



# National Textile University

*Department of Computer Science*

Subject:

**Operating System**

Submitted To:

**Sir Nasir Mehmood**

Submitted By:

**Hafsa Tayyab**

Registration No:

**23-NTU-CS-1163**

Lab No:

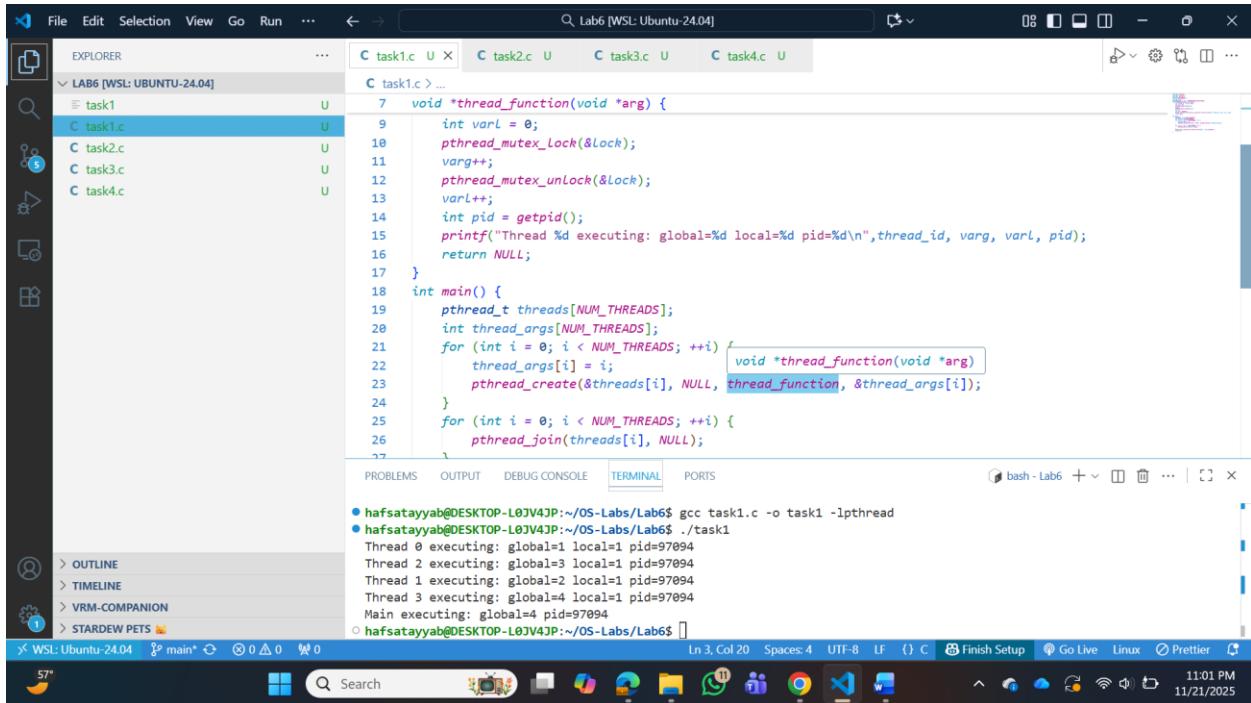
**6**

Semester:

