**National University of Computer & Emerging Sciences**

**Karachi Campus**



**Mutex Lock (System Call)**

**Project Report**

**Operating Systems**

**Section: 4F**

**Group Members:**

**21K-4656 Muhammad Talha**

**21K-4654 Ayush Kumar**

**21K-3413 Avinash**

**Submitted to: Sir Ghufran Ahmed**

**Title: Mutex Lock**

**Objective:**

The objective of the mutex lock system call project for readers-writers issue is to demonstrate how to use mutex locks in a Linux kernel module to synchronize access to shared resources between multiple readers and writers. The readers-writers problem is a classic synchronization problem in computer science that deals with coordinating access to a shared resource between multiple readers and writers.

**Introduction and Background:**

The concept of mutex locks is an essential aspect of operating systems and plays a crucial role in synchronizing access to shared resources. Operating systems use mutex locks to manage the access of multiple processes to shared resources such as files, databases, and other system resources. In this project, we aim to provide a comprehensive understanding of mutex locks and their implementation in operating systems.

We will be analyzing the different types of mutex locks and their characteristics, including spin locks, semaphores, and mutexes. We will also be examining the underlying mechanisms used in mutex locks, including test-and-set instructions and compare-and-swap operations.

Additionally, we will be implementing a mutex lock in a real-world scenario and testing its efficiency and performance. Overall, this project aims to provide a detailed understanding of mutex locks and their importance in modern operating systems. The insights gained from this project will be invaluable in developing efficient and robust systems that can handle concurrency effectively.

**Platform and Languages:**

The project at hand involves the implementation of mutex locks in operating systems using the C programming language. For the development of this project, we utilized the Linux operating system and its command-line interface as our primary development environment.

**Methodology:**

In this particular implementation using the C programming language, the methodology involves using two mutex locks, one for the readers and one for the writers. When a reader wants to read the shared variable, it locks the reader mutex, increments the count of active readers, and checks if it's the first reader. If it is, it locks the writer mutex to prevent any writer from accessing the shared variable. After reading the shared variable, it decrements the count of active readers and checks if there are no more readers, in which case it unlocks the writer mutex.

When a writer wants to write to the shared variable, it locks the writer mutex, and then it waits until there are no more readers or writers accessing the shared variable. Once it acquires the mutex, it increments the count of active writers and writes to the shared variable. After writing, it decrements the count of active writers and unlocks the writer mutex.

**Project Specifications:**

The primary goal of this project is to implement mutex locks in operating systems and test their performance in a real-world scenario. To achieve this goal, we have defined the following project specifications:

1. Functionality: The project will provide all necessary functionalities related to mutex locks, such as lock and unlock operations, blocking and non-blocking mutex locks, and support for multiple threads and processes.
2. Performance: We will conduct a comprehensive performance analysis of the implemented mutex locks to measure their efficiency, scalability, and robustness under different workloads.

**Description:**

The program has a shared integer variable, shared\_variable, which is accessed by multiple reader and writer threads. The readers only read the shared variable, while the writers modify it. The goal is to ensure that the readers and writers do not interfere with each other and access the shared variable in a synchronized manner.

The program creates two types of threads - reader threads and writer threads. The reader threads access the shared variable in a read-only mode, while the writer threads modify it.

The program creates NUM\_READERS reader threads and NUM\_WRITERS writer threads using the pthread\_create function. Each reader and writer thread executes its respective function - reader\_func or writer\_func.

