

Lab 8 (Part 2 - Brownie Points)

Team Members :

Rounak Das – SE20UCSE149

Nikhita Rapolu – SE20UCSE115

Neethu Vangapalli – SE20UCSE110

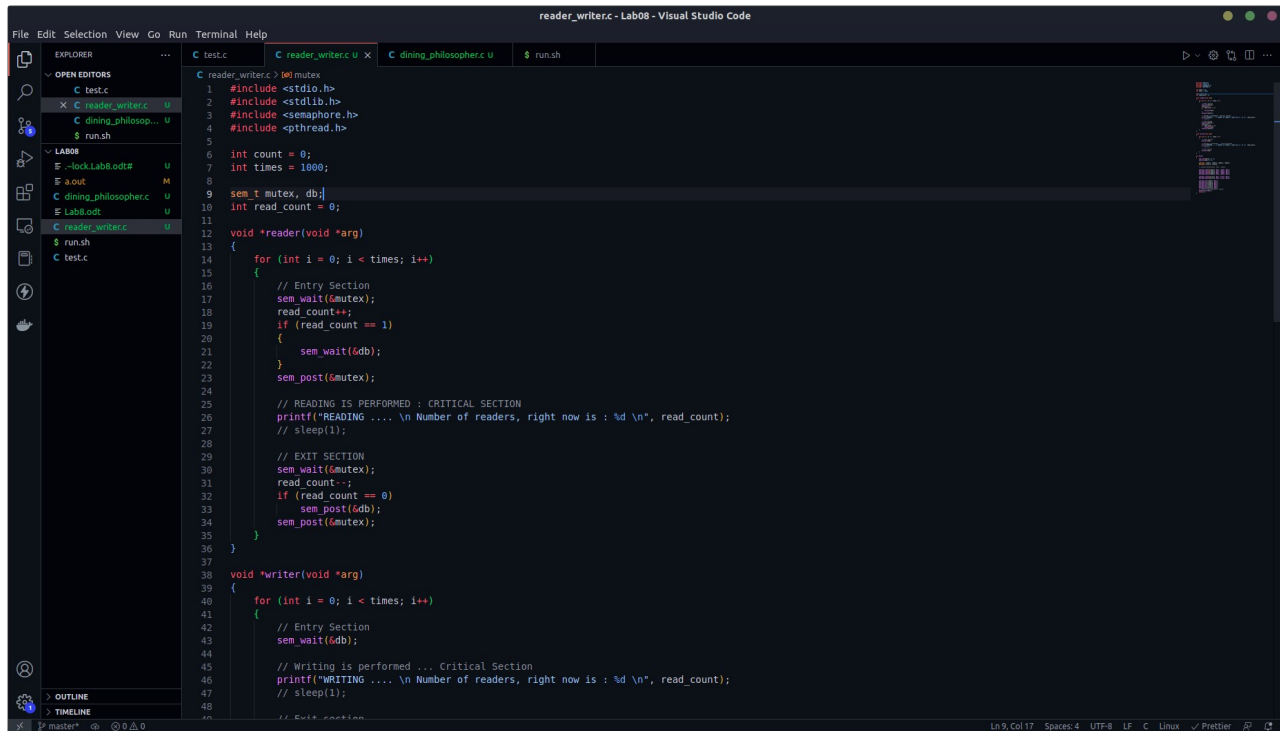
Niharika Kakumanu – SE20UCSE114

Kartik MVS – SE20UCSE040

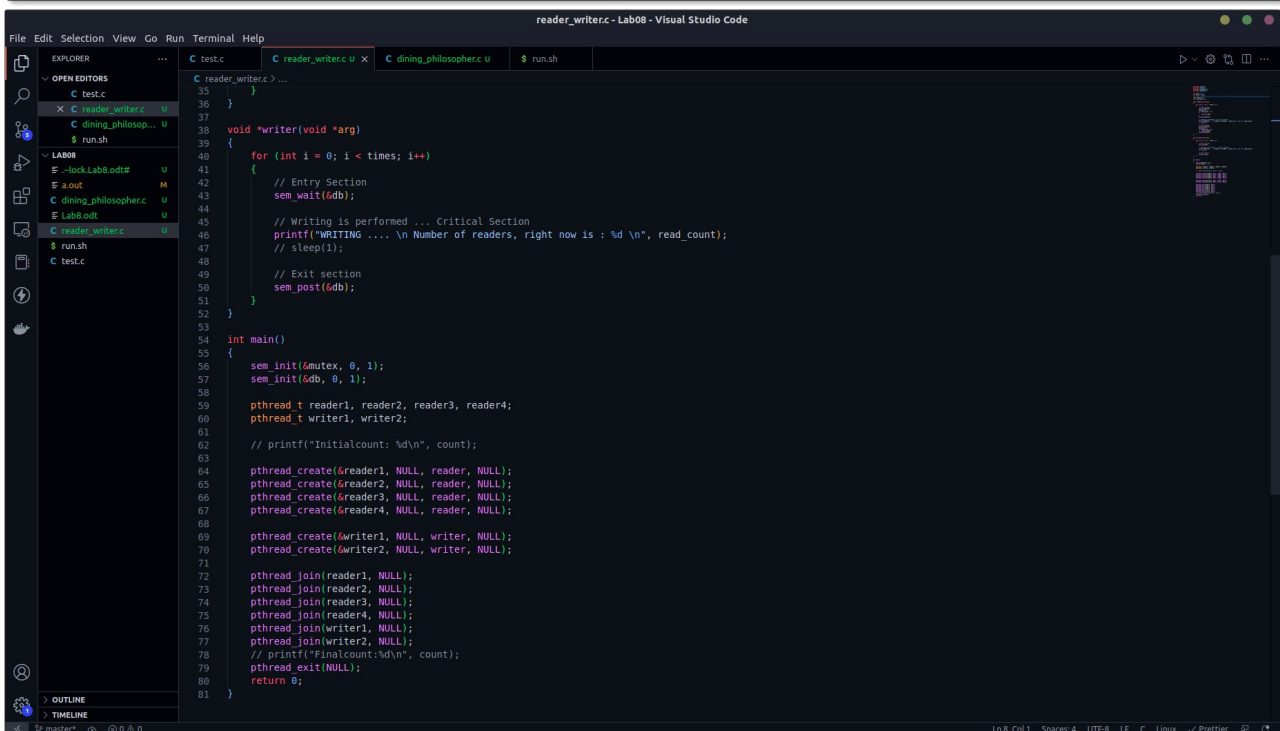
Mrinmoy Das – SE20UCSE101

Nitya Gaddala – SE20UCSE117

Reader & Writer Problem ...



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <semaphore.h>
4 #include <pthread.h>
5
6 int count = 0;
7 int times = 1000;
8
9 sem_t mutex, db;
10 int read_count = 0;
11
12 void *reader(void *arg)
13 {
14     for (int i = 0; i < times; i++)
15     {
16         // Entry Section
17         sem_wait(&mutex);
18         read_count++;
19         if (read_count == 1)
20         {
21             sem_wait(&db);
22         }
23         sem_post(&mutex);
24
25         // READING IS PERFORMED : CRITICAL SECTION
26         printf("READING .... \n Number of readers, right now is : %d \n", read_count);
27         // sleep(1);
28
29         // EXIT SECTION
30         sem_wait(&mutex);
31         read_count--;
32         if (read_count == 0)
33         {
34             sem_post(&db);
35             sem_post(&mutex);
36         }
37     }
38
39 void *writer(void *arg)
40 {
41     for (int i = 0; i < times; i++)
42     {
43         // Entry Section
44         sem_wait(&db);
45
46         // Writing is performed ... Critical Section
47         printf("WRITING .... \n Number of readers, right now is : %d \n", read_count);
48         // sleep(1);
49
50         // Exit section
51         sem_post(&db);
52     }
53 }
```



