Readers-Writers Problem

(With Writer Priority)

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

Problem Statement:

"A database is shared among some processes, a few of which have the right to write also. Any writer has priority over readers. The system will belong to the writer/s whenever any writer is ready."

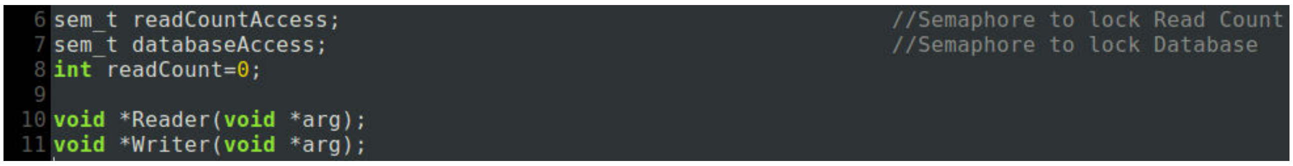
In others words its the Second Readers-Writers Problem.

Second readers-writers problem:

The constraint is added that no writer, once added to the queue, shall be kept waiting longer than absolutely necessary. This is also called writers-preference.

Second readers-writers problem's solution:

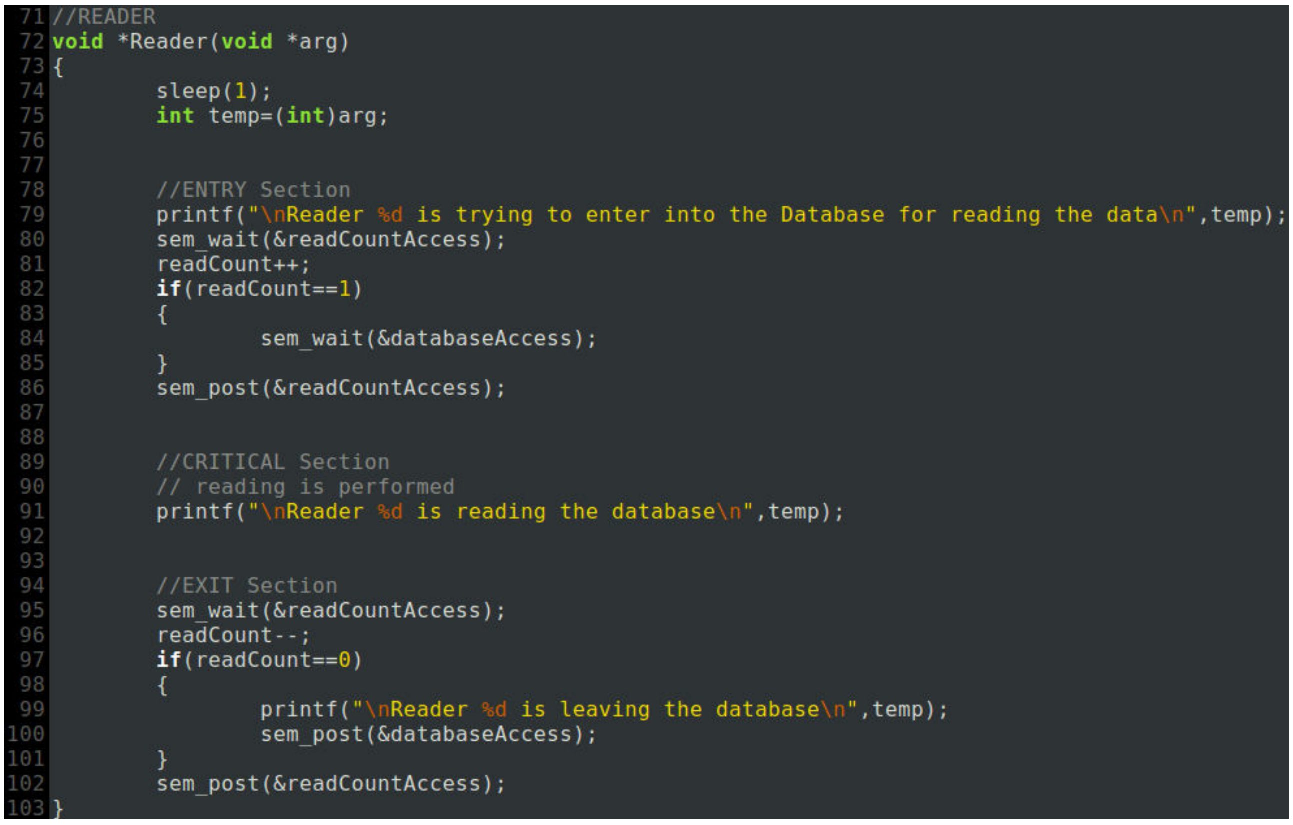
The solution to first readers-writers problem is suboptimal, because it is possible that a reader might have the lock, a writer is waiting for the lock, and then a reader requests access. It would be unfair for next reader to jump in immediately, ahead of the Writer; if that happened often enough,the Writer would starve. Instead, the Writer should start as soon as possible. This is the motivation for the second readers-writers problem.



```
sem_t readCountAccess;                                          //Semaphore to lock Read Count
sem_t databaseAccess;                                           //Semaphore to lock Database
int readCount=0;

void *Reader(void *arg);
void *Writer(void *arg);
```