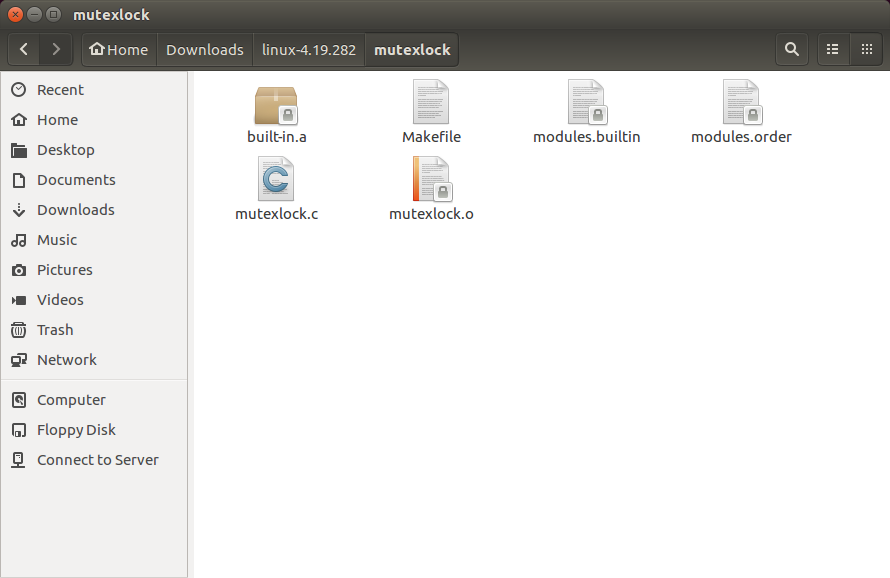
In the reader\_func, a reader thread tries to acquire a mutex lock on the shared resource before reading the shared variable. If a writer is already modifying the shared variable, then the reader thread waits until the writer releases the mutex lock. After the reader reads the shared variable, it releases the mutex lock.

In the writer\_func, a writer thread tries to acquire a mutex lock on the shared resource before modifying the shared variable. If any other thread is holding the mutex lock, the writer waits until it is released. After modifying the shared variable, the writer releases the mutex lock.

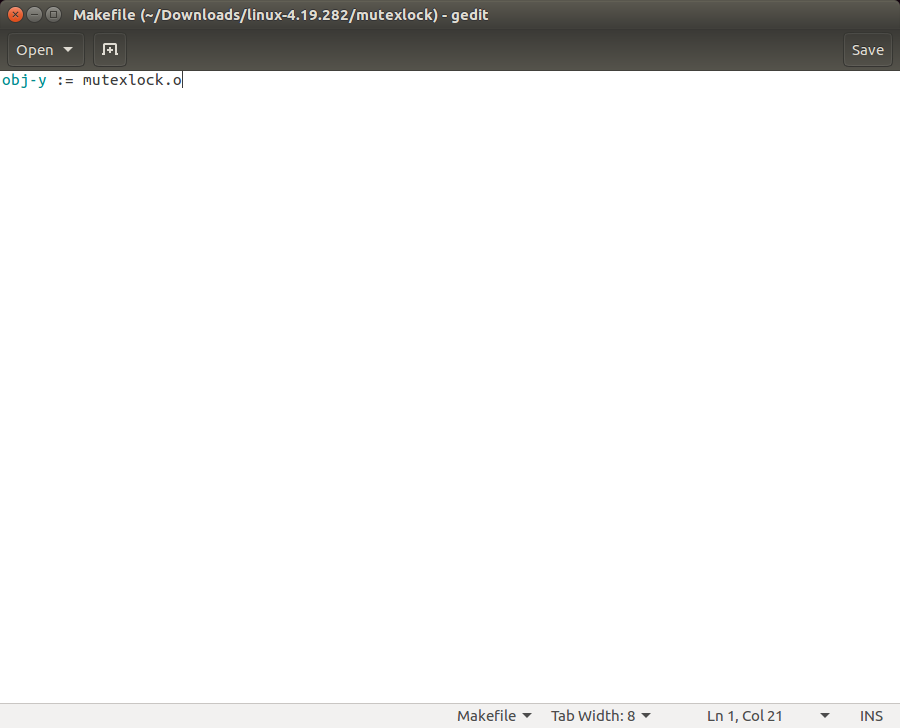
The program uses two mutex locks - mutex\_lock and writers\_lock to ensure synchronization between the reader and writer threads. The program uses the pthread\_mutex\_lock and pthread\_mutex\_unlock functions to acquire and release the mutex locks.

