

National Textile University, Faisalabad



Department of Computer Science

Name	Dawood Saif
Class	SE-5 th (A)
Reg. No.	23-NTU-CS-1145
Course	Operating system
Submitted To	Sir Nasir Mehmood
Submission Date	21/11/2025
Lab No.	9 (After Mids.)

Lab No. 9 Operating system:

Binary Semaphores:

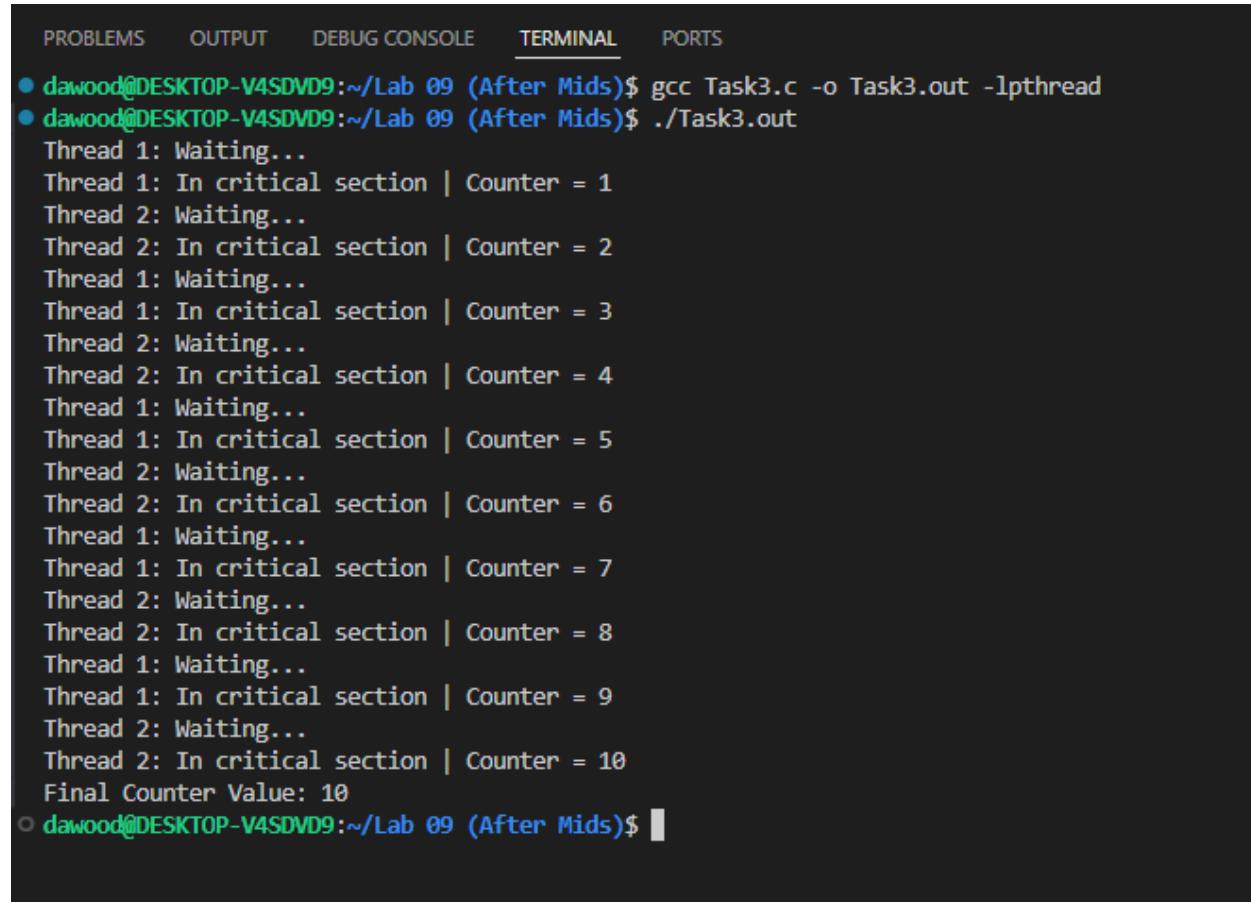
A binary semaphore is a synchronization mechanism in an operating system that can take only two integer **values, 0 or 1**, to control access to a single shared resource. It is primarily used to enforce **mutual exclusion**, ensuring that only one process or thread can access a critical section of code at a time, thereby preventing race conditions.

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;  
}
```

Outputs:



```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS  
● dawood@DESKTOP-V4SDVD9:~/Lab 09 (After Mids)$ gcc Task3.c -o Task3.out -lpthread  
● dawood@DESKTOP-V4SDVD9:~/Lab 09 (After Mids)$ ./Task3.out  
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  
○ dawood@DESKTOP-V4SDVD9:~/Lab 09 (After Mids)$
```

After Changing the

```
sem_init(&mutex, 0, 1); to      sem_init(&mutex, 1, 0);
```

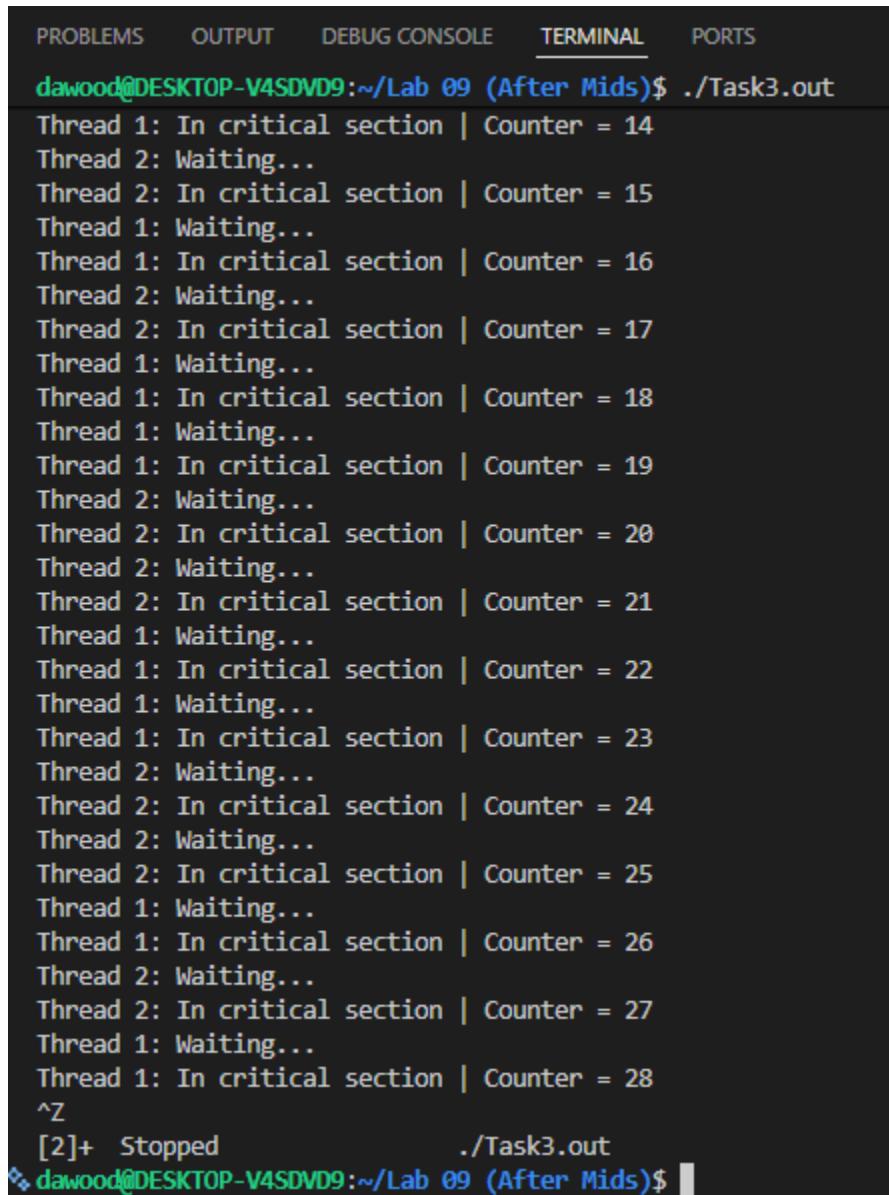
we will get:

```
22 int main() {
23     sem_init(&mutex, 1, 0); // Binary semaphore initialized to 1
24     pthread_t t1, t2;
25     int id1 = 1, id2 = 2;
26     pthread_create(&t1, NULL, thread_function, &id1);
27     pthread_create(&t2, NULL, thread_function, &id2);
28     pthread_join(t1, NULL);
29     pthread_join(t2, NULL);
30     printf("Final Counter Value: %d\n", counter);
31     sem_destroy(&mutex);
32 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
● dawood@DESKTOP-V4SDVD9:~/Lab 09 (After Mids)$ gcc Task3.c -o Task3.out -lpthread
❖ dawood@DESKTOP-V4SDVD9:~/Lab 09 (After Mids)$ ./Task3.out
Thread 1: Waiting...
Thread 2: Waiting...
[]
```

Secondly if we comment out or remove the sem_wait statement, what will we get:



```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS
dawood@DESKTOP-V4SDVD9:~/Lab 09 (After Mids)$ ./Task3.out
Thread 1: In critical section | Counter = 14
Thread 2: Waiting...
Thread 2: In critical section | Counter = 15
Thread 1: Waiting...
Thread 1: In critical section | Counter = 16
Thread 2: Waiting...
Thread 2: In critical section | Counter = 17
Thread 1: Waiting...
Thread 1: In critical section | Counter = 18
Thread 1: Waiting...
Thread 2: In critical section | Counter = 19
Thread 2: Waiting...
Thread 2: In critical section | Counter = 20
Thread 2: Waiting...
Thread 2: In critical section | Counter = 21
Thread 1: Waiting...
Thread 1: In critical section | Counter = 22
Thread 1: Waiting...
Thread 1: In critical section | Counter = 23
Thread 2: Waiting...
Thread 2: In critical section | Counter = 24
Thread 2: Waiting...
Thread 2: In critical section | Counter = 25
Thread 1: Waiting...
Thread 1: In critical section | Counter = 26
Thread 2: Waiting...
Thread 2: In critical section | Counter = 27
Thread 1: Waiting...
Thread 1: In critical section | Counter = 28
^Z
[2]+  Stopped                  ./Task3.out
dawood@DESKTOP-V4SDVD9:~/Lab 09 (After Mids)$
```

Similarly if we remove sem_post statement from code , we can see

```
11 sem_wait(&mutex); // Acquire
12 // Critical section
13 counter++;
14 printf("Thread %d: In critical section | Counter = %d\n", id,
15 counter);
16 sleep(1);
17 //sem_post(&mutex); // Release
18 sleep(1);
`
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
● dawood@DESKTOP-V4SDVD9:~/Lab 09 (After Mids)$ gcc Task3.c -o Task3.out -lpthread
❖ dawood@DESKTOP-V4SDVD9:~/Lab 09 (After Mids)$ ./Task3.out
Thread 2: Waiting...
Thread 2: In critical section | Counter = 1
Thread 1: Waiting...
Thread 2: Waiting...
```

Task 2:

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;
}
void *thread_function1(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,t3,t4;
    int id1 = 1, id2 = 2, id3 = 3, id4 =4 ;

    pthread_create(&t1, NULL, thread_function, &id1);
    pthread_create(&t2, NULL, thread_function, &id2);
    pthread_create(&t3, NULL, thread_function1, &id3);
    pthread_create(&t4, NULL, thread_function1, &id4);
    pthread_join(t1, NULL);

    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);
    printf("Final Counter Value: %d\n", counter);
    sem_destroy(&mutex);
    return 0;
}
```

Output Screenshot:

```
dawood@DESKTOP-V4SDVD9:~/Lab 09 (After Mids)$ ./Task5.out
Thread 1: Waiting...
Thread 4: In critical section | Counter = -1
Thread 3: Waiting...
Thread 2: In critical section | Counter = 0
Thread 4: Waiting...
Thread 4: In critical section | Counter = -1
Thread 2: Waiting...
Thread 2: In critical section | Counter = 0
Thread 4: Waiting...
Thread 4: In critical section | Counter = -1
Thread 2: Waiting...
Thread 2: In critical section | Counter = 0
Thread 4: Waiting...
Thread 4: In critical section | Counter = -1
Thread 2: Waiting...
Thread 2: In critical section | Counter = 0
Thread 4: Waiting...
Thread 4: In critical section | Counter = -1
Thread 2: Waiting...
Thread 2: In critical section | Counter = 0
Thread 1: In critical section | Counter = 1
Thread 3: In critical section | Counter = 0
Thread 1: Waiting...
Thread 1: In critical section | Counter = 1
Thread 3: Waiting...
Thread 3: In critical section | Counter = 0
Thread 1: Waiting...
Thread 1: In critical section | Counter = 1
Thread 3: Waiting...
Thread 3: In critical section | Counter = 0
Thread 1: Waiting...
Thread 1: In critical section | Counter = 1
Thread 3: Waiting...
Thread 3: In critical section | Counter = 0
Final Counter Value: 0
```

Task 3:

Difference Between Binary Semaphore and Mutex Lock

Feature	Binary Semaphore	Mutex
Value	0 or 1	Locked or unlocked
Ownership	No ownership (any thread can post)	Ownership (only locking thread should unlock)
Use case	Signalling + mutual exclusion	Only mutual exclusion
Danger	Incorrect <code>sem_post()</code> by any thread	Safer due to ownership
Can it wake another thread?	YES	NO (only unlocks)