

OS Lab

Lab 2 Report

Pthreads Synchronization

Vibhuti Raman 180010040

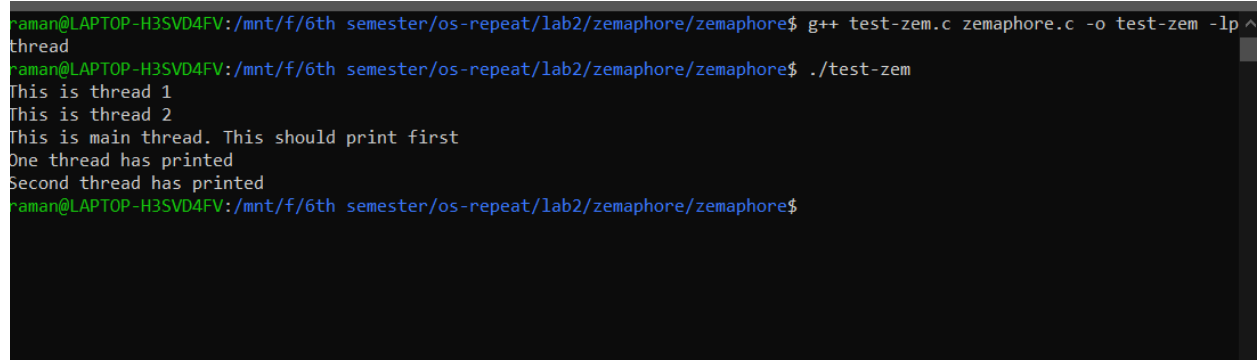
Vipul Nikam 180010041

In this assignment, our aim was to implement the basics of multi-threaded programming, and synchronizing multiple threads using locks and condition variables. We used the pthreads thread API in this assignment.

PART 1

We compiled our first file (Screenshot 2) and got the desired output. Here we can verify correctness of code with output. By comparing (Screenshot 1) We can see the main thread is printed and then the subsequent threads are printed.

Screenshot 1

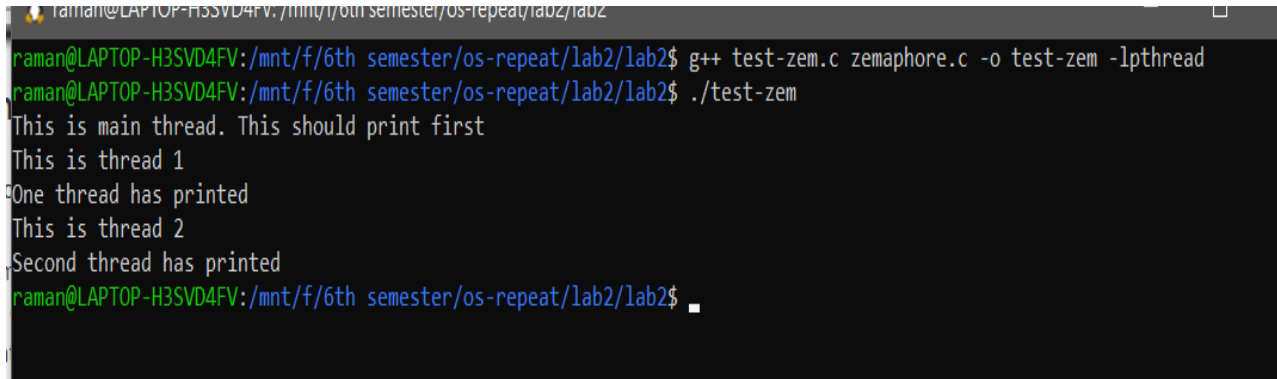


```
aman@LAPTOP-H3SVD4FV:/mnt/f/6th semester/os-repeat/lab2/zemaphore/zemaphore$ g++ test-zem.c zemaphore.c -o test-zem -lpthread
aman@LAPTOP-H3SVD4FV:/mnt/f/6th semester/os-repeat/lab2/zemaphore/zemaphore$ ./test-zem
This is thread 1
This is thread 2
This is main thread. This should print first
One thread has printed
Second thread has printed
aman@LAPTOP-H3SVD4FV:/mnt/f/6th semester/os-repeat/lab2/zemaphore/zemaphore$
```

Creating thread is running multiple processes at the same time. But we can not control the sequence in which the process will run if we just create a thread. To prevent this from happening, we need to create a lock.