Finally, the program waits for all the reader and writer threads to finish executing using the pthread\_join function. The program then destroys the mutex locks and exits.

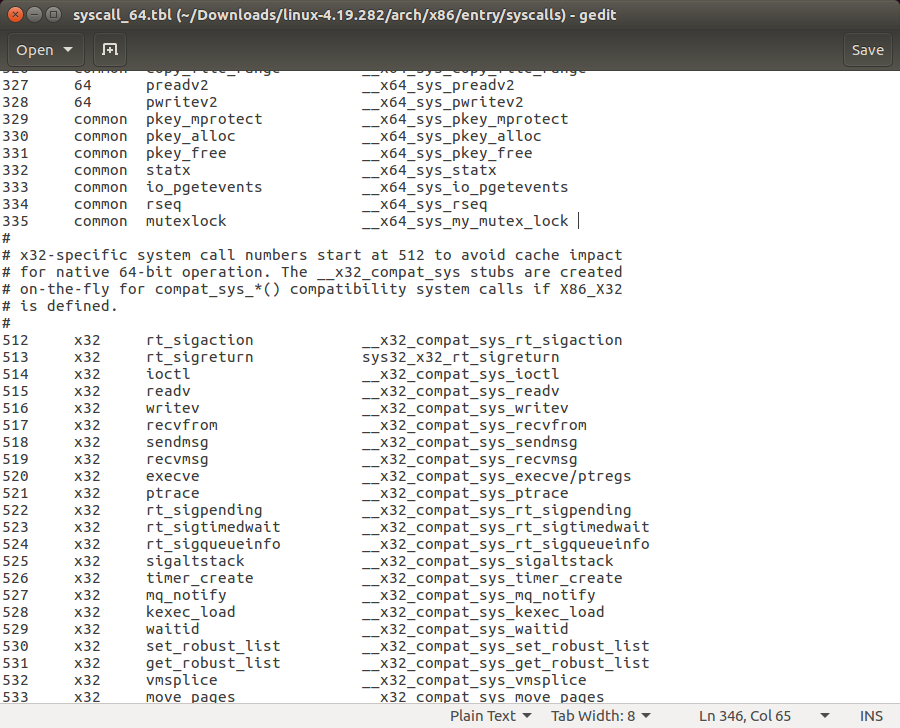
