = 1;

Sresource.sem\_flg = SEM\_UNDO;

Now start the reader code, first put a wait on the mutex to block other readers:

semop(semid1, &Wmutex,1);

//+++++++critical section 1

Reader will Increment the read count :

\*ptr\_read\_count = \*ptr\_read\_count + 1;

Check if this is the first reader, if so we will block the resource from other writers:

if( \*ptr\_read\_count == 1 ){

semop(semid2, &Wresource,1);

}

//+++++++exit critical section 1

Release the mutex as critical section 1 is exited and we have incremented the counter which tells we are here, now we can perform the reading:

//-------------reading

char input;

cout<<"\n Do u want to read to file [y/n] => ";

cin>> input;

cout<<endl;

if (input == 'y'){

string line;

ifstream myfile ("example.txt");

if(myfile.is\_open()){

cout<<"\nContents of file are below:"<<endl;

while(getline (myfile,line) ) {

cout<<line <<endl;

}

myfile.close();

}

else cout<<"\nUnable to open file"<<endl;

}

//----------------------

After reading complete we need to decrement the counter, so we first need to put a wait on the mutex.

semop(semid1, &Wmutex,1);

//++++++++critical section 2

\*ptr\_read\_count = \*ptr\_read\_count - 1;

Next check if we are the last reader in order to release the resource.

if( \*ptr\_read\_count == 0 ){

semop(semid2, &Sresource,1);

}

//++++++++exit critical section 2

Lastly, we need to release the mutex as we have decremented the counter and finished.

semop(semid1, &Smutex,1);

return 0;

## Writer Code

We must use the same keys for the semaphores, we can use the same code except the initialization lines are removed as this was done in reader 1.

Note since only the resource semaphore is used in writer, we didn’t have to declare any of the other semaphores as they are not used here.

#define SEMKEY1 151

#define SEMKEY2 152

#define SHM\_KEY 9876

int main()

{

printf("\n--------main is Starting for writer --------------------");

int shmid,semid1,semid2;

char \*ptr\_read\_count;

/\*semid1 will be the mutex \*/

semid1 = semget(SEMKEY1, 1 , 0777|IPC\_CREAT);

/\*semid2 will be the resource \*/

semid2 = semget(SEMKEY2, 1 , 0777|IPC\_CREAT);

struct sembuf Wmutex, Smutex,Wresource, Sresource;

Wmutex.sem\_num = 0;

Wmutex.sem\_op = -1;

Wmutex.sem\_flg = SEM\_UNDO;

Smutex.sem\_num = 0;

Smutex.sem\_op = 1;

Smutex.sem\_flg = SEM\_UNDO;

Wresource.sem\_num = 0;

Wresource.sem\_op = -1;

Wresource.sem\_flg = SEM\_UNDO;

Sresource.sem\_num = 0;

Sresource.sem\_op = 1;

Sresource.sem\_flg = SEM\_UNDO;

Now start the writer code, first put a wait on the resource to block other writers on the resource before preforming the writing process :

semop(semid2, &Wresource,1);

//-------------writing is done

char input;

cout<<"\n Do u want to write to file [y/n] => ";

cin>> input;

cout<<endl;

string line;

ofstream myfile ("example.txt", fstream::app);

if(myfile.is\_open() ){

cout<<"\nEnter input to save to file => ";

cin>>line;

myfile <<line <<"\n ";

myfile.close();

}

//----------------------

After completion release the resource by preforming a signal on resource:

semop(semid2, &Sresource,1);

return 0;

## Flowcharts for reader preference

Diagram

Description automatically generatedDiagram

Description automatically generated

# Programming Writers-Preference

To start we will need 4 semaphores, one for *rmutex* (read mutex) , *wmutex* (write mutex) , *readTry* (used by every reader) and *resource*.

We also require 2 shared memory locations for a reader counter and writer counter.

To declared these, we define 4 SEMKEY’s and 2 SHM\_KEY’s for the shared memory’s.

All initialisation is done in writer 1 (w1) file.

## Writer Code

#define SEMKEY1 151

#define SEMKEY2 152

#define SEMKEY3 153

#define SEMKEY4 154

#define SHM\_KEY1 9876

#define SHM\_KEY2 9877

int main()

{

printf("\n--------main is Starting for writer 1 ------------\n");

int shmid1,shmid2, semid1,semid2, semid3,semid4;

int \*ptr\_read\_count, \*ptr\_write\_count;

/\*shmid1 will be the rmutex \*/

semid1 = semget(SEMKEY1, 1 , 0777|IPC\_CREAT);

semctl(semid1,0,SETVAL,1);

/\*shmid2 will be the wmutex \*/

semid2 = semget(SEMKEY2, 1 , 0777|IPC\_CREAT);

semctl(semid2,0,SETVAL,1);

/\*shmid3 will be the readTry \*/

semid3 = semget(SEMKEY3, 1 , 0777|IPC\_CREAT);

semctl(semid3,0,SETVAL,1);

/\*shmid4 will be the resource \*/

semid4 = semget(SEMKEY4, 1 , 0777|IPC\_CREAT);

semctl(semid4,0,SETVAL,1);

Next set up the shared memory for the two counters:

/\* shmid1 is shared memory for the readcounter \*/

shmid1 = shmget(SHM\_KEY1, 256 , 0777|IPC\_CREAT);

ptr\_read\_count = (int\*)shmat(shmid1, 0, 0);

\*ptr\_read\_count = 0;

/\* shmid2 is shared memory for the writecount \*/

shmid2 = shmget(SHM\_KEY2, 256 , 0777|IPC\_CREAT);

ptr\_write\_count = (int\*)shmat(shmid2, 0, 0);

\*ptr\_write\_count = 0;

Now define and initialise a number of variables of type *sembuf*:

struct sembuf Wrmutex,Srmutex, Wwmutex, Swmutex,WreadTry,SreadTry,Wresource,Sresource;

Wrmutex.sem\_num = 0;

Wrmutex.sem\_op = -1;

Wrmutex.sem\_flg = SEM\_UNDO;

Srmutex.sem\_num = 0;

Srmutex.sem\_op = 1;

Srmutex.sem\_flg = SEM\_UNDO;

Wwmutex.sem\_num = 0;

Wwmutex.sem\_op = -1;

Wwmutex.sem\_flg = SEM\_UNDO;

Swmutex.sem\_num = 0;

Swmutex.sem\_op = 1;

Swmutex.sem\_flg = SEM\_UNDO;

WreadTry.sem\_num = 0;

WreadTry.sem\_op = -1;

WreadTry.sem\_flg = SEM\_UNDO;

SreadTry.sem\_num = 0;

SreadTry.sem\_op = 1;

SreadTry.sem\_flg = SEM\_UNDO;

Wresource.sem\_num = 0;

Wresource.sem\_op = -1;

Wresource.sem\_flg = SEM\_UNDO;

Sresource.sem\_num = 0;

Sresource.sem\_op = 1;

Sresource.sem\_flg = SEM\_UNDO;

Now start the writer code, first reserve the entry section by preforming a wait on the *Wmutex* before we increment the writer counter:

semop(semid2, &Wwmutex,1);

Increment the writer counter, letting others know we are here.

\*ptr\_write\_count = \*ptr\_write\_count + 1;

Up next, we need to check if we are the first writer, if so then we need to lock the readers out from the upcoming critical section using *WreadTry* semaphore.

if( \*ptr\_write\_count == 1 ){

semop(semid3, &WreadTry,1);

}

Release the mutex as we are finished with the write count and done the check:

semop(semid2, &Swmutex,1);

Now lock the resource from others before we commence writing:

semop(semid4, &Wresource,1);

This is the critical section where writing is preformed:

//Entry Critical Section (CS)

//-------------writing is done

char input;

cout<<"\n Do u want to write to file [y/n] => ";

cin>> input;

cout<<endl;

string line;

ofstream myfile ("example.txt", fstream::app);

if(myfile.is\_open() ){

cout<<"\nEnter input to save to file => ";

cin>>line;

myfile <<line <<"\n ";

myfile.close();

}

//----------------------

//Exit Critical Section (CS)

After exiting the critical section, release the resource:

semop(semid4, &Sresource,1);

Before we exit, we must decrement the write count, to do this we reserve the *Wmutex:*

semop(semid2, &Wwmutex,1);

Preform the decrement:

\*ptr\_write\_count = \*ptr\_write\_count - 1;

Check if we are the last writer leaving, if true them we release the *readTry*:

if( \*ptr\_write\_count == 0 ){

semop(semid3, &SreadTry,1);

}

Release the Write mutex for others before exit:

semop(semid2, &Swmutex,1);

return 0;

## Reader Code

We must use the same keys for the semaphores, we can use the same code except the initialization lines are removed as this was done in writer 1.

Note since only the *resource* & *Wmutex* semaphore is used in writer, we didn’t have to declare any of the other semaphores as they are not used here.

#define SEMKEY1 151

#define SEMKEY2 152

#define SEMKEY3 153

#define SEMKEY4 154

#define SHM\_KEY1 9876

#define SHM\_KEY2 9877

int main()

{

printf("\n--------main is Starting for reader 1 ------------\n");

int shmid1,shmid2, semid1,semid2, semid3,semid4;

int \*ptr\_read\_count, \*ptr\_write\_count;

/\*shmid1 will be the rmutex \*/

semid1 = semget(SEMKEY1, 1 , 0777|IPC\_CREAT);

/\*shmid2 will be the wmutex \*/

semid2 = semget(SEMKEY2, 1 , 0777|IPC\_CREAT);

/\*shmid3 will be the readTry \*/

semid3 = semget(SEMKEY3, 1 , 0777|IPC\_CREAT);

/\*shmid4 will be the resource \*/

semid4 = semget(SEMKEY4, 1 , 0777|IPC\_CREAT);

/\* shmid1 is shared memory for the readcounter \*/

shmid1 = shmget(SHM\_KEY1, 256 , 0777|IPC\_CREAT);

ptr\_read\_count = (int\*)shmat(shmid1, 0, 0);

/\* shmid2 is shared memory for the writecount \*/

shmid2 = shmget(SHM\_KEY2, 256 , 0777|IPC\_CREAT);

ptr\_write\_count = (int\*)shmat(shmid2, 0, 0);

Now define and initialise a number of variables of type *sembuf*:

struct sembuf Wrmutex, Srmutex, Wwmutex, Swmutex, WreadTry,SreadTry,Wresource, Sresource;

Wrmutex.sem\_num = 0;

Wrmutex.sem\_op = -1;

Wrmutex.sem\_flg = SEM\_UNDO;

Srmutex.sem\_num = 0;

Srmutex.sem\_op = 1;

Srmutex.sem\_flg = SEM\_UNDO;

Wwmutex.sem\_num = 0;

Wwmutex.sem\_op = -1;

Wwmutex.sem\_flg = SEM\_UNDO;

Swmutex.sem\_num = 0;

Swmutex.sem\_op = 1;

Swmutex.sem\_flg = SEM\_UNDO;

WreadTry.sem\_num = 0;

WreadTry.sem\_op = -1;

WreadTry.sem\_flg = SEM\_UNDO;

SreadTry.sem\_num = 0;

SreadTry.sem\_op = 1;

SreadTry.sem\_flg = SEM\_UNDO;

Wresource.sem\_num = 0;

Wresource.sem\_op = -1;

Wresource.sem\_flg = SEM\_UNDO;

Sresource.sem\_num = 0;

Sresource.sem\_op = 1;

Sresource.sem\_flg = SEM\_UNDO;

Now start the reader code, first reserve the entry section by preforming a wait on the *WreadTry* to indicate a reader is trying to enter the section.

semop(semid3, &WreadTry,1);

Lock the read mutex before go and increment the read counter:

semop(semid1, &Wrmutex,1);

Preform the increment to let others know we are here.

\*ptr\_read\_count = \*ptr\_read\_count + 1;

Do a check to see if we are the first reader if so, we need to lock the resource:

if( \*ptr\_read\_count == 1 ){

semop(semid4, &Wresource,1);

}

Now release this for other readers to use:

semop(semid1, &Srmutex,1);

Preform a signal on the *readTry* to indicate we are done trying to access the resource:

semop(semid3, &SreadTry,1);

Enter the critical section next and preform the reading:

//Enter Critical section

//-------------reading

char input;

cout<<"\n Do u want to read to file [y/n] => ";

cin>> input;

cout<<endl;

if (input == 'y'){

string line;

ifstream myfile ("example.txt");

if(myfile.is\_open()){

cout<<"\nContents of file are below:"<<endl;

while(getline (myfile,line) ) {

cout<<line <<endl;

}

myfile.close();

}

else cout<<"\nUnable to open file"<<endl;

}

//----------------------

//Exit Critical section

Before exiting we must decrement the counter, to do this reserve the exit section:

semop(semid1, &Wrmutex,1);

Preform the decrement:

\*ptr\_read\_count = \*ptr\_read\_count - 1;

Check if we are the last reader leaving if so, we must release the resource:

if( \*ptr\_read\_count == 0 ){

semop(semid4, &Sresource,1);

}

Lastly, we release the read mutex also:

semop(semid1, &Srmutex,1);

return 0;

## Flowchart for Writer-Preference

Diagram

Description automatically generatedDiagram

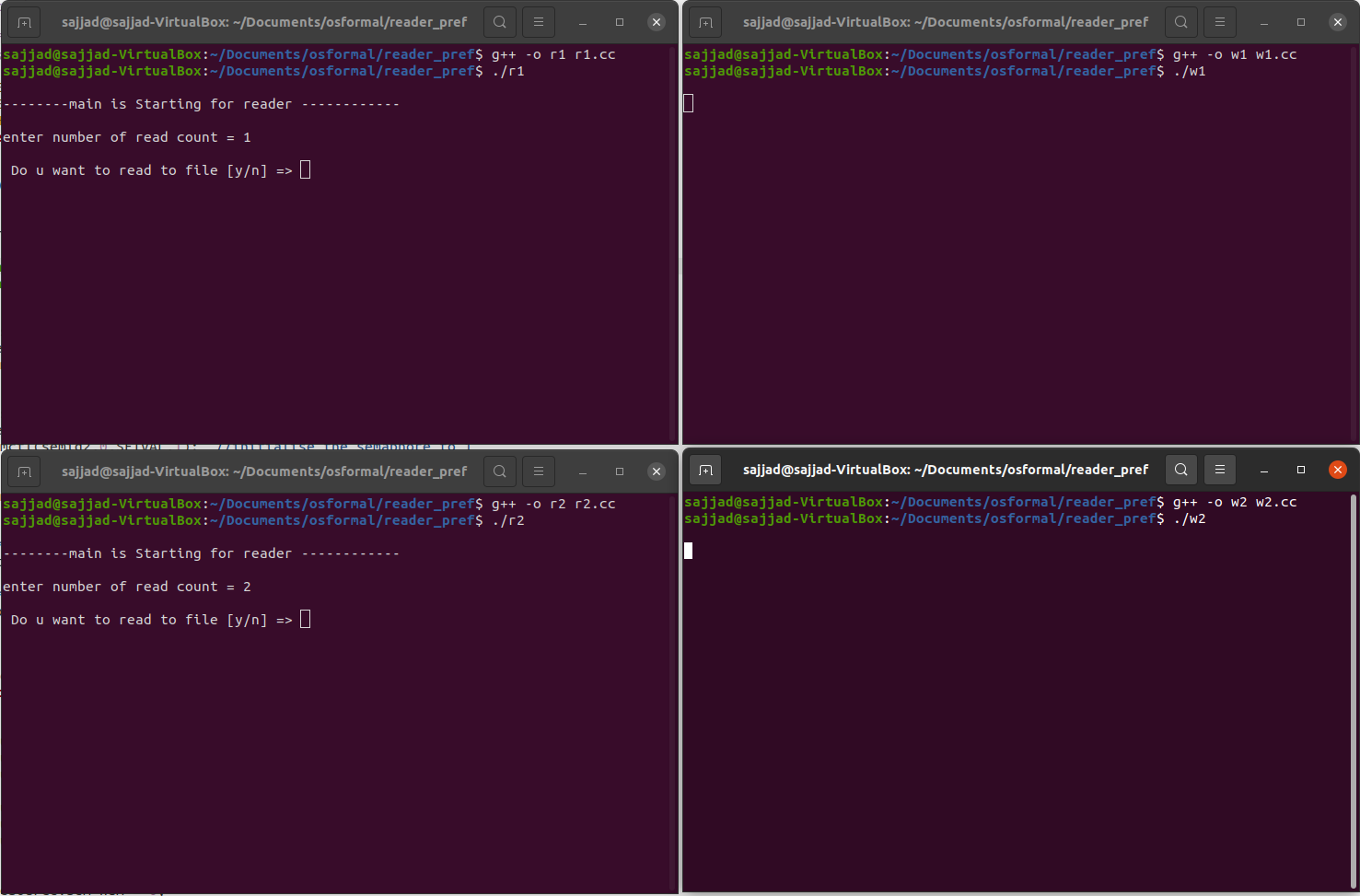
Description automatically generated

# Testing Readers Preference

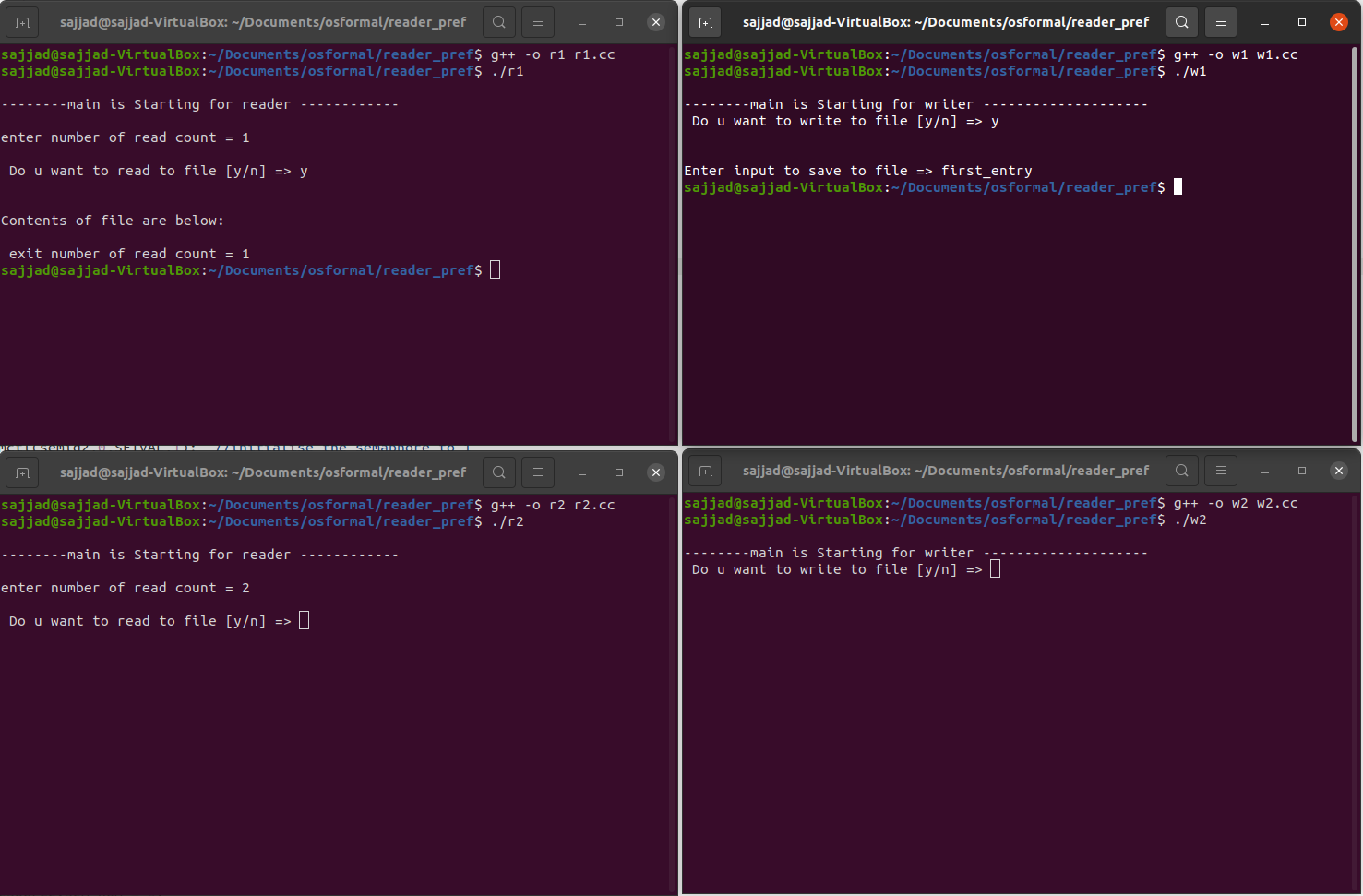
Start by opening 4 terminals, run **r1** first as initialization is done here, then **W1** then **R2** then **W2**.

