



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() {

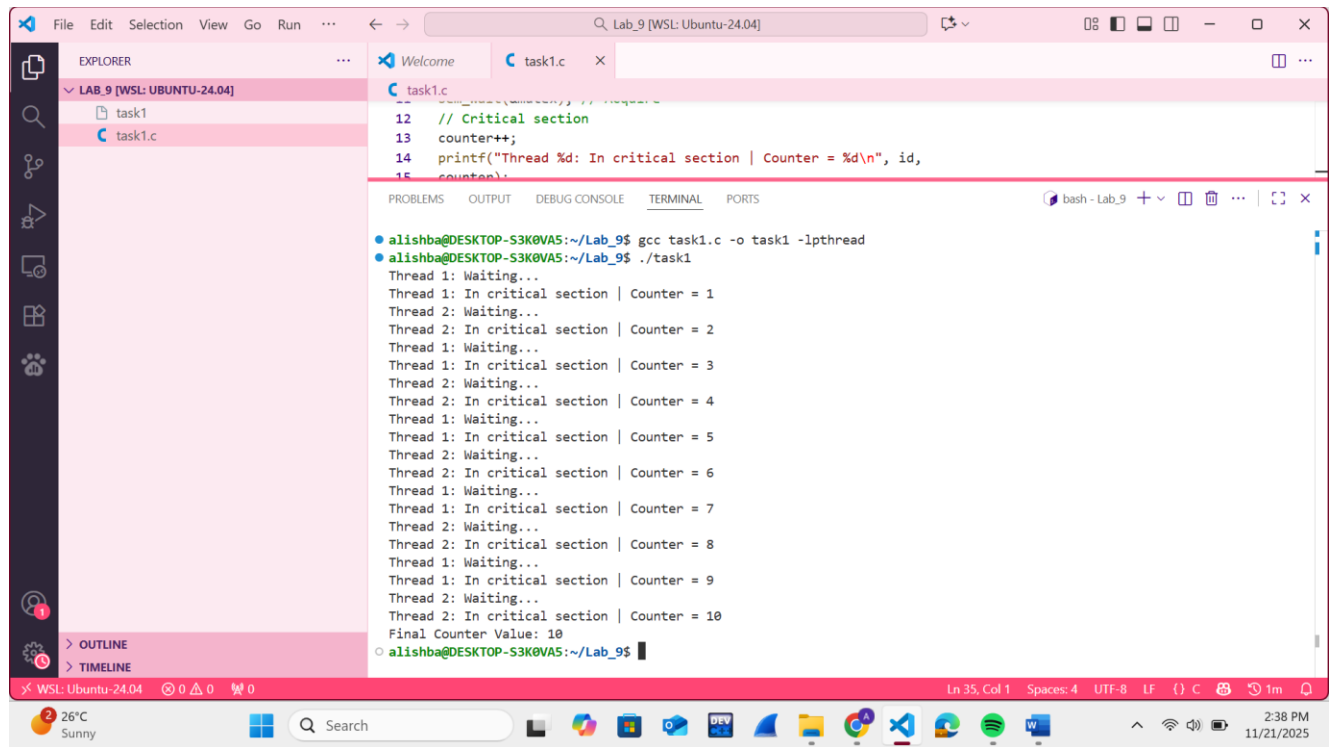
    sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1
    pthread_t t1, t2;
    int id1 = 1, id2 = 2;

    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:




```
task1.c
12 // Critical section
13 counter++;
14 printf("Thread %d: In critical section | Counter = %d\n", id,
15 counter);

alishba@DESKTOP-S3K0VA5:~/Lab_9$ gcc task1.c -o task1 -lpthread
alishba@DESKTOP-S3K0VA5:~/Lab_9$ ./task1
Thread 1: Waiting...
Thread 1: In critical section | Counter = 1
Thread 2: Waiting...
Thread 2: In critical section | Counter = 2
Thread 1: Waiting...
Thread 1: In critical section | Counter = 3
Thread 2: Waiting...
Thread 2: In critical section | Counter = 4
Thread 1: Waiting...
Thread 1: In critical section | Counter = 5
Thread 2: Waiting...
Thread 2: In critical section | Counter = 6
Thread 1: Waiting...
Thread 1: In critical section | Counter = 7
Thread 2: Waiting...
Thread 2: In critical section | Counter = 8
Thread 1: Waiting...
Thread 1: In critical section | Counter = 9
Thread 2: Waiting...
Thread 2: In critical section | Counter = 10
Final Counter Value: 10
alishba@DESKTOP-S3K0VA5:~/Lab_9$
```

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...".



```
alishba@DESKTOP-S3K0VA5:~/Lab_9$ gcc task1.c -o task1 -lpthread
alishba@DESKTOP-S3K0VA5:~/Lab_9$ ./task1
Thread 1: Waiting...
Thread 2: Waiting...
[Freezing]
```