**Testing:**

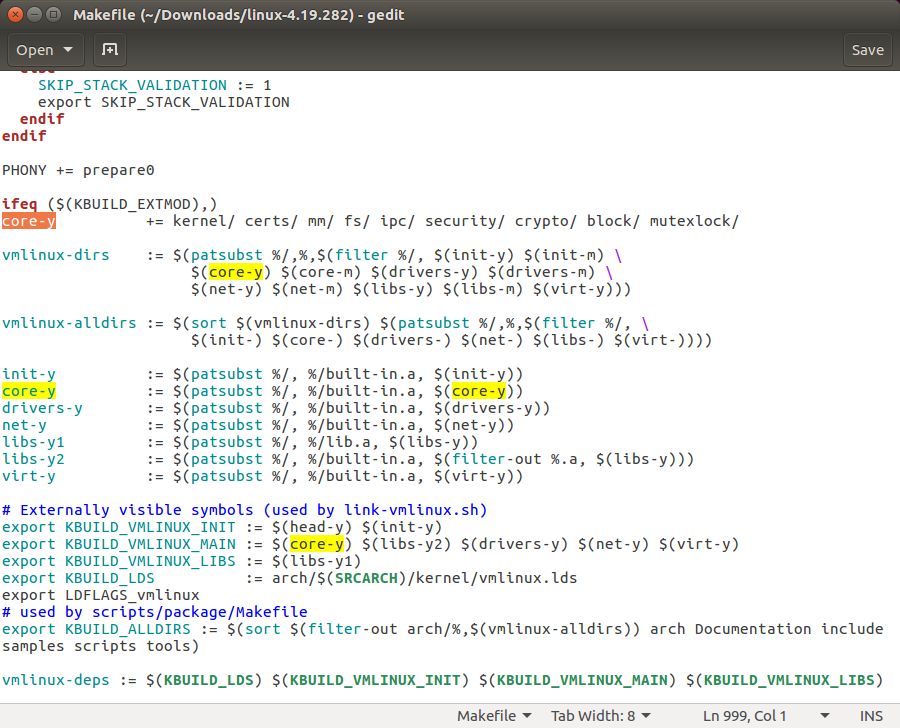


**Contents of “MakeFile”:**

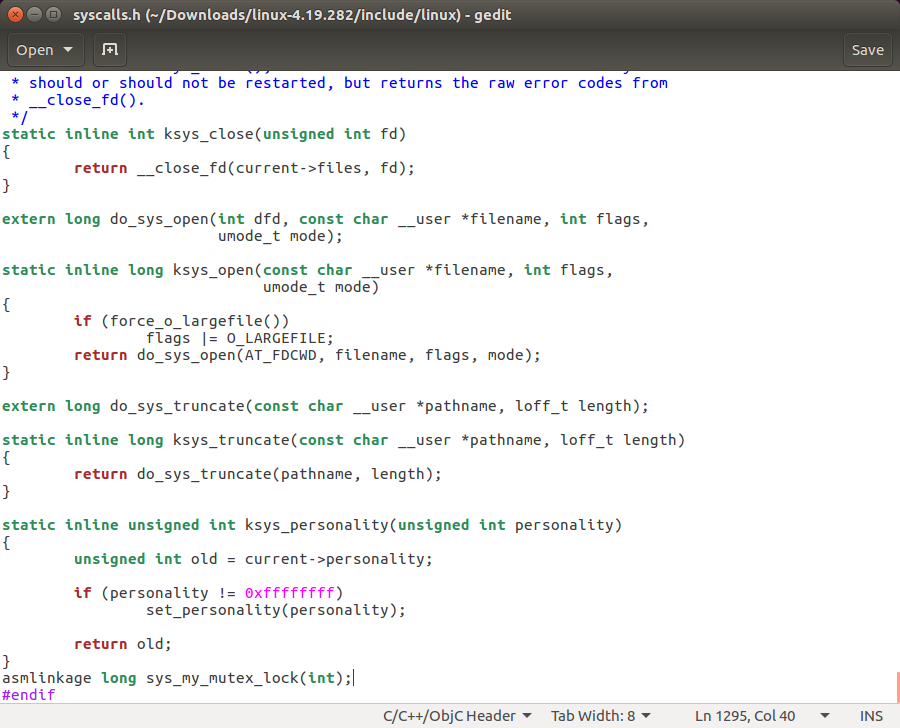


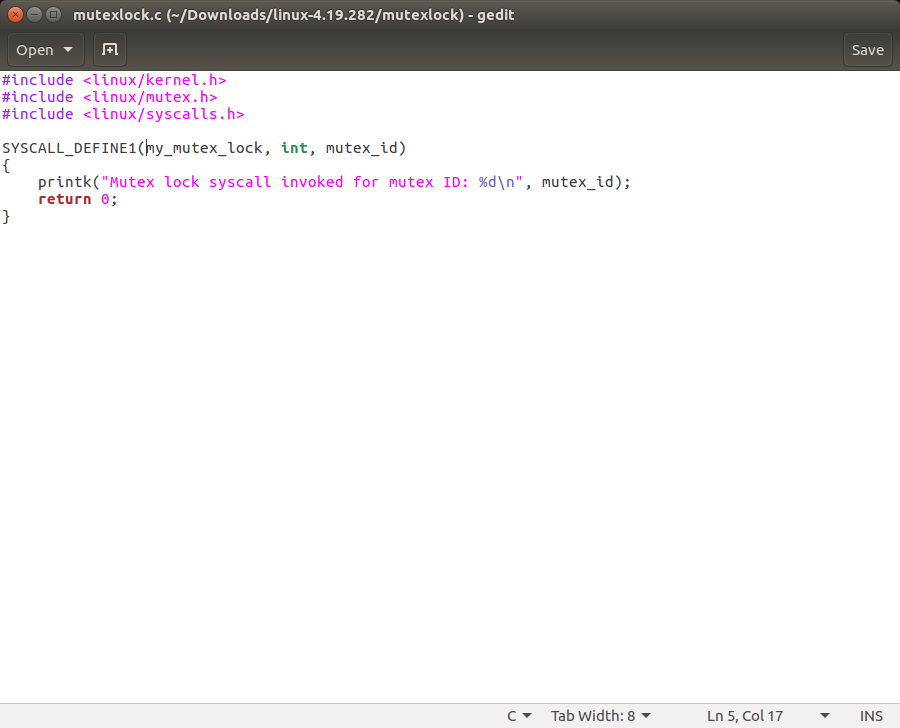
**Adding a new System call (line no. 335):**