The Reader Function:



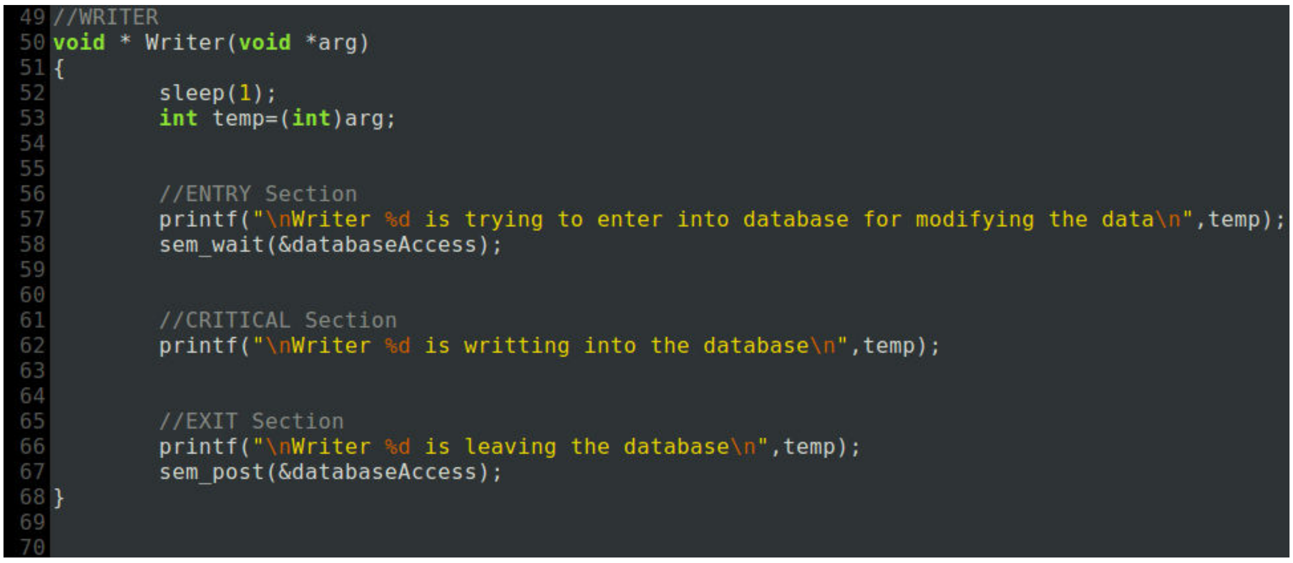
```c
//READER
void *Reader(void *arg)
{
        sleep(1);
        int temp=(int)arg;


        //ENTRY Section
        printf("\nReader %d is trying to enter into the Database for reading the data\n",temp);
        sem_wait(&readCountAccess);
        readCount++;
        if(readCount==1)
        {
                sem_wait(&databaseAccess);
        }
        sem_post(&readCountAccess);


        //CRITICAL Section
        // reading is performed
        printf("\nReader %d is reading the database\n",temp);


        //EXIT Section
        sem_wait(&readCountAccess);
        readCount--;
        if(readCount==0)
        {
                printf("\nReader %d is leaving the database\n",temp);
                sem_post(&databaseAccess);
        }
        sem_post(&readCountAccess);
}
```

