National Textile University

Department of Computer Science

Subject:
Operating system

Submitted to:

Sir Nasir Mehmood

Submitted by:

Areeba Tariq

Reg number:

23-NTU-CS-1139

Lab no:
6th

Semester:

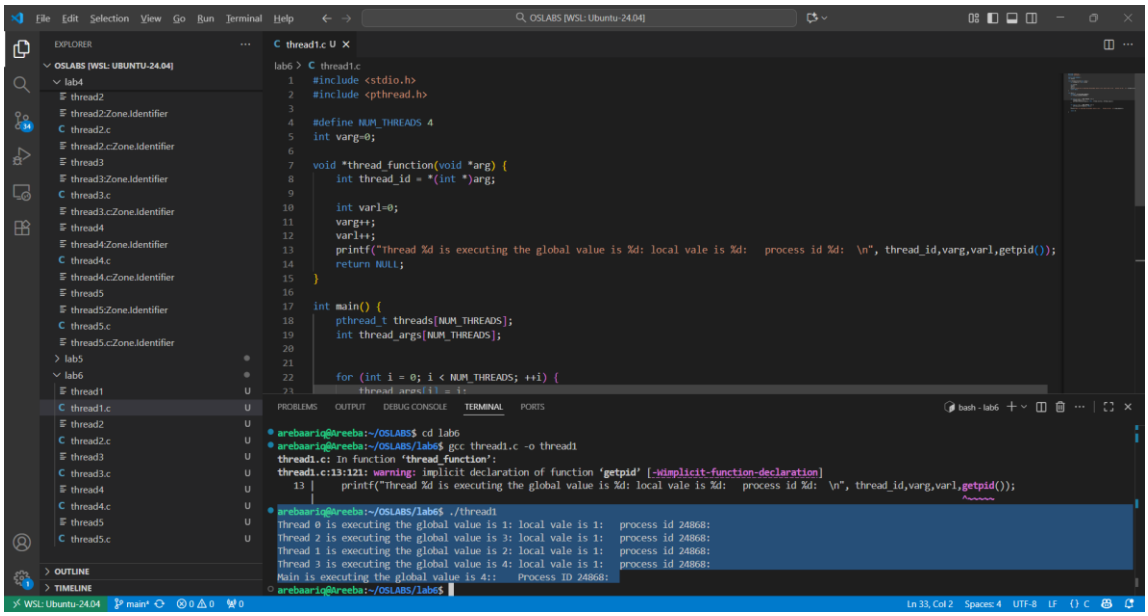
5th

Task 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 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



