LOVELY PROFESSIONAL UNIVERSITY

Assignment

Operating system Lab

Department of Computer Science & Engineering

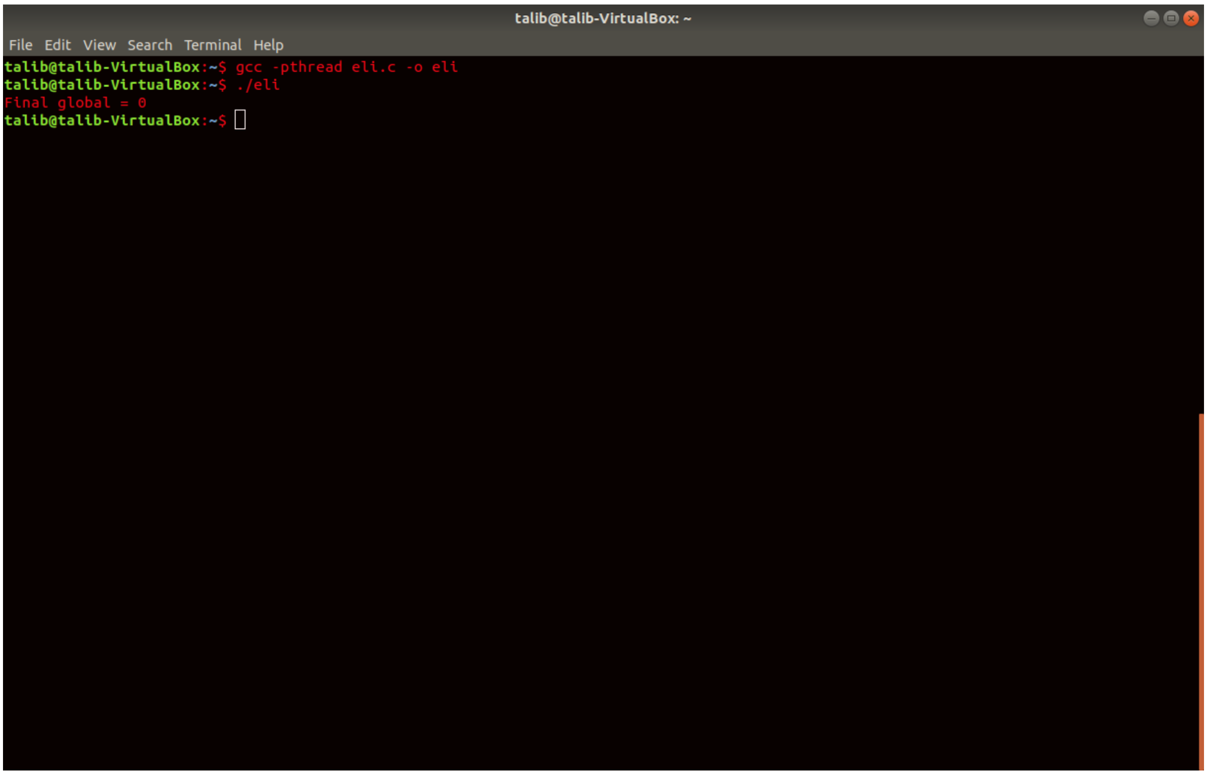
Submitted By:-	Project In Charge:-
Talebul Islam	Priyanka mittal
Roll No:- 58 Section : GT	(Assistant Professor)
Reg No :- 11904195	

Github Link : - https://github.com/talebulislam/Lab-Operating-

2.Write a program to eliminate race condition using semaphores

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h> //compile and link with -pthread
#include <semaphore.h>
int global = 0;
sem_t mutex;
void *inc( void *ptr )
{
        int i;
        for(i = 0; i < 9000000; i++)
        {
                sem_wait(&mutex);
                global++;
                sem_post(&mutex);
        }
}

void *dec( void *ptr )
{
        int i;
        for(i = 0; i < 9000000; i++)
        {
                sem_wait(&mutex);
                global--;
                sem_post(&mutex);
        }
}

