****

**Project Report**

CS2006 Operating System

**Zainab Talha 21k-491**

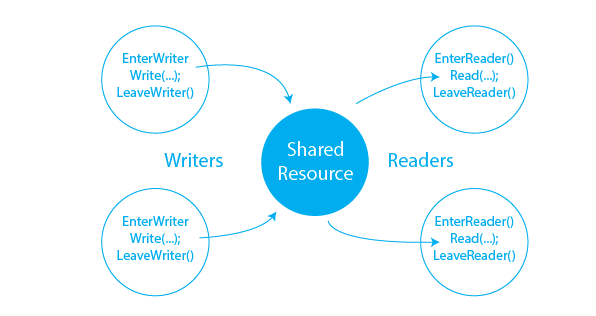
**Maheen shakeel 21k-4533**

Contents

* Introduction:
* Background:
* Platform & languages
* Algorithm
* Methodology:
* REAL TIME EXAPMLE:
* Project aspects in Future technology:
* Conclusion:
* References
* **Introduction:**

The reader-writer problem is a common synchronization problem in multi-threaded applications. It involves multiple threads that either read or write to a shared resource. The reader-writer problem is challenging because it requires balancing the need for concurrent access to the shared resource while ensuring data consistency and preventing race conditions.

In this project, we implement a reader-writer lock in the Linux kernel using C. Our implementation uses a spinlock to prevent race conditions and ensures fairness in granting access to the shared resource. We evaluate the performance of our implementation using a benchmark test that measures the throughput and latency of the lock.



* **Background:**

The reader-writer problem involves multiple threads that either read or write to a shared resource. The problem is challenging because it requires balancing the need for concurrent access to the shared resource while ensuring data consistency and preventing race conditions. One solution to the reader-writer problem is to use a reader-writer lock, which allows multiple readers to access the shared resource simultaneously while ensuring that only one writer can access the resource at a time.

A reader-writer lock can be implemented using different mechanisms, such as semaphores, mutex, or spinlocks.

* **Platform & Languages**

The code is implemented in **C** and is designed to run on a **Linux platform**. It includes headers specific to Linux, such as <linux/kernel.h>, <linux/syscalls.h>, and <sys/syscall.h>. The code utilizes Linux kernel functions and system calls.

* **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.

* **Methodology:**

1. **Downloading a kernel:**

First of all, we downloaded a kernel from kernel.org by using the

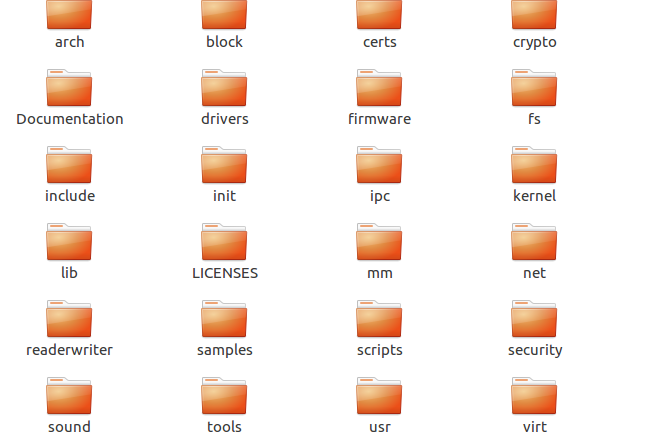
“**wget**” command.

1. **Extracting the kernel:**

Then we went to the folder where the kernel was downloaded and extract it

1. **Making a new folder called readerwriter:**

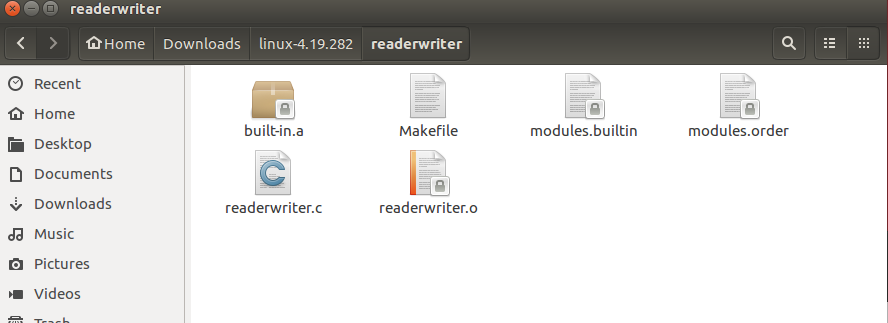
We then get inside the kernel’s folder and created a new directory by opening the terminal there and typing “**mkdir readerwriter**”.