Remarks

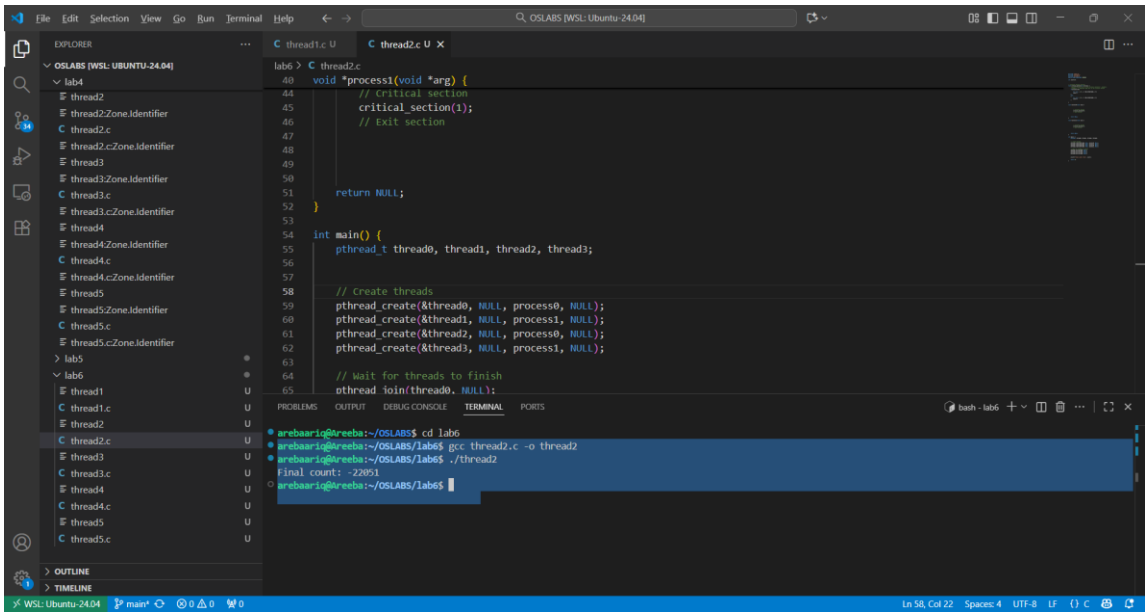
Task 2

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



Remarks

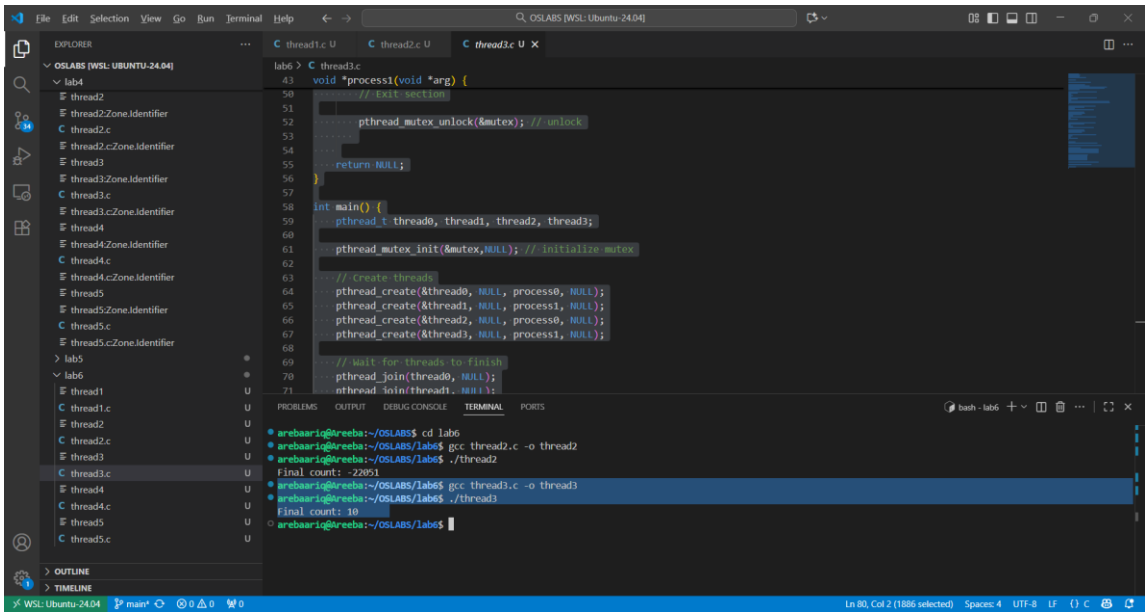
Task 3

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
    {
        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;
}

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

    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, 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);

