**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-System>

**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);

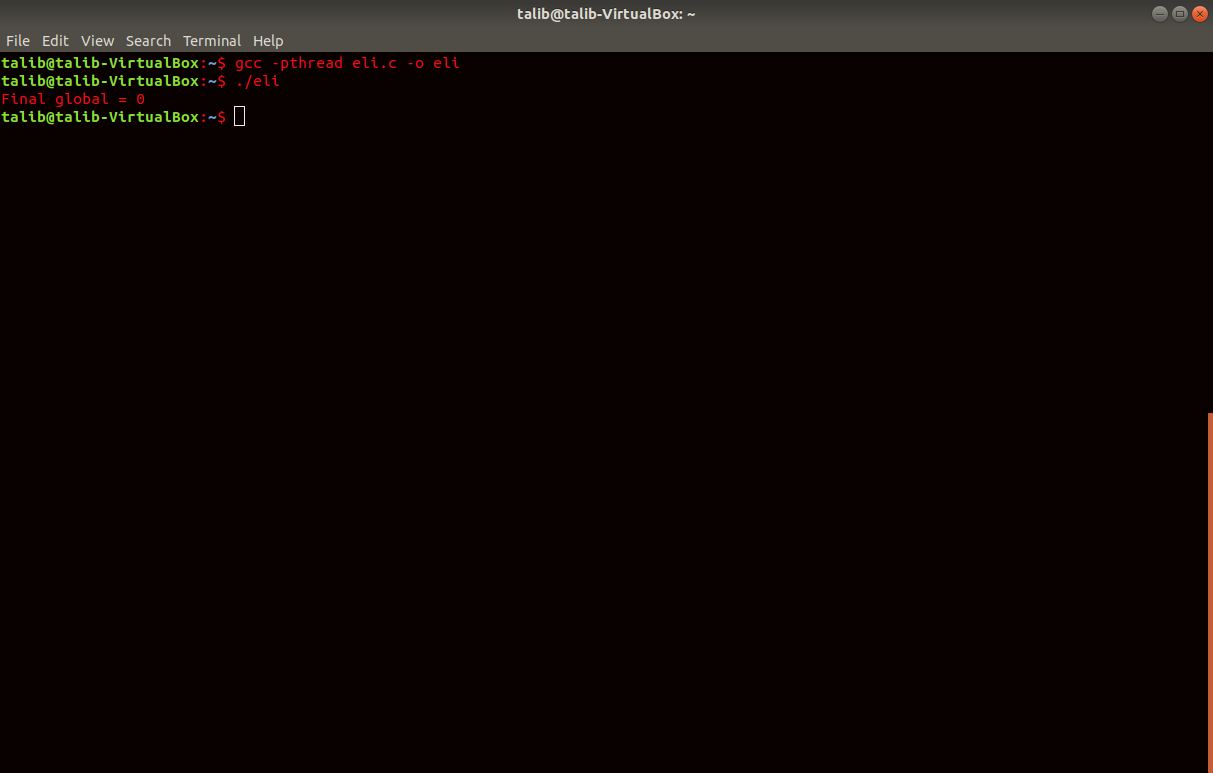
sem\_destroy(&mutex);

printf("Final global = %d\n", global);

exit(0);

}

Output : -



**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);

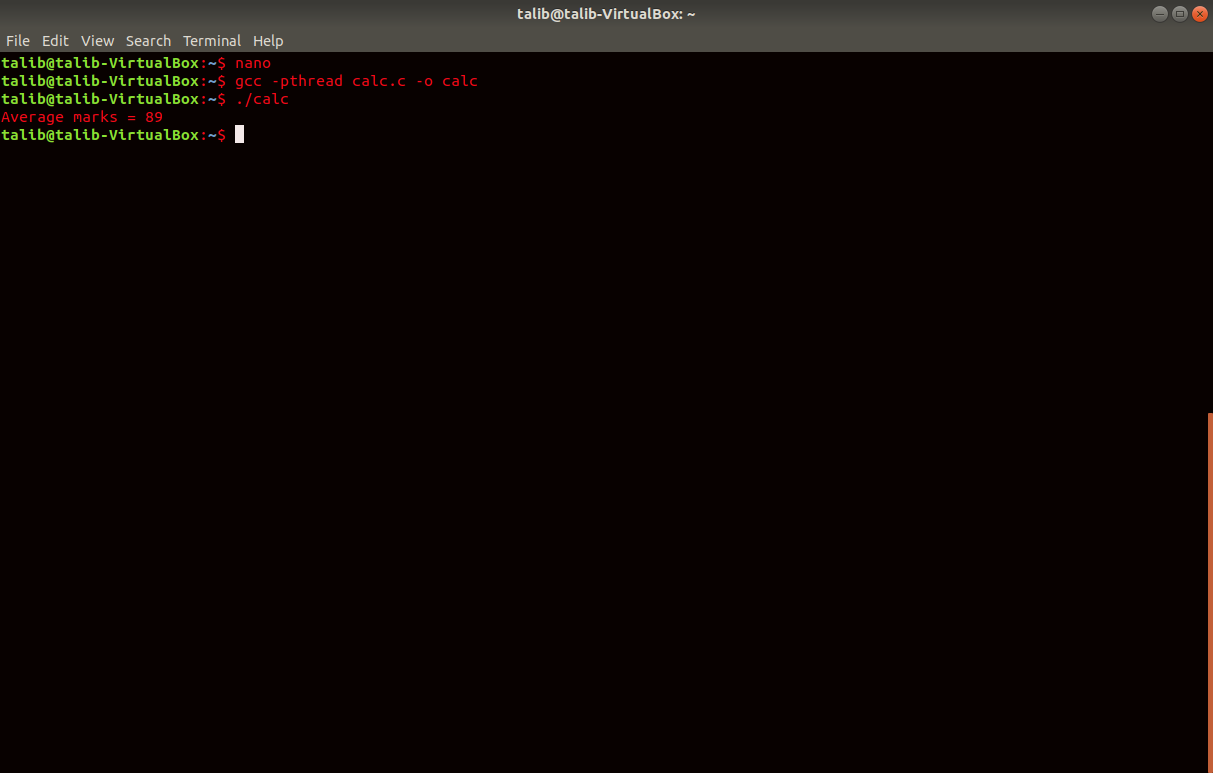
sem\_destroy(&mutex);

