National Textile University

Department of Computer Science

Subject:

Operating Systems

Submitted To:

Sir Nasir Mehmood

Submitted By:

Alishba Riasat

Registration No:

23-NTU-CS-1135

Lab No:

6

Semester:

5th

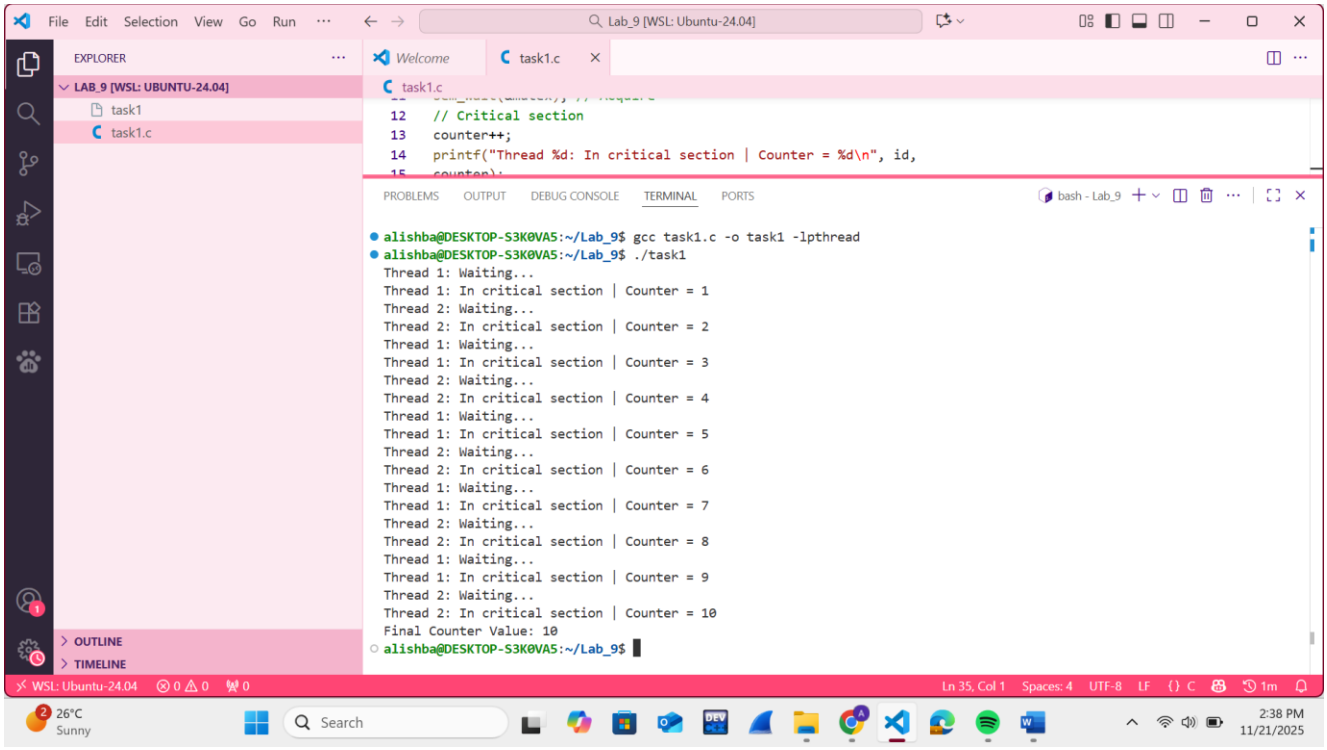
Task 1:

Code:

```
#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;
}
```

Output:



Changes demonstration:

When I changed sem_init(&mutex, 0, 1) to sem_init(&mutex, 0, 0)

When the semaphore starts at 0, both threads get stuck at sem_wait(). Since no thread reaches sem_post() first, the semaphore never becomes 1. This causes a deadlock, and the program freezes right after printing “Waiting...”.

When I removed sem_post(&mutex)

Here the semaphore gets locked by the first sem_wait() and never gets unlocked again. So after the first iteration, the semaphore stays at 0 permanently, and both threads block on their next sem_wait(). This again leads to a **deadlock**, but this time because the lock is never released.

When I removed sem_wait(&mutex) but kept sem_post(&mutex)

Now the critical section is not protected at all. Both threads enter freely, and every loop increases the semaphore value because only sem_post() is running. This removes any mutual exclusion, so the counter is updated without synchronization.

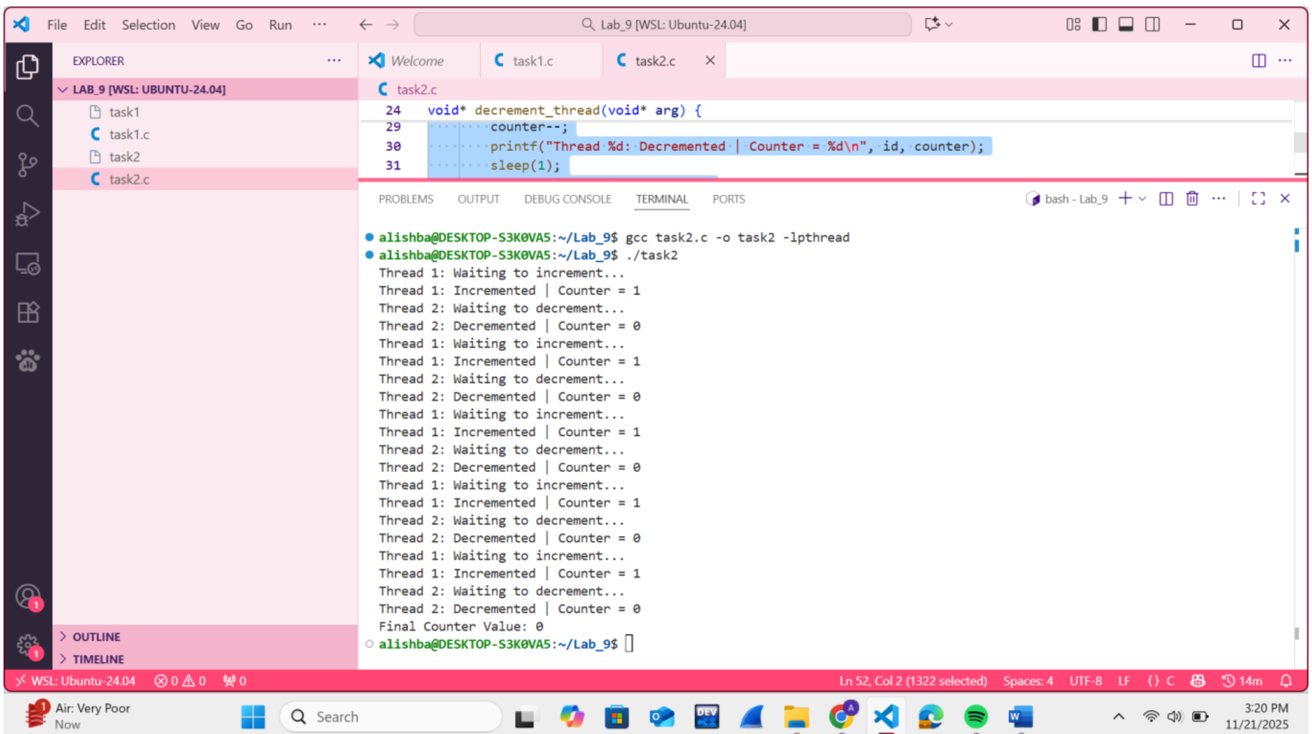
Task 2:

Code:

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex; // Binary semaphore
int counter = 0;

// Thread that acts like a producer
void* increment_thread(void* arg) {
    int id=*(int*)arg;
    for (int i=0; i<5; i++) {
        printf("Thread %d: Waiting to increment...\n", id);
        sem_wait(&mutex); // acquire
        counter++;
        printf("Thread %d: Incremented | Counter = %d\n", id, counter);
        sleep(1);
        sem_post(&mutex); // release
        sleep(1);
    }
    return NULL;
}

// Thread that acts like a consumer
void* decrement_thread(void* arg) {
    int id=*(int*)arg;
    for (int i=0; i<5; i++) {
        printf("Thread %d: Waiting to decrement...\n", id);
        sem_wait(&mutex); // acquire
        counter--;
        printf("Thread %d: Decremented | Counter = %d\n", id, counter);
        sleep(1);
        sem_post(&mutex); // release
        sleep(1);
    }
    return NULL;
}
int main() {
    sem_init(&mutex, 0, 1);   // semaphore = 1
    pthread_t t1, t2;
    int id1=1, id2=2;
```

```c
    pthread_create(&t1, NULL, increment_thread, &id1);
    pthread_create(&t2, NULL, decrement_thread, &id2);

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

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

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

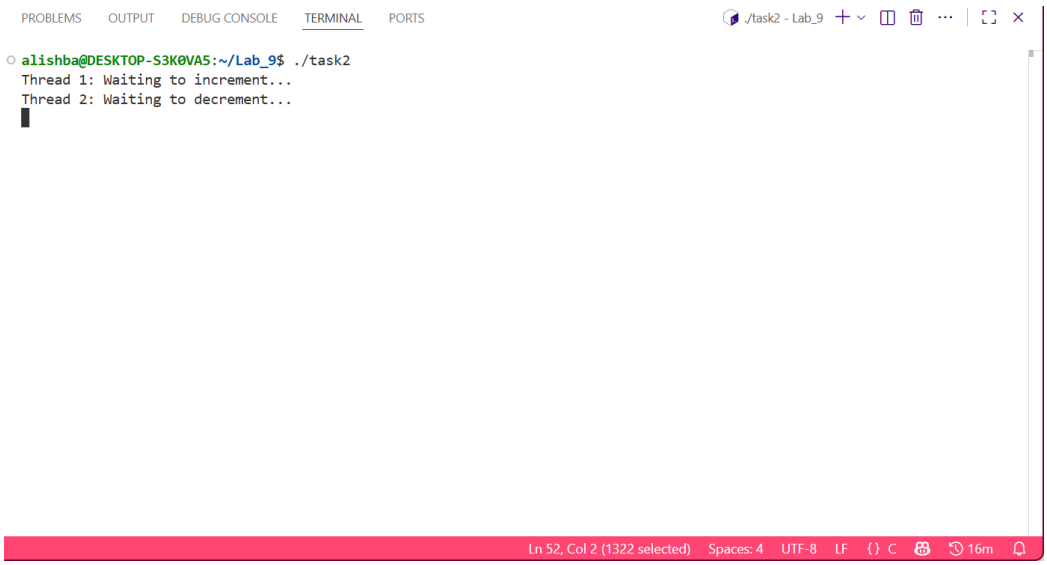
Output:



Changes demonstration:

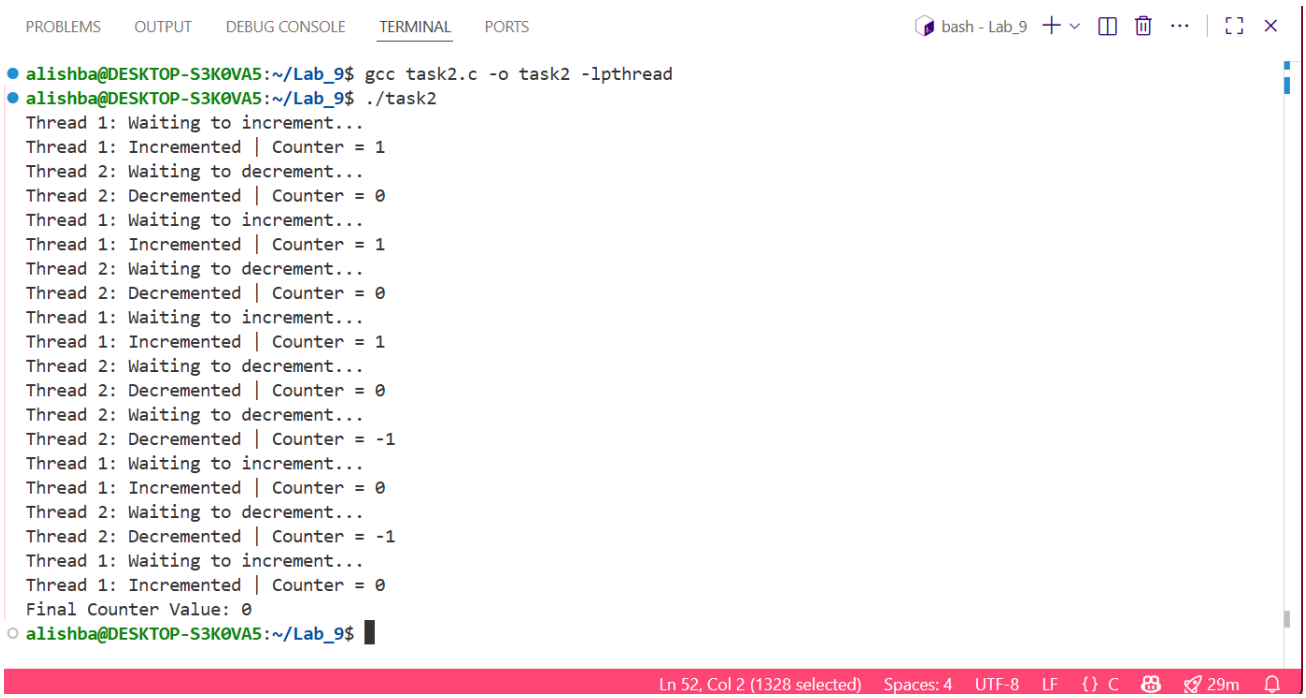
When I changed sem_init(&mutex, 0, 1) to sem_init(&mutex, 0, 0)