int main()
{
        pthread_t thread1, thread2;
        sem_init(&mutex, 0, 1);
        pthread_create( &thread1, NULL, inc, NULL);
        pthread_create( &thread2, NULL, dec, NULL);
        pthread_join( thread1, NULL);
        pthread_join( thread2, NULL);
        sem_destroy(&mutex);
        printf("Final global = %d\n", global);
        exit(0);
}
```

Output : -



talib@talib-VirtualBox: ~
File Edit View Search Terminal Help

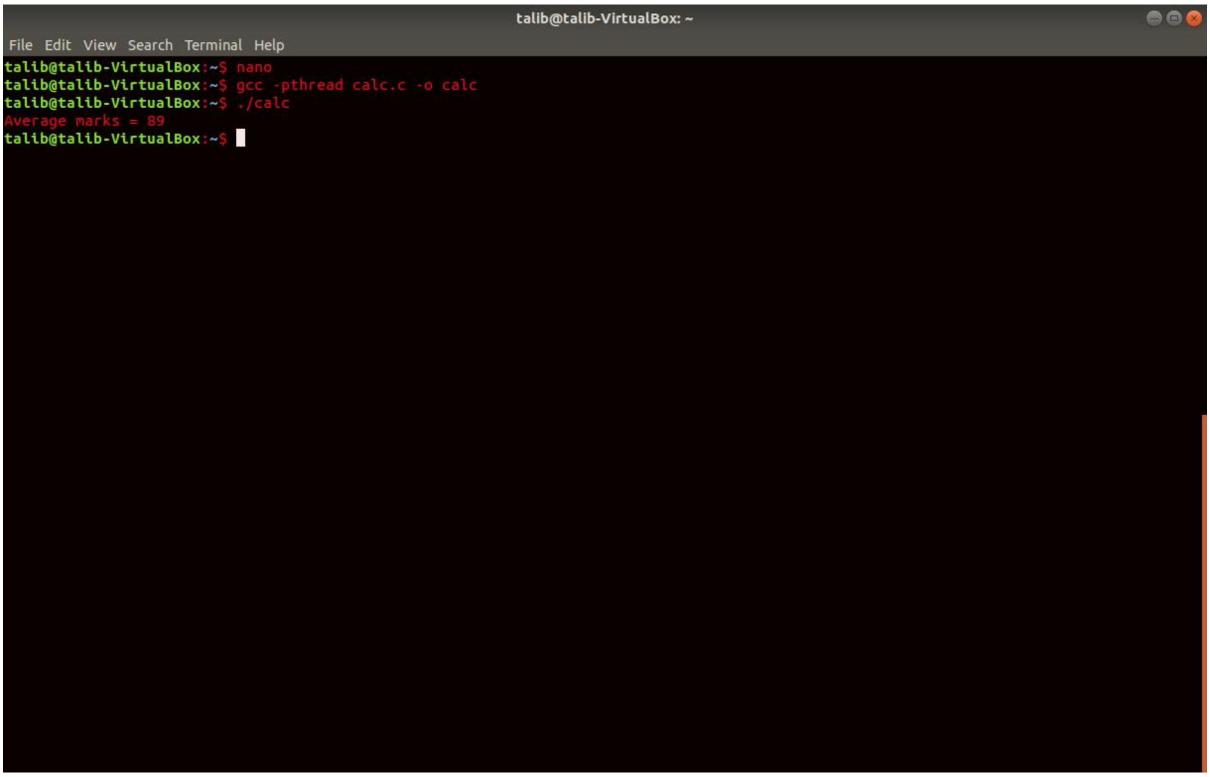
```
talib@talib-VirtualBox:~$ gcc -pthread eli.c -o eli
talib@talib-VirtualBox:~$ ./eli
Final global = 0
talib@talib-VirtualBox:~$ 
```

4.Write a program to create a thread that calculates the average Marks of a student. The result is passed back to the main program For printing.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h> //compile and link with -pthread
#include <semaphore.h>
int global = 0;
sem_t mutex;
void *inc( void *ptr )
{
        int i;
        for(i = 0; i < 9000000; i++)
        {
                sem_wait(&mutex);
                global++;
                sem_post(&mutex);
        }
}
```

```
void *dec( void *ptr )
{
        int i;
        for(i = 0; i < 9000000; i++)
        {
                sem_wait(&mutex);
                global--;
                sem_post(&mutex);
        }
}
int main()
{
        pthread_t thread1, thread2;
        sem_init(&mutex, 0, 1);
        pthread_create( &thread1, NULL, inc, NULL);
        pthread_create( &thread2, NULL, dec, NULL);
        pthread_join( thread1, NULL);
        pthread_join( thread2, NULL);
        sem_destroy(&mutex);
        printf("Final global = %d\n", global);
        exit(0);
}
```