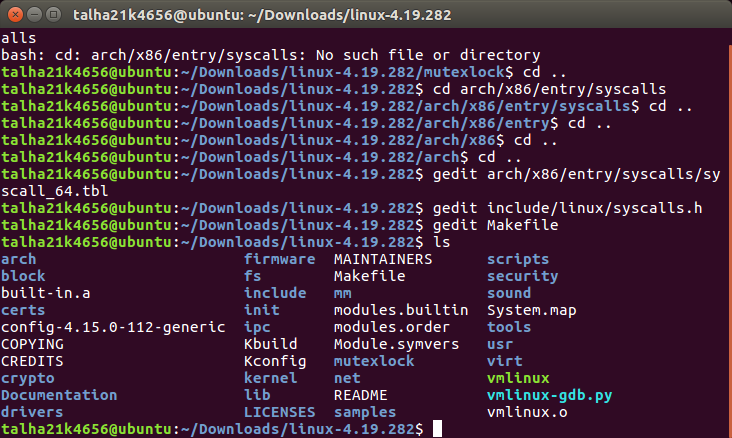
****

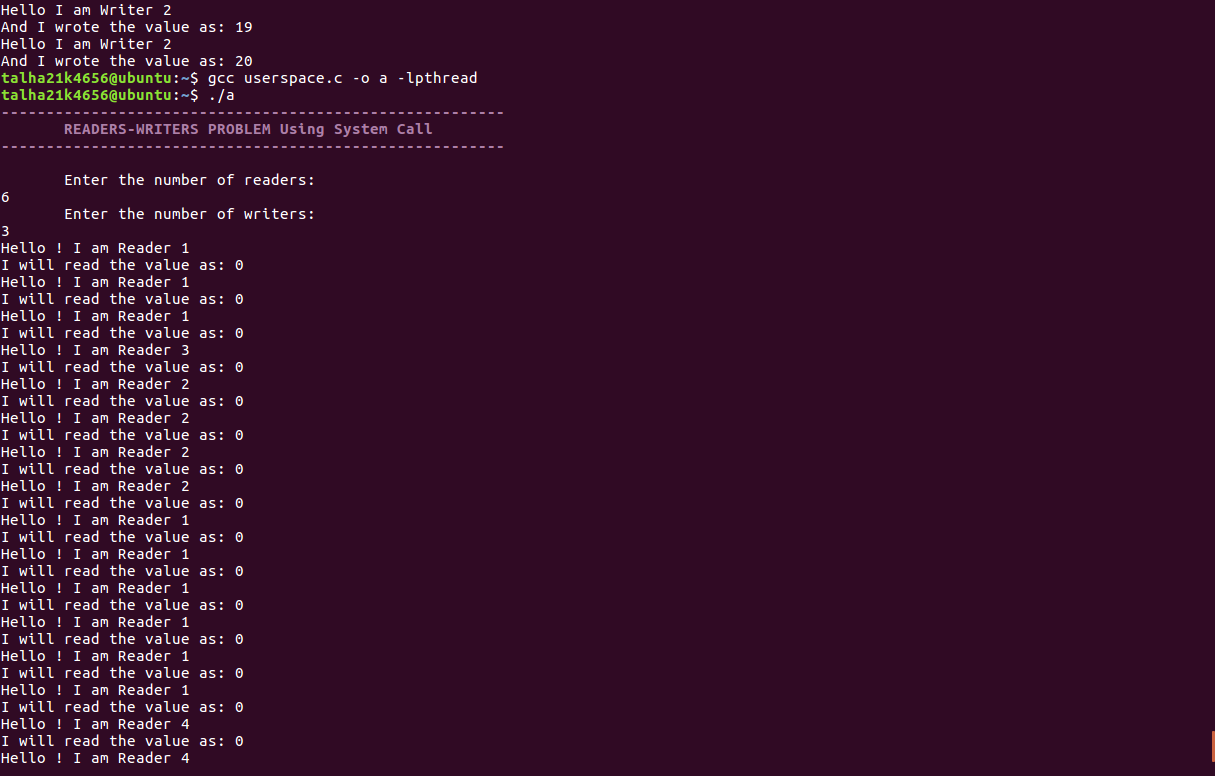
