National Textile University

Department of Computer Science

Subject: Operating System

Submitted to: Sir Nasir Mahmood

Submitted by: Nimra Tanveer

Reg number:23-NTU-CS-1201

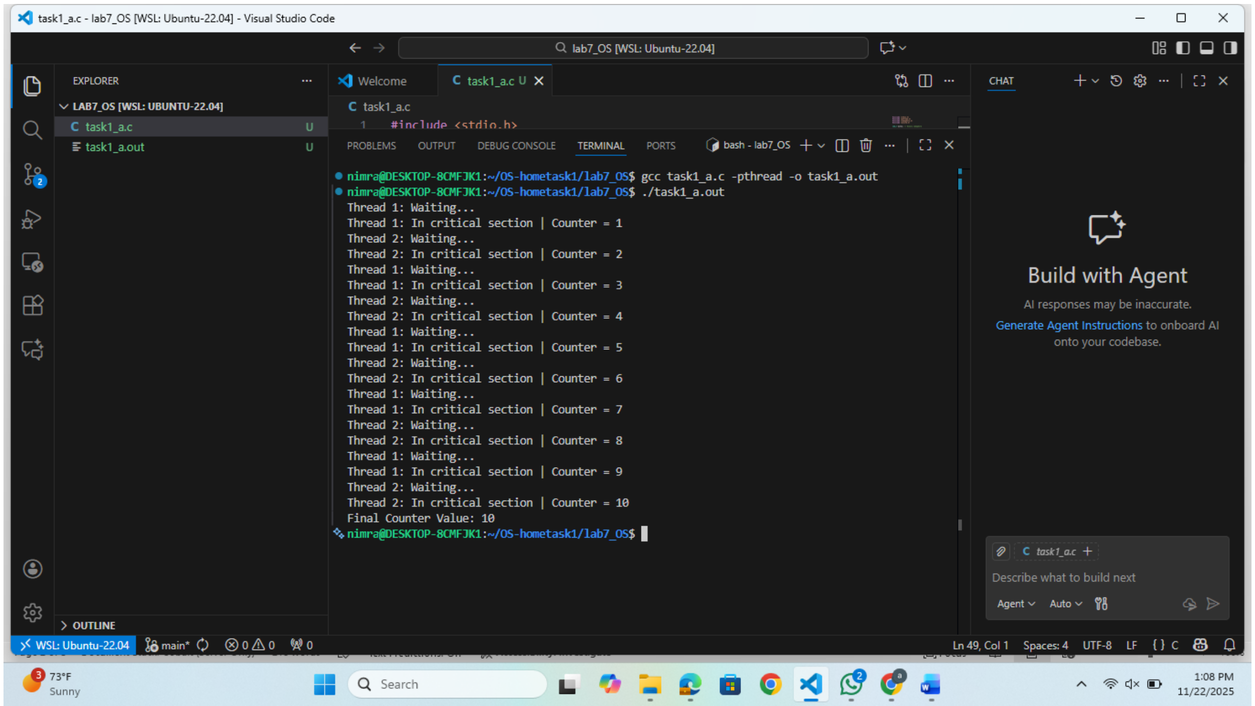
Lab no .07

Task 1: 4.3. Binary Semaphore Example

Part a

- **Two threads run at the same time, but a binary semaphore makes sure only one thread enters the critical section at a time.**
- **Each thread increases to a shared variable counter safely without conflict.**
- **In the end, the program prints the final counter value after both threads finish.**