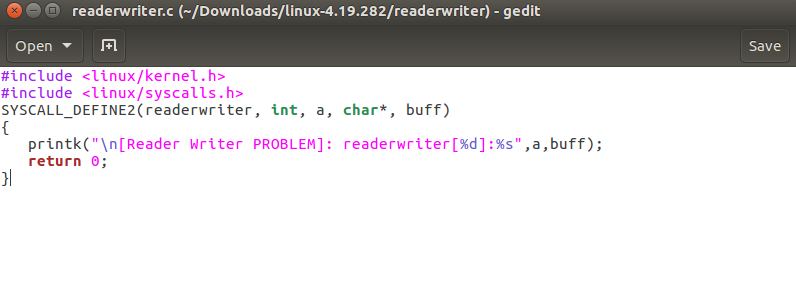


1. **Adding a C code for the system call:**

We go inside the readerwriter folder and open the terminal there and created a new C code file by typing “**gedit readerwriter.c**” and write the following code there

*#include <linux/kernel.h>  
#include <linux/syscalls.h>  
SYSCALL\_ DEFINE2(readerwriter, int, a, char\*, buff)  
{  
   printk("\n[ Reader Writer PROBLEM]: readerwriter[%d]:%s",a,buff);  
   return 0; }*





1. **Code explanation:**

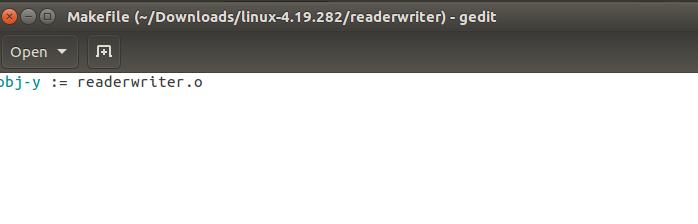
The code defines a custom system call named "**readerwriter**" in the Linux kernel. It takes two arguments: an integer a and a character pointer buff. When the system call is invoked, it logs a message to the kernel log, indicating the values of the arguments passed. The system call always returns 0, indicating success.

1. **Creating a Makefile for the C code:**

We created a Makefile for our new folder to ensure that the code in the folder is

always compiled whenever the kernel is compiled. In order to do this, we typed in our

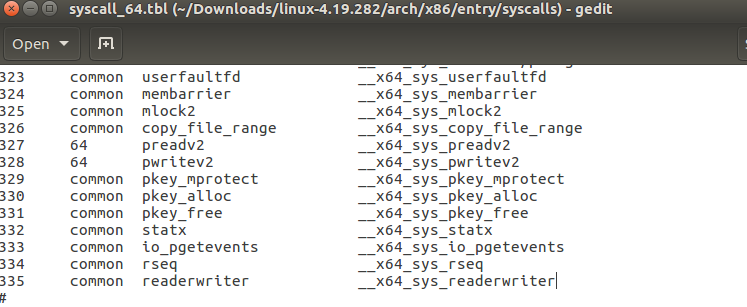
terminal **“gedit Makefile”** and put **“obj-y :=readerwriter.o**



1. **Adding the new code into the system table file:**

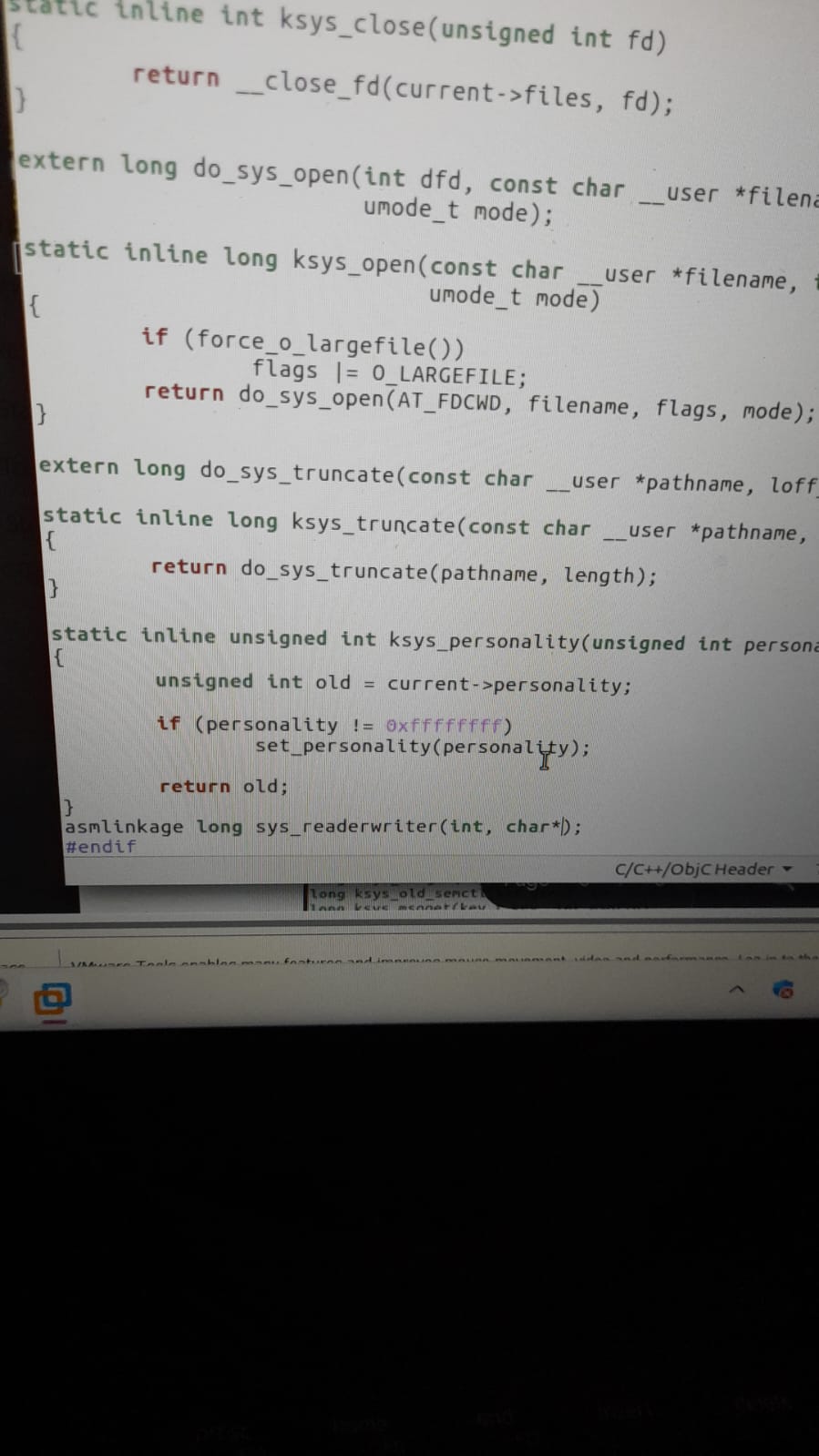
Since we were creating a 64-bit system call so we added the system call entry into the **syscall\_64.tbl** file which keeps the name of all the system calls in our system.