****

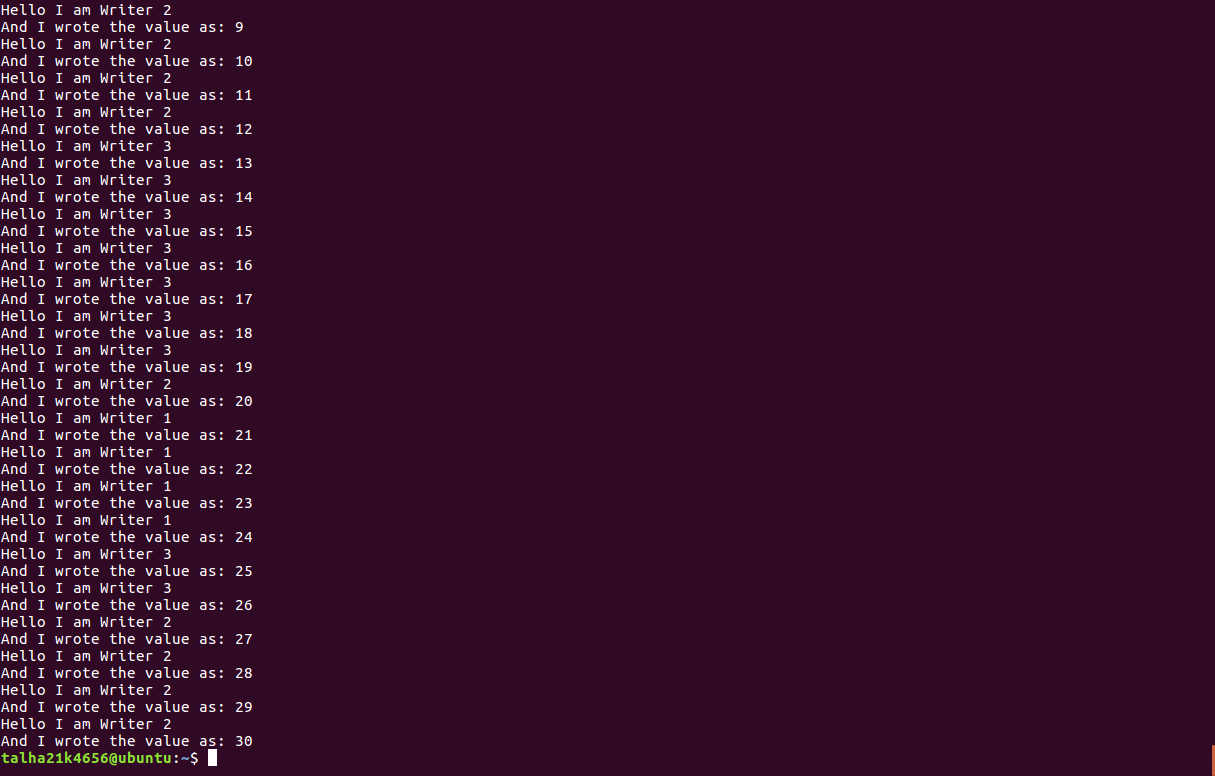
**The ‘asmlinkage’ function declaration:**

