**Lab 6**

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

Code:

#include <iostream>

#include <thread>

#include <atomic>

using namespace std;

const int NUM\_THREADS = 2;

atomic<int> turn;

atomic<bool> flag[NUM\_THREADS];

void lock(int threadId) {

int other = 1 - threadId;

flag[threadId].store(true);

turn.store(other);

while (flag[other].load() && turn.load() == other) {

// Busy waiting

}

}

void unlock(int threadId) {

flag[threadId].store(false);

}

void criticalSection(int threadId) {

cout << "Thread " << threadId << " is in critical section." << endl;

this\_thread::sleep\_for(chrono::milliseconds(100));

}

void threadFunction(int threadId) {

for (int i = 0; i < 5; ++i) {

lock(threadId);

criticalSection(threadId);

unlock(threadId);

this\_thread::sleep\_for(chrono::milliseconds(50));

}

}

int main() {

turn.store(0);

flag[0].store(false);

flag[1].store(false);

thread t1(threadFunction, 0);

thread t2(threadFunction, 1);

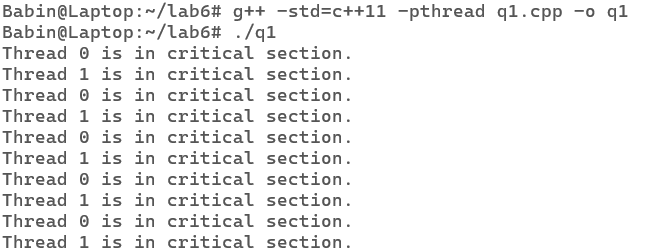
t1.join();

t2.join();

return 0;

}

Output:



2. Write a program to avoid racing conditions using semaphore.

Code:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define NUM\_THREADS 2

#define COUNT 10

int shared\_variable = 0;

sem\_t semaphore;

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

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

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

sem\_wait(&semaphore); // Wait until semaphore is available

// Critical section

printf("Thread %d in critical section (iteration %d)\n", thread\_id, i+1);

shared\_variable++;

printf("Shared variable value after increment: %d\n", shared\_variable);

sem\_post(&semaphore); // Release semaphore

}

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[NUM\_THREADS];

int thread\_ids[NUM\_THREADS] = {0, 1};

sem\_init(&semaphore, 0, 1); // Initialize semaphore with initial value 1

// Create threads

for (int i = 0; i < NUM\_THREADS; ++i) {

int result = pthread\_create(&threads[i], NULL, thread\_function, &thread\_ids[i]);

if (result != 0) {

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

return 1;

}

}

// Join threads

for (int i = 0; i < NUM\_THREADS; ++i) {

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

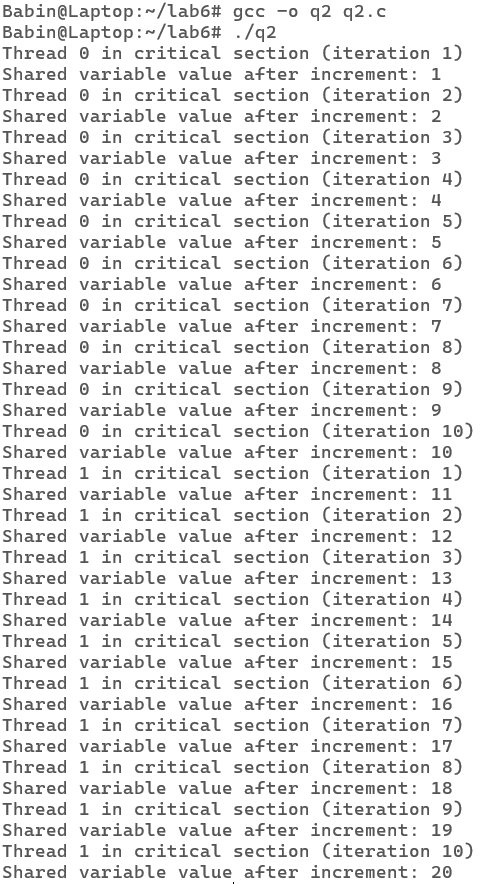
}

sem\_destroy(&semaphore); // Destroy semaphore

return 0;

}

Output:



3. WAP to simulate producer consumer problem using mutex.

Code:

#include <stdio.h>

#include <stdlib.h>

// Initialize a mutex to 1

int mutex = 1;

// Number of full slots as 0

int full = 0;

// Number of empty slots as size of buffer

int empty = 10, x = 0;

void producer()

{

// Decrease mutex value by 1

--mutex;

// Increase the number of full slots by 1

++full;

// Decrease the number of empty slots by 1

--empty;

// Item produced

x++;

printf("\nProducer produces item %d", x);

// Increase mutex value by 1

++mutex;

}

void consumer()

{

// Decrease mutex value by 1

--mutex;

// Decrease the number of full slots by 1

--full;

// Increase the number of empty slots by 1

++empty;

printf("\nConsumer consumes item %d", x);

x--;

// Increase mutex value by 1

++mutex;

}

int main()

{

int n, i;

printf("\n1. Press 1 for Producer"

"\n2. Press 2 for Consumer"

"\n3. Press 3 for Exit");

for (i = 1; i > 0; i++) {

printf("\nEnter your choice:");

scanf("%d", &n);

switch (n) {

case 1:

if ((mutex == 1) && (empty != 0)) {

producer();

}

else {

printf("Buffer is full!");

}

break;

case 2:

if ((mutex == 1) && (full != 0)) {

consumer();

}

else {

printf("Buffer is empty!");

}

break;

case 3:

exit(0);

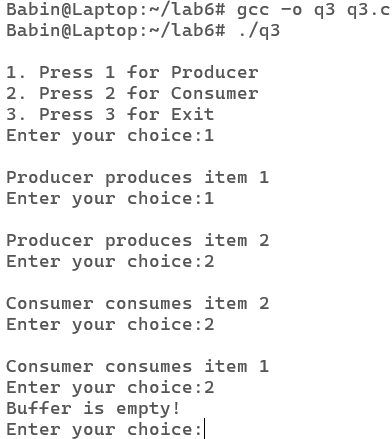
break;

}

}

}

Output:



4. Write a program using fork system call such that parent process should calculate the sum of the two numbers and child process should calculate the multiplication of the same number.

Code:

#include <iostream>

#include <unistd.h>

#include <sys/wait.h> // For wait()

using namespace std;

int main() {

int num1, num2;

cout << "Enter two numbers: ";

cin >> num1 >> num2;

pid\_t pid = fork();

if (pid < 0) {

cout << "Fork failed!" << endl;

return 1;

}

if (pid > 0) {

// Parent process waits for the child to complete

wait(NULL); // Wait for the child process to complete

int sum = num1 + num2;

cout << "Parent process:\n";

cout << "Sum of " << num1 << " and " << num2 << " is " << sum << endl;

} else {

// Child process

int product = num1 \* num2;

cout << "Child process:\n";

cout << "Product of " << num1 << " and " << num2 << " is " << product << endl;

}

return 0;

}

Output:

