Assignment No. 04(B)

Date:

TITLE: Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader-Writer problem with reader priority.

OBJECTIVE:

- Study the concepts of thread synchronization and mutual exclusion using mutex in Linux.
- Learn to implement mutual exclusion with thread synchronization in the Linux kernel.

SOFTWARE REQUIREMENTS:

1. Ubuntu 16.04
2. GNU C Compiler

THEORY:

Readers writer problem is example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource.

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one mutex **m** and a semaphore **w**. An integer variable **read_count** is used to maintain the number of readers currently accessing the resource. The variable **read_count** is initialized to 0. A value of 1 is given initially to **m** and **w**.

Instead of having the process to acquire lock on the shared resource, we use the mutex **m** to make the process to acquire and release lock whenever it is updating the **read_count** variable.

The code for the writer process looks like this:

```
while(TRUE) {
   wait(w);
   /*perform the write operation */
   signal(w);
}
```

The code for the reader process looks like this:

```
while(TRUE) {
   wait(m);   //acquire lock read_count++;
   if(read_count == 1)
          wait(w);
   signal(m);  //release lock
   /* perform the reading operation */
   wait(m);   // acquire lock
   read_count--;
   if(read_count == 0)
          signal(w);
   signal(m);  // release lock
}
```

Code Explained:

- ❿ As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- ❿ After performing the write operation, it increments **w** so that the next writer can access the resource.
- ❿ On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.
- ❿ When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- ❿ The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- ❿ The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- ❿ Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

Conclusion: Thus we have learned to implement mutual exclusion with thread synchronization in the Linux kernel and demonstrated Reader-Writer problem with reader priority.