LAB 7 REPORT

Shouvanik Chakrabarti (130050072)
Krishna Harsha (130050076)

PART – A :

1.
The results of the three runs are:

Time: 2276
Count: 66153

Time: 2408
Count: 54437

Time: 1415
Count: 80459

The average time is: 2033
The average value of count is: 67016

2.
No, count does not match the ideal value in any run.

The lines that cause the race condition in the code are:
count++;
There is a race conditon between adding 1 to the value of count and assigning it to the original variable.
Consider count = A and two threads attempting to increment it. Suppose the first thread checks the value of A and calculates A+1 as the new value. However, before ti can update the value it is context switched out and a second thread increments A to A+1. Now the first thread is rescheduled and it sets the value of count to A+1 that it had calculated at an earlier time. Thus, the new value of count is A+1 instead of A+2 after two expected increments leading to a less than ideal value of count.

PART – B:

1.
The results of the three runs are:

Time: 3116
Count: 54029

Time: 7597
Count: 76911

Time: 829
Count: 97437

The average time is: 3847
The average value of count is: 76125

2.
No, count does not match the ideal value in any run.

The lines that cause the race condition in the code are:

```
    while(locked);
    locked=1;
    count++;
    locked=0;
```

Here the race condition is between checking the value of locked and updating it. One thread finds locked to be 0, and moves into the critical section. However, before it sets locked to 1 , it can be switched out and other scheduled threads can move into the critical section. Now the same race condition as in PART-A can occur between the different threads that have entered the critical section, and results in a less than ideal valueof count.

PART – C:

1.
The results of the three runs are:

Time: 15284
Count: 100000

Time: 14079
Count: 100000

Time: 17420
Count: 100000

The average time is: 15594
The average value of count is: 100000s

2.
Yes, count matches the ideal value in every run.

There is no race condition in the code. This is because the thread function locks a mutex before entering the critical section (count ++;) and unlocks it after completing it. This ensures that only one thread is in the critical section at a time, avoiding the race conditions of PART-A, as no other thread cannot calculate or update the value of count, while another thread is in the process of incrementing it.
Thus we end up with the ideal value of count, in every run.