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.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
alishba@DESKTOP-S3K0VA5:~/Lab_9$ gcc task1.c -o task1 -lpthread
alishba@DESKTOP-S3K0VA5:~/Lab_9$ ./task1
Thread 1: Waiting...
Thread 1: In critical section | Counter = 1
Thread 2: Waiting...
Thread 1: Waiting...
█
```

Ln 45, Col 1 (801 selected) Spaces: 4 UTF-8 LF {} C 6m

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.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
alishba@DESKTOP-S3K0VA5:~/Lab_9$ gcc task1.c -o task1 -lpthread
alishba@DESKTOP-S3K0VA5:~/Lab_9$ ./task1
Thread 1: Waiting...
Thread 1: In critical section | Counter = 1
Thread 2: Waiting...
Thread 2: In critical section | Counter = 2
Thread 1: Waiting...
Thread 1: In critical section | Counter = 3
Thread 2: Waiting...
Thread 2: In critical section | Counter = 4
Thread 2: Waiting...
Thread 2: In critical section | Counter = 5
Thread 1: Waiting...
Thread 1: In critical section | Counter = 6
Thread 2: Waiting...
Thread 2: In critical section | Counter = 7
Thread 1: Waiting...
Thread 1: In critical section | Counter = 8
Thread 1: Waiting...
Thread 1: In critical section | Counter = 9
Thread 2: Waiting...
Thread 2: In critical section | Counter = 10
Final Counter Value: 10
alishba@DESKTOP-S3K0VA5:~/Lab_9$ █
```

Ln 45, Col 1 (811 selected) Spaces: 4 UTF-8 LF {} C 10m

Task 2:

Code:

```
#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: Decrementing | 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;
```

Output:

Air: Very Poor Now

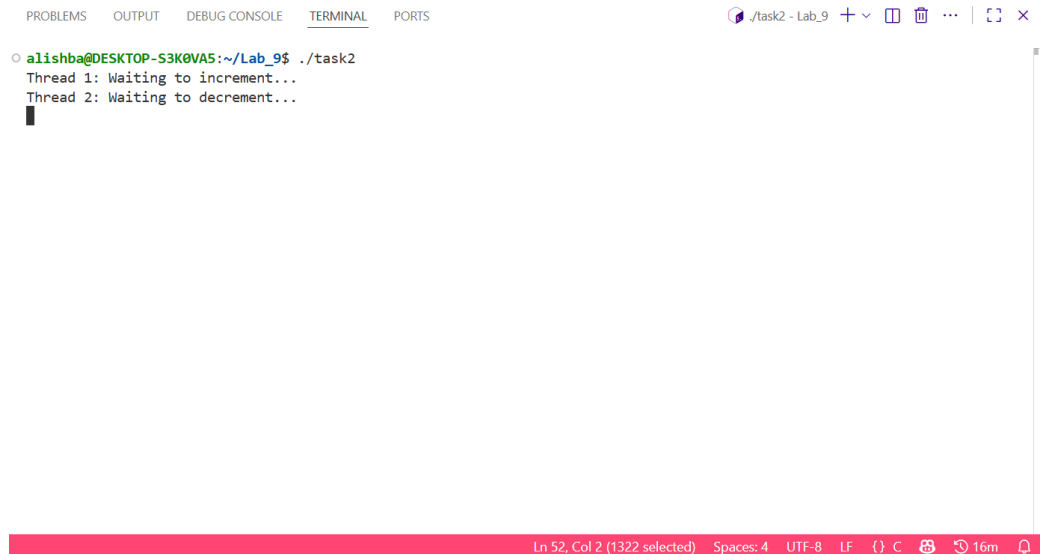
Search

3:20 PM 11/21/2025

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.



