



# National Textile University

*Department of Computer Science*

**Subject:**

Operating system

**Submitted To:**

Sir Nasir

**Submitted By:**

Noor Ul Ain

**Registration No:**

23-NTU-CS-1221


**Lab No:**

9

**Semester:**

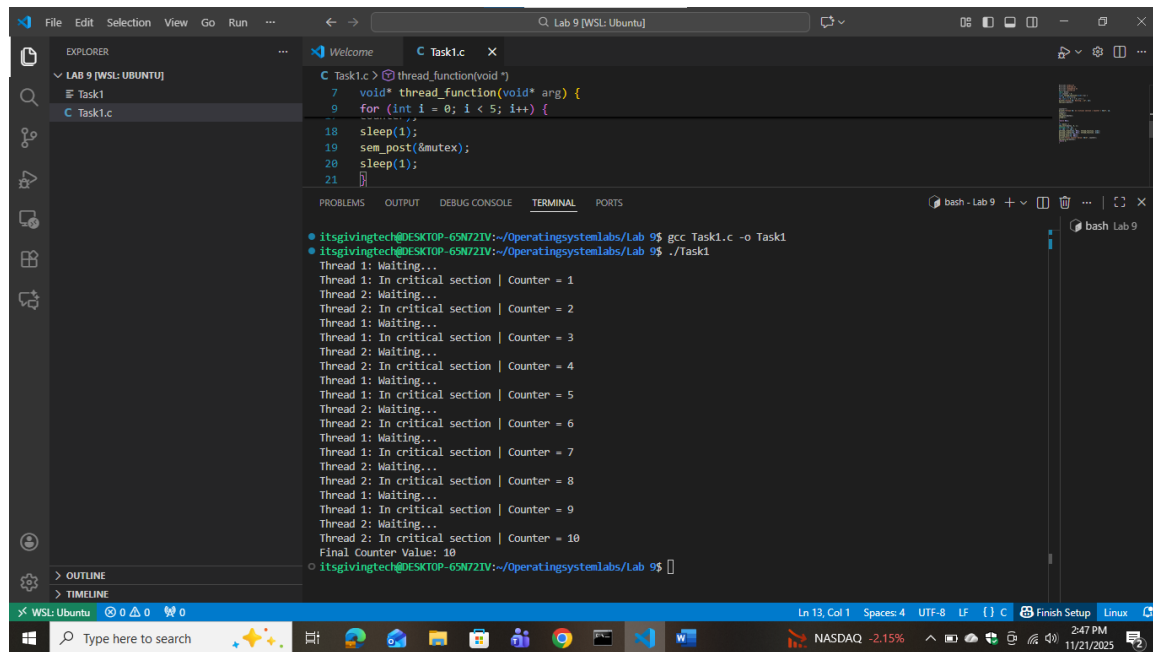
5

## Task 1:



```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <semaphore.h>
4  #include <unistd.h>
5  sem_t mutex;
6  int counter = 0;
7  void* thread_function(void* arg) {
8  int id = *(int*)arg;
9  for (int i = 0; i < 5; i++) {
10 printf("Thread %d: Waiting...\n", id);
11 sem_wait(&mutex);
12
13
14
15 counter++;
16 printf("Thread %d: In critical section | Counter = %d\n", id,
17 counter);
18 sleep(1);
19 sem_post(&mutex);
20 sleep(1);
21 }
22 return NULL;
23 }
24 int main() {
25 sem_init(&mutex, 0, 1);
26 pthread_t t1, t2;
27 int id1 = 1, id2 = 2;
28 pthread_create(&t1, NULL, thread_function, &id1);
29 pthread_create(&t2, NULL, thread_function, &id2);
30 pthread_join(t1, NULL);
31 pthread_join(t2, NULL);
32 printf("Final Counter Value: %d\n", counter);
33 sem_destroy(&mutex);
34 return 0;
35 }
```

## Output:



The screenshot shows a Visual Studio Code editor window with a C file named `Task1.c`. The code defines a `thread_function` that increments a shared counter in a loop, with a semaphore value of 1. The terminal output shows the execution of the program, where two threads alternate in entering the critical section and incrementing the counter, resulting in a final value of 10.

```
Task1.c:7: void* thread_function(void* arg) {
Task1.c:9:     for (int i = 0; i < 5; i++) {
Task1.c:18:         sleep(1);
Task1.c:19:         sem_post(&mutex);
Task1.c:20:         sleep(1);
Task1.c:21:     }
```

```
itsgivingtech@DESKTOP-65N72IV:~/OperatingSystemLabs/Lab 9$ gcc Task1.c -o Task1
itsgivingtech@DESKTOP-65N72IV:~/OperatingSystemLabs/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
itsgivingtech@DESKTOP-65N72IV:~/OperatingSystemLabs/Lab 9$
```

## Remarks:

In this code, two threads are trying to increment the shared counter in the same function, here semaphore value is one means one thread can enter the lock at a time, when one thread enters and increