



National Textile University

Department of Computer Science

Subject:

Operating system

Submitted To:

Sir Nasir

Submitted By:

Noor ul Ain

Registration No:

23-NTU-CS-1221

Lab No:

6

Semester:

5th

Task1:

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 vale is %d: process id %d: \n", thread_id, varg, varl, getpid());
    return NULL;
}

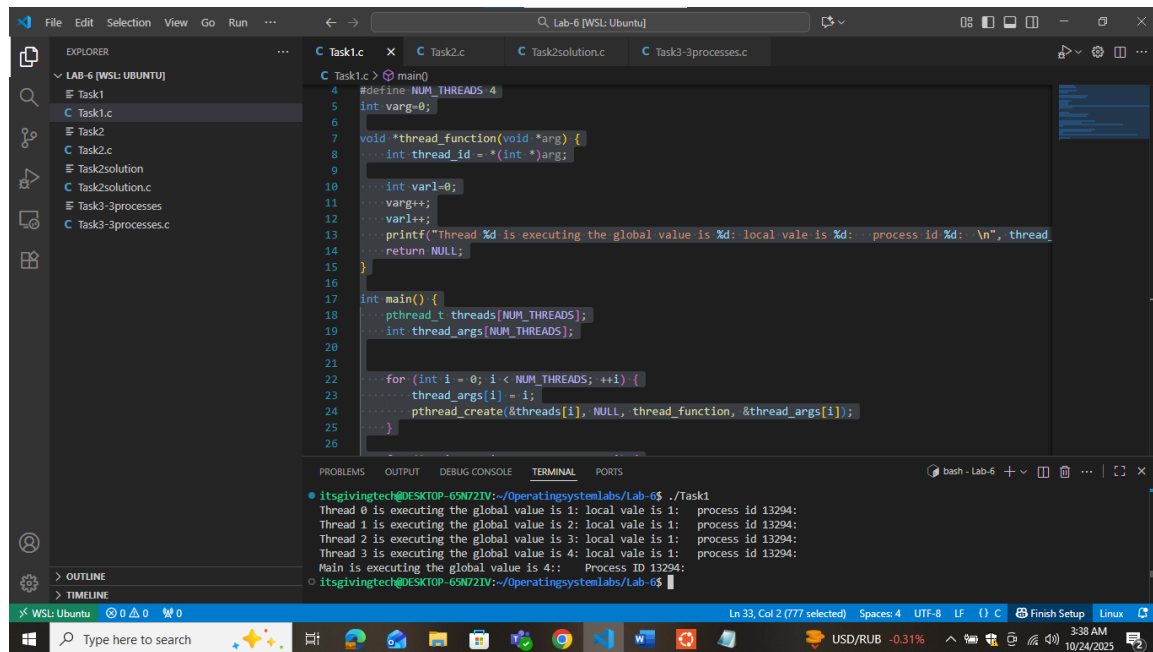
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];

    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 a Visual Studio Code editor with a C program named `Task1.c` open. The program defines 4 threads and a global variable `var`. The `main` function creates 4 threads, each of which increments `var` and prints its value along with the thread ID and process ID. The terminal output shows the execution of the program, with each thread printing its value and the main function printing the final value of `var`.

```
4 #define NUM_THREADS 4
5 int var=0;
6
7 void *thread_function(void *arg) {
8     int thread_id = *(int *)arg;
9
10    int var1=0;
11    var++;
12    var1++;
13    printf("Thread %d is executing the global value is %d: local vale is %d: process id %d: \n", thread_id, var, var1, getpid());
14    return NULL;
15 }
16
17 int main() {
18     pthread_t threads[NUM_THREADS];
19     int thread_args[NUM_THREADS];
20
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
26     pthread_join(threads[0], NULL);
27     pthread_join(threads[1], NULL);
28     pthread_join(threads[2], NULL);
29     pthread_join(threads[3], NULL);
30
31     printf("Main is executing the global value is %d: Process ID %d\n", var, getpid());
32 }
```

