



National Textile University

Department of Computer Science

Subject:

Operating System

Submitted to:

Nasir Mahmood

Submitted by:

Afia Maham

Reg number:

23-NTU-CS-1126

Lab no:

05

Semester:

5th

Task01:

```
#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;
}

```

The screenshot shows a Visual Studio Code editor window titled "Operating System [WSL: Ubuntu-24.04]". The Explorer panel on the left shows a project structure with folders "Lab-3", "Lab-4", "Lab-5", "Lab-6", and "Lab-6-hometask". Under "Lab-6", there are files "task01-out", "task01.c", "task02-out", "task02.c", "task03.c", "task04.c", and "task04-out". The main editor window displays the code for "task01.c". The code defines a global array "thread_args" and a function "thread_function" that increments a global variable "varg" and prints the thread ID and varg. The main function creates four threads and joins them. The Output panel at the bottom shows the execution results, including the compilation command and the output of the program.

```

// task01.c
17 int main() {
18     int thread_args[NUM_THREADS];
19
20     for (int i = 0; i < NUM_THREADS; ++i) {
21         thread_args[i] = i;
22         pthread_create(&threads[i], NULL, thread_function, &thread_args[i]);
23     }
24
25     for (int i = 0; i < NUM_THREADS; ++i) {
26         pthread_join(threads[i], NULL);
27     }
28     printf("Main is executing the global value is %d:: Process ID %d: \n", varg, getpid());
29 }
30
// Output
aflamham@DESKTOP-190QPQH:~/Operating System$ gcc Lab-6/task01.c
aflamham@DESKTOP-190QPQH:~/Operating System$ ./Lab-6/task01-out
Thread 0 is executing the global value is 1: local vale is 1: process id 1778:
Thread 2 is executing the global value is 3: local vale is 1: process id 1778:
Thread 3 is executing the global value is 4: local vale is 1: process id 1778:
Thread 1 is executing the global value is 2: local vale is 1: process id 1778:
Main is executing the global value is 4:: Process ID 1778:
aflamham@DESKTOP-190QPQH:~/Operating System$

```

Task 02:

```

#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;
}

```

The screenshot shows a Visual Studio Code editor window titled "Operating System [WSL: Ubuntu-24.04]". The Explorer pane on the left shows a project structure with folders "assignment-1", "Lab-3", "Lab-4", "Lab-5", "Lab-6", and "Lab6-hometask". Under "Lab-6", there are files "task01.c", "task02.c", "task03.c", and "task04.c". The main editor window displays the code for "task02.c", which includes headers for stdio, pthread, and unistd, defines NUM_ITERATIONS as 100000, and contains a critical_section function. The terminal at the bottom shows the compilation and execution of the program, resulting in a final count of -67887.

```

1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h>
4  #define NUM_ITERATIONS 100000
5
6  int count=10;
7
8  // Critical section function
9  void critical_section(int process) {
10     //printf("Process %d is in the critical section\n", process);
11     //sleep(1); // Simulate some work in the critical section
12     if(process==0){
13
14     }
15 }

```

```

aflamaham@DESKTOP-190QPQH:~/Operating System$ gcc Lab-6/task02.c -o Lab-6/task02-out
aflamaham@DESKTOP-190QPQH:~/Operating System$ ./Lab-6/task02-out
Final count: -67887
aflamaham@DESKTOP-190QPQH:~/Operating System$ Lab-6/task02-out
Final count: -247307
aflamaham@DESKTOP-190QPQH:~/Operating System$ Lab-6/task02-out
Final count: -276101
aflamaham@DESKTOP-190QPQH:~/Operating System$