Output : -



talib@talib-VirtualBox: ~
File Edit View Search Terminal Help

```
talib@talib-VirtualBox:~$ nano
talib@talib-VirtualBox:~$ gcc -pthread calc.c -o calc
talib@talib-VirtualBox:~$ ./calc
Average marks = 89
talib@talib-VirtualBox:~$ 
```

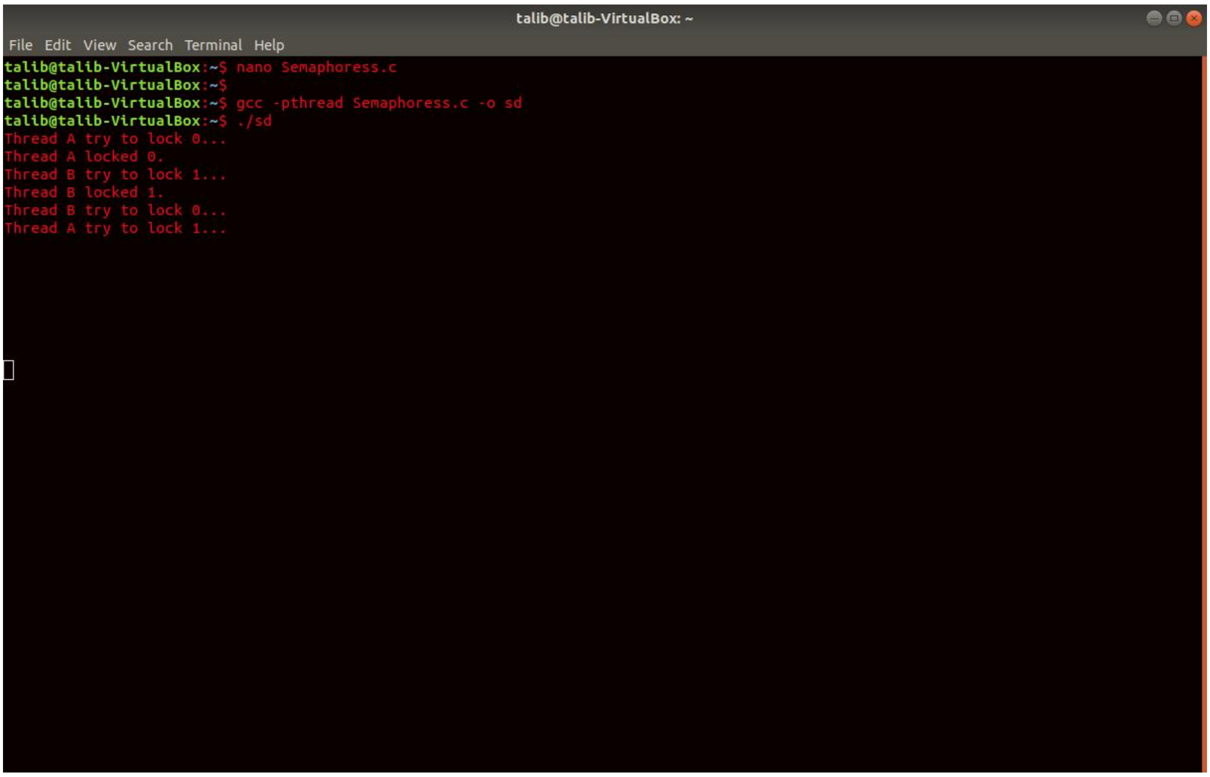
6.Writeaprogram to generate the deadlock condition using semaphore.

```
#include <stdio.h>      // Input/Output
#include <unistd.h>     // Time management (usleep)
#include <pthread.h>    // Threads management
#include <sys/sem.h>    // Semaphores management

#define PERMS 0660  // -rw permissions for group and user

int semId;

int initSem(int semId, int semNum, int initValue)
{
        return semctl(semId, semNum, SETVAL, initValue);
}

/* An operation list is structured like this :
*   { semphore index, operation, flags }
* The operation is an integer value interpreted like this :
*   >= 0 : Rise the semaphore value by this value.
*       This trigger the awakening of semaphores waiting for a rise.
*   == 0 : Wait for the semaphore to be at value 0.
*    < 0 : Substract abs(value) to the semaphore.
*       If then the semaphore is negative, wait for a rise.
*/

// Try to take a resource, wait if not available
int P(int semId, int semNum)
{
   // Operation list of 1 operation, taking resource, no flag
   struct sembuf operationList[1];
   operationList[0].sem_num = semNum;
   operationList[0].sem_op = -1;
   operationList[0].sem_flg = 0;
   return semop(semId, operationList, 1);
}

// Release a resource

int V(int semId, int semNum)
{
   // Operation list of 1 operation, releasing resource, no flag
```

```
    struct sembuf operationList[1];
    operationList[0].sem_num = semNum;
    operationList[0].sem_op = 1;
    operationList[0].sem_flg = 0;

    return semop(semId, operationList, 1);
}

void* funcA(void* nothing)
{
    printf("Thread A try to lock 0...\n");