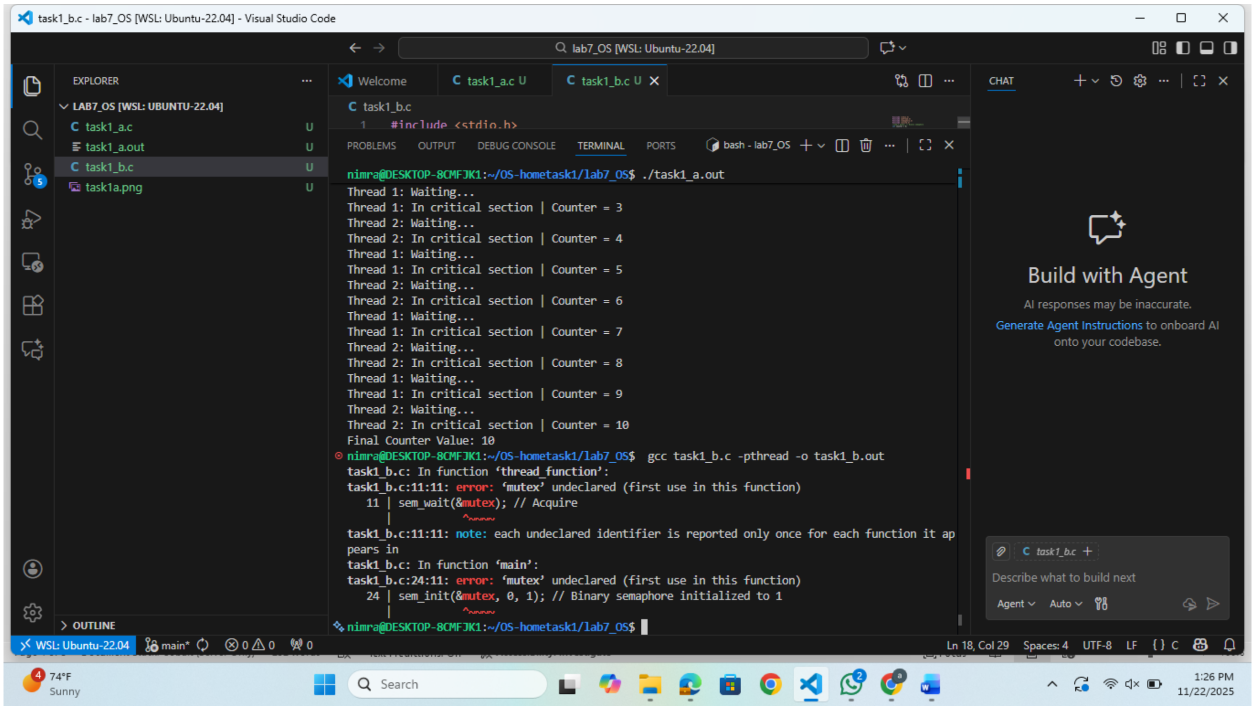
```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex; // Binary semaphore
int counter = 0;

void* thread_function(void* arg) {
    int id = *(int*)arg;

    for (int i = 0; i < 5; i++) {
        printf("Thread %d: Waiting...\n", id);
        sem_wait(&mutex); // Acquire

        // 4.4 Counting Semaphore Example
        // A counting semaphore with initial value = 3 allows
        // up to 3 threads to access a resource simultaneously.

        // Critical section
        counter++;
        printf("Thread %d: In critical section | Counter = %d\n", id, counter);
        sleep(1);

        sem_post(&mutex); // Release
        sleep(1);
    }

    return NULL;
}

int main() {
    sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1

    pthread_t t1, t2;
    int id1 = 1, id2 = 2;

    pthread_create(&t1, NULL, thread_function, &id1);
    pthread_create(&t2, NULL, thread_function, &id2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final Counter Value: %d\n", counter);

    sem_destroy(&mutex);
    return 0;
}
```



Task b: commenting sem_t mutex

- **Commenting on sem_t mutex means the semaphore does not exist, so a compile-time error will occur.**
- **The program will not run because there is no defined semaphore named mutex.**