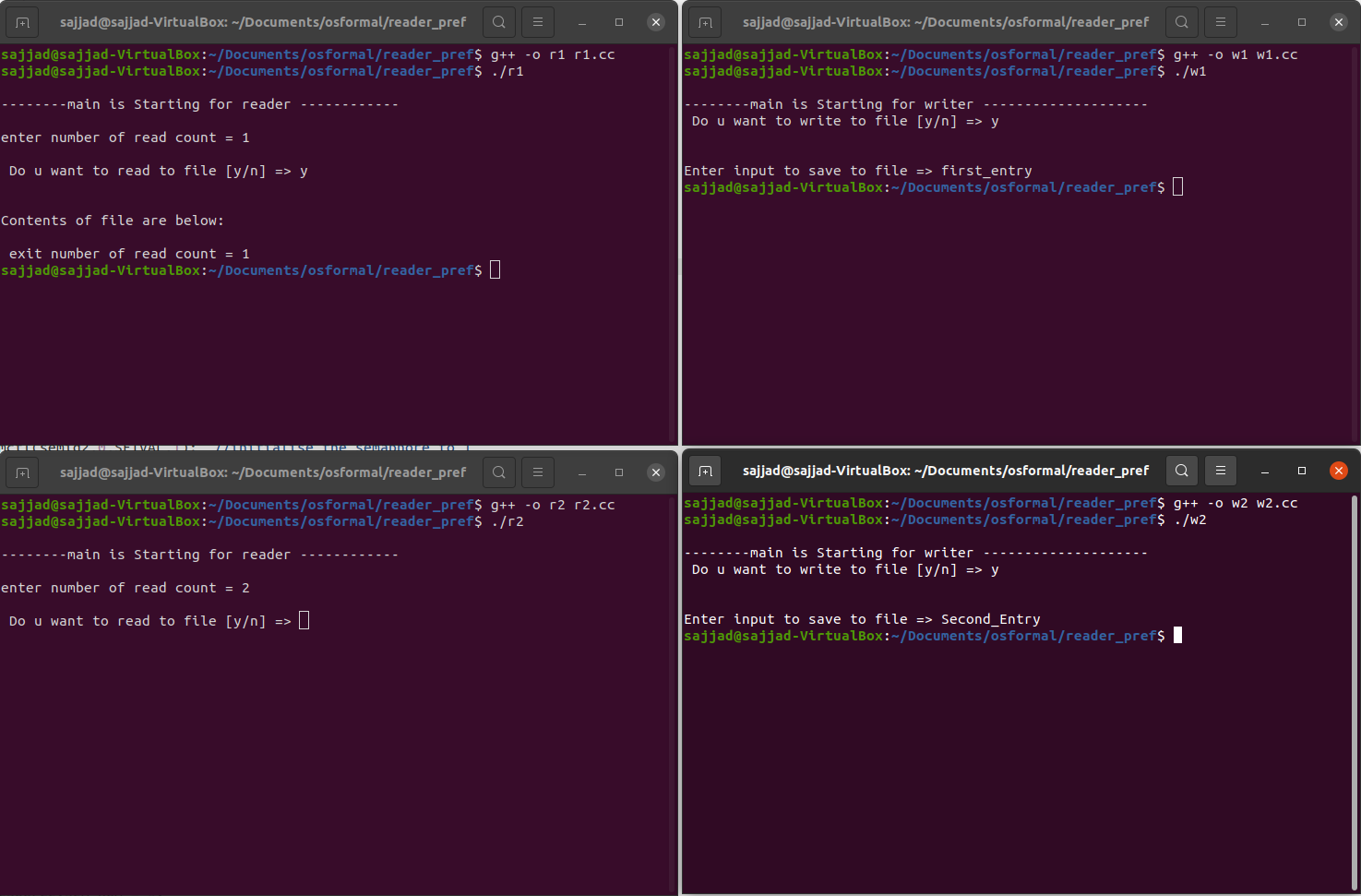
As seen below when reader 1 starts the read count is 1 and the writers are blocked.