This tbl file is located inside the kernel folder in **/arch/x86/entry/syscalls/syscall\_64.tbl**. We went into this directory by using **cd** and then edit the file by typing “**gedit syscall\_64.tbl**”



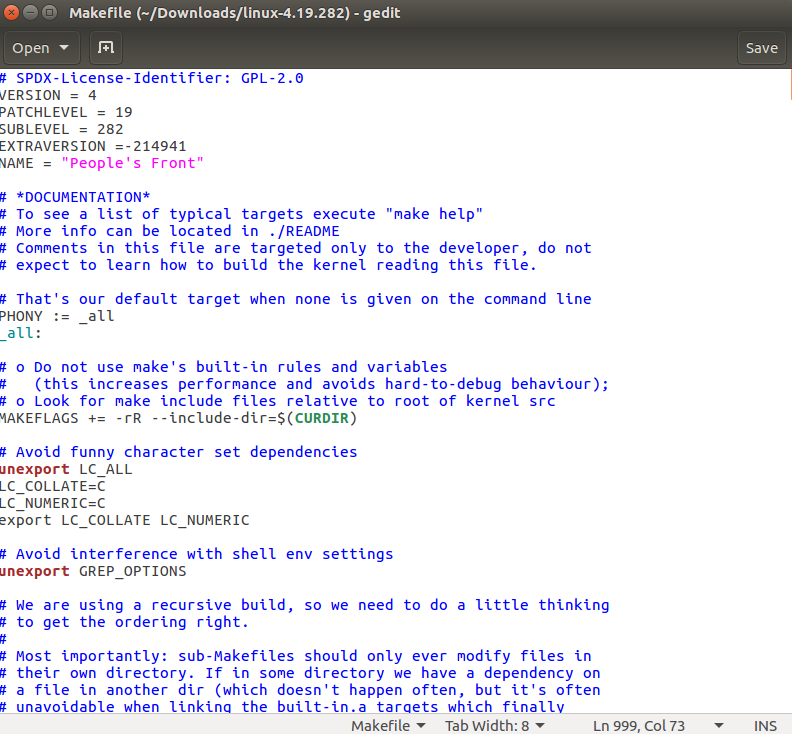
1. **Adding the prototype of the new system call into the system calls header file:**

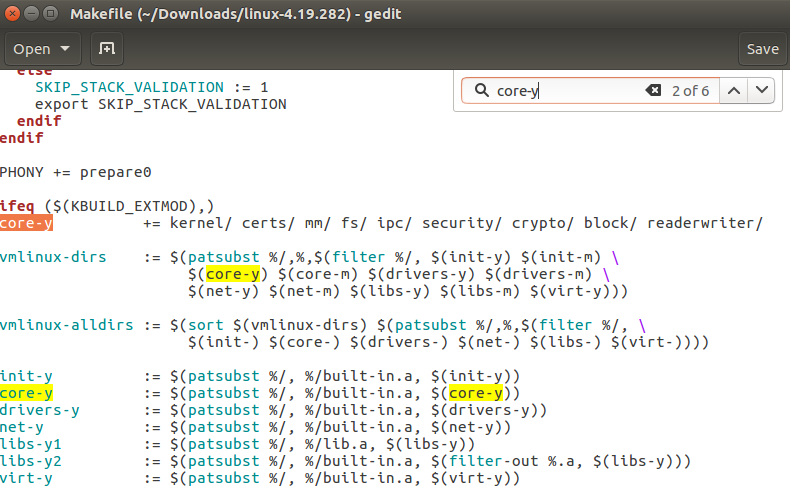
Then we added the prototype of our system call in the system’s header file which is located in the kernel folder then we did **gedit “include/linux/syscalls.h”**. We added the proto type of our system call function in this file.