Terminal Output:

```
itgivingtech@DESKTOP-65N72IV:~/OperatingSystemLabs/Lab-6$ ./Task1
Thread 0 is executing the global value is 1: local vale is 1: process id 13294:
Thread 1 is executing the global value is 2: local vale is 1: process id 13294:
Thread 2 is executing the global value is 3: local vale is 1: process id 13294:
Thread 3 is executing the global value is 4: local vale is 1: process id 13294:
Main is executing the global value is 4: Process ID 13294:
itgivingtech@DESKTOP-65N72IV:~/OperatingSystemLabs/Lab-6$
```

Task 2:

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 a Visual Studio Code editor window with a file explorer on the left and a terminal at the bottom. The file explorer shows a project named 'LAB-6 [WSL: UBUNTU]' with files 'Task1.c', 'Task2.c', 'Task2solution.c', 'Task3-3processes.c', and 'Task3-3processes.c'. The 'Task2.c' file is open in the editor, showing a C program that creates four threads (thread0, thread1, thread2, thread3) and joins them. The program prints 'Final count: 8713'. The terminal at the bottom shows the command 'bash - Lab-6' and the output 'Final count: 8713'.

```

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;
}

```

```

bash - Lab-6
Final count: 8713

```

Task 3:

With peterson's 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) {
    //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++;
    }
    // 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) {

    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);

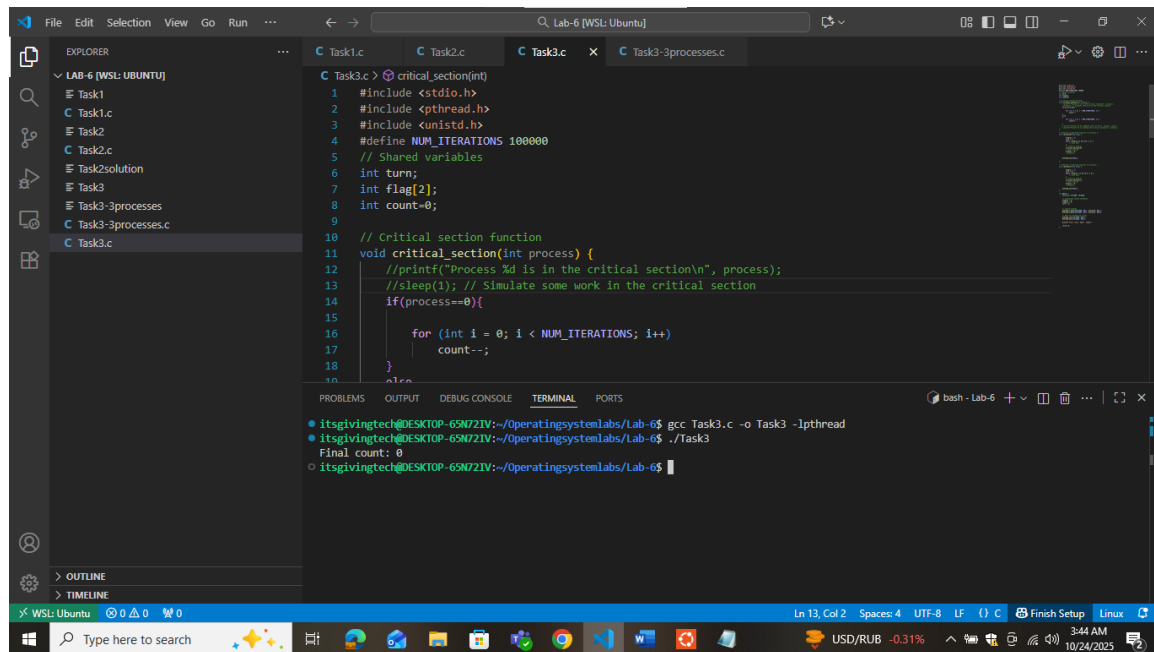
    // Wait for threads to finish
    pthread_join(thread0, NULL);
    pthread_join(thread1, NULL);