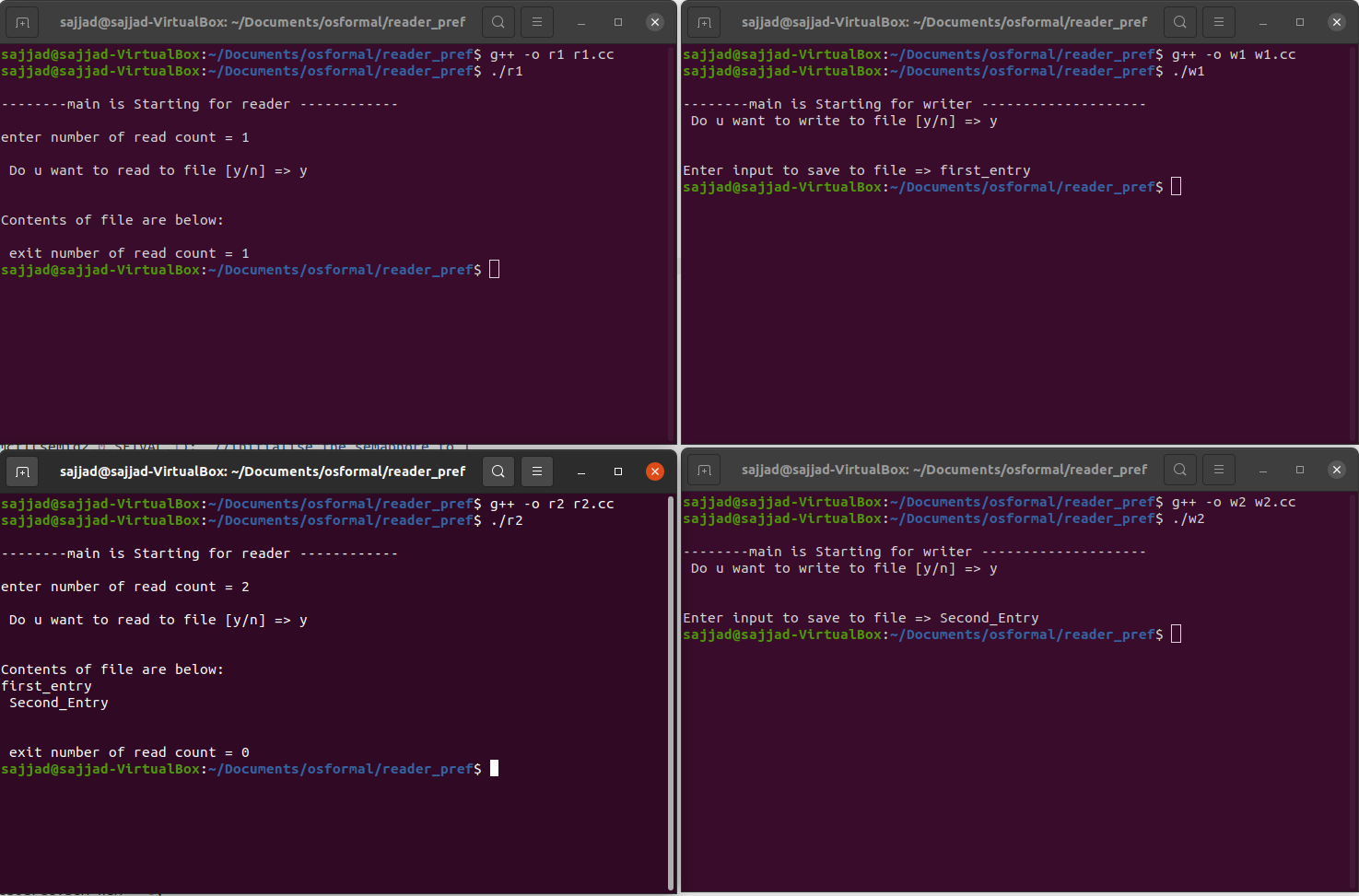
When R2 runs its read count is 2 due to r1 already there and incremented the counter.

Both writers are blocked.

We run r1, enter ‘y’ for yes to view the contents of the file which is initially empty.

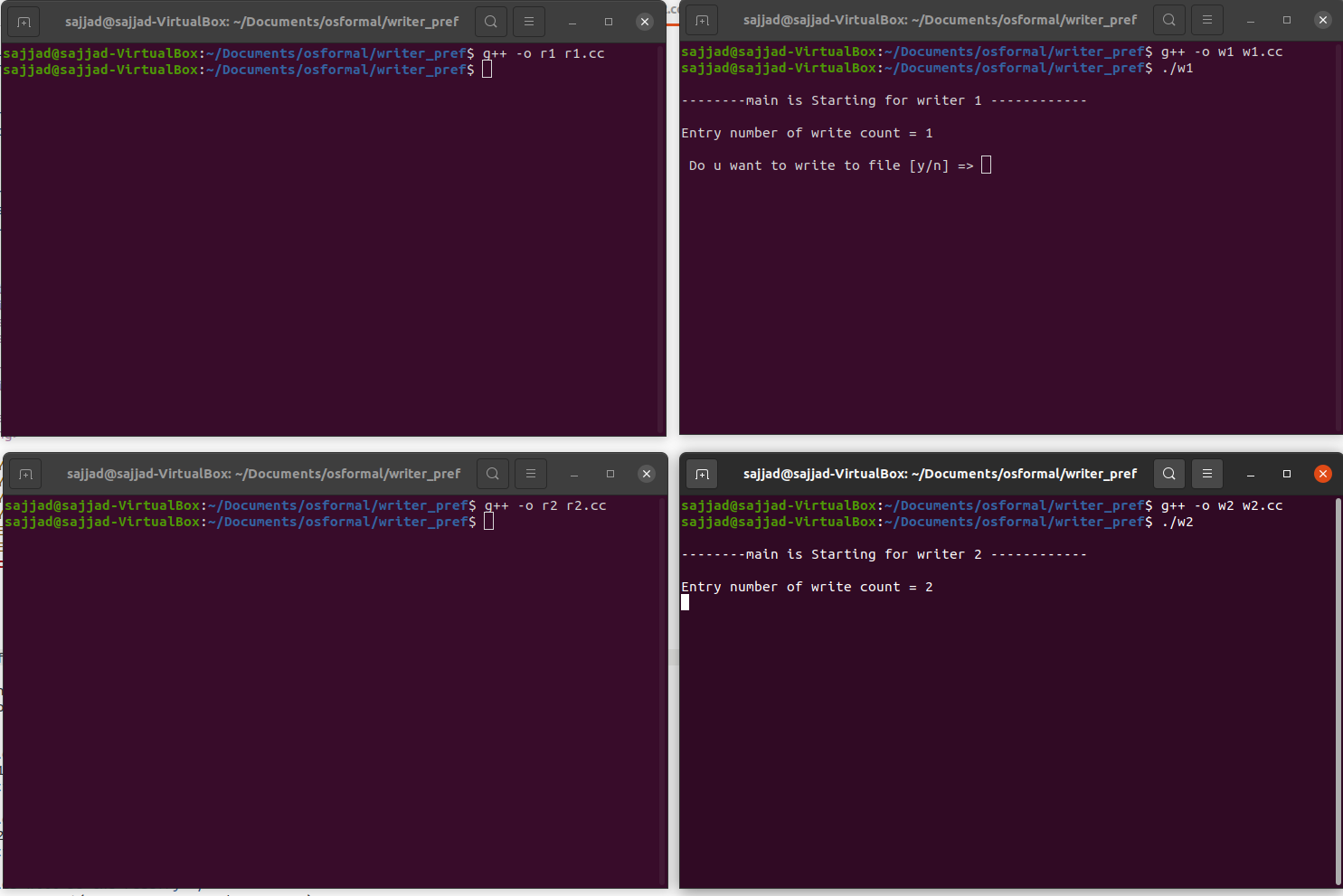
This now frees up w1 where we enter ‘y’ to insert text to the file as “first\_entry”

Up next, we use w2 to enter another line of text to the file, this text is “second\_entry”

Finally, we use r2 to view the contents of the file which does indeed show the input saved from w1 & w2.