1. **Changing version and adding the hello folder in the kernel’s Makefile:**

We added our roll number in the extraversion of the kernel’s make file and we added the new module that we created into our kernel’s make file. For this, we open **Makefile** of the kernel and searched for **“core-y”** and go to it’s second instance which is under **“KBUILD\_EXTMOD”** and added our new module which is **“readerwriter”** at the end of it.



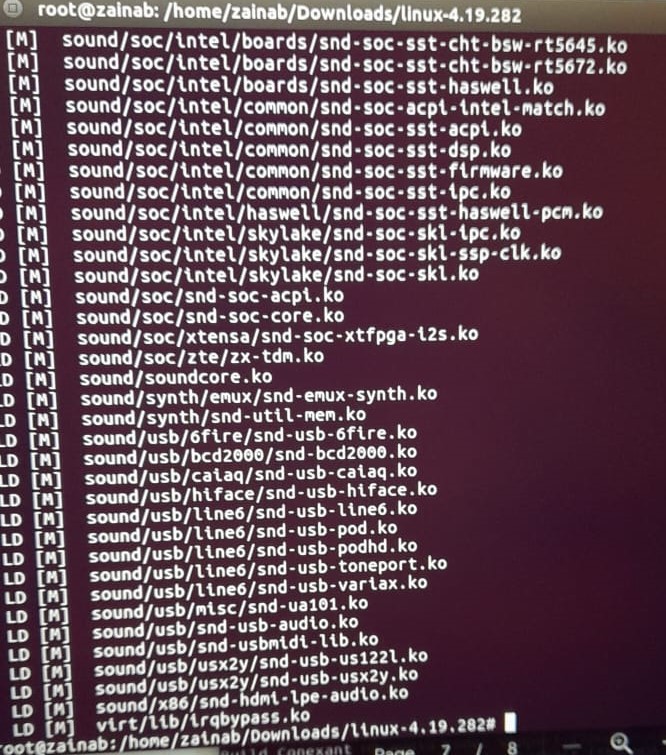


1. **Creating a config file:**

We created a config file for our kernel. On terminal we typed “**ls /boot | grep config**” then we typed **cp /boot/config-4.15.0-112-generic /home/zainab/kernel/linux-4.19.282** (current working directory). Then we created the old config by typing **“yes "" | make oldconfig -j4”**, by doing so, the system automatically created the new config for us and select the default option for everything.

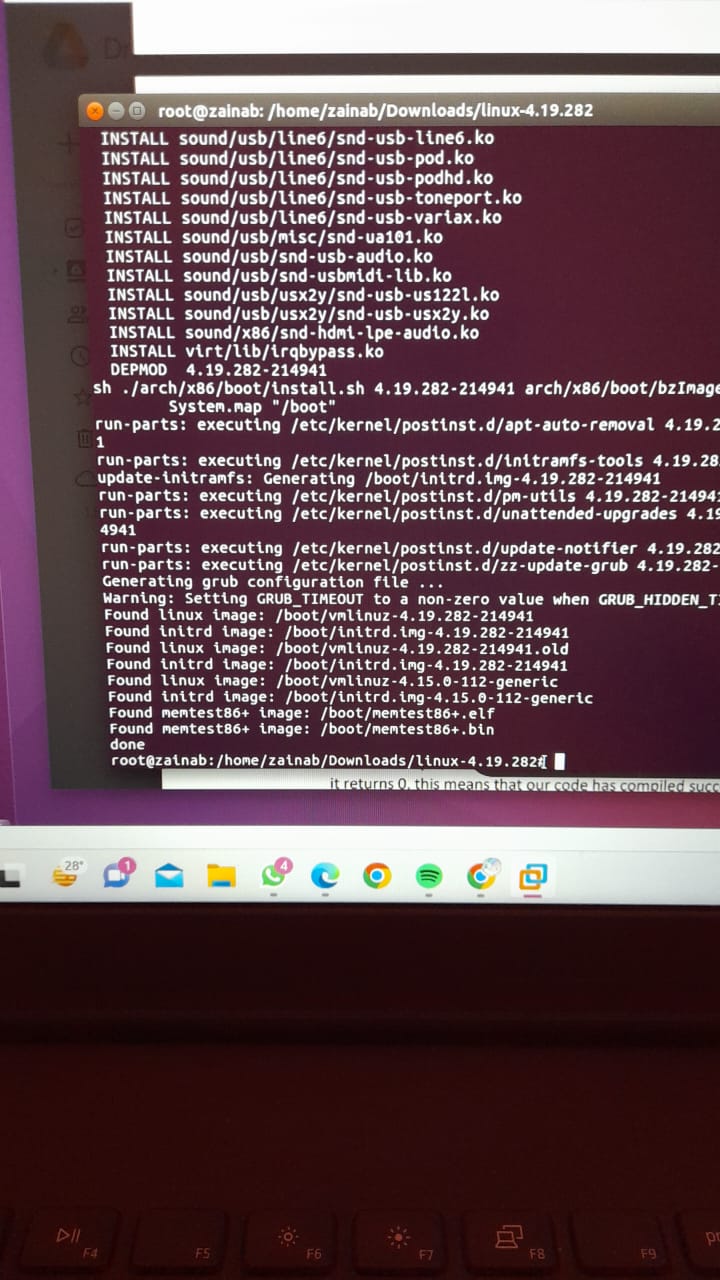
1. **Cleaning and Compiling the kernel:**

We cleaned all of our existing object and executable file because compiler sometimes link or compile files incorrectly and to avoid this, we deleted all of our old object and executable files by typing **“make clean -j4”** and when this all is done, we typed “**make -j4**” to start building our kernel



1. **Installing modules:**

We installed the kernel by typing “**make modules\_install install**” which installed the kernel and updated our grub as well. When all done and the terminal said **“done”**, then we restarted our laptop by typing **“shutdown -r now”** and hold the **“Shift”** key while it was restarting to open up the grub menu and switch to the new kernel which we just installed.



1. **Checking if the System call is Working Properly:**

After logging into the newly compiled kernel, we checked the system call by making a C code named “**userspace.c**” and put the following code in it:

*#include<stdio.h>*

*#include<pthread.h>*

*#include<unistd.h>*

*#include<semaphore.h>*

*#include<string.h>  
#include<linux/kernel.h>*

*#include<sys/syscall.h>*

*sem\_t mutex; // Semaphore for mutual exclusion of readcount*

*sem\_t wblock; // Semaphore for writer exclusion*

*int x = 0; // Shared resource*

*int readcount = 0;*

*void \*reader(void \*arg) {*

*int \*t = (int \*)arg; // Temporary variable t*

*int f = \*t;*

*// Entry section*

*sem\_wait(&mutex); // Acquire mutex to update readcount*

*readcount++;*

*if (readcount == 1) {*

*sem\_wait(&wblock); // If first reader, acquire wblock to block writers*

*}*

*sem\_post(&mutex); // Release mutex after updating readcount*

