



National Textile University

Department of Computer Science

Subject:

Operating Systems

Submitted To:

Sir Nasir Mehmood

Submitted By:

Alishba Riasat

Registration No:

23-NTU-CS-1135

Lab No:

6

Semester:

5th

Program 1:

Code:

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

#define NUM_THREADS 4
int varg=0;

void *thread_function(void *arg) {
    int thread_id = *(int *)arg;

    int varl=0;
    varg++;
    varl++;
    printf("Thread %d is executing the global value is %d: local  
vare is %d: process id %d: \n", thread_id, varg, varl, getpid());
    return NULL;
}

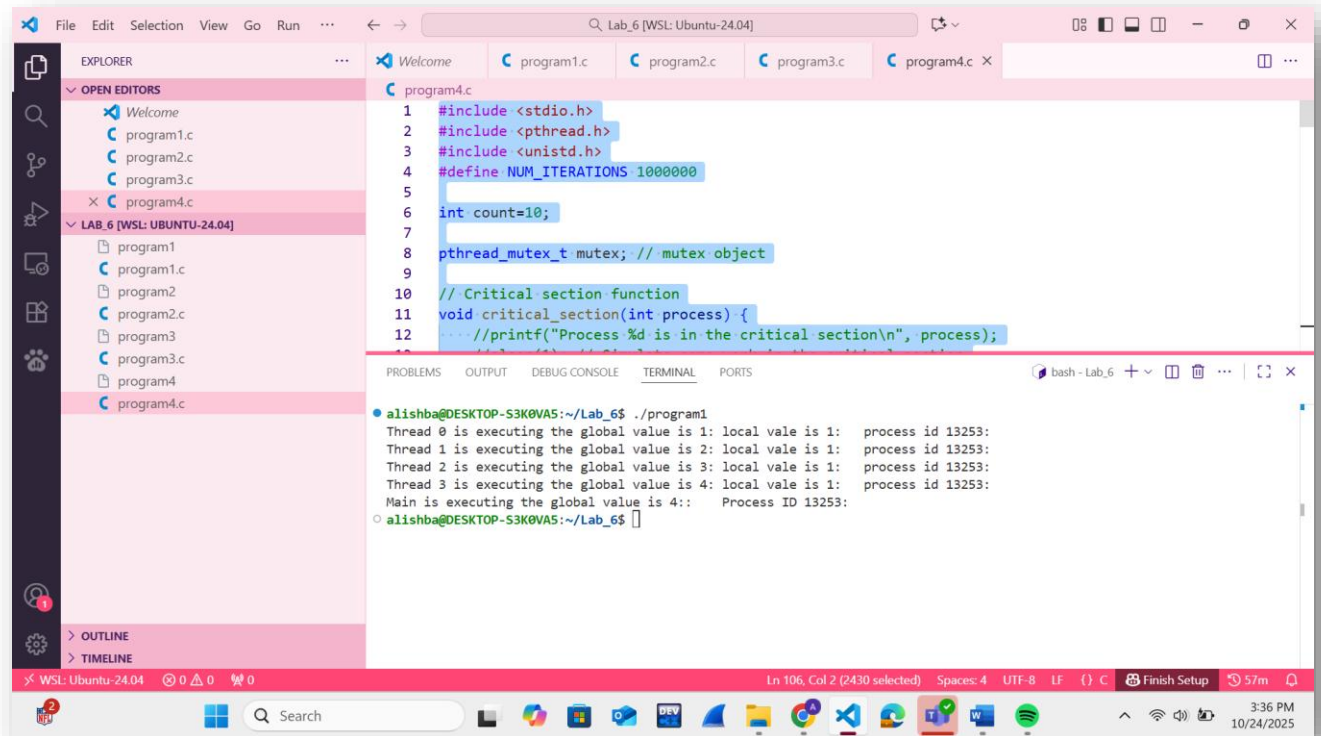
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; ++i) {
        thread_args[i] = i;
        pthread_create(&threads[i], NULL, thread_function,
&thread_args[i]);
    }

