<u>Lab 8</u> (Part 2 - Brownie Points)

Team Members :
Rounak Das – SE20UCSE149
Nikhita Rapolu – SE20UCSE115
Neethu Vangapalli – SE20UCSE110
Niharika Kakumanu – SE20UCSE114
Kartik MVS – SE20UCSE040
Mrinmoy Das – SE20UCSE101
Nitya Gaddala – SE20UCSE117

<u>Reader & Writer Problem …</u>

reader_writer.c - Lab08 - Visual Studio Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>

int count = 0;
int times = 1000;

sem_t mutex, db;
int read_count = 0;

void *reader(void *arg)
{
    for (int i = 0; i < times; i++)
    {
        // Entry Section
        sem_wait(&mutex);
        read_count++;
        if (read_count == 1)
        {
            sem_wait(&db);
        }
        sem_post(&mutex);

        // READING IS PERFORMED : CRITICAL SECTION
        printf("READING .... \n Number of readers, right now is : %d \n", read_count);
        // sleep(1);

        // EXIT SECTION
        sem_wait(&mutex);
        read_count--;
        if (read_count == 0)
            sem_post(&db);
        sem_post(&mutex);
    }
}

void *writer(void *arg)
{
    for (int i = 0; i < times; i++)
    {
        // Entry Section
        sem_wait(&db);

        // Writing is performed ... Critical Section
        printf("WRITING .... \n Number of readers, right now is : %d \n", read_count);
        // sleep(1);
```

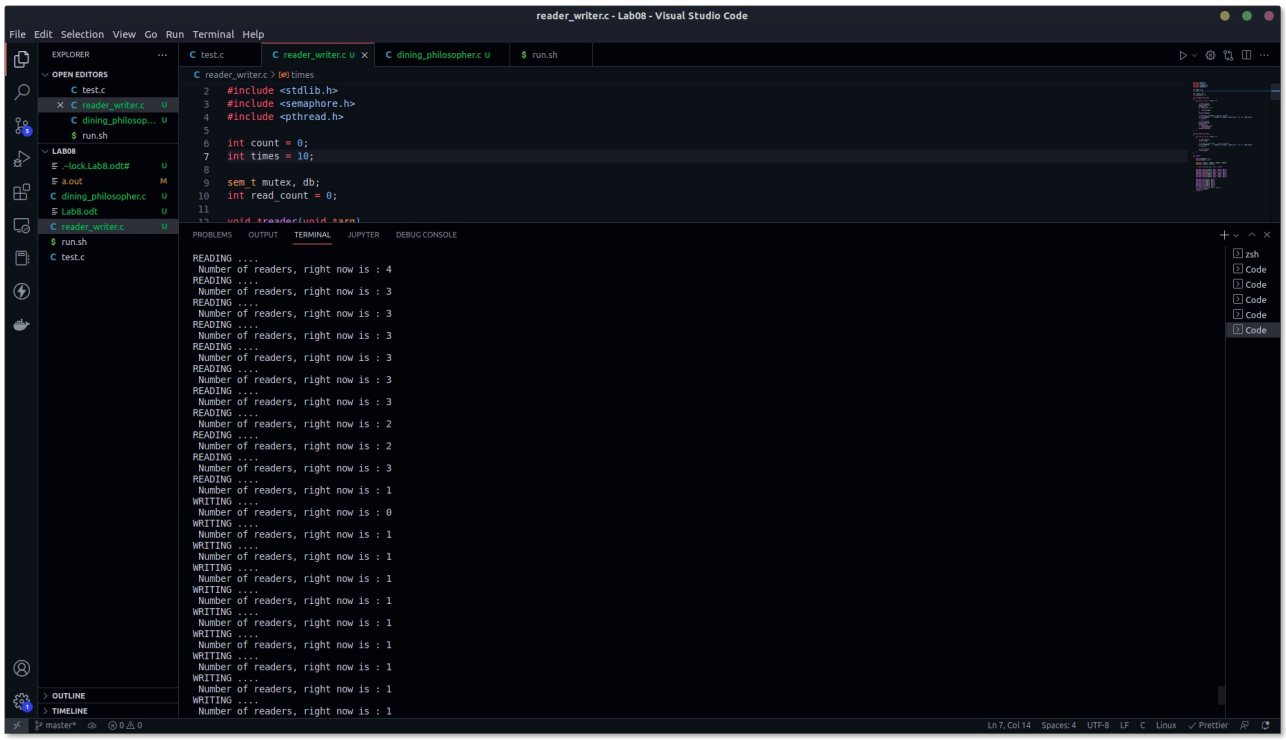
reader_writer.c - Lab08 - Visual Studio Code