*// Critical section (Reading shared resource)*

*printf("Data read by the reader %d is %d\n", f, x);*

*syscall(335,f,”Data read by the reader\n”);*

*// Exit section*

*sem\_wait(&mutex); // Acquire mutex to update readcount*

*readcount--;*

*if (readcount == 0) {*

*sem\_post(&wblock); // If last reader, release wblock to allow writers*

*}*

*sem\_post(&mutex); // Release mutex after updating readcount*

*}*

*void \*writer(void \*arg) {*

*int \*t = (int \*)arg;*

*int f = \*t;*

*// Entry section*

*sem\_wait(&wblock); // Acquire wblock to block other writers*

*// Critical section (Writing shared resource)*

*x++; // Modify the shared resource (writing)*

*printf("Data written by the writer %d is %d\n", f, x);*

*syscall(335,f,”Data written by the writer\n”);*

*// Exit section*

*sem\_post(&wblock); // Release wblock to allow other writers*

*}*

*int main() {*

*int i;*

*pthread\_t id1[5], id2[5]; // id1 = reader , id2 = writer threads*

*//Initializing*

*sem\_init(&mutex, 0, 1);*

*sem\_init(&wblock, 0, 1);*

*//Create reader and writer threads*

*for (i = 0; i <= 2; i++) {*

*pthread\_create(&id2[i], NULL, writer, &i);*

*pthread\_create(&id1[i], NULL, reader, &i);*

*}*

*//Join reader and writer threads*

*for (i = 0; i <= 2; i++) {*

*pthread\_join(id2[i], NULL);*

*pthread\_join(id1[i], NULL);*

*}*

*//Destroy semaphores*

*sem\_destroy(&mutex);*

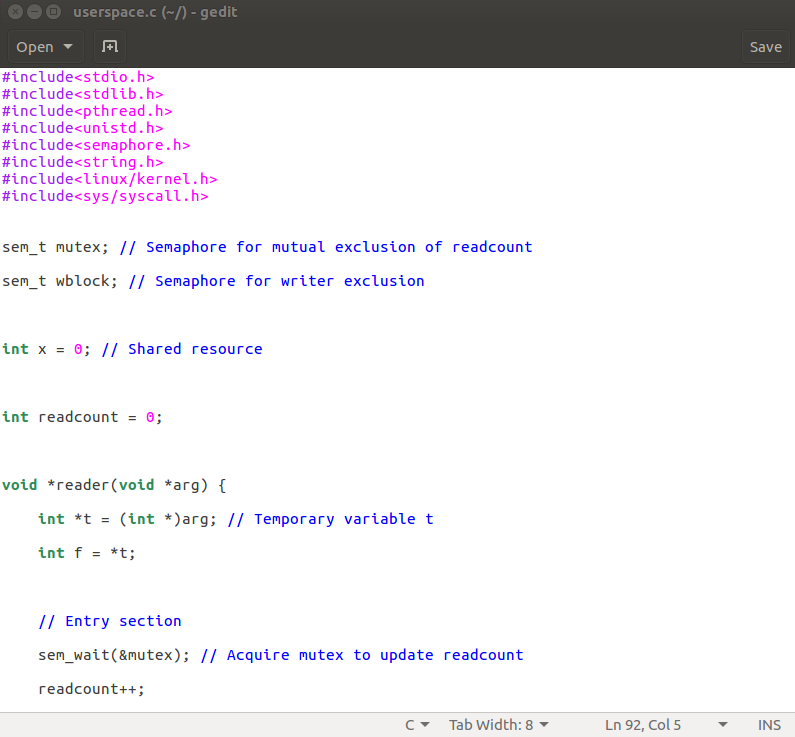
*sem\_destroy(&wblock);*

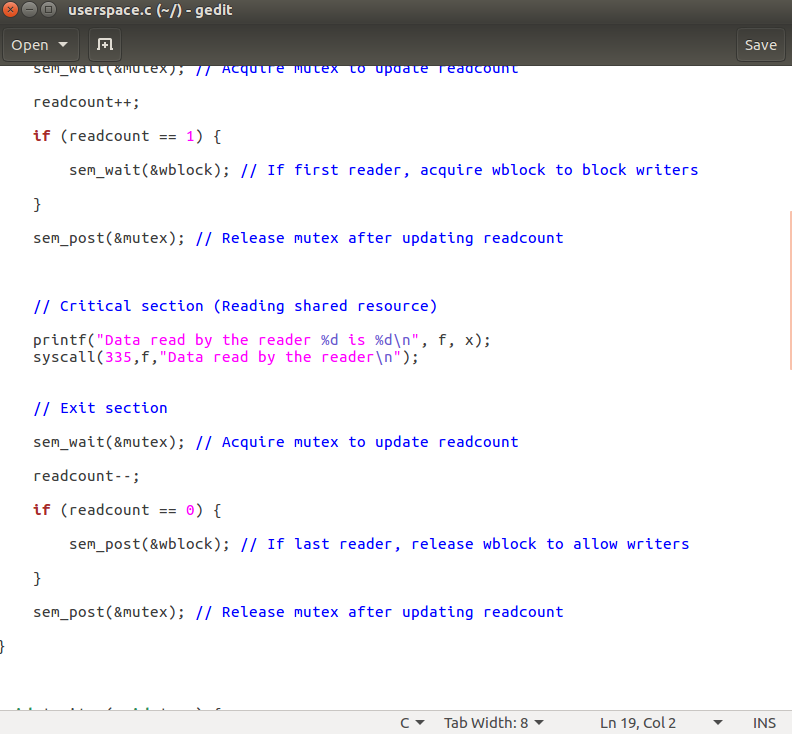
*return 0;*

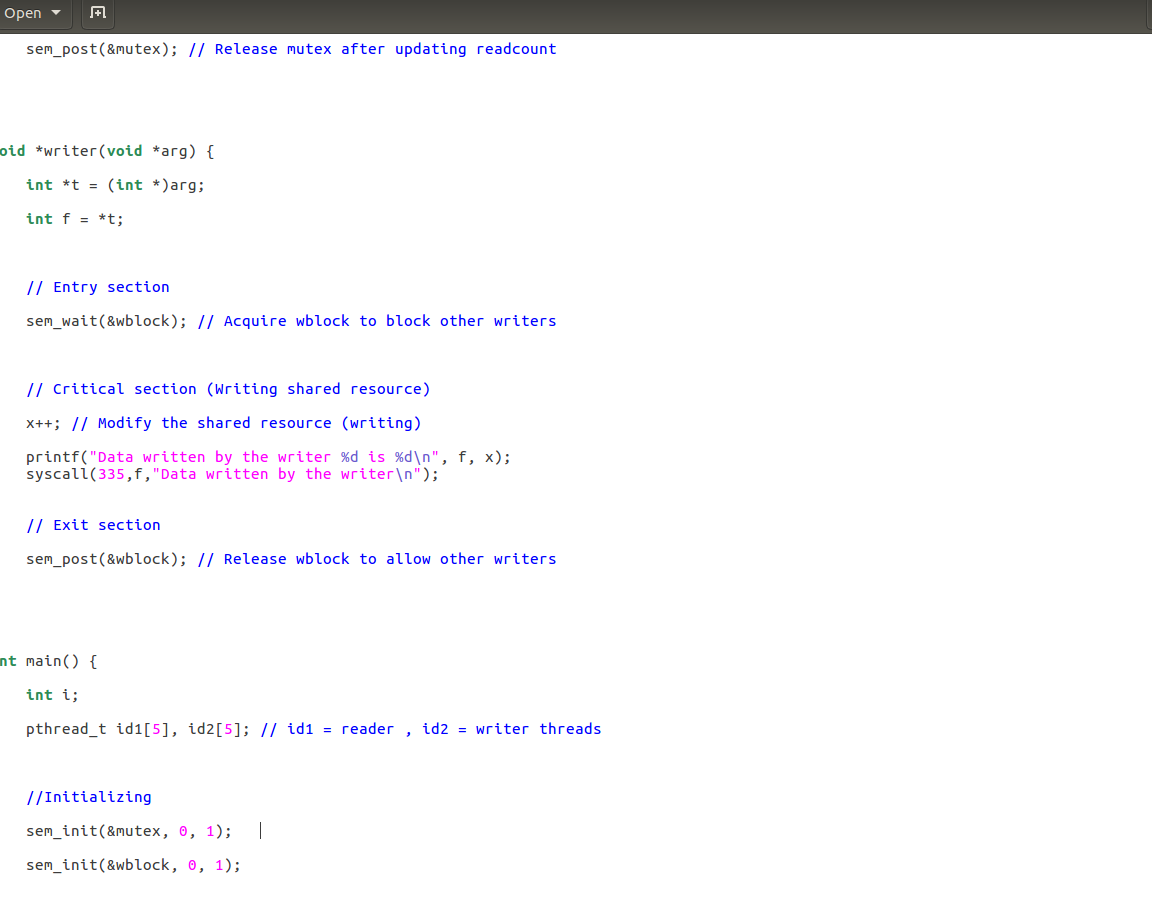
*}*