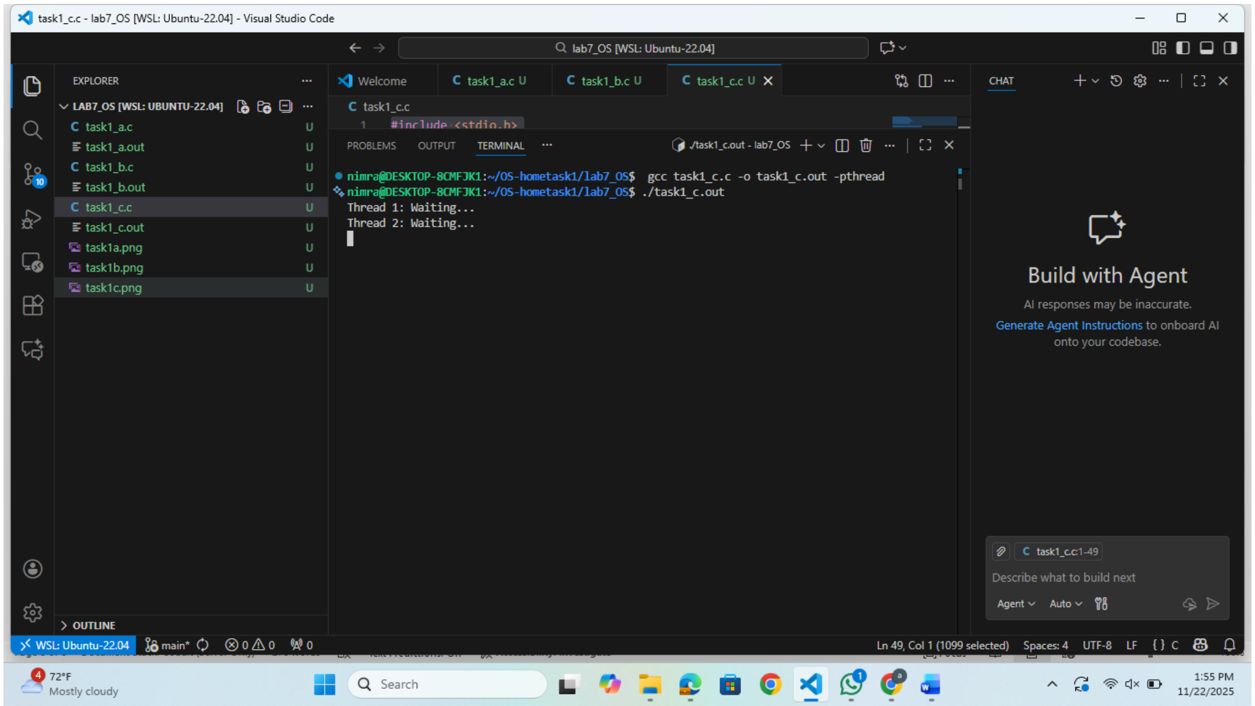
```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
//sem_t mutex; // Binary semaphore
int counter = 0;
void* thread_function(void* arg) {
int id = *(int*)arg;
for (int i = 0; i < 5; i++) {
printf("Thread %d: Waiting...\n", id);
sem_wait(&mutex); // Acquire

// Critical section
counter++;
printf("Thread %d: In critical section | Counter = %d\n", id,
counter);
sleep(1);
sem_post(&mutex); // Release
sleep(1);
}
return NULL;
}
int main() {
sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1
pthread_t t1, t2;
int id1 = 1, id2 = 2;
pthread_create(&t1, NULL, thread_function, &id1);
pthread_create(&t2, NULL, thread_function, &id2);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
printf("Final Counter Value: %d\n", counter);
sem_destroy(&mutex);
return 0;
}
```



Task c:

- With initial value **0**, the semaphore is locked from the start.
- Both threads get stuck in sem_wait() and the program never continues.

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex; // Binary semaphore
int counter = 0;

void* thread_function(void* arg) {
    int id = *(int*)arg;

    for (int i = 0; i < 5; i++) {
        printf("Thread %d: Waiting...\n", id);
        sem_wait(&mutex); // Acquire

        // 4.4 Counting Semaphore Example
        // A counting semaphore with initial value = 3 allows
        // up to 3 threads to access a resource simultaneously.

        // Critical section
        counter++;
        printf("Thread %d: In critical section | Counter = %d\n", id, counter);
        sleep(1);

        sem_post(&mutex); // Release
        sleep(1);
    }

    return NULL;
}

int main() {
    sem_init(&mutex, 0, 0); // Binary semaphore initialized to 1

    pthread_t t1, t2;
    int id1 = 1, id2 = 2;

    pthread_create(&t1, NULL, thread_function, &id1);
    pthread_create(&t2, NULL, thread_function, &id2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final Counter Value: %d\n", counter);