    pthread_mutex_destroy(&mutex); // destroy mutex

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

    return 0;
}
```

Output



Remarks

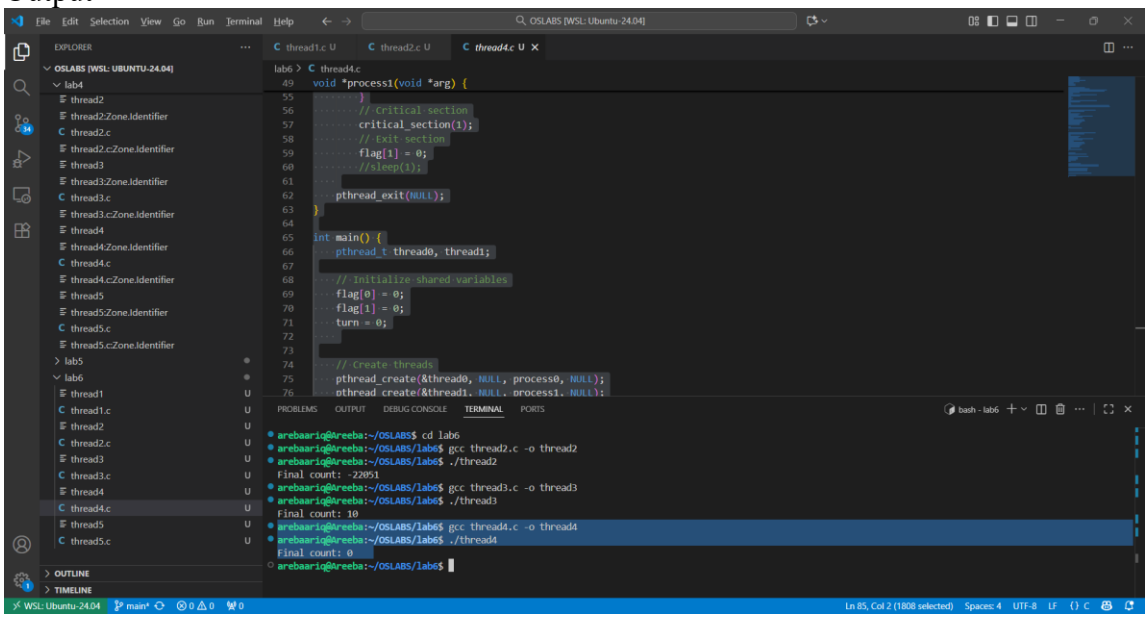
Task 4

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



Remarks

This program uses **Peterson's Algorithm** to safely control access to a shared variable count between two threads. It ensures **mutual exclusion** using flags and a turn variable so both threads update count without race conditions. The final value of count confirms correct synchronization.

Task 6

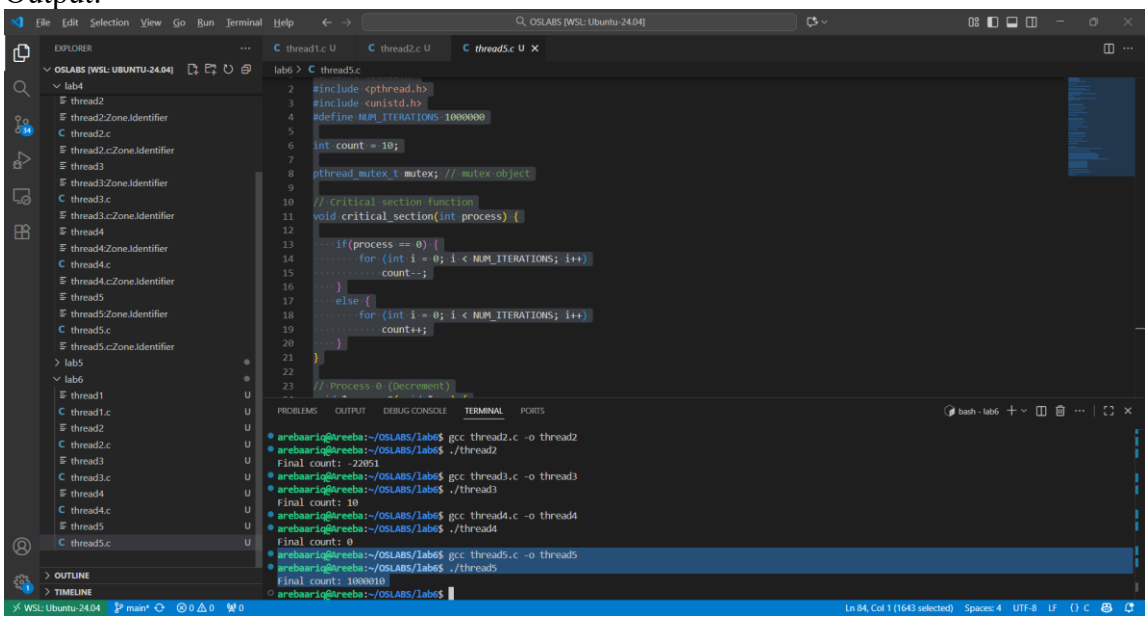
Code:

```
#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) {

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

// Process 0 (Decrement)
void *process0(void *arg) {

   pthread_mutex_lock(&mutex);

   critical_section(0);
```

```
    pthread_mutex_unlock(&mutex);

    return NULL;
}

// Process 1 (Increment)
void *process1(void *arg) {

    pthread_mutex_lock(&mutex);

    critical_section(1);

    pthread_mutex_unlock(&mutex);

    return NULL;
}

// ✅ New Process 2 (Increment)
void *process2(void *arg) {

    pthread_mutex_lock(&mutex);

    critical_section(2);

    pthread_mutex_unlock(&mutex);

    return NULL;
}

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

    pthread_mutex_init(&mutex, NULL);

    // Create threads (same pattern + 1 more)
    pthread_create(&thread0, NULL, process0, NULL);
    pthread_create(&thread1, NULL, process1, NULL);
    pthread_create(&thread2, NULL, process0, NULL);
    pthread_create(&thread3, NULL, process1, NULL);
    pthread_create(&thread4, NULL, process2, NULL); // 👈 new thread

    // 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_mutex_destroy(&mutex);

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

    return 0;
```

}

Output:



Remarks

This program uses **mutex locking** to protect a shared variable count during updates by multiple threads. Three different thread functions modify count by incrementing or decrementing it, while pthread_mutex prevents race conditions and ensures correct final output.