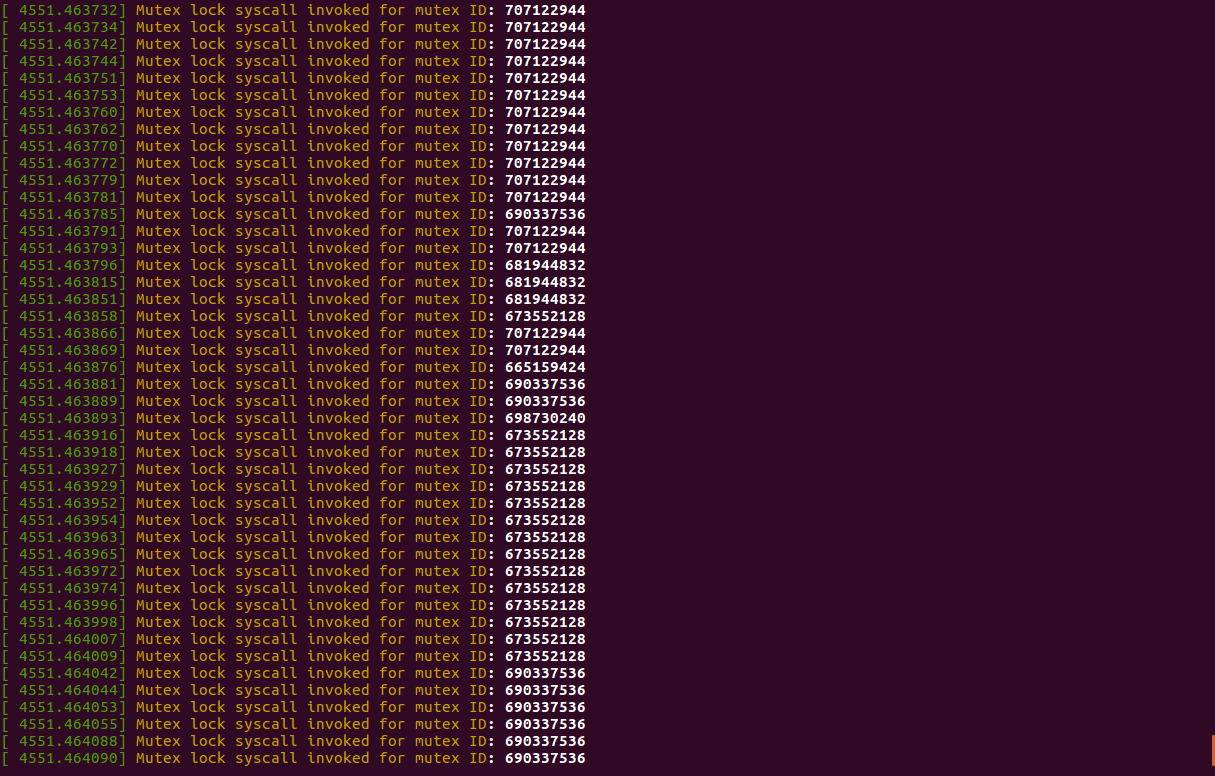


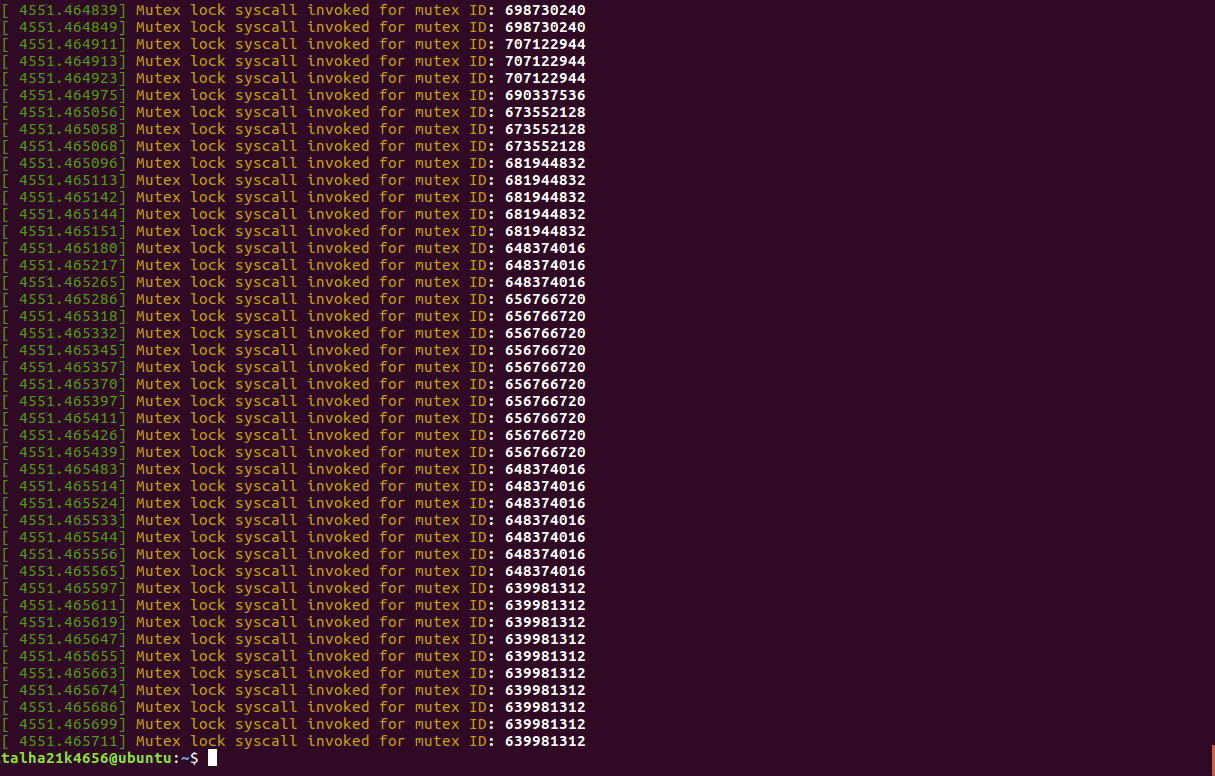












**Conclusion:**

In conclusion, the implementation of mutex locks through system call in the reader-writer problem has proven to be an effective approach to ensure mutual exclusion and synchronization between multiple threads. The reader-writer problem involves maintaining the correct balance between read and write operations, while preventing conflicting access and preserving data integrity.

While mutex locks provide an effective solution to the reader-writer problem, it is essential to consider potential drawbacks. Excessive lock contention can lead to performance degradation, as threads may be forced to wait for access to the shared resource even when there is no conflict. Therefore, it is crucial to strike a balance between synchronization and performance by minimizing the time spent holding locks and maximizing concurrency where possible.

In summary, the implementation of mutex locks in the reader-writer problem has proven to be a reliable and efficient approach for achieving synchronization and mutual exclusion. By carefully managing access to shared resources, mutex locks ensure data integrity, prevent conflicts, and maintain fairness in thread access.