    sem_destroy(&mutex);
    return 0;
}
```

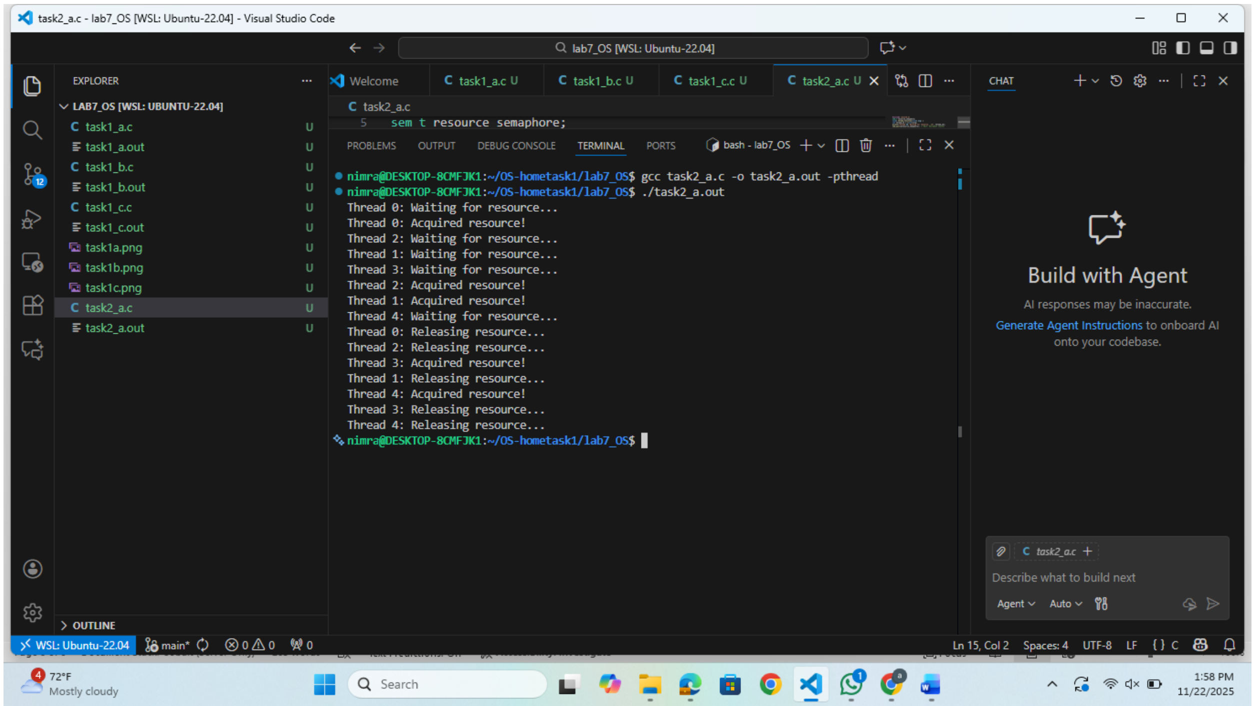


Task2: 4.4. Counting Semaphore Example

Task a:

- **At the start, semaphore value is 3.**
- **So, Thread 0, Thread 1, Thread 2 will immediately enter.**
- **Thread 3 and Thread 4 will have to wait until one of the first threads releases the resource.**
- **This code allows only 3 threads to run inside the critical section at the same time using a counting semaphore.**

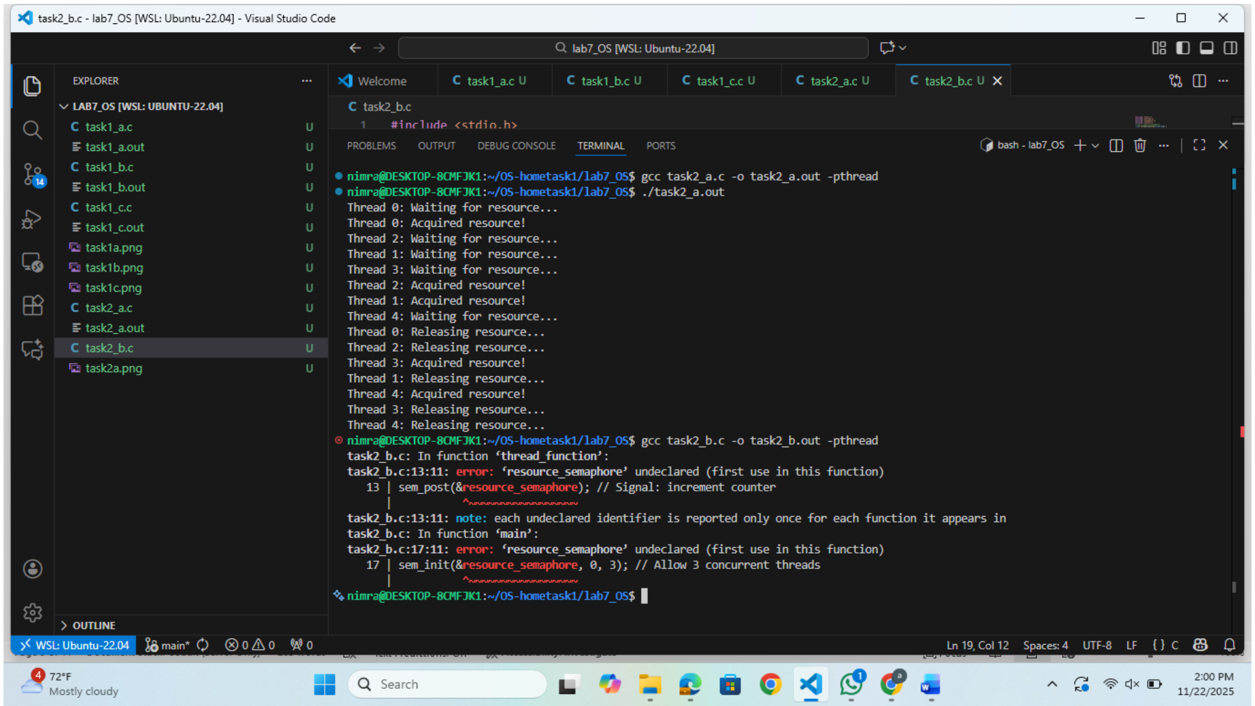
```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t resource_semaphore;
void* thread_function(void* arg) {
int thread_id = *(int*)arg;
printf("Thread %d: Waiting for resource...\n", thread_id);
sem_wait(&resource_semaphore); // Wait: decrement counter
printf("Thread %d: Acquired resource!\n", thread_id);
sleep(2); // Use resource
printf("Thread %d: Releasing resource...\n", thread_id);
sem_post(&resource_semaphore); // Signal: increment counter
return NULL;
}
int main() {
sem_init(&resource_semaphore, 0, 3); // Allow 3 concurrent threads
pthread_t threads[5];
int ids[5];
for (int i = 0; i < 5; i++) {
ids[i] = i;
pthread_create(&threads[i], NULL, thread_function, &ids[i]);
}
for (int i = 0; i < 5; i++) {
pthread_join(threads[i], NULL);
}
sem_destroy(&resource_semaphore);
return 0;
}
```



Task b: commenting mutex

- **If you comment out sem_t resource_semaphore then the variable no longer exists, so every place that uses it sem_wait gives an undeclared error.**
- **Because the semaphore is removed but still used in the code, the program cannot compile and therefore cannot run.**

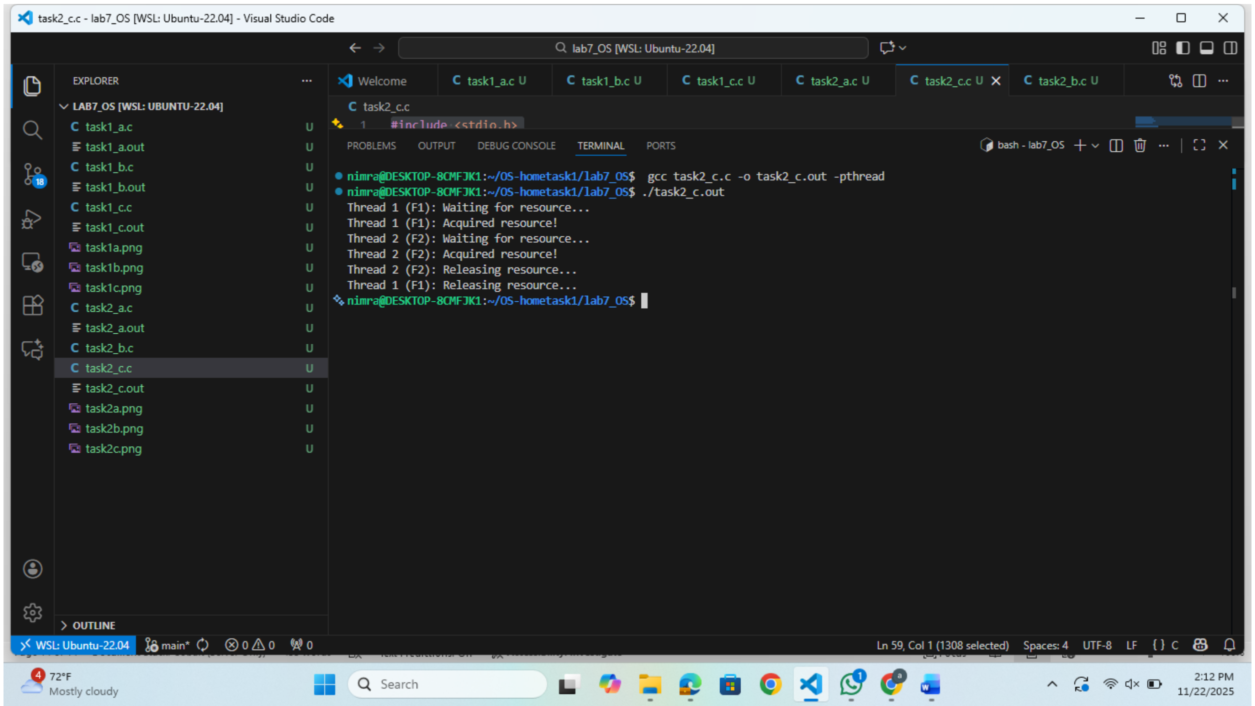