printf("Final global = %d\n", global);

exit(0);

}

Output : -



**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

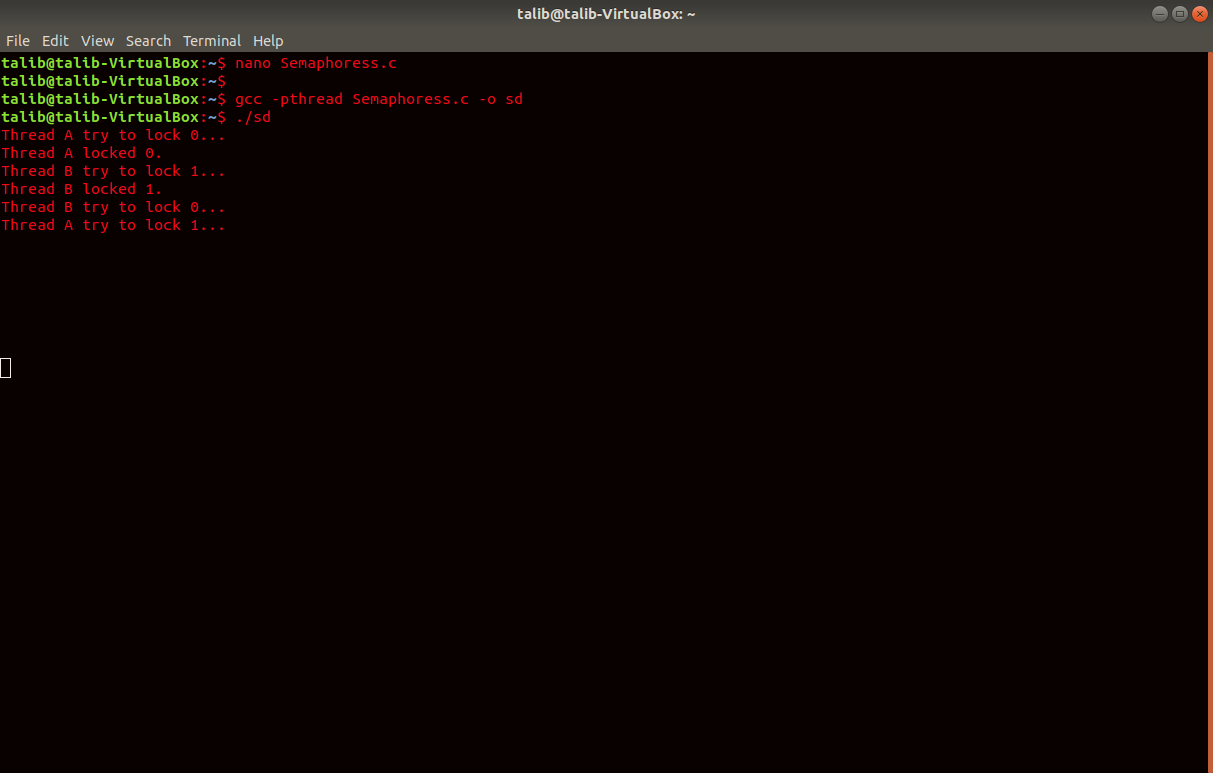
semctl(semId, 0, IPC\_RMID, 0);

semctl(semId, 1, IPC\_RMID, 0);

return 0;

}

Output : -



**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 :-