# Testing Writers Preference

Start by opening 4 terminals, run **w1** first as initialization is done here, then **r1** then **w2** then **r2**.

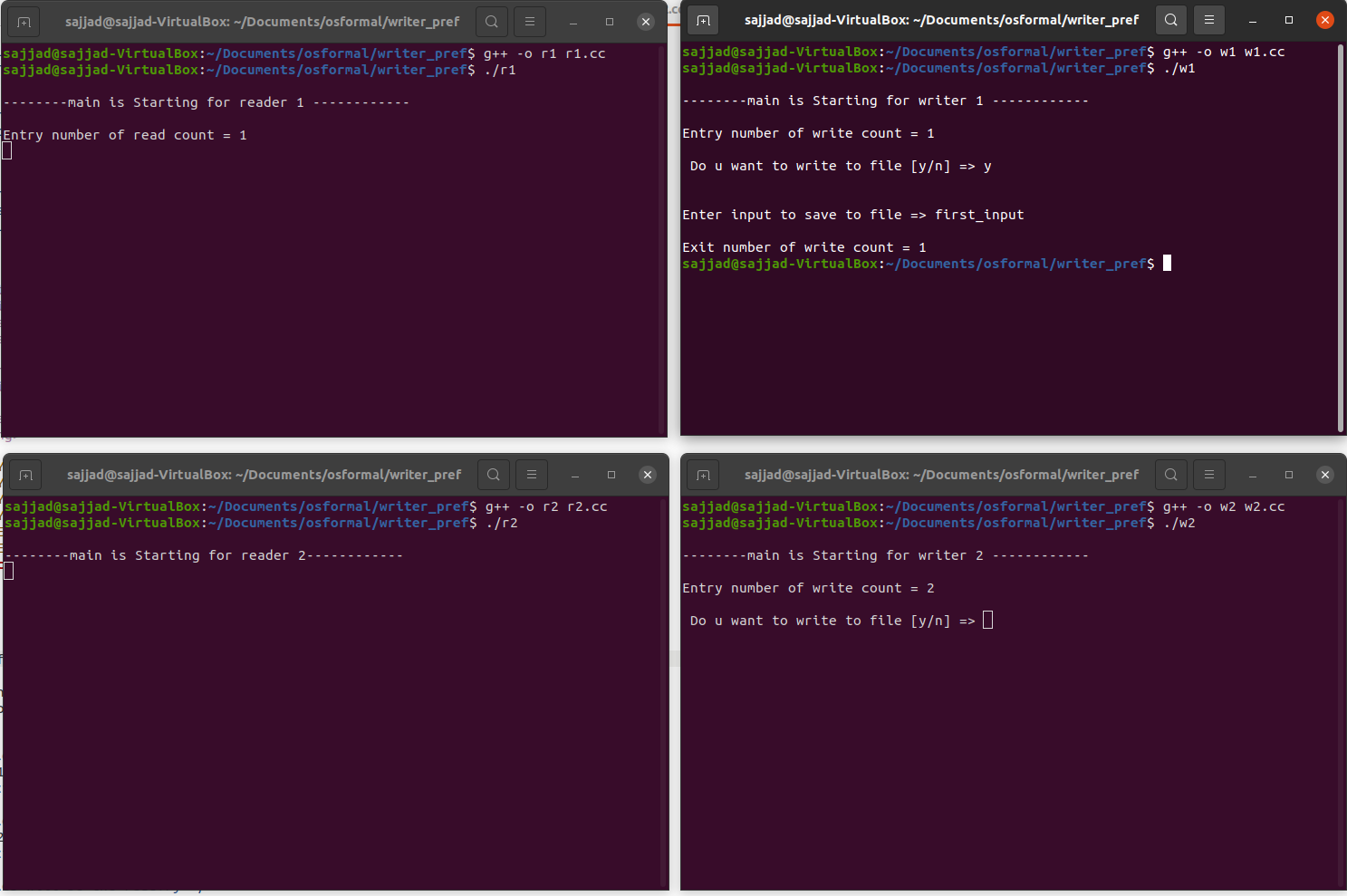


When a writer is in, the readers are blocked.

We start by entering ‘y’ into w1 and insert text “first\_input” to save to the file.

When w2 runs it is the second writer and the counter is at 2 now.