**5<sup>th</sup>**

# Lab 6: Synchronization

## Program 1: Fixing Race Condition using Mutex



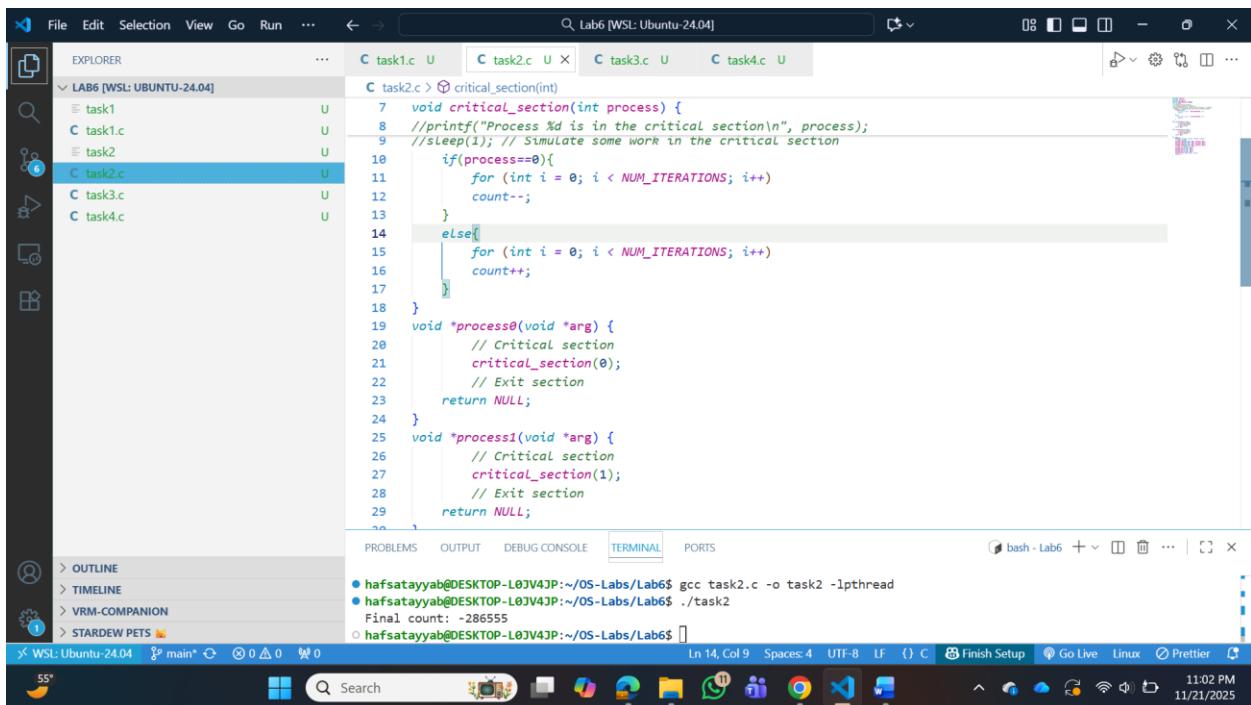
```
task1.c
task2.c
task3.c
task4.c

task1.c > ...
7 void *thread_function(void *arg) {
9     int varl = 0;
10    pthread_mutex_lock(&lock);
11    varg++;
12    pthread_mutex_unlock(&lock);
13    varl++;
14    int pid = getpid();
15    printf("Thread %d executing: global=%d local=%d pid=%d\n", thread_id, varg, varl, pid);
16    return NULL;
17 }
18 int main() {
19     pthread_t threads[NUM_THREADS];
20     int thread_args[NUM_THREADS];
21     for (int i = 0; i < NUM_THREADS; ++i) {
22         thread_args[i] = i;
23         pthread_create(&threads[i], NULL, thread_function, &thread_args[i]);
24     }
25     for (int i = 0; i < NUM_THREADS; ++i) {
26         pthread_join(threads[i], NULL);
27     }
}

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

● hafsatayyab@DESKTOP-L0JV4JP:~/OS-Labs/Lab6$ gcc task1.c -o task1 -lpthread
● hafsatayyab@DESKTOP-L0JV4JP:~/OS-Labs/Lab6$ ./task1
Thread 0 executing: global=1 local=1 pid=97094
Thread 2 executing: global=3 local=1 pid=97094
Thread 1 executing: global=2 local=1 pid=97094
Thread 3 executing: global=4 local=1 pid=97094
Main executing: global=4 pid=97094
hafsatayyab@DESKTOP-L0JV4JP:~/OS-Labs/Lab6$ 
```