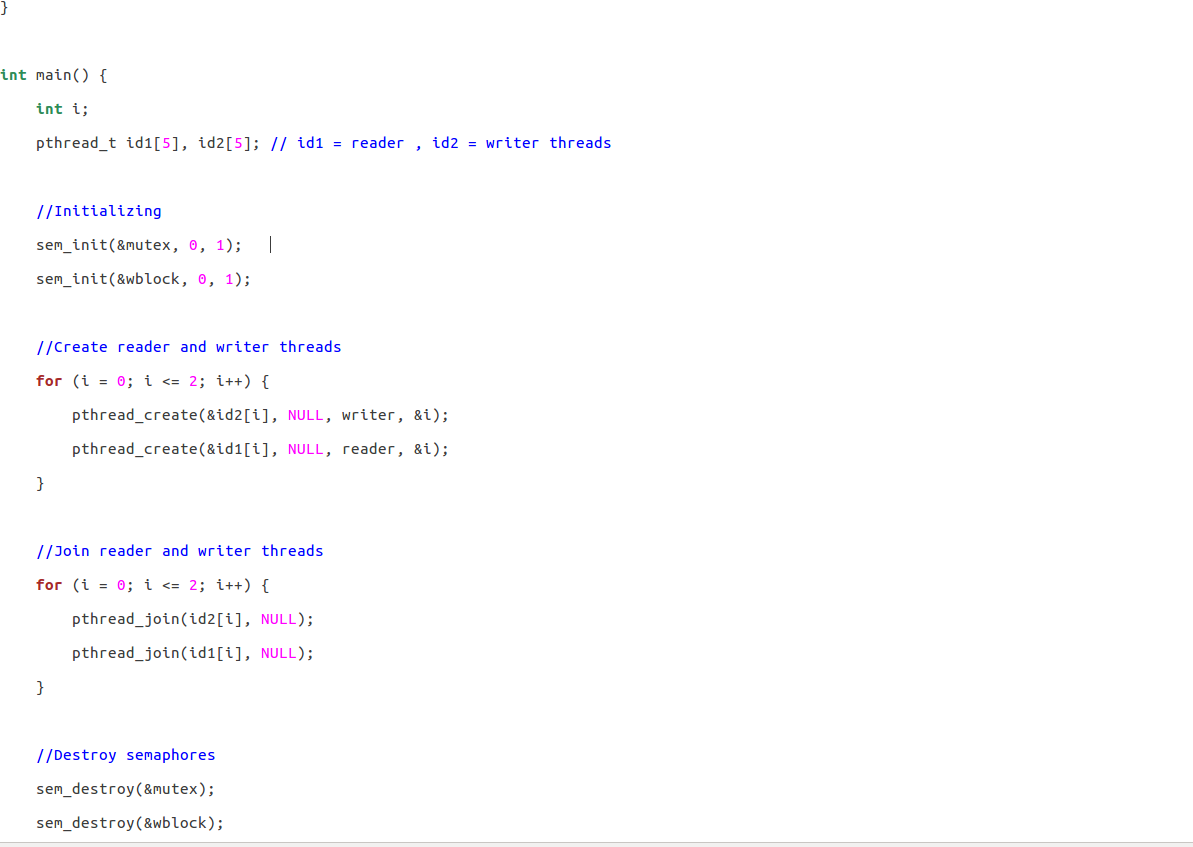
1. **Code explanation:**

This code addresses the Reader-Writer problem by utilizing threads and semaphores. The program creates multiple reader and writer threads that access a shared resource x. Readers acquire a mutex semaphore to update the readcount variable, which keeps track of the number of active readers. If a reader is the first to access the resource, it also acquires a writer exclusion semaphore to block writers. Readers then read the value of x and release the mutex semaphore. Upon exiting the critical section, readers acquire the mutex again to update the readcount and release the writer exclusion semaphore if they are the last reader. Writers, on the other hand, acquire the writer exclusion semaphore to block other writers, increment x, and release the semaphore. This synchronization ensures that multiple readers can access the resource simultaneously, but only one writer can modify it at a time, preventing data inconsistency.



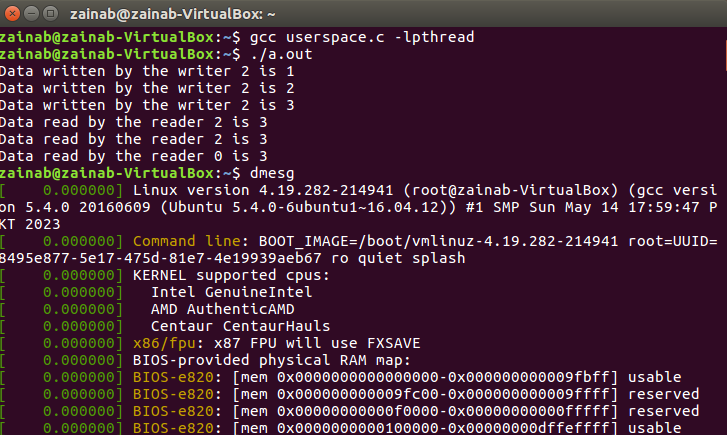


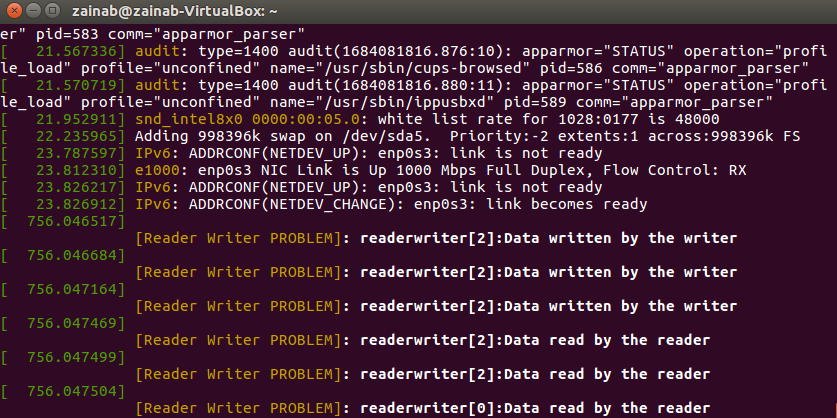




1. **Results:**

We compiled the code by typing **“gcc userspace.c -lpthread”** and execute it by typing **“./a.out”**. Then it displayed the working of reader and writer then we run **“dmesg”** to see the kernel messages and it displayed the following output





* **REAL TIME EXAPMLE:**

A practical example of a Readers and Writers problem is an airline reservation system consisting of a huge data base with many processes that read and write the data. Reading information from the data base will not cause a problem since no data is changed. The problem lies in writing information to the data base.

* **Project aspects in Future technology:**

1. **Performance Optimization**:

Future technology often introduces new optimization opportunities. Analyzing the code for potential bottlenecks and identifying areas for optimization can lead to improved performance. Techniques like cache optimization, algorithmic improvements, and reducing unnecessary synchronization overhead can enhance overall system efficiency.

1. **Parallelism and Multithreading:**

Future technology emphasizes utilizing the full potential of multi-core processors and parallel computing. Enhancing the code to leverage parallelism and multithreading can improve performance and efficiency. Techniques like thread-level parallelism, task-based parallelism, or utilizing specialized parallel processing frameworks can be explored.

* **Conclusion:**

In conclusion, the Reader-Writer project successfully addresses synchronization between multiple readers and writers accessing a shared resource. The code ensures mutual exclusion and maintains data consistency using threads and semaphores. Future enhancements can focus on scalability, performance optimization, integration with advanced technologies, and ensuring security and privacy. The project provides a foundation for further development and adaptation to future technological advancements.

* **REFRENCES:**

From Wikipedia:

* <https://en.m.wikipedia.org/wiki/Readers%E2%80%93writers_problem>
* <https://en.m.wikipedia.org/wiki/Semaphore_(programming)>