**LAB 6**

1. **Write a program to simulate Peterson’s solution.**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define MAX\_THREADS 10

#define ITERATIONS 5

int turn;

int flag[MAX\_THREADS];

int count = 0; // Actual count

int expected\_count = 0; // Expected count

void lock(int id) {

int other = 1 - id;

flag[id] = 1;

turn = id;

// Busy-wait until the other thread releases the lock

while (flag[other] == 1 && turn == id) {

// Do nothing

}

}

void unlock(int id) {

flag[id] = 0;

}

void\* thread\_function(void\* arg) {

int thread\_id = \*((int\*)arg);

// Enter critical section

lock(thread\_id);

// Critical Section

printf("Thread %d is in the critical section. Actual count: %d, Expected count: %d\n", thread\_id, count, expected\_count);

// Perform some critical section operations

count++;

expected\_count++;

// Exit critical section

unlock(thread\_id);

pthread\_exit(NULL);

}

int main() {

int num\_threads;

printf("Enter the number of threads (up to %d): ", MAX\_THREADS);

scanf("%d", &num\_threads);

if (num\_threads < 1 || num\_threads > MAX\_THREADS) {

fprintf(stderr, "Invalid number of threads. Exiting.\n");

exit(EXIT\_FAILURE);

}

pthread\_t threads[MAX\_THREADS];

int thread\_ids[MAX\_THREADS];

for (int i = 0; i < num\_threads; i++) {

thread\_ids[i] = i;

flag[i] = 0;

}

for (int i = 0; i < num\_threads; i++) {

if (pthread\_create(&threads[i], NULL, thread\_function, &thread\_ids[i]) != 0) {

fprintf(stderr, "Error creating thread %d.\n", i);

exit(EXIT\_FAILURE);

}

}

for (int i = 0; i < num\_threads; i++) {

pthread\_join(threads[i], NULL);

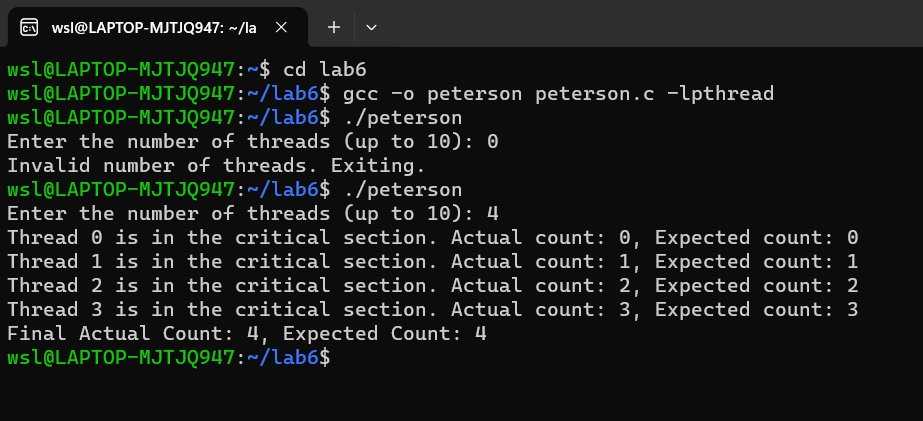
}

printf("Final Actual Count: %d, Expected Count: %d\n", count, expected\_count);

return 0;

}

**Output:**



1. **Write a program to avoid racing conditions using semaphore.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#define KEY 0x1111

union semun {

int val;

struct semid\_ds \*buf;

unsigned short \*array;

};

struct sembuf p = { 0, -1, SEM\_UNDO};

struct sembuf v = { 0, +1, SEM\_UNDO};

int main() {

int id = semget(KEY, 1, 0666 | IPC\_CREAT);

if(id < 0) {

perror("semget"); exit(11);

}

union semun u;

u.val = 1;

if(semctl(id, 0, SETVAL, u) < 0) {

perror("semctl"); exit(12);

}

int pid;

pid = fork();

srand(pid);

if(pid < 0) {

perror("fork"); exit(1);

}

else if(pid) {

char \*s = "redpanda";

int l = strlen(s);

for(int i = 0; i < l; ++i) {

if(semop(id, &p, 1) < 0) {

perror("semop p"); exit(13);

}

putchar(s[i]);

fflush(stdout);

sleep(rand() % 2);

putchar(s[i]);

fflush(stdout);

if(semop(id, &v, 1) < 0) {

perror("semop p"); exit(14);

}

sleep(rand() % 2);

}

}

else {

char \*s = "REDPANDA";

int l = strlen(s);

for(int i = 0; i < l; ++i) {

if(semop(id, &p, 1) < 0) {

perror("semop p"); exit(15);

}

putchar(s[i]);

fflush(stdout);

sleep(rand() % 2);

putchar(s[i]);

fflush(stdout);

if(semop(id, &v, 1) < 0) {

perror("semop p"); exit(16);

}

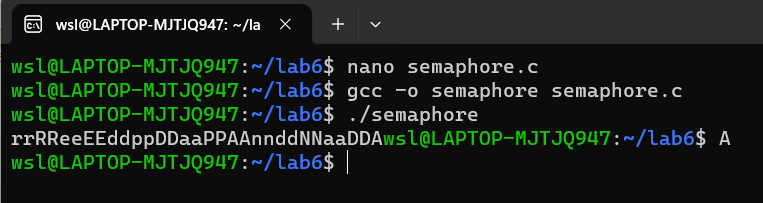
sleep(rand() % 2);

}

}

}

**Output:**

****