```

Task 03:

```

#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
```

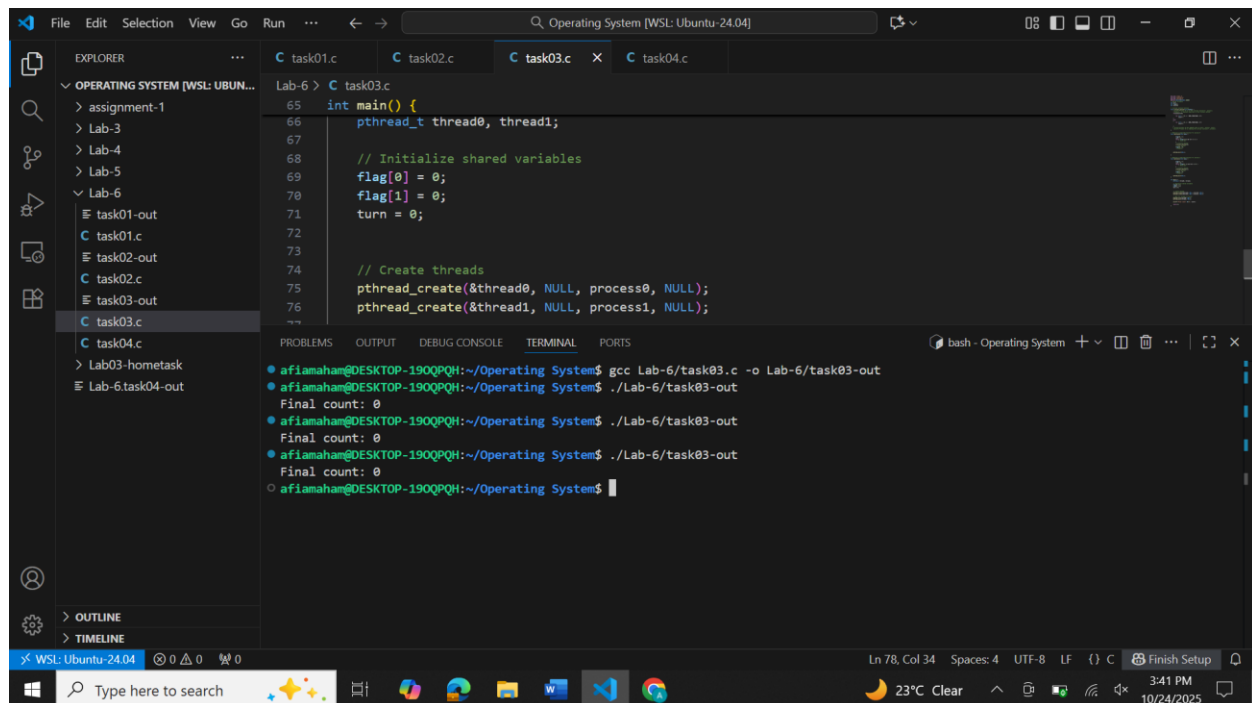
```
pthread_join(thread0, NULL);
```

```
pthread_join(thread1, NULL);
```

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

```
return 0;
```

```
}
```



The screenshot shows the Visual Studio Code interface with a C file named `task03.c` open. The code defines a `main` function that creates two threads, `thread0` and `thread1`, using `pthread_create`. Both threads call `process0` and `process1` respectively. The `main` function then joins both threads using `pthread_join` and prints the final count. The terminal output shows the compilation and execution of the program, resulting in a final count of 0.

```
65 int main() {
66     pthread_t thread0, thread1;
67
68     // Initialize shared variables
69     flag[0] = 0;
70     flag[1] = 0;
71     turn = 0;
72
73
74     // Create threads
75     pthread_create(&thread0, NULL, process0, NULL);
76     pthread_create(&thread1, NULL, process1, NULL);
```

```
bash - Operating System
aflamahan@DESKTOP-190QPQH:~/Operating System$ gcc Lab-6/task03.c -o Lab-6/task03-out
aflamahan@DESKTOP-190QPQH:~/Operating System$ ./Lab-6/task03-out
Final count: 0
aflamahan@DESKTOP-190QPQH:~/Operating System$ ./Lab-6/task03-out
Final count: 0
aflamahan@DESKTOP-190QPQH:~/Operating System$ ./Lab-6/task03-out
Final count: 0
aflamahan@DESKTOP-190QPQH:~/Operating System$
```

Task 04:

```
#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 += 5;
        }
    }
    //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(1);
    // 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;  
}
```

```
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     if(process==0){
15
16         for (int i = 0; i < NUM_ITERATIONS; i++)
17             count--;
18     }
19     else if (process == 1)
```

```
aflamham@DESKTOP-190QPQH:~/Operating System$ gcc Lab-6/task04.c -o Lab-6/task04-out
aflamham@DESKTOP-190QPQH:~/Operating System$ ./Lab-6/task04-out
Final count: 2000010
aflamham@DESKTOP-190QPQH:~/Operating System$ ./Lab-6/task04-out
Final count: 2000010
aflamham@DESKTOP-190QPQH:~/Operating System$ ./Lab-6/task04-out
Final count: 2000010
aflamham@DESKTOP-190QPQH:~/Operating System$
```

Comparison between Peterson's Algorithm and Mutex

1. Type of Synchronization:

Peterson's Algorithm is a software-based method that uses shared variables to make sure only one process enters the critical section at a time. A mutex, on the other hand, is an operating system feature that relies on hardware support. Because of this, mutexes are much more reliable and are commonly used in real-world programs.

2. Process Limitation:

Peterson's Algorithm is designed to work with only two processes. If you try to use it with three or more, it becomes complicated and unreliable. Mutexes don't have this issue, they can easily handle multiple threads or processes at once, which makes them more practical for modern systems.

3. Efficiency:

Peterson's Algorithm uses something called *busy waiting*, meaning it keeps checking repeatedly to see if it can enter the critical section. This wastes CPU time. Mutexes are smarter, they make a thread wait quietly until the lock becomes available, which saves CPU resources and improves overall performance.