The Writer Function:



```c
//WRITER
void * Writer(void *arg)
{
        sleep(1);
        int temp=(int)arg;


        //ENTRY Section
        printf("\nWriter %d is trying to enter into database for modifying the data\n",temp);
        sem_wait(&databaseAccess);


        //CRITICAL Section
        printf("\nWriter %d is writting into the database\n",temp);


        //EXIT Section
        printf("\nWriter %d is leaving the database\n",temp);
        sem_post(&databaseAccess);
}


```

In this solution, preference is given to the writers. This is accomplished by forcing every reader to lock and release the readtry semaphore individually. The writers on the other hand don't need to lock it individually. Only the first writer will lock the readtry and then all subsequent writers can simply use the resource as it gets freed by the previous writer. The very last writer must release the readtry semaphore, thus opening the gate for readers to try reading.

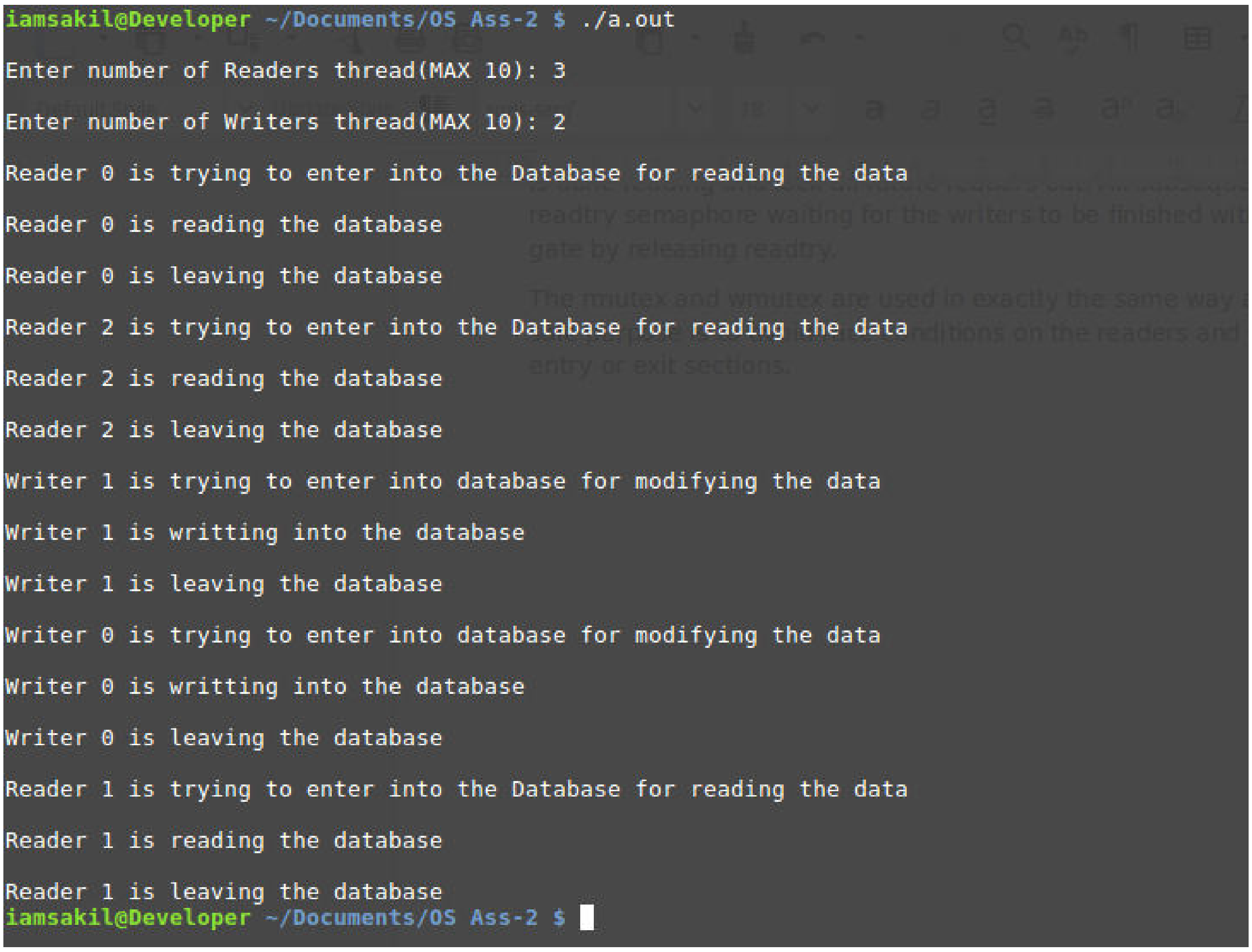
No reader can engage in the entry section if the readtry semaphore has been set by a writer previously. The reader must wait for the last writer to unlock the resource and readtry semaphores. On the other hand, if a particular reader has locked the readtry semaphore, this will indicate to any potential concurrent writer that there is a reader in the entry section. So the writer will wait for the reader to release the readtry and then the writer will immediately lock it for itself and all subsequent writers. However, the writer will not be able to access the resource until the current reader has released the resource, which only occurs after the reader is finished with the resource in the critical section.