    P(semId, 0);        // Take resource/semaphore 0 of semID
    printf("Thread A locked 0.\n");

    usleep(50*1000);  // Wait 50 ms

    printf("Thread A try to lock 1...\n");
    P(semId, 1);        // Take resource/semaphore 1 of semID
    printf("Thread A locked 1.\n");

    V(semId, 0);        // Release resource/semaphore 0 of semID
    V(semId, 1);        // Release resource/semaphore 1 of semID
    return NULL;
}

void* funcB(void* nothing)
{
    printf("Thread B try to lock 1...\n");
    P(semId, 1);        // Take resource/semaphore 0 of semID
    printf("Thread B locked 1.\n");

    usleep(5*1000);  // Wait 50 ms

    printf("Thread B try to lock 0...\n");
    P(semId, 0);        // Take resource/semaphore 1 of semID
    printf("Thread B locked 0.\n");

    V(semId, 0);        // Release resource/semaphore 0 of semID
    V(semId, 1);        // Release resource/semaphore 1 of semID
    return NULL;
}

// Main function
```

```
int main(int argc, char* argv[])
{
   int i;          // Iterator

   // We create a set of 2 semaphores
   // ftok generates a key based on the program name and a char value
   // This avoid to pick an arbitrary key already existing
   semId = semget(ftok(argv[0], 'A'), 2, IPC_CREAT | PERMS);

   // Set the semaphore at index 0 to value 1 (= available for use)

   initSem(semId, 0, 1);

   // Set the semaphore at index 1 to value 1 (= available for use)

   initSem(semId, 1, 1);

   pthread_t thread[2];    // Array of threads

   pthread_create(&thread[0], NULL, funcA, NULL);
   pthread_create(&thread[1], NULL, funcB, NULL);

   // Wait until threads are all complete
   for (i = 0 ; i < 2 ; i++)
  {
        pthread_join(thread[i], NULL);
   }
   printf("This is not printed in case of deadlock\n");

   // Free the semaphores

   semctl(semId, 0, IPC_RMID, 0);
   semctl(semId, 1, IPC_RMID, 0);

   return 0;
}
```