```c
    }
}

void *writer(void *arg)
{
    for (int i = 0; i < times; i++)
    {
        // Entry Section
        sem_wait(&db);

        // Writing is performed ... Critical Section
        printf("WRITING .... \n Number of readers, right now is : %d \n", read_count);
        // sleep(1);

        // Exit section
        sem_post(&db);
    }
}

int main()
{
    sem_init(&mutex, 0, 1);
    sem_init(&db, 0, 1);

    pthread_t reader1, reader2, reader3, reader4;
    pthread_t writer1, writer2;

    // printf("Initialcount: %d\n", count);

    pthread_create(&reader1, NULL, reader, NULL);
    pthread_create(&reader2, NULL, reader, NULL);
    pthread_create(&reader3, NULL, reader, NULL);
    pthread_create(&reader4, NULL, reader, NULL);

    pthread_create(&writer1, NULL, writer, NULL);
    pthread_create(&writer2, NULL, writer, NULL);

    pthread_join(reader1, NULL);
    pthread_join(reader2, NULL);
    pthread_join(reader3, NULL);
    pthread_join(reader4, NULL);
    pthread_join(writer1, NULL);
    pthread_join(writer2, NULL);
    // printf("Finalcount:%d\n", count);
    pthread_exit(NULL);
    return 0;
}
```

Created two writers and four readers. Simulated the threads for 1000 times. R-R allowed, W-W not allowed, R-W not allowed. That's why the db mutex.

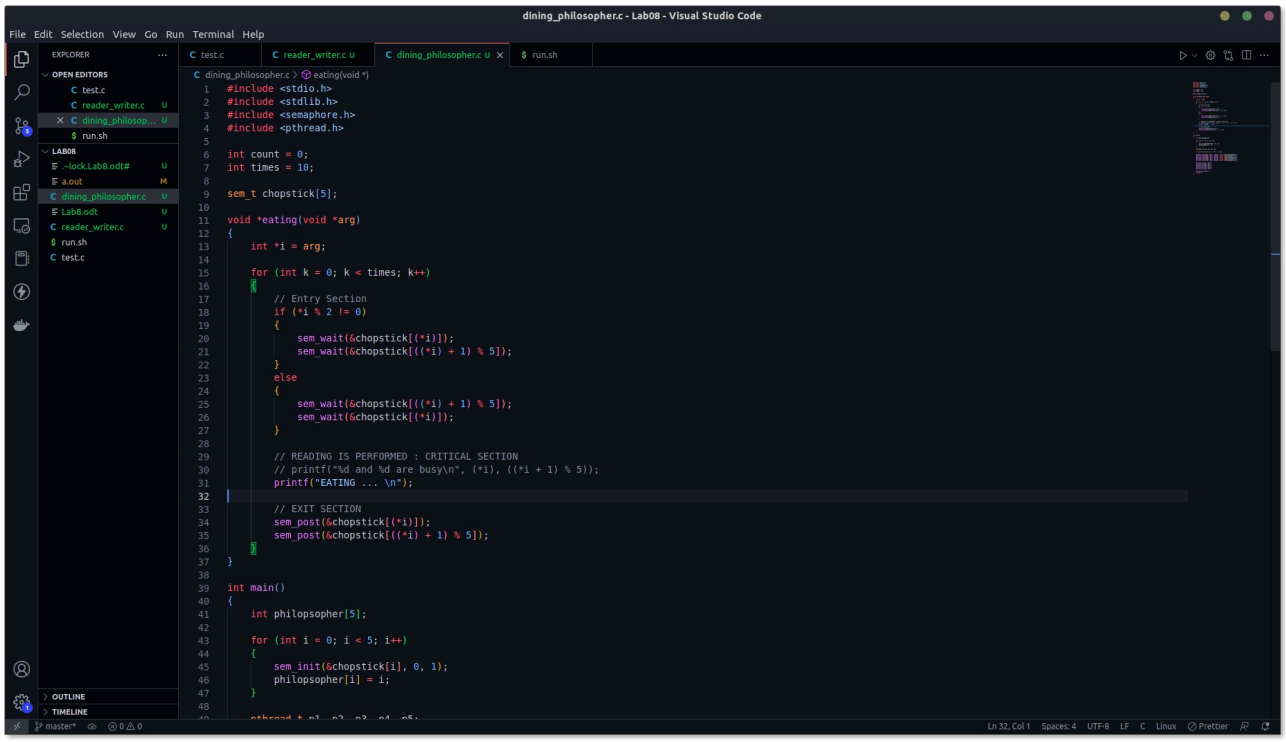
(**Elaborate these from the book, I used the algorithm same as book almost.)**

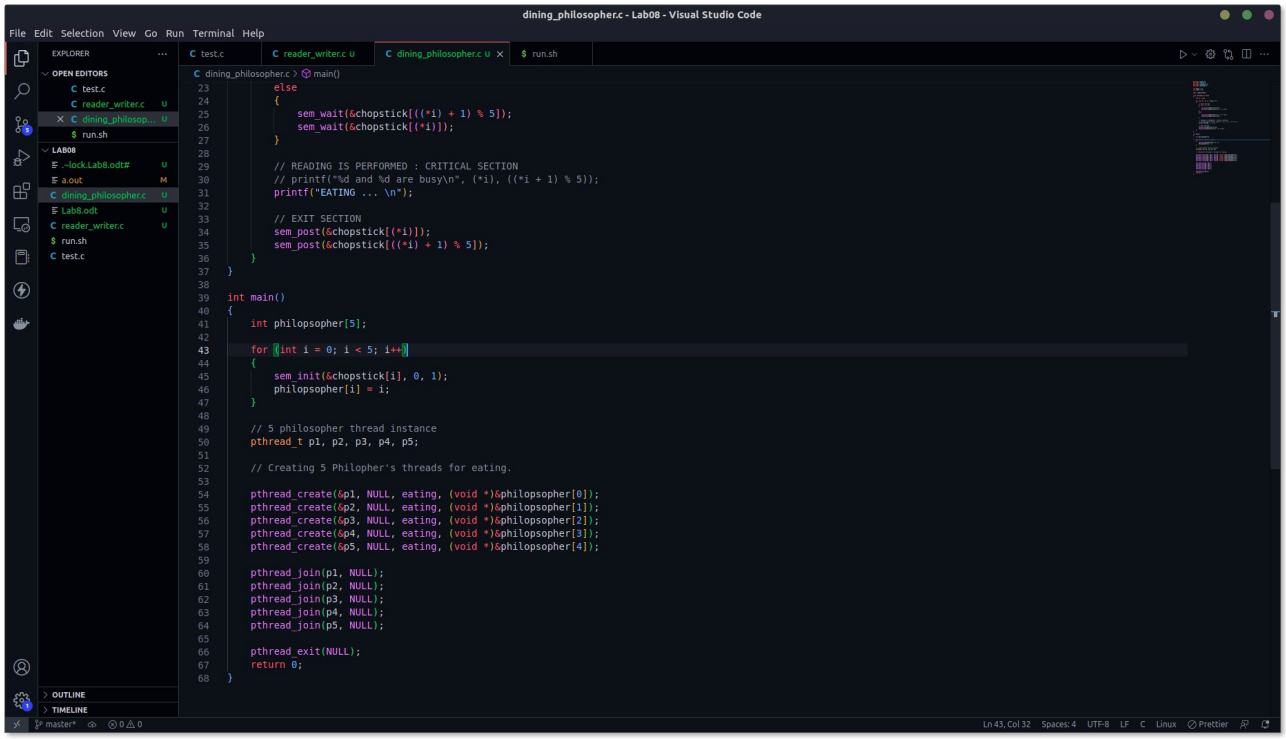
Output :



As we can see from the above output, when one reader is reading, several other readers are also allowed to exist in the critical section. But, when writer is there is it's critical section, no reader or any other writer is allowed to execute in it's critical section.

Dining Philosophers' problem





Here index inside the function eating is 'i' . So, ith philosopher takes ith chopstick and (i+1)%5 th chopstick. The modulus sign shows that the philosophers are sitting in a round table.

To avoid deadlock situation, even indexed philosopher picks up right chopstick first and then left chopstick. Odd indexed philosophers do it the other way round. This helps to avoid deadlock.