```
35 }
36 }
37
38 void *writer(void *arg)
39 {
40     for (int i = 0; i < times; i++)
41     {
42         // Entry Section
43         sem_wait(&db);
44
45         // Writing is performed ... Critical Section
46         printf("WRITING .... \n Number of readers, right now is : %d \n", read_count);
47         // sleep(1);
48
49         // Exit section
50         sem_post(&db);
51     }
52 }
53
54 int main()
55 {
56     sem_init(&mutex, 0, 1);
57     sem_init(&db, 0, 1);
58
59     pthread_t reader1, reader2, reader3, reader4;
60     pthread_t writer1, writer2;
61
62     // printf("Initialcount: %d\n", count);
63
64     pthread_create(&reader1, NULL, reader, NULL);
65     pthread_create(&reader2, NULL, reader, NULL);
66     pthread_create(&reader3, NULL, reader, NULL);
67     pthread_create(&reader4, NULL, reader, NULL);
68
69     pthread_create(&writer1, NULL, writer, NULL);
70     pthread_create(&writer2, NULL, writer, NULL);
71
72     pthread_join(reader1, NULL);
73     pthread_join(reader2, NULL);
74     pthread_join(reader3, NULL);
75     pthread_join(reader4, NULL);
76     pthread_join(writer1, NULL);
77     pthread_join(writer2, NULL);
78     // printf("Finalcount: %d\n", count);
79     pthread_exit(NULL);
80     return 0;
81 }
```

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.)

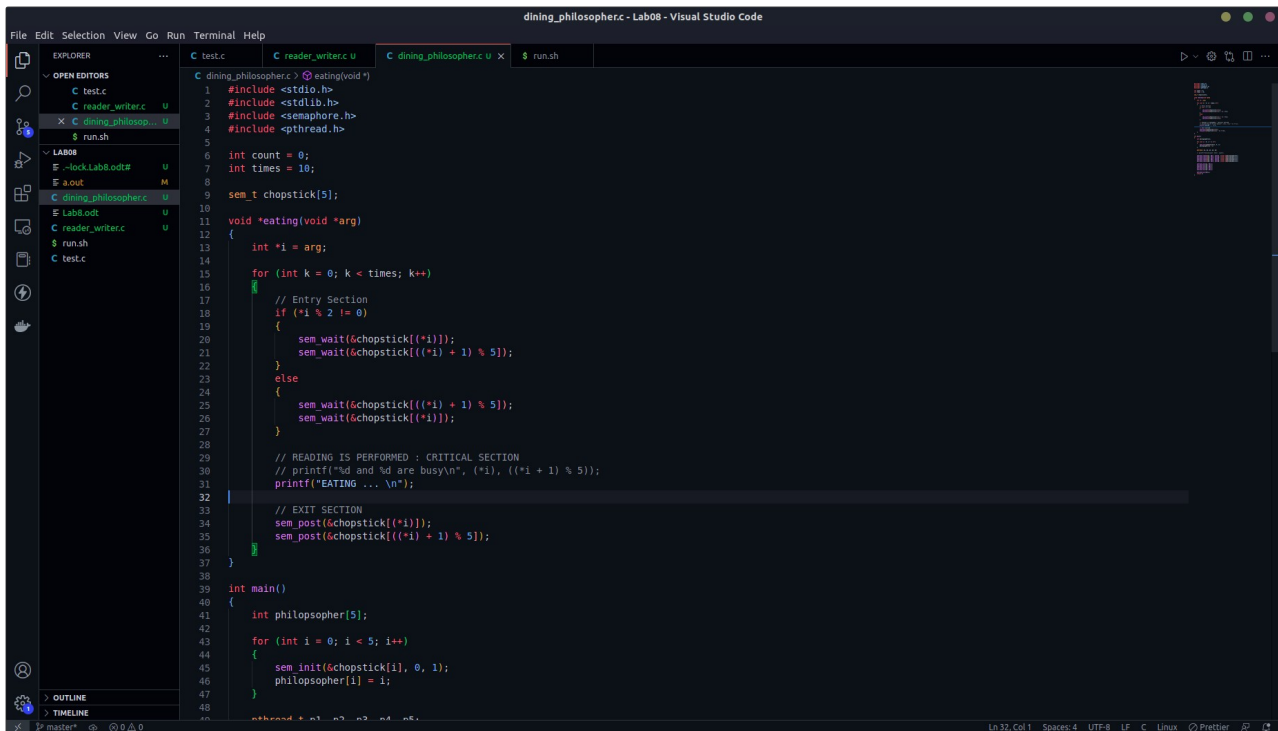
Output :

```
reader_writer.c - Lab08 - Visual Studio Code
C test.c  C reader_writer.c x  C dining_philosopher.c u  $ run.sh

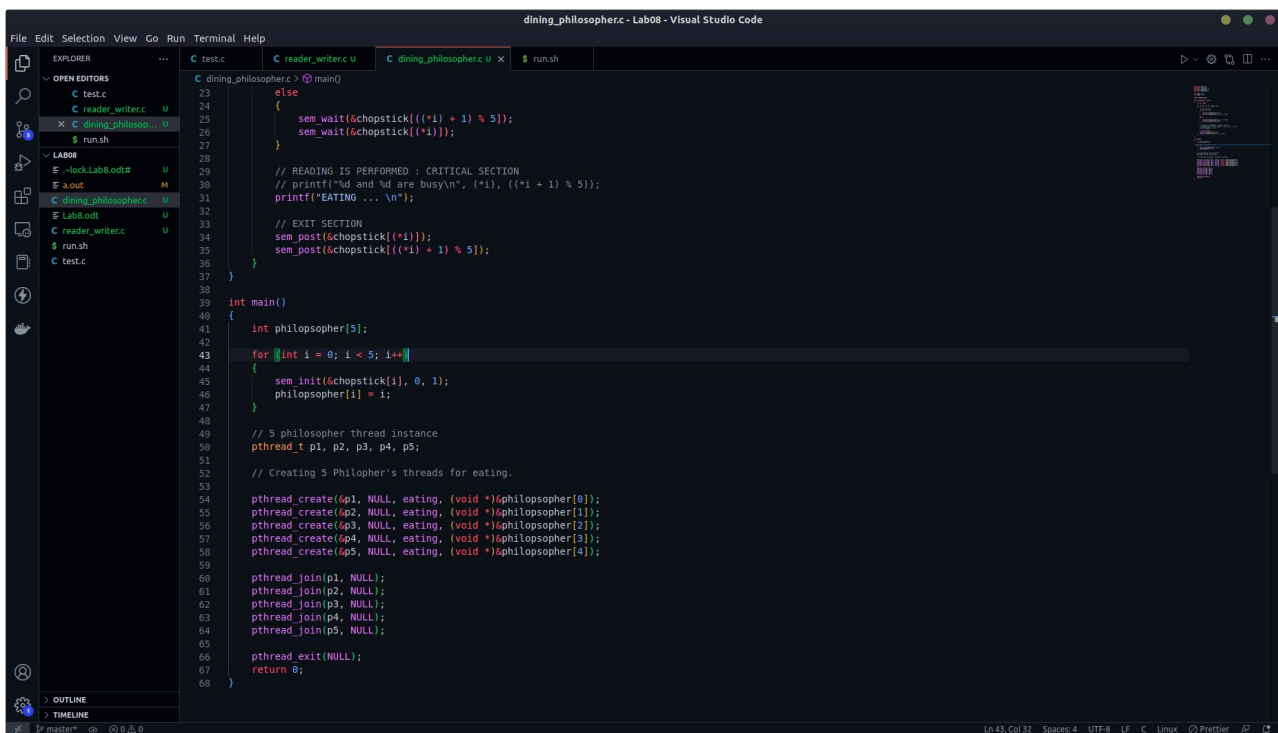
2 #include <stdlib.h>
3 #include <semaphore.h>
4 #include <pthread.h>
5
6 int count = 0;
7 int times = 10;
8
9 sem_t mutex, db;
10 int read_count = 0;
11
12 void *reader(void *arg)
13 {
14     int i = 0;
15     while (i < times)
16     {
17         printf("READING ...");
18         while (1)
19         {
20             pthread_mutex_lock(&mutex);
21             if (read_count > 0)
22             {
23                 pthread_mutex_unlock(&mutex);
24                 continue;
25             }
26             read_count++;
27             pthread_mutex_unlock(&mutex);
28             printf("Number of readers, right now is : %d\n", read_count);
29             sleep(1);
30         }
31         pthread_mutex_lock(&db);
32         pthread_mutex_unlock(&db);
33         i++;
34     }
35 }
36
37 void *writer(void *arg)
38 {
39     int i = 0;
40     while (i < times)
41     {
42         printf("WRITING ...");
43         while (1)
44         {
45             pthread_mutex_lock(&mutex);
46             if (count > 0)
47             {
48                 pthread_mutex_unlock(&mutex);
49                 continue;
50             }
41             count++;
52             pthread_mutex_unlock(&mutex);
53             printf("Number of writers, right now is : %d\n", count);
54             sleep(1);
55         }
56         pthread_mutex_lock(&db);
57         pthread_mutex_unlock(&db);
58         i++;
59     }
60 }
61
62 int main()
63 {
64     pthread_t reader, writer;
65     pthread_create(&reader, NULL, reader, NULL);
66     pthread_create(&writer, NULL, writer, NULL);
67     pthread_join(reader, NULL);
68     pthread_join(writer, NULL);
69     return 0;
70 }
```

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



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <semaphore.h>
4 #include <pthread.h>
5
6 int count = 0;
7 int times = 10;
8
9 sem_t chopstick[5];
10
11 void *eating(void *arg)
12 {
13     int *i = arg;
14
