**Lab 6: Process synchronization and inter-process communication in operating system**

**Objectives:**

The objective of these programming tasks is to explore and implement fundamental concepts in process synchronization and inter-process communication in operating systems. The first task involves simulating Peterson’s solution, a classic algorithm used to achieve mutual exclusion between two processes. The second task focuses on avoiding race conditions using semaphores, a synchronization primitive that ensures controlled access to shared resources. The third task simulates the producer-consumer problem using mutexes, demonstrating how mutual exclusion can be achieved in multi-threaded environments. Finally, the fourth task involves using the fork system call to create a parent and child process, where each process performs different arithmetic operations on shared data, illustrating process creation and communication. Together, these exercises provide a practical understanding of key synchronization techniques and process management in operating systems.

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

**Code:**

#include <iostream>

#include <pthread.h>

#include <unistd.h>  // *for usleep*

using namespace std;

*const* int NUM\_THREADS = 2;

*volatile* bool flag[NUM\_THREADS] = {false, false};

*volatile* int turn = 0;

void *\**criticalSection(void *\**threadId) {

    int id = \*(int \*)threadId;

    int other = 1 - id;

    for (int i = 0; i < 5; ++i) {

// *Lock*

        flag[id] = true;

        turn = other;

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

// *Busy-wait*

        }

// *Critical Section*

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

        usleep(100000);// *Sleep for 100 milliseconds*

// *Unlock*

        flag[id] = false;

// *Sleep to simulate work outside critical section*

        usleep(50000);// *Sleep for 50 milliseconds*

    }

    pthread\_exit(NULL);

}

int main() {

    pthread\_t threads[NUM\_THREADS];

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

// *Create threads*

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

        pthread\_create(&threads[i], NULL, criticalSection, (void \*)&threadIds[i]);

    }

// *Wait for threads to complete*

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

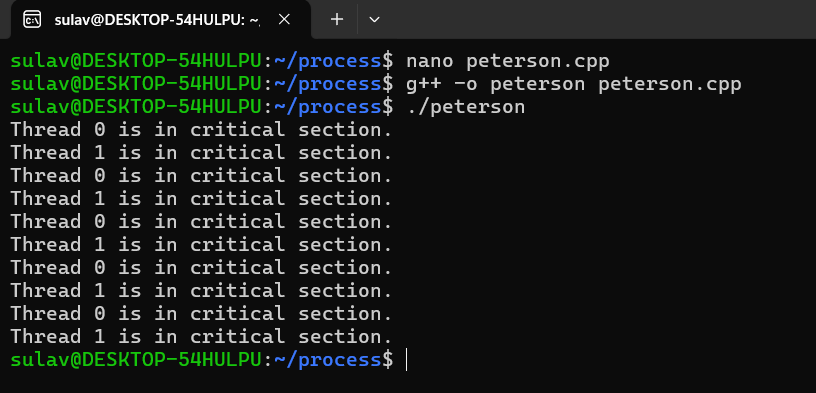
        pthread\_join(threads[i], NULL);

    }

    return 0;

}

**Output:**

****

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

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUM\_THREADS 2

#define COUNT 10

int shared\_variable = 0;

pthread\_mutex\_t mutex;

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

    int thread\_id = \*((int \*)arg);

    for (int i = 0; i < COUNT; ++i) {

        pthread\_mutex\_lock(&mutex);// *Lock mutex*

// *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);

        pthread\_mutex\_unlock(&mutex);// *Unlock mutex*

    }

    pthread\_exit(NULL);

}

int main() {

    pthread\_t threads[NUM\_THREADS];

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

// *Initialize mutex*

    if (pthread\_mutex\_init(&mutex, NULL) != 0) {

        perror("pthread\_mutex\_init");

        return 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: %d\n", i, result);

            return 1;