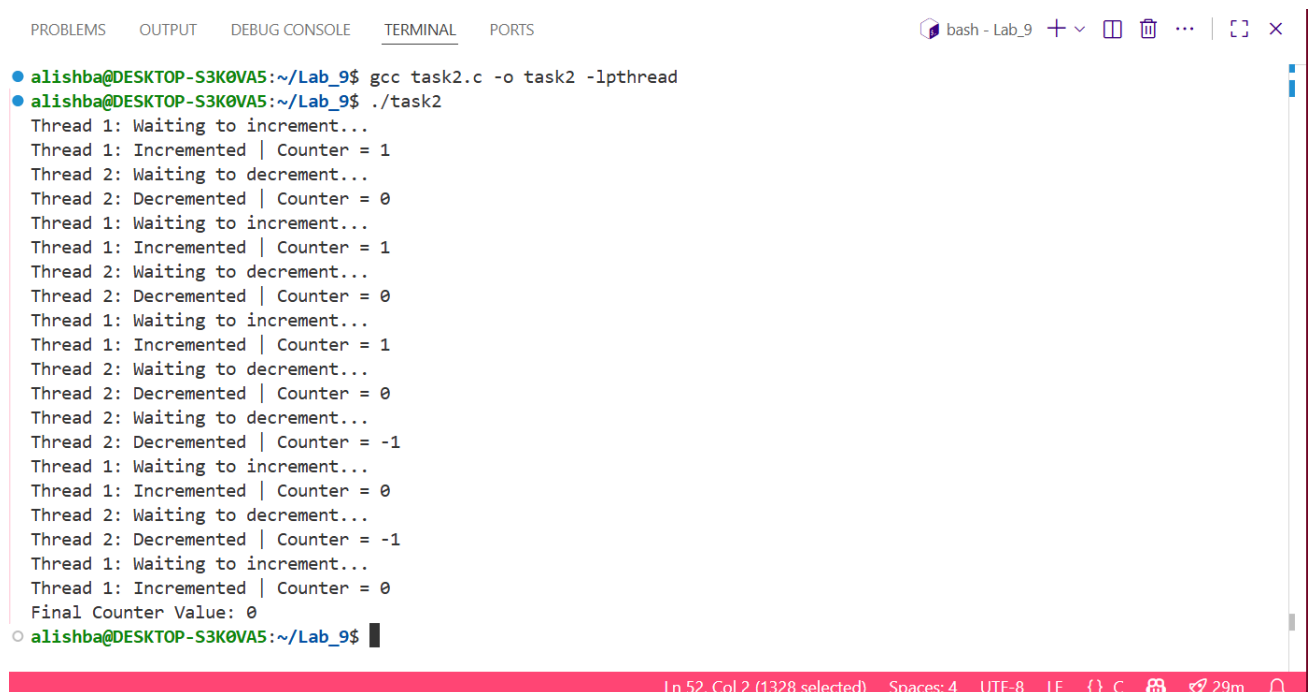
A screenshot of a terminal window titled `./task2 - Lab_9`. The terminal shows the output of running `./task2`. The output is:

```
alishba@DESKTOP-S3K0VA5:~/Lab_9$ ./task2
Thread 1: Waiting to increment...
Thread 2: Waiting to decrement...
```

The terminal is in a state where both threads are waiting, indicating a deadlock. The status bar at the bottom shows `Ln 52, Col 2 (1322 selected)`, `Spaces: 4`, `UTF-8`, `LF`, `{}`, `C`, and a timer at `16m`.

Remove `sem_wait()`:

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



A screenshot of a terminal window titled `bash - Lab_9`. The terminal shows the output of running `gcc task2.c -o task2 -lpthread` and `./task2`. The output shows the threads executing concurrently, with the counter being incremented and decremented multiple times. The final output is:

```
alishba@DESKTOP-S3K0VA5:~/Lab_9$ gcc task2.c -o task2 -lpthread
alishba@DESKTOP-S3K0VA5:~/Lab_9$ ./task2
Thread 1: Waiting to increment...
Thread 1: Incremented | Counter = 1
Thread 2: Waiting to decrement...
Thread 2: Decrement | Counter = 0
Thread 1: Waiting to increment...
Thread 1: Incremented | Counter = 1
Thread 2: Waiting to decrement...
Thread 2: Decrement | Counter = 0
Thread 1: Waiting to increment...
Thread 1: Incremented | Counter = 1
Thread 2: Waiting to decrement...
Thread 2: Decrement | Counter = 0
Thread 2: Waiting to decrement...
Thread 2: Decrement | Counter = -1
Thread 1: Waiting to increment...
Thread 1: Incremented | Counter = 0
Thread 2: Waiting to decrement...
Thread 2: Decrement | Counter = -1
Thread 1: Waiting to increment...
Thread 1: Incremented | Counter = 0
Final Counter Value: 0
alishba@DESKTOP-S3K0VA5:~/Lab_9$
```

The status bar at the bottom shows `Ln 52, Col 2 (1328 selected)`, `Spaces: 4`, `UTF-8`, `LF`, `{}`, `C`, and a timer at `29m`.

Remove sem_post():

Deadlock occurs after first iteration.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
alishba@DESKTOP-S3K0VA5:~/Lab_9$ gcc task2.c -o task2 -lpthread
alishba@DESKTOP-S3K0VA5:~/Lab_9$ ./task2
Thread 1: Waiting to increment...
Thread 1: Incremented | Counter = 1
Thread 2: Waiting to decrement...
Thread 1: Waiting to increment...
█
```

Comparison between Mutex and Semaphores:

Similarities:

- Both of them protect the shared variable counter from access at the same time. They ensure that only one thread at the same time can enter the critical section 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