## Program 2: Without Peterson's Algorithm



```
task1.c
task2.c
task3.c
task4.c

task2.c > critical_section(int)
7 void critical_section(int process) {
8 //printf("Process %d is in the critical section\n", process);
9 //sleep(1); // Simulate some work in the critical section
10 if(process==0){
11     for (int i = 0; i < NUM_ITERATIONS; i++)
12         count--;
13 }
14 else{
15     for (int i = 0; i < NUM_ITERATIONS; i++)
16         count++;
17 }
18 void *process0(void *arg) {
19     // Critical section
20     critical_section(0);
21     // Exit section
22     return NULL;
23 }
24 void *process1(void *arg) {
25     // Critical section
26     critical_section(1);
27     // Exit section
28     return NULL;
29 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

● hafsatayyab@DESKTOP-L0JV4JP:~/OS-Labs/Lab6$ gcc task2.c -o task2 -lpthread
● hafsatayyab@DESKTOP-L0JV4JP:~/OS-Labs/Lab6$ ./task2
Final count: -286555
hafsatayyab@DESKTOP-L0JV4JP:~/OS-Labs/Lab6$ 
```

## Program 3: With Peterson's Algorithm

A screenshot of the Visual Studio Code interface running in WSL Ubuntu-24.04. The code editor shows four files: task1.c, task2.c, task3.c, and task4.c. task3.c is the active file, containing the Peterson's algorithm implementation. The terminal below shows the build and run commands for task3.c.

```
task3.c // Critical section(int)
10 void critical_section(int process) {
11 }
12 // Peterson's Algorithm function for process 0
13 void *process0(void *arg) {
14     flag[0] = 1;
15     turn = 1;
16     while (flag[1]==1 && turn == 1) {
17         // Busy wait
18     }
19     // Critical section
20     critical_section(0);
21     // Exit section
22     flag[0] = 0;
23     //sleep(1);
24     pthread_exit(NULL);
25 }
26 // Peterson's Algorithm function for process 1
27 void *process1(void *arg) {
28     flag[1] = 1;
29     turn = 0;
30     while (flag[0]==1 && turn == 0) {
31         // Busy wait
32     }
33     // Critical section
34     critical_section(1);
35     // Exit section
36     flag[1] = 0;
37     //sleep(1);
38     pthread_exit(NULL);
39 }
```

Terminal output:

```
hafsatayyab@DESKTOP-L03V4JP:~/OS-Labs/Lab6$ gcc task3.c -o task3 -lpthread
hafsatayyab@DESKTOP-L03V4JP:~/OS-Labs/Lab6$ ./task3
Final count: 0
hafsatayyab@DESKTOP-L03V4JP:~/OS-Labs/Lab6$
```

## Program 4: With Mutex

A screenshot of the Visual Studio Code interface running in WSL Ubuntu-24.04. The code editor shows four files: task1.c, task2.c, task3.c, and task4.c. task4.c is the active file, containing a mutex-based algorithm implementation. The terminal below shows the build and run commands for task4.c.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define NUM_ITERATIONS 1000000
int count=0;
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++;
    }
}
```

Terminal output:

```
hafsatayyab@DESKTOP-L03V4JP:~/OS-Labs/Lab6$ gcc task4.c -o task4 -lpthread
hafsatayyab@DESKTOP-L03V4JP:~/OS-Labs/Lab6$ ./task4
Final count: 10
hafsatayyab@DESKTOP-L03V4JP:~/OS-Labs/Lab6$
```

Aspect	Peterson's Algorithm	Mutex ( <code>pthread_mutex</code> , OS lock)
Type	Software-only mutual exclusion	OS-supported + hardware-assisted lock
Thread Support	Only 2 threads (classic version)	Any number of threads
Implementation	Uses shared variables: <code>flag[]</code> , <code>turn</code>	Uses atomic CPU instructions + kernel support
Correctness on Modern CPUs	Not guaranteed due to weak memory ordering	Guaranteed — includes memory barriers
Performance	Busy waiting (spinlock) → wastes CPU	Blocks when waiting → efficient
Fairness / Bounded Waiting	Guaranteed theoretically	Depends on implementation but generally fair
Reliability	Fragile: affected by compiler optimizations, caching	Very reliable and widely used
Scalability	Poor (only 2 threads, slow)	Excellent
Use Case	Educational purposes only	Real-world synchronization in programs
Ease of Use	Hard to implement correctly	Easy API: <code>pthread_mutex_lock()</code>
Memory Requirements	Few shared variables	Mutex structure (small but more overhead)
Kernel Involvement	None (pure userspace spinlock)	May block and involve scheduler
Where It Works	Only in strictly sequentially consistent models	Works everywhere (Linux, Windows, macOS, multicore CPUs)