        }

    }

// *Join threads*

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

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

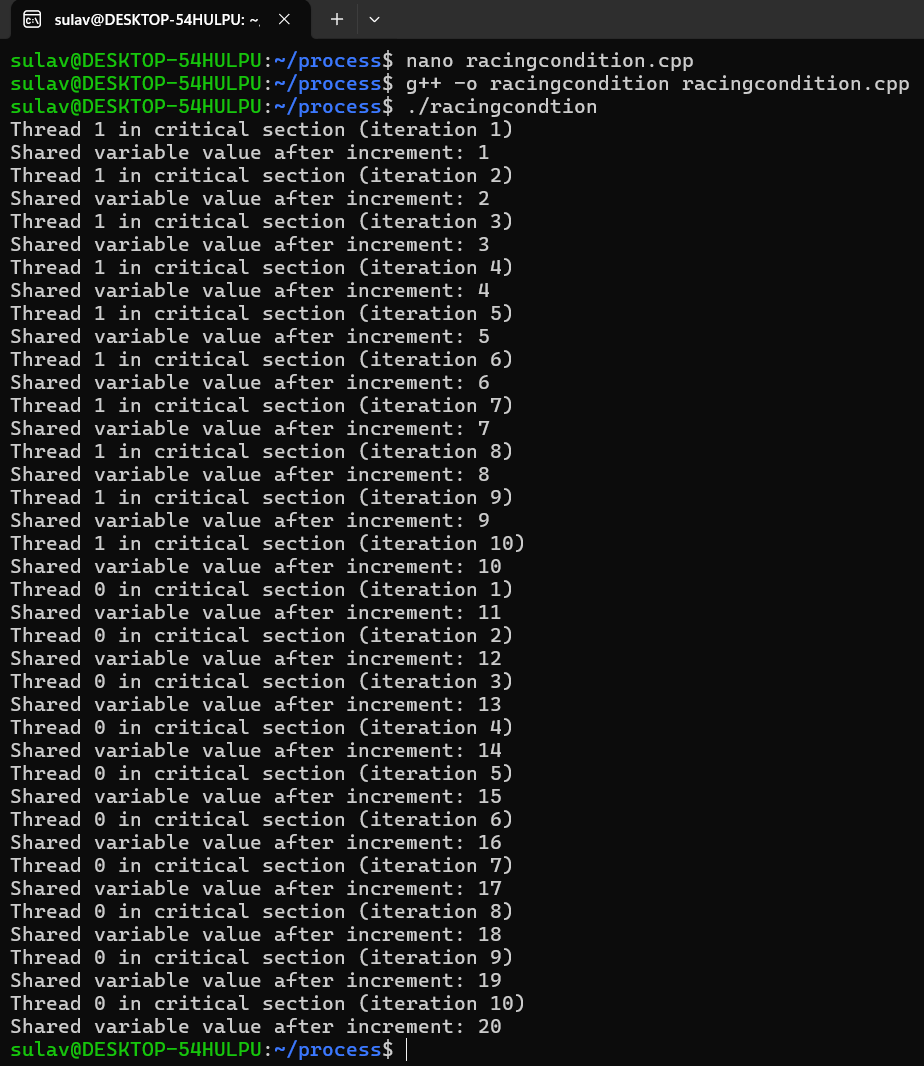
    }

// *Destroy mutex*

    pthread\_mutex\_destroy(&mutex);

    return 0;

}

**Output: **

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

**Code:**

#include <stdio.h>

#include <stdlib.h>

// *Simulate a mutex with an integer*

int mutex = 1;

int full = 0;

int empty = 10;

int x = 0;

void producer() {

    if (mutex == 1 && empty != 0) {

        --mutex;// *Lock the mutex*

        ++full;// *Increase the number of full slots*

        --empty;// *Decrease the number of empty slots*

        x++;// *Produce an item*

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

        ++mutex;// *Release the mutex*

    } else {

        printf("\nBuffer is full!");

    }

}

void consumer() {

    if (mutex == 1 && full != 0) {

        --mutex;// *Lock the mutex*

        --full;// *Decrease the number of full slots*

        ++empty;// *Increase the number of empty slots*

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

        x--;// *Consume an item*

        ++mutex;// *Release the mutex*

    } else {

        printf("\nBuffer is empty!");