```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
//sem_t resource_semaphore;
void* thread_function(void* arg) {
int thread_id = *(int*)arg;
printf("Thread %d: Waiting for resource...\n", thread_id);
//sem_wait(&resource_semaphore); // Wait: decrement counter
printf("Thread %d: Acquired resource!\n", thread_id);
sleep(2); // Use resource
printf("Thread %d: Releasing resource...\n", thread_id);
sem_post(&resource_semaphore); // Signal: increment counter
return NULL;
}
int main() {
sem_init(&resource_semaphore, 0, 3); // Allow 3 concurrent threads
pthread_t threads[5];
int ids[5];
for (int i = 0; i < 5; i++) {
ids[i] = i;
pthread_create(&threads[i], NULL, thread_function, &ids[i]);
}
for (int i = 0; i < 5; i++) {
pthread_join(threads[i], NULL);
}
sem_destroy(&resource_semaphore);
return 0;
}
```



Task c: Making two threads function

- **The program creates two threads, each running its own thread function that waits for a semaphore,**
- **Uses the shared resource, and then release it.**
- **The semaphore controls access so that both thread functions follow the same safe pattern while using the shared resource.**

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t resource_semaphore;

// FIRST THREAD FUNCTION
void* thread_function1(void* arg) {
    int thread_id = *(int*)arg;

    printf("Thread %d (F1): Waiting for resource...\n", thread_id);
    sem_wait(&resource_semaphore);

    printf("Thread %d (F1): Acquired resource!\n", thread_id);
    sleep(2);

    printf("Thread %d (F1): Releasing resource...\n", thread_id);
    sem_post(&resource_semaphore);

    return NULL;
}