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

    return 0;
}

```

Output:



The screenshot shows a Visual Studio Code editor window titled 'Lab-6 [WSL: Ubuntu]'. The Explorer panel on the left shows a project structure with files 'Task1.c', 'Task2.c', 'Task3.c', 'Task3-3processes.c', and 'Task3.c'. The main editor area displays the code for 'Task3.c', which includes headers for `<stdio.h>`, `<pthread.h>`, and `<unistd.h>`, and defines `NUM_ITERATIONS` as 100000. It declares shared variables `int turn;`, `int flag[2];`, and `int count=0;`. A critical section function is defined as follows:

```
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     if(process==0){
15         for (int i = 0; i < NUM_ITERATIONS; i++)
16             count--;
17     }
18 }
```

The bottom panel shows the TERMINAL output with the following commands and results:

```
itsgivingtech@DESKTOP-65N72IV:~/OperatingSystemLabs/Lab-6$ gcc Task3.c -o Task3 -lpthread
itsgivingtech@DESKTOP-65N72IV:~/OperatingSystemLabs/Lab-6$ ./Task3
Final count: 0
itsgivingtech@DESKTOP-65N72IV:~/OperatingSystemLabs/Lab-6$
```

Task 4:

3 processes:

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 {
            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;
}

int main() {
    pthread_t thread0, thread1, thread2, thread3, thread4, thread5;

    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, process0, NULL);
    pthread_create(&thread4, NULL, process1, NULL);
    pthread_create(&thread5, NULL, process2, 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_mutex_destroy(&mutex); // destroy mutex

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

    return 0;
}

```

output:

```

File Edit Selection View Go Run ...
Lab-6 [WSL: Ubuntu]
EXPLORER
  Lab-6 [WSL: Ubuntu]
    Task1
    Task1.c
    Task2
    Task2.c
    Task2solution
    Task3
    Task3-3processes
    Task3.c
    Task4
    Task4.c
  C Task4.c
    critical_section(int)
    #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)
      {
    }
  }
  PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
  bash - Lab-6
  itsgivingtech@DESKTOP-65N72IV:~/OperatingSystemLabs/Lab-6$ gcc Task4.c -o Task4 -lpthread
  itsgivingtech@DESKTOP-65N72IV:~/OperatingSystemLabs/Lab-6$ ./Task4
  Final count: -1999990
  itsgivingtech@DESKTOP-65N72IV:~/OperatingSystemLabs/Lab-6$
  Ln 27, Col 16 Spaces: 4 UTF-8 LF C Finish Setup Linux
  Type here to search USD/RUB -0.31% 3:45 AM 10/24/2023

```

Remarks:

Difference between Mutex and Peterson algorithm:

Peterson's Algorithm	Mutex
A software-only method for handling mutual exclusion between two processes.	A built-in synchronization tool provided by the operating system.
Works only for two processes at a time.	Can easily handle many processes or threads .
Uses two shared variables (<i>flag</i> and <i>turn</i>) to decide which process enters the critical section.	Uses lock and unlock functions to control access automatically.
Wastes CPU time because processes keep checking (busy waiting) until they get access.	Doesn't waste CPU — threads simply wait until the lock is available.
Purely a software solution — doesn't rely on OS or hardware support.	Depends on the operating system and hardware for thread management.
Not very efficient; mainly used for understanding how synchronization works.	Much faster and efficient for real-world programs.
Harder to write and more error-prone.	Simple and easy to use with built-in thread libraries.
Mostly used for learning and theory in computer science.	Commonly used in real applications and operating systems .

Peterson's Algorithm	Mutex
Makes sure both processes take turns using the <code>turn</code> variable.	Fairness is managed by the OS; some systems add features like priority control .