And thus we tackled the issue of sequence which we can compare in screenshot 1 & 2.

Screenshot 2

A screenshot of a terminal window with a dark background. The prompt is 'raman@LAPTOP-H3SVD4FV:/mnt/f/6th semester/os-repeat/lab2/lab2\$'. The user enters 'g++ test-zem.c semaphore.c -o test-zem -lpthread'. The prompt changes to './test-zem'. The output shows: 'This is main thread. This should print first', 'This is thread 1', 'One thread has printed', 'This is thread 2', 'Second thread has printed'. The prompt returns to './test-zem\$' with a cursor. The window title bar shows 'Taman@LAPTOP-H3SVD4FV: /mnt/f/6th semester/os-repeat/lab2/lab2' and standard window controls.

```
raman@LAPTOP-H3SVD4FV:/mnt/f/6th semester/os-repeat/lab2/lab2$ g++ test-zem.c semaphore.c -o test-zem -lpthread
raman@LAPTOP-H3SVD4FV:/mnt/f/6th semester/os-repeat/lab2/lab2$ ./test-zem
This is main thread. This should print first
This is thread 1
One thread has printed
This is thread 2
Second thread has printed
raman@LAPTOP-H3SVD4FV:/mnt/f/6th semester/os-repeat/lab2/lab2$
```

The program test-zem.c spawns two threads, and all three threads (the main thread and the two new threads) share a semaphore. Before you implement the semaphore logic, the new threads will print to screen before the main thread. However, after implementing the semaphore correctly, the main thread will print first, owing to the synchronization enabled by the semaphore.

PART 2

This is the code with N threads. Thread i must print number i in a continuous loop. Without any synchronization between the threads, the threads will print their numbers in any order (Screenshot 3). And after adding the synchronization (Screenshot 5) to the code, the numbers are printed in the order 1, 2, ..., N , 1, 2, ..., N , and so on. Following screenshots (3 & 5) are for N = 2 as can be seen below.

Screenshot 3

```
raman@LAPTOP-H3SVD4FV:/mnt/f/6th semester/os-repeat/lab2/zemaphore/zemaphore$ clear
raman@LAPTOP-H3SVD4FV:/mnt/f/6th semester/os-repeat/lab2/zemaphore/zemaphore$ g++ test-toggle.c zemaphore.c -o test-toggle -lpthread
raman@LAPTOP-H3SVD4FV:/mnt/f/6th semester/os-repeat/lab2/zemaphore/zemaphore$ ./test-toggle
This is thread 0
This is thread 0
This is thread 0
This is thread 0
This is thread 0
This is thread 0
This is thread 0
This is thread 0
This is thread 0
This is thread 0
This is thread 1
This is thread 1
This is thread 1
This is thread 1
This is thread 1
This is thread 1
This is thread 1
This is thread 1
This is thread 1
This is thread 1
This is thread 2
This is thread 2
This is thread 2
This is thread 2
This is thread 2
This is thread 2
This is thread 2
This is thread 2
This is thread 2
This is thread 2
raman@LAPTOP-H3SVD4FV:/mnt/f/6th semester/os-repeat/lab2/zemaphore/zemaphore$
```

Screenshot 4

```
zemlist = (zem_t *)malloc(sizeof(zem_t) * NUM_THREADS);
for (int i = 0; i < NUM_THREADS; i++)
{
    zem_init(&zemlist[i], 0);
}
for(int i = 0; i < NUM_THREADS; i++)
{
    mythread_id[i] = i;
    pthread_create(&mythreads[i], NULL, justprint, (void *)&mythread_id[i]);
}
pthread_join(&zemlist[0]);
for(int i = 0; i < NUM_THREADS; i++)
{
    pthread_join(mythreads[i], NULL);
}
```

So as we can see when threads were not organised (screenshot 3), we saw output unordered. But when we added the lock, we can see 0,1,2,0,1,2 and so on (Screenshot 5) as N is 2. And by comparing both the screenshots (3 & 5) we can verify the correctness of our code.

Screenshot 4 is the main function of the modified program. malloc is a subroutine for performing dynamic memory allocation. We can see the malloc() used from stdlib.h library.

Screenshot 5

[illegible]