// SECOND THREAD FUNCTION (same style)
void* thread_function2(void* arg) {
    int thread_id = *(int*)arg;

    printf("Thread %d (F2): Waiting for resource...\n", thread_id);
    sem_wait(&resource_semaphore);

    printf("Thread %d (F2): Acquired resource!\n", thread_id);
    sleep(2);

    printf("Thread %d (F2): Releasing resource...\n", thread_id);
    sem_post(&resource_semaphore);

    return NULL;
}

int main() {
    sem_init(&resource_semaphore, 0, 3);

    pthread_t t1, t2;
    int id1 = 1, id2 = 2;

    // FIRST function thread
    pthread_create(&t1, NULL, thread_function1, &id1);

    // SECOND function thread
    pthread_create(&t2, NULL, thread_function2, &id2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    sem_destroy(&resource_semaphore);

    return 0;
}
```



Task 3: Difference between Semaphore and Mutex

Feature	Semaphore	Mutex
Definition	A signaling mechanism that uses a counter to control access to multiple resources.	A locking mechanism that allows only one thread or process to access a resource at a time.
Type	Can be Counting or Binary	Always Binary
Resource Access	Can allow multiple threads at the same time (based on counter).	Allows only ONE thread at a time.

Ownership	No ownership, any thread can release a semaphore.	Ownership exists, only the thread that locked the mutex can unlock it.
Usage	Used for resource pools, limiting access (e.g., 5 database connections).	Used for mutual exclusion on shared data.
Risk	If released incorrectly, resource inconsistency.	If not released by the owner, deadlock can occur.
Value Range	Value can be 0 to N	Only 0 or 1 (locked or unlocked).