Both threads will block immediately on sem_wait() that will cause deadlock.

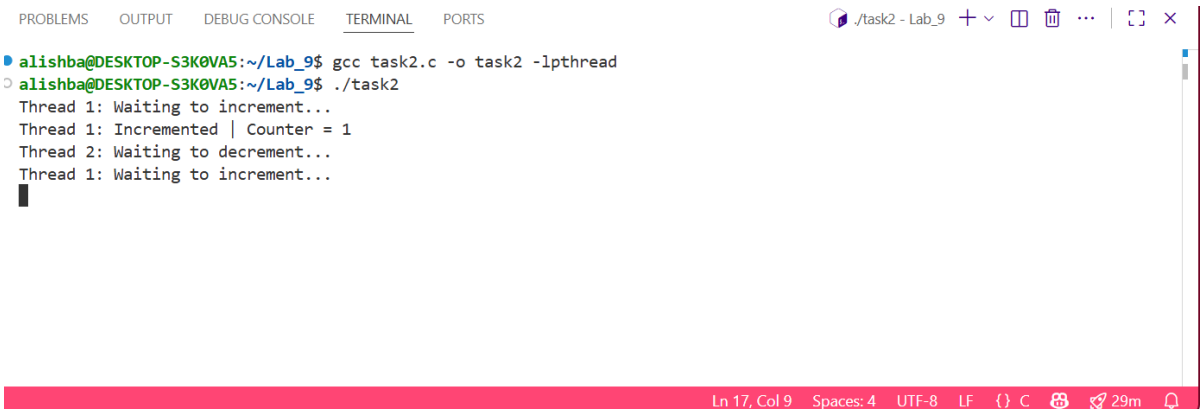


Remove sem_wait():

There will be no mutual exclusion and both threads can increment/decrement at the same time. Counter updates may become inconsistent.



Remove sem_post():
Deadlock occurs after first iteration.



Comparison between Mutex and Semaphores:

Similarities:

- Both of them protect the shared variable counter from access as the same time. They ensure that only one thread at same time can enter the critical selection preventing race condition.
- Both have explicit lock/wait and unlock/post operations:
  Mutex: pthread_mutex_lock() / pthread_mutex_unlock()
  Semaphore: sem_wait() / sem_post()

Differences:

Feature	Mutex	Semaphore
Declaration	pthread_mutex_t_lock;	sem_t mutex;
Initialization	pthread_mutex_init (&lock, null)	sem_init &mutex, 0, 1)
Lock mechanism	Only thread that locks can unlock	Any thread can sem_post() even if it didn't sem_wait()
Flexibility	Only binary lock	Can be binary or counting semaphore
Usage in code	counter++ protected by pthread_mutex_lock()	counter++ protected by sem_wait()
Ownership	Owned by thread that locks it	No ownership; any thread can signal