The resource semaphore can be locked by both the writer and the reader in their entry section. They are only able to do so after first locking the readtry semaphore, which can only be done by one of them at a time.

If there are no writers wishing to get to the resource, as indicated to the reader by the status of the readtry semaphore, then the readers will not lock the resource. This is done to allow a writer to immediately take control over the resource as soon as the current reader is finished reading. Otherwise, the writer would need to wait for a queue of readers to be done before the last one can unlock the readtry semaphore. As soon as a writer shows up, it will try to set the readtry and hang up there waiting for the current reader to release the readtry. It will then take control over the resource as soon as the current reader is done reading and lock all future readers out. All subsequent readers will hang up at the readtry semaphore waiting for the writers to be finished with the resource and to open the gate by releasing readtry.

The rmutex and wmutex are used in exactly the same way as in the first solution. Their sole purpose is to avoid race conditions on the readers and writers while they are in their entry or exit sections.

The Output:



```
iamsakil@Developer ~/Documents/OS Ass-2 $ ./a.out

Enter number of Readers thread(MAX 10): 3

Enter number of Writers thread(MAX 10): 2

Reader 0 is trying to enter into the Database for reading the data

Reader 0 is reading the database

Reader 0 is leaving the database

Reader 2 is trying to enter into the Database for reading the data

Reader 2 is reading the database

Reader 2 is leaving the database

Writer 1 is trying to enter into database for modifying the data

Writer 1 is writting into the database

Writer 1 is leaving the database

Writer 0 is trying to enter into database for modifying the data

Writer 0 is writting into the database

Writer 0 is leaving the database

Reader 1 is trying to enter into the Database for reading the data

Reader 1 is reading the database

Reader 1 is leaving the database
iamsakil@Developer ~/Documents/OS Ass-2 $
```

By:
SAKIL MALLICK
B.CSE-III
ROLL-001510501050