15     for (int k = 0; k < times; k++)
16     {
17         // Entry Section
18         if (*i % 2 != 0)
19         {
20             sem_wait(&chopstick[*i]);
21             sem_wait(&chopstick[( (*i) + 1) % 5]);
22         }
23         else
24         {
25             sem_wait(&chopstick[( (*i) + 1) % 5]);
26             sem_wait(&chopstick[*i]);
27         }
28
29         // READING IS PERFORMED : CRITICAL SECTION
30         // printf("id and %d are busy\n", (*i), (( *i) + 1) % 5));
31         printf("EATING ... \n");
32
33         // EXIT SECTION
34         sem_post(&chopstick[*i]);
35         sem_post(&chopstick[( (*i) + 1) % 5]);
36     }
37 }
38
39 int main()
40 {
41     int philosopher[5];
42
43     for (int i = 0; i < 5; i++)
44     {
45         sem_init(&chopstick[i], 0, 1);
46         philosopher[i] = i;
47     }
48 }
```



```
23 else
24 {
25     sem_wait(&chopstick[( (*i) + 1) % 5]);
26     sem_wait(&chopstick[*i]);
27 }
28
29 // READING IS PERFORMED : CRITICAL SECTION
30 // printf("id and %d are busy\n", (*i), (( *i) + 1) % 5));
31 printf("EATING ... \n");
32
33 // EXIT SECTION
34 sem_post(&chopstick[*i]);
35 sem_post(&chopstick[( (*i) + 1) % 5]);
36 }
37
38
39 int main()
40 {
41     int philosopher[5];
42
43     for (int i = 0; i < 5; i++)
44     {
45         sem_init(&chopstick[i], 0, 1);
46         philosopher[i] = i;
47     }
48
49     // 5 philosopher thread instance
50     pthread_t p1, p2, p3, p4, p5;
51
52     // Creating 5 Philosopher's threads for eating.
53
54     pthread_create(&p1, NULL, eating, (void *)(&philosopher[0]));
55     pthread_create(&p2, NULL, eating, (void *)(&philosopher[1]));
56     pthread_create(&p3, NULL, eating, (void *)(&philosopher[2]));
57     pthread_create(&p4, NULL, eating, (void *)(&philosopher[3]));
58     pthread_create(&p5, NULL, eating, (void *)(&philosopher[4]));
59
60     pthread_join(p1, NULL);
61     pthread_join(p2, NULL);
62     pthread_join(p3, NULL);
63     pthread_join(p4, NULL);
64     pthread_join(p5, NULL);
65
66     pthread_exit(NULL);
67     return 0;
68 }
```

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.