    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], NULL);
    }
    printf("Main is executing the global value is %d:: Process ID  
%d: \n", varg, getpid());

    return 0;
}
```

Output:



The screenshot shows the Visual Studio Code editor with a C program named `program4.c` open. The program includes `<stdio.h>`, `<pthread.h>`, and `<unistd.h>`, and defines `NUM_ITERATIONS` as `1000000`. It initializes a global `count` to 10 and a `pthread_mutex_t` mutex. A `critical_section` function is defined, which prints the process ID and sleeps for 1 second. The terminal output shows the execution of `./program1` on a system with 4 threads (Thread 0, 1, 2, 3) and the main process, all with process ID 13253. The output shows the global value being updated from 1 to 4, and the local value being updated from 1 to 1 for each thread.

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4 #define NUM_ITERATIONS 1000000
5
6 int count=10;
7
8 pthread_mutex_t mutex; // mutex object
9
10 // Critical section function
11 void critical_section(int process) {
12     //printf("Process %d is in the critical section\n", process);
13     //sleep(1); // Simulate some work in the critical section
14 }
```

```
alishba@DESKTOP-S3K0VAS:~/Lab_6$ ./program1
Thread 0 is executing the global value is 1: local vale is 1: process id 13253:
Thread 1 is executing the global value is 2: local vale is 1: process id 13253:
Thread 2 is executing the global value is 3: local vale is 1: process id 13253:
Thread 3 is executing the global value is 4: local vale is 1: process id 13253:
Main is executing the global value is 4:: Process ID 13253:
alishba@DESKTOP-S3K0VAS:~/Lab_6$
```

Program 2 (Without Peterson Algorithm):

Code:

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define NUM_ITERATIONS 1000000

int count=10;

// Critical section function
void critical_section(int process) {
    //printf("Process %d is in the critical section\n", process);
    //sleep(1); // Simulate some work in the critical section
}
```

```

    if(process==0){

        for (int i = 0; i < NUM_ITERATIONS; i++)
            count--;
    }
    else
    {
        for (int i = 0; i < NUM_ITERATIONS; i++)
            count++;
    }
}

void *process0(void *arg) {

    // Critical section
    critical_section(0);
    // Exit section

    return NULL;
}

void *process1(void *arg) {

    // Critical section
    critical_section(1);
    // Exit section

    return NULL;
}

int main() {
    pthread_t thread0, thread1, thread2, thread3;

```

```

// Create threads
pthread_create(&thread0, NULL, process0, NULL);
pthread_create(&thread1, NULL, process1, NULL);
pthread_create(&thread2, NULL, process0, NULL);
pthread_create(&thread3, NULL, process1, NULL);

// Wait for threads to finish
pthread_join(thread0, NULL);
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
pthread_join(thread3, NULL);

printf("Final count: %d\n", count);

return 0;
}

```

Output:

The screenshot shows the Visual Studio Code editor with a C program that uses pthreads to create four threads. The program is named 'program2.c' and is located in a directory named 'Lab_6'. The code defines two functions: 'critical_section' and 'process0'. The 'process0' function calls 'critical_section' twice. The 'critical_section' function increments a global 'count' variable. The main function creates four threads, each calling 'process0'. After all threads have finished, the main function prints the final count.

```

11 void critical_section(int process) {
24
25 }
26
27 void *process0(void *arg) {
28
29
30
31     // Critical section
32     critical_section(0);
33     // Exit section

```

The terminal output shows the command being executed and the final result:

```

alishba@DESKTOP-S3K0VAS:~/Lab_6$ ./program2
Final count: 37078
alishba@DESKTOP-S3K0VAS:~/Lab_6$

```

Program 3 (With Peterson Algorithm):

Code:

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define NUM_ITERATIONS 100000
// Shared variables
int turn;
int flag[2];
int count=0;

// Critical section function
void critical_section(int process) {

    if(process==0){

        for (int i = 0; i < NUM_ITERATIONS; i++)
            count--;
    }
    else
    {
        for (int i = 0; i < NUM_ITERATIONS; i++)
            count++;
    }

}

// Peterson's Algorithm function for process 0
void *process0(void *arg) {

    flag[0] = 1;
    turn = 1;
    while (flag[1]==1 && turn == 1) {
        // Busy wait
    }
    // Critical section
```

```

        critical_section(0);
        // Exit section
        flag[0] = 0;
        //sleep(1);

pthread_exit(NULL);

}

// Peterson's Algorithm function for process 1
void *process1(void *arg) {

    flag[1] = 1;
    turn = 0;
    while (flag[0] == 1 && turn == 0) {
        // Busy wait
    }
    // Critical section
    critical_section(1);
    // Exit section
    flag[1] = 0;
    //sleep(1);

    pthread_exit(NULL);
}

int main() {
    pthread_t thread0, thread1;

    // Initialize shared variables
    flag[0] = 0;
    flag[1] = 0;
    turn = 0;

    // Create threads
    pthread_create(&thread0, NULL, process0, NULL);
    pthread_create(&thread1, NULL, process1, NULL);

    // Wait for threads to finish
    pthread_join(thread0, NULL);
    pthread_join(thread1, NULL);
}

```

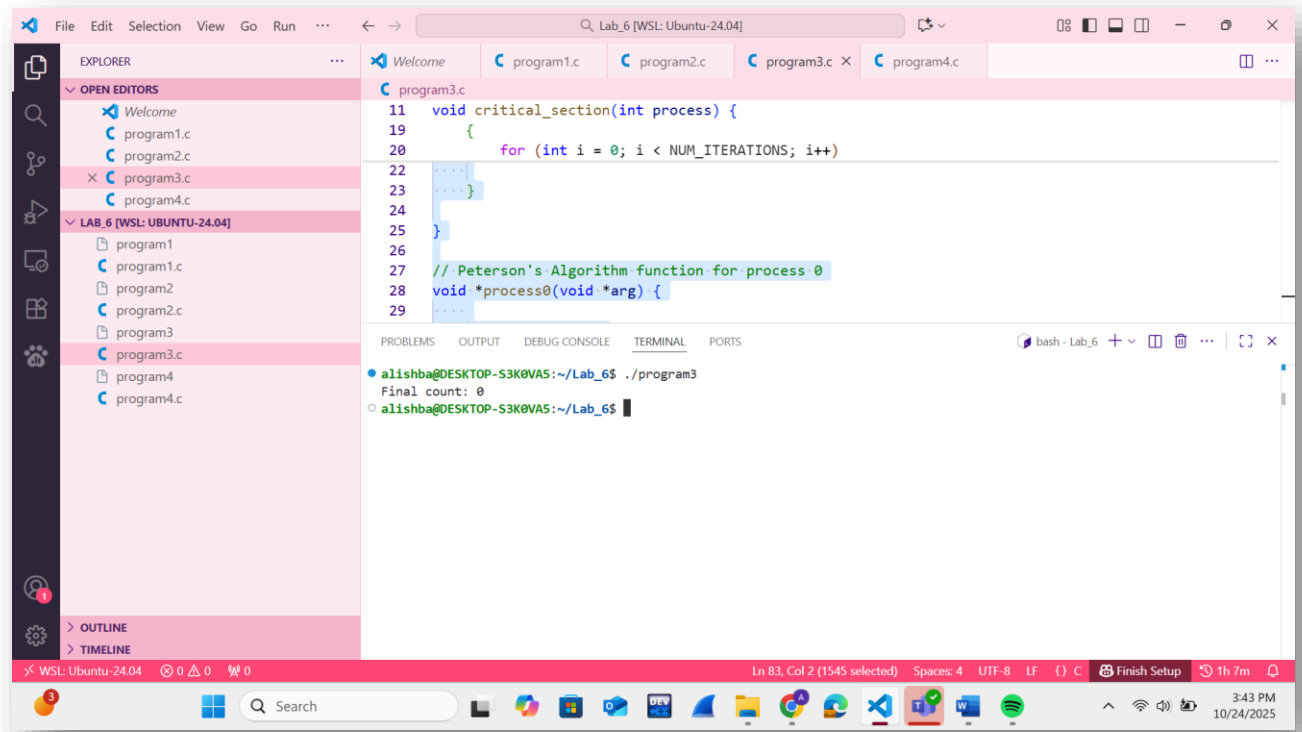
```

printf("Final count: %d\n", count);

return 0;
}