Output : -



talib@talib-VirtualBox: ~
File Edit View Search Terminal Help

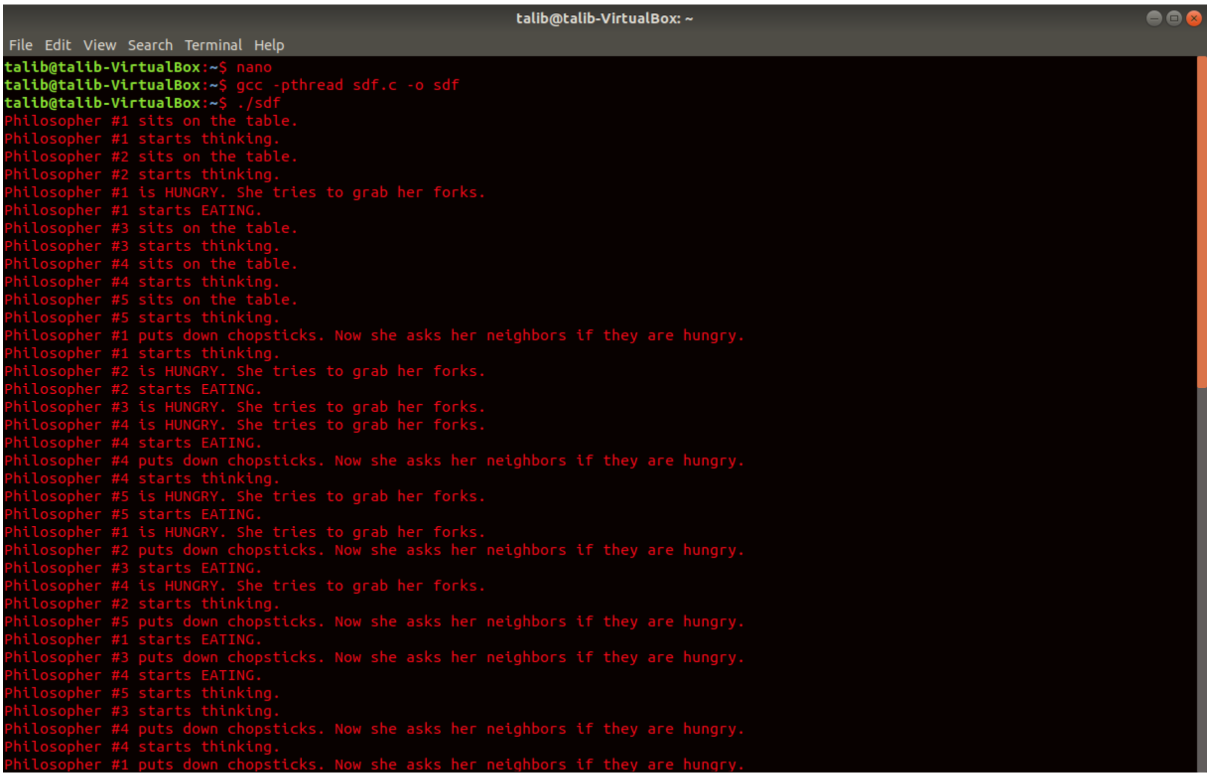
```
talib@talib-VirtualBox:~$ nano Semaphoress.c
talib@talib-VirtualBox:~$
talib@talib-VirtualBox:~$ gcc -pthread Semaphoress.c -o sd
talib@talib-VirtualBox:~$ ./sd
Thread A try to lock 0...
Thread A locked 0.
Thread B try to lock 1...
Thread B locked 1.
Thread B try to lock 0...
Thread A try to lock 1...
```

8.Write a program using semaphore to avoid the race condition.

With the help of dining philosopher problems

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

#define N_PHILOSOPHERS 5
#define LEFT (ph_num + N_PHILOSOPHERS - 1) % N_PHILOSOPHERS
#define RIGHT (ph_num + 1) % N_PHILOSOPHERS

pthread_mutex_t mutex;
pthread_cond_t condition[N_PHILOSOPHERS];

enum
{
        THINKING, HUNGRY, EATING } state[N_PHILOSOPHERS];
        int phil_num[N_PHILOSOPHERS];

        void *philosophing (void *arg);
        void pickup_forks(int ph_num);
        void return_forks(int ph_num);
        void test(int ph_num);

        int main(int argc, char *argv[])
{
        pthread_t ph_thread[N_PHILOSOPHERS];
        pthread_mutex_init(&mutex, NULL);

        for (int i = 0; i < N_PHILOSOPHERS; i++)
        {
                pthread_cond_init(&condition[i], NULL);
                phil_num[i] = i;
        }