    }

}

int main() {

    int n, i;

    printf("\n1. Press 1 for Producer");

    printf("\n2. Press 2 for Consumer");

    printf("\n3. Press 3 for Exit");

    for (i = 1; i > 0; i++) {

        printf("\nEnter your choice: ");

        scanf("%d", &n);

        switch (n) {

        case 1:

            producer();

            break;

        case 2:

            consumer();

            break;

        case 3:

            exit(0);

        default:

            printf("Invalid choice! Please enter 1, 2, or 3.");

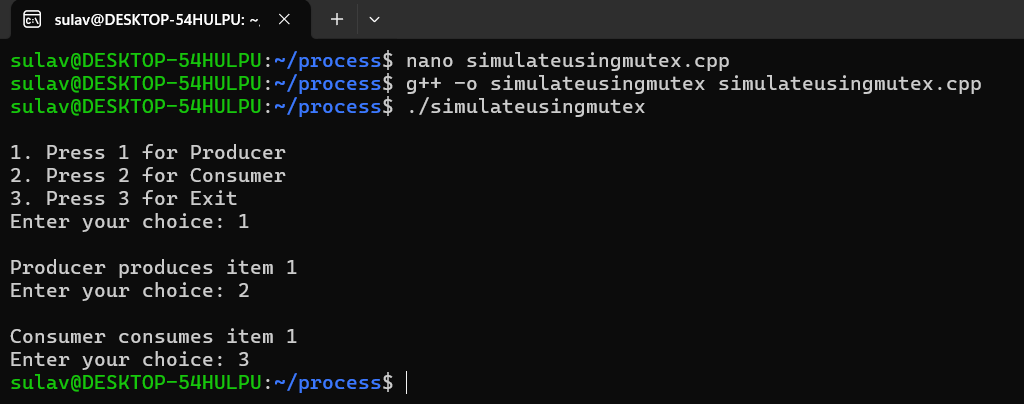
        }

    }

    return 0;

}

**Output:**

****

1. **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> // for fork()

#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) {

// Fork failed

cerr << "Fork failed!" << endl;

return 1;

} else if (pid == 0) {

// Child process

int product = num1 \* num2;

cout << "Child process:\n";

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

} else {

// Parent process

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

int sum = num1 + num2;

cout << "Parent process:\n";

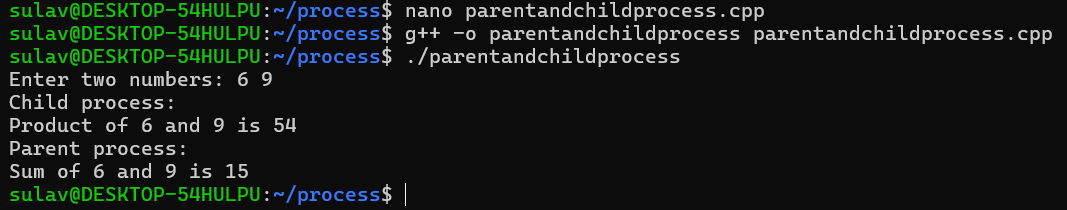
cout << "Sum of " << num1 << " and " << num2 << " is " << sum << endl;

}

return 0;

}

**Output:**

****

**Conclusion:**

In conclusion, these exercises provided hands-on experience with essential synchronization techniques and process management in operating systems. By simulating Peterson’s solution, we learned how to ensure mutual exclusion between processes, preventing conflicts in shared resource access. Implementing semaphore-based synchronization highlighted how to avoid race conditions effectively. The producer-consumer problem, tackled with mutexes, reinforced the importance of controlling thread interactions in a shared environment. Finally, using the fork system call to perform arithmetic operations in separate processes illustrated how parent and child processes can work together while maintaining distinct execution paths. Overall, these tasks deepened our understanding of concurrent programming and the critical role of synchronization in maintaining system stability.