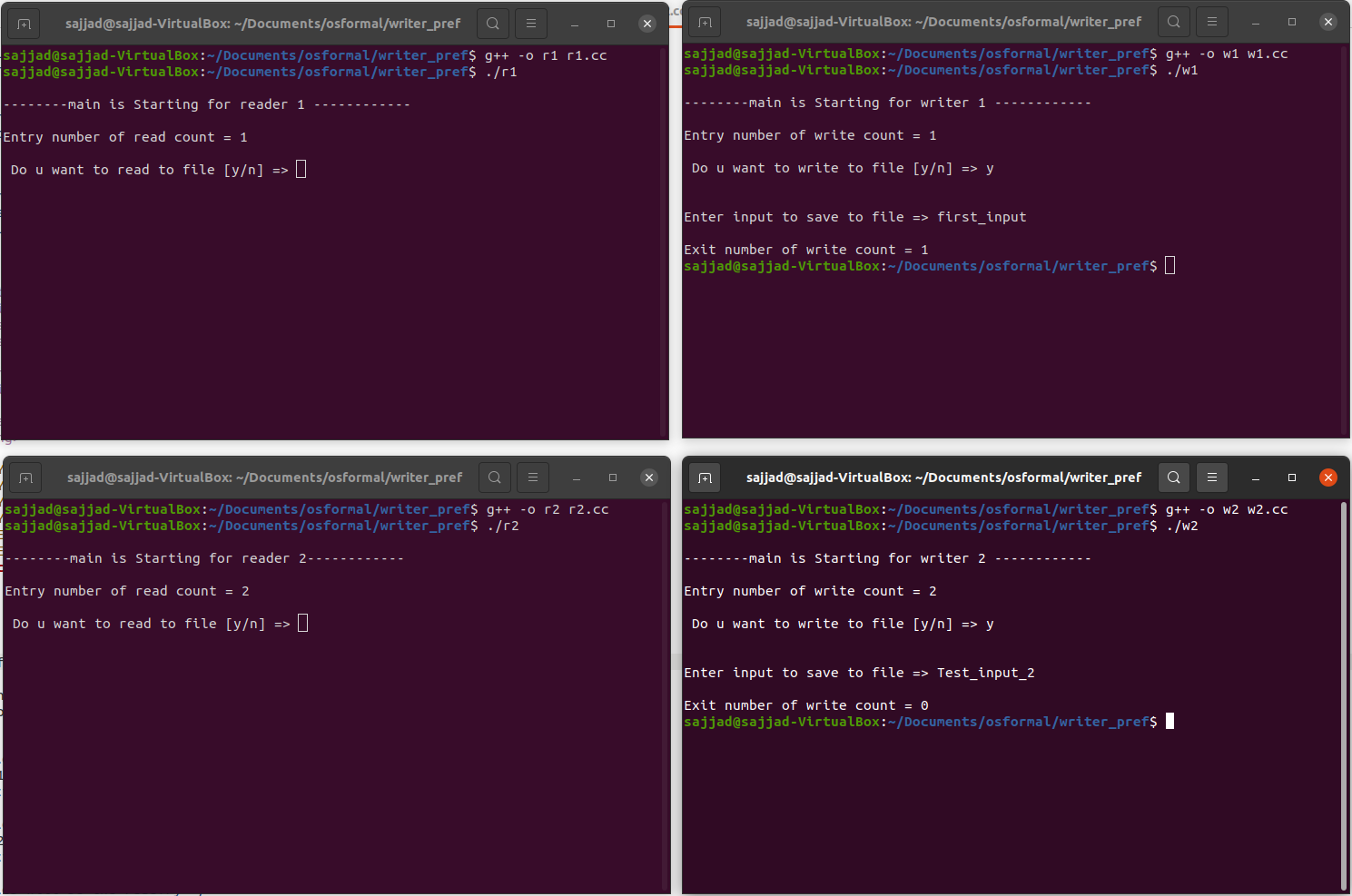
The exit number of write count seen in w1 indicates there is still 1 writer still writing.



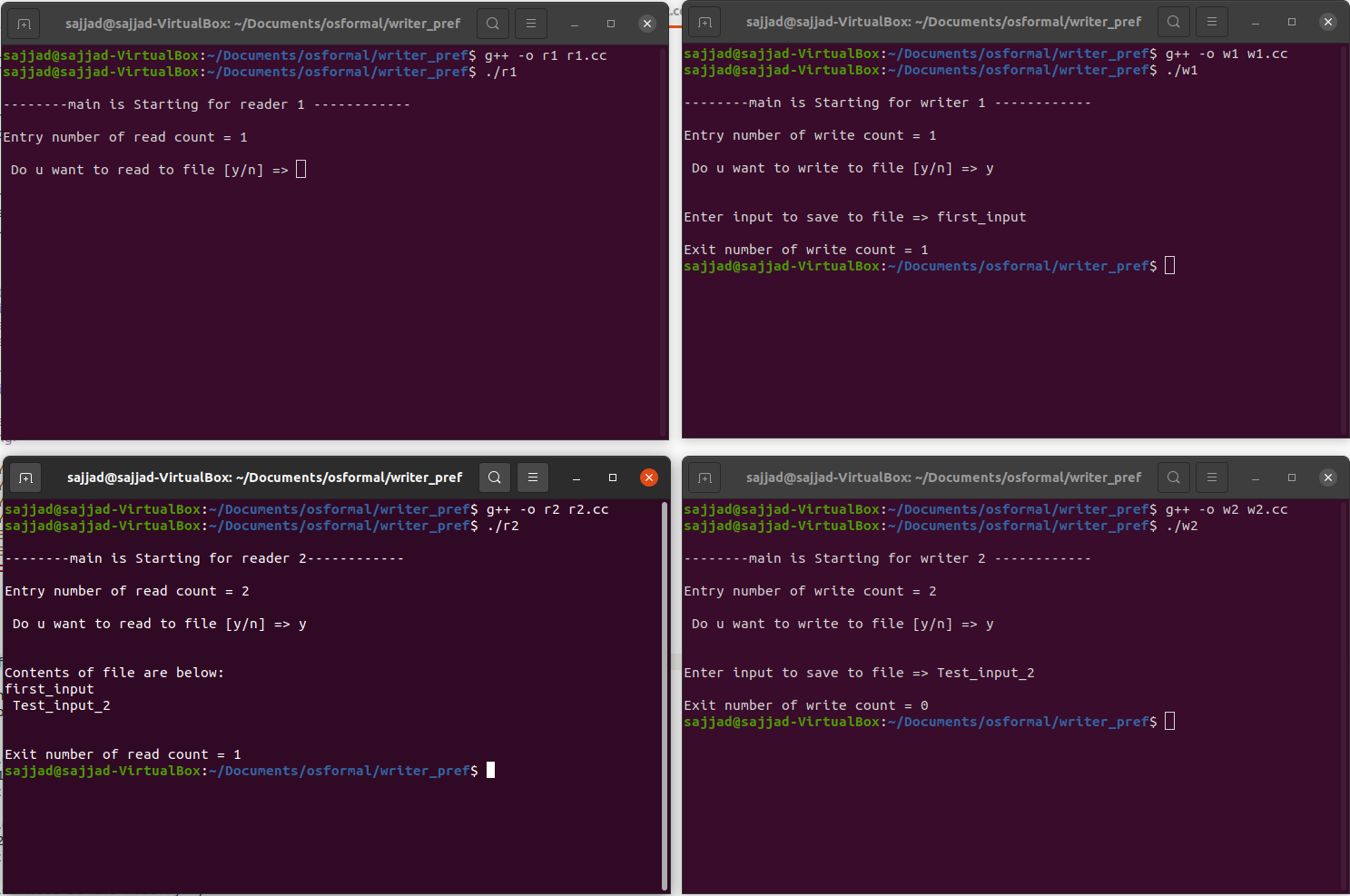
Now use w2 to enter another line “test\_input\_2”.

After pressing enter for w2 this will now free up reader 1 &2 to read.

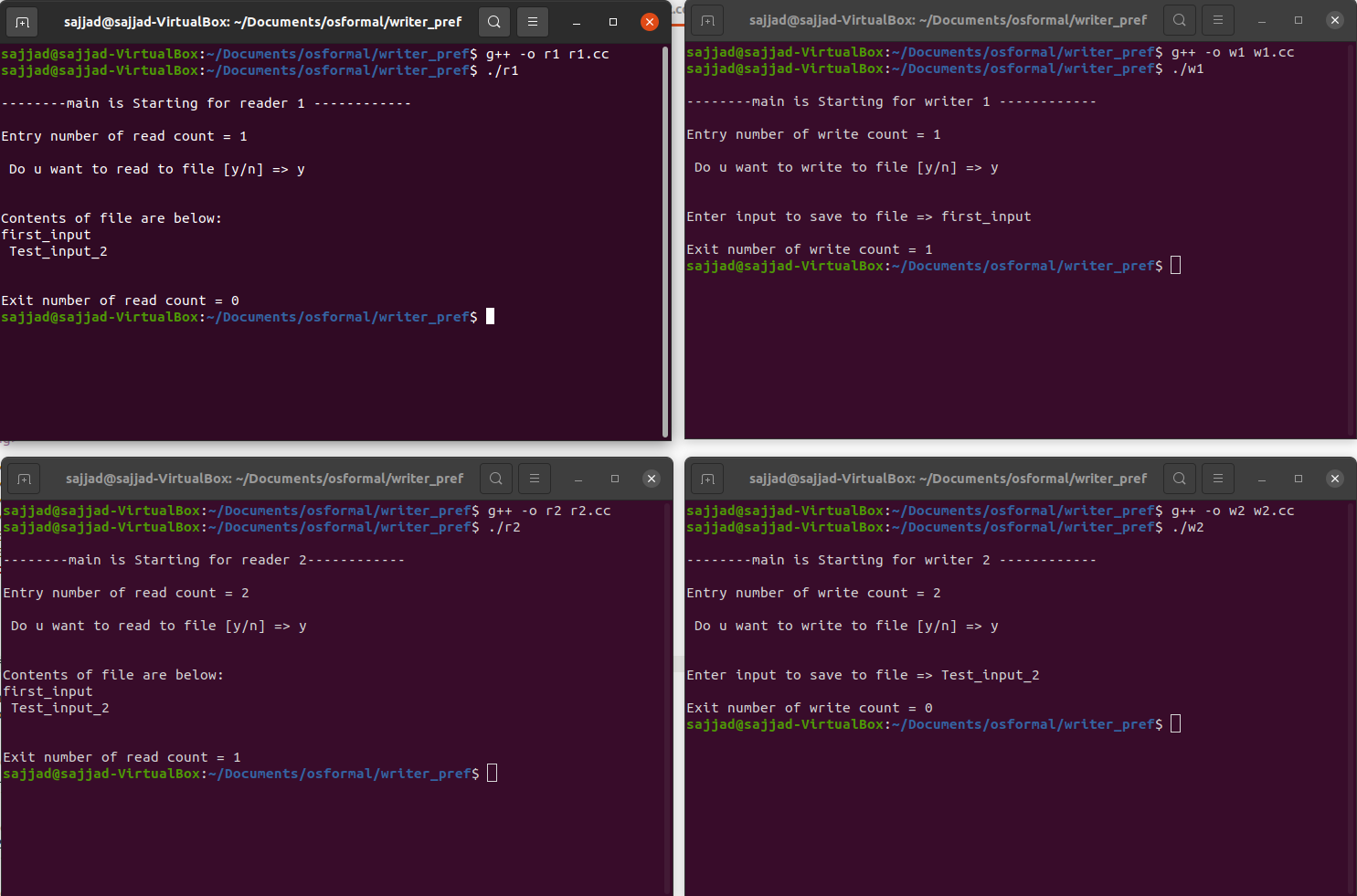
We can enter ‘y’ to either r1 or r2 this does not matter.



If we use r2 first it will read the contents and decrement the counter before exiting.

The exit number of read counter is 1 indicating there is still one reader left.

Lastly run r1 which displays the contents of the file also, when this exists the counter is zero as this was the last reader.



# Final changes to both solutions

We have met all the objectives of this assignment; last modification is to make the readers/writers sections into an infinite while loop.

**Objectives**

* These programs will be run in **separate terminal** windows.
* The output from the processes should demonstrate that solution solves the problem.
* The Writer process should **open a text file** and **append** some **text** to the file.
* The text will be some **input** from the keyboard.
* When the Writer is finished writing to the file the file should be closed.
* The Reader should type out the text in the file line by line, in response to a request from the user.
* When the Reader is finished reading the file should be closed.
* The reader/writer sections will be in an infinite while loop.

We achieve this my encompassing the readers writers code section into a while loop like show below for the readers preference, writers code.

while(1){

semop(semid2, &Wresource,1);

//critical section

//-------------writing is done

char input;

cout<<"\n Do u want to write to file [y/n] => ";

cin>> input;

cout<<endl;

string line;

ofstream myfile ("example.txt", fstream::app);

if(myfile.is\_open() ){

cout<<"\nEnter input to save to file => ";

cin>>line;

myfile <<line <<"\n ";

myfile.close();

}

//----------------------

//exit critical section

/\* release the shared resource by signal on resource \*/

semop(semid2, &Sresource,1);

}

# Conclusion

In conclusion we have coded and demonstrated two solutions to the readers-writers problem, that is the readers preference and writers’ preference. It is possible in the first solution for writers to be starved and in the second for readers to be starved thus there is a third solution that was not included in this report where no thread starves by allocating time slots therefore the threads will always terminate in a bounded amount of time.

# Bibliography

* P. Courtois, F. Heymans and D. Parnas, "Concurrent control with “readers” and “writers”", *Communications of the ACM*, vol. 14, no. 10, 1971. Available: 10.1145/362759.362813 [Accessed 21 December 2021].
* Tanenbaum and A. Woodhull, *Operating systems*. Upper Sadle River: Pearson/Prentice Hall, 2008.
* "Readers–writers problem - Wikipedia", *En.wikipedia.org*. [Online]. Available: https://en.wikipedia.org/wiki/Readers%E2%80%93writers\_problem#cite\_note-1. [Accessed: 21- Dec- 2021].