```

Output:



Program 4 (With Mutex):

Code:

```

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define NUM_ITERATIONS 1000000

int count=10;

pthread_mutex_t mutex; // mutex object

```



```

// Critical section function
void critical_section(int process) {
    //printf("Process %d is in the critical section\n", process);
    //sleep(1); // Simulate some work in the critical section
    if(process==0){

        for (int i = 0; i < NUM_ITERATIONS; i++)
            count--;
    }
    else if (process==1)
    {
        for (int i = 0; i < NUM_ITERATIONS; i++)
            count++;
    }
    else if(process==2)
    {
        for (int i = 0; i < NUM_ITERATIONS; i++)
            count--;
    }
    else{
        for (int i = 0; i < NUM_ITERATIONS; i++)
            count++;
    }
    //printf("Process %d has updated count to %d\n", process,
count);
    //printf("Process %d is leaving the critical section\n",
process);
}

// Peterson's Algorithm function for process 0
void *process0(void *arg) {

    pthread_mutex_lock(&mutex); // lock

    // Critical section
    critical_section(0);
    // Exit section

    pthread_mutex_unlock(&mutex); // unlock

    return NULL;
}

```

```
// Peterson's Algorithm function for process 1
void *process1(void *arg) {

    pthread_mutex_lock(&mutex); // lock

    // Critical section
    critical_section(1);
    // Exit section

    pthread_mutex_unlock(&mutex); // unlock

    return NULL;
}

void *process2(void *arg) {

    pthread_mutex_lock(&mutex); // lock

    // Critical section
    critical_section(2);
    // Exit section

    pthread_mutex_unlock(&mutex); // unlock

    return NULL;
}

void *process3(void *arg) {

    pthread_mutex_lock(&mutex); // lock

    // Critical section
    critical_section(3);
    // Exit section

    pthread_mutex_unlock(&mutex); // unlock
```

```

        return NULL;
    }
    int main() {
        pthread_t thread0, thread1, thread2, thread3, thread4, thread5,
        thread6, thread7;

        pthread_mutex_init(&mutex, NULL); // initialize mutex

        // Create threads
        pthread_create(&thread0, NULL, process0, NULL);
        pthread_create(&thread1, NULL, process1, NULL);
        pthread_create(&thread2, NULL, process2, NULL);
        pthread_create(&thread3, NULL, process3, NULL);
        pthread_create(&thread4, NULL, process0, NULL);
        pthread_create(&thread5, NULL, process1, NULL);
        pthread_create(&thread6, NULL, process2, NULL);
        pthread_create(&thread7, NULL, process3, NULL);

        // Wait for threads to finish
        pthread_join(thread0, NULL);
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
        pthread_join(thread3, NULL);
        pthread_join(thread4, NULL);
        pthread_join(thread5, NULL);
        pthread_join(thread6, NULL);
        pthread_join(thread7, NULL);

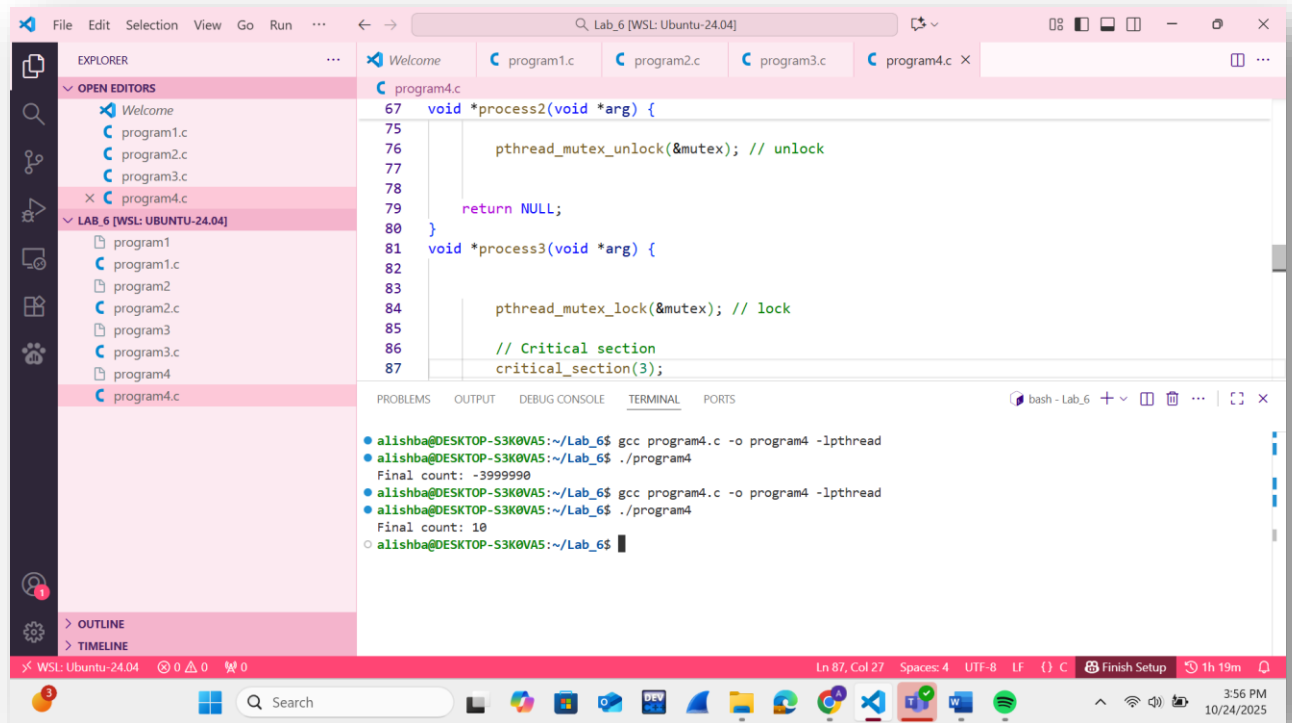
        pthread_mutex_destroy(&mutex); // destroy mutex

        printf("Final count: %d\n", count);

        return 0;
    }