        for (int i = 0; i < N_PHILOSOPHERS; i++)
        {
                 pthread_create(&ph_thread[i], NULL, philosophing,
                &phil_num[i]);
                printf("Philosopher #%d sits on the table.\n", i + 1);
                sleep(1);
        }
  for (int i = 0; i < N_PHILOSOPHERS; i++)
  pthread_join(ph_thread[i], NULL);


  pthread_mutex_destroy(&mutex);
  for (int i = 0; i < N_PHILOSOPHERS; i++)
  pthread_cond_destroy(&condition[i]);

  return(0);
}

void *philosophing(void *arg)
{
  while(1)
{
     int *ph_num = arg;
     printf("Philosopher #%d starts thinking.\n", *ph_num + 1);
     sleep(2);
     pickup_forks(*ph_num);
     return_forks(*ph_num);
  }
}

void pickup_forks(int ph_num)
{
        pthread_mutex_lock(&mutex);

        printf("Philosopher #%d is HUNGRY. She tries to grab her forks.\n",
        ph_num + 1);
        state[ph_num] = HUNGRY;
        test(ph_num);
        while (state[ph_num] != EATING)
        pthread_cond_wait(&condition[ph_num], &mutex);

        pthread_mutex_unlock(&mutex);
  }

void return_forks(int ph_num)
{
        pthread_mutex_lock(&mutex);

        printf("Philosopher #%d puts down chopsticks. Now she asks her
        neighbors if they are hungry.\n", ph_num + 1);
        state[ph_num] = THINKING;
        test(LEFT);
        test(RIGHT);

        pthread_mutex_unlock(&mutex);
 }

void test(int ph_num)
{
        if (state[ph_num] == HUNGRY && state[LEFT] != EATING &&
        state[RIGHT] != EATING)
        {
                printf("Philosopher #%d starts EATING.\n", ph_num + 1);
                state[ph_num] = EATING;
                sleep(3);
                pthread_cond_signal(&condition[ph_num]);
        }
}
```

Output :-


talib@talib-VirtualBox: ~
File Edit View Search Terminal Help

```
talib@talib-VirtualBox:~$ nano
talib@talib-VirtualBox:~$ gcc -pthread sdf.c -o sdf
talib@talib-VirtualBox:~$ ./sdf
Philosopher #1 sits on the table.
Philosopher #1 starts thinking.
Philosopher #2 sits on the table.
Philosopher #2 starts thinking.
Philosopher #1 is HUNGRY. She tries to grab her forks.
Philosopher #1 starts EATING.
Philosopher #3 sits on the table.
Philosopher #3 starts thinking.
Philosopher #4 sits on the table.
Philosopher #4 starts thinking.
Philosopher #5 sits on the table.
Philosopher #5 starts thinking.
Philosopher #1 puts down chopsticks. Now she asks her neighbors if they are hungry.
Philosopher #1 starts thinking.
Philosopher #2 is HUNGRY. She tries to grab her forks.
Philosopher #2 starts EATING.
Philosopher #3 is HUNGRY. She tries to grab her forks.
Philosopher #4 is HUNGRY. She tries to grab her forks.
Philosopher #4 starts EATING.
Philosopher #4 puts down chopsticks. Now she asks her neighbors if they are hungry.
Philosopher #4 starts thinking.
Philosopher #5 is HUNGRY. She tries to grab her forks.
Philosopher #5 starts EATING.
Philosopher #1 is HUNGRY. She tries to grab her forks.
Philosopher #2 puts down chopsticks. Now she asks her neighbors if they are hungry.
Philosopher #3 starts EATING.
Philosopher #4 is HUNGRY. She tries to grab her forks.
Philosopher #2 starts thinking.
Philosopher #5 puts down chopsticks. Now she asks her neighbors if they are hungry.
Philosopher #1 starts EATING.
Philosopher #3 puts down chopsticks. Now she asks her neighbors if they are hungry.
Philosopher #4 starts EATING.
Philosopher #5 starts thinking.
Philosopher #3 starts thinking.
Philosopher #4 puts down chopsticks. Now she asks her neighbors if they are hungry.
Philosopher #4 starts thinking.
Philosopher #1 puts down chopsticks. Now she asks her neighbors if they are hungry.
```