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

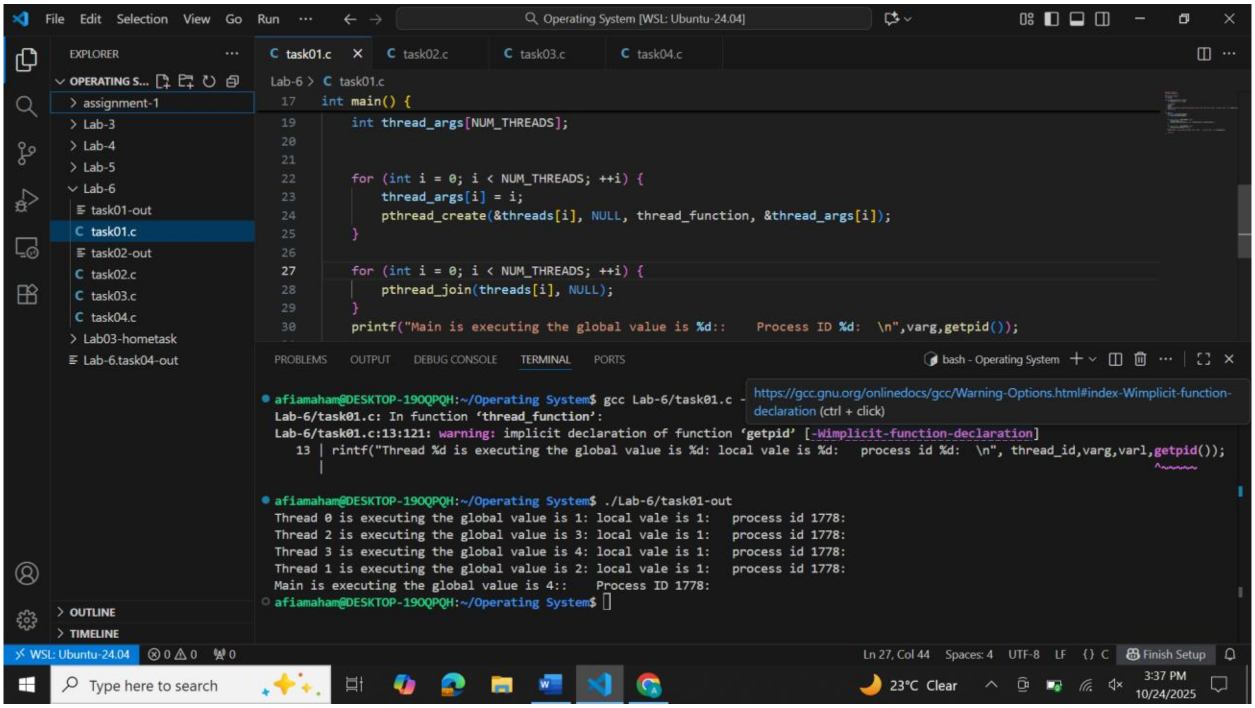
Semester:

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;
}
```

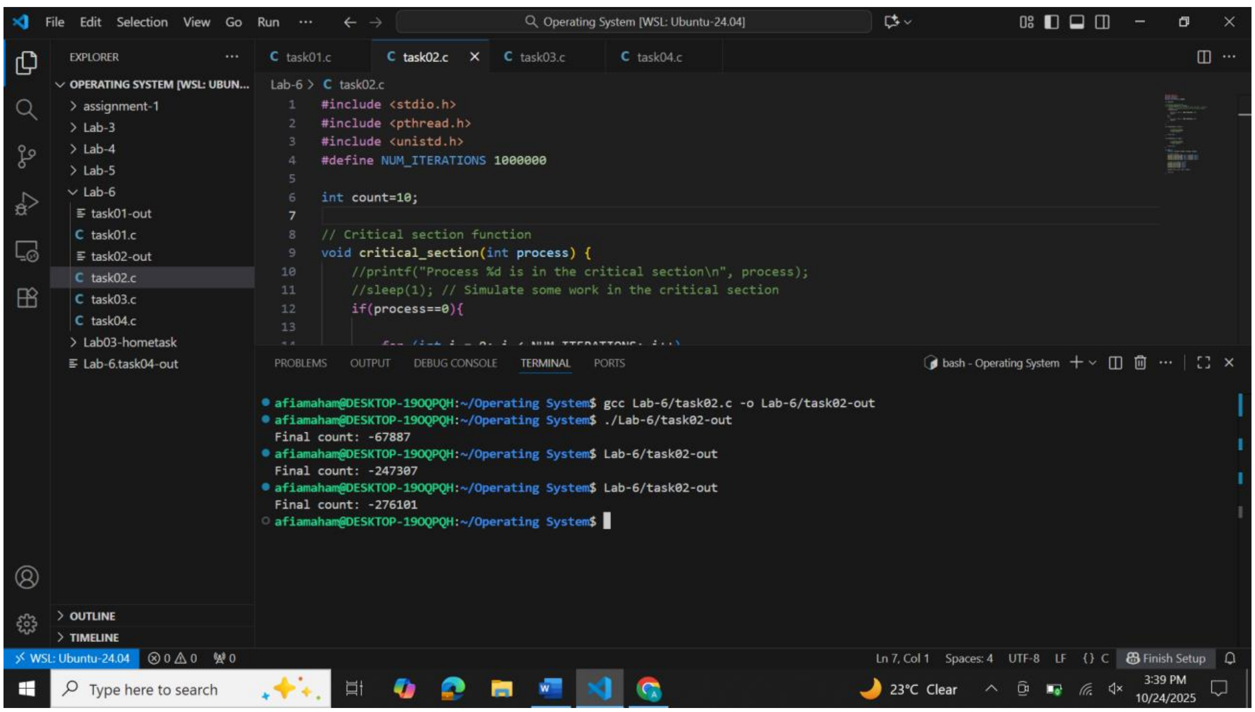


Task 02:

```
#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);
```

```c
    // 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;
}
```



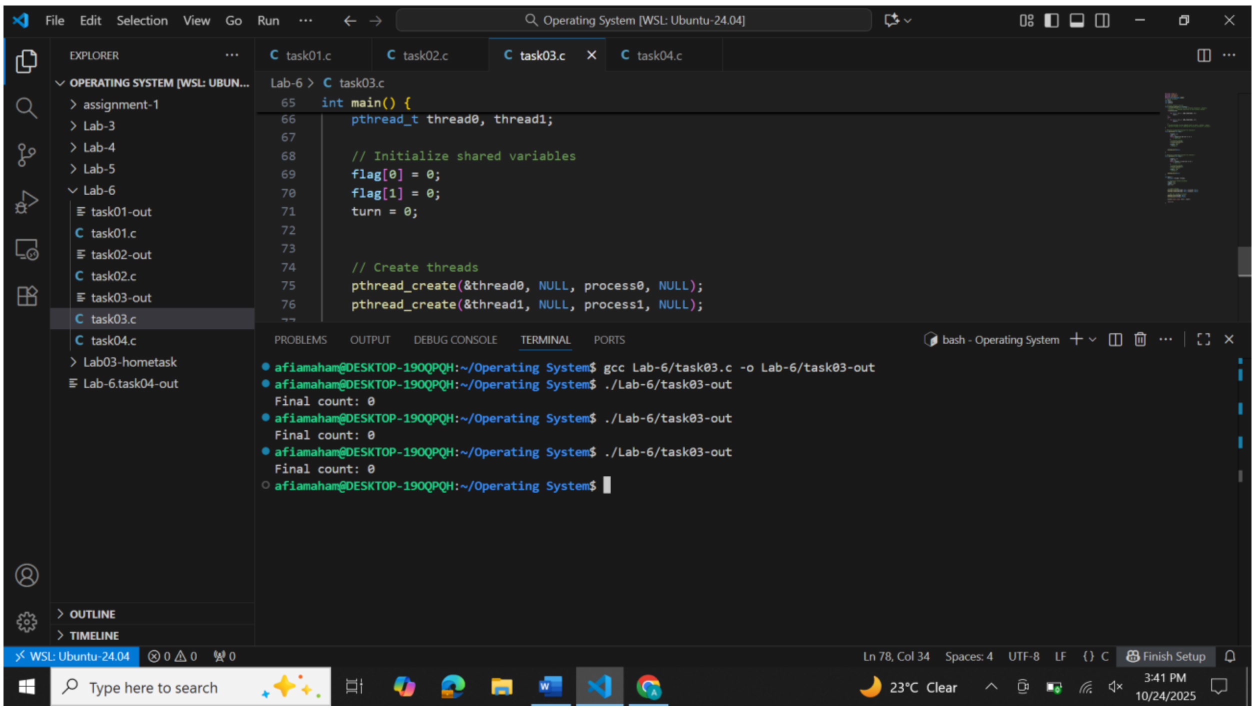
Task 03:

```c
#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;
}
```



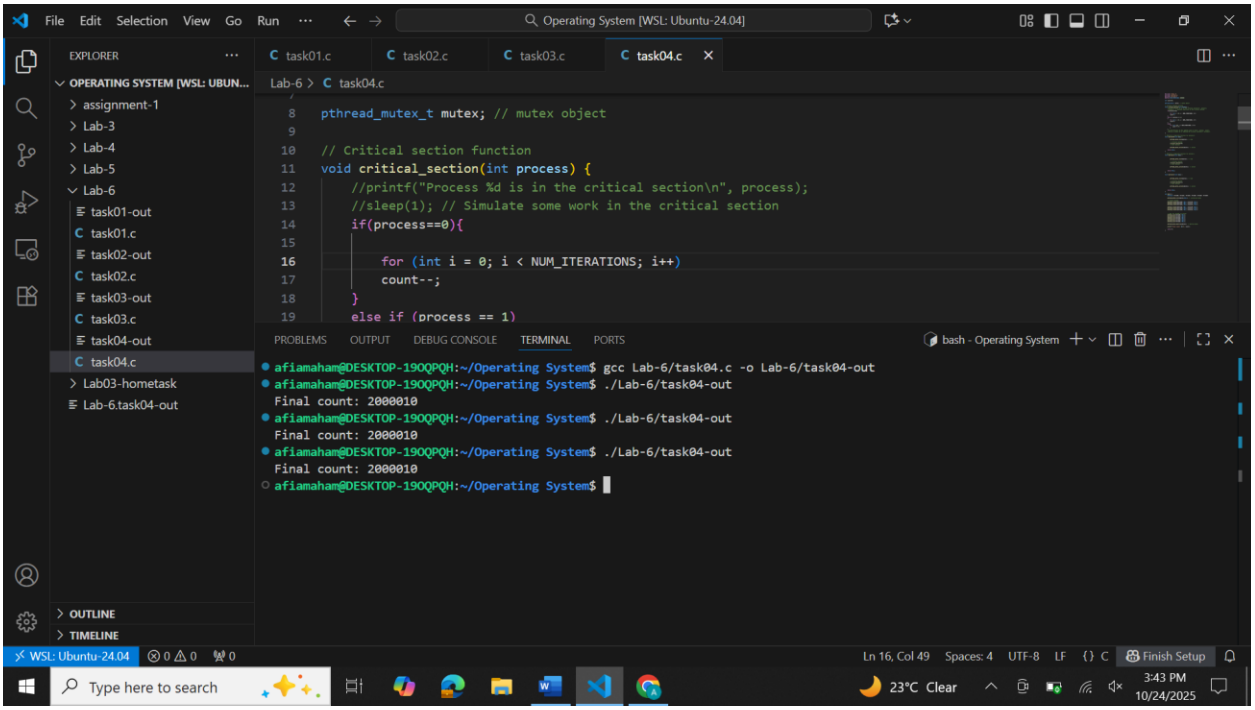
Task 04:

```
#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;
}
```



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.