```

Output:



```
67 void *process2(void *arg) {
75     pthread_mutex_unlock(&mutex); // unlock
76
77
78
79     return NULL;
80 }
81 void *process3(void *arg) {
82
83     pthread_mutex_lock(&mutex); // lock
84
85     // Critical section
86     critical_section(3);
87 }
```

```
alishba@DESKTOP-S3K0VA5:~/Lab_6$ gcc program4.c -o program4 -lpthread
alishba@DESKTOP-S3K0VA5:~/Lab_6$ ./program4
Final count: -39999990
alishba@DESKTOP-S3K0VA5:~/Lab_6$ gcc program4.c -o program4 -lpthread
alishba@DESKTOP-S3K0VA5:~/Lab_6$ ./program4
Final count: 10
alishba@DESKTOP-S3K0VA5:~/Lab_6$
```

Comparison Between Peterson Algorithm and one with Mutex:

Feature	Peterson's Algorithm	Mutex Version
Critical Section Protection	Software-only: flag[] + turn	OS-level: pthread_mutex_lock/unlock
Number of Threads Supported	2 only	Any number (e.g., 8 threads)
Waiting Mechanism	Busy wait: while(flag[1]==1 && turn==1)	Blocking: pthread_mutex_lock() suspends the thread
Code Complexity	Requires flag[], turn, and busy wait logic	Simple: lock → critical section → unlock
Scalability	Hard to extend beyond 2 threads	Easy to extend to many threads/processes
CPU Usage While Waiting	High (spins continuously)	Low (thread sleeps until lock is free)
Ease of Maintenance	Harder to read and extend	Easy to read, maintain, and reuse
Example Thread Creation	Only 2 threads (thread0, thread1)	Multiple threads (thread0–thread7)

Use Case	Educational, software mutual exclusion demo	Practical, real-world multithreading
Final Count Control	Works for 2 threads if increments/decrements are balanced	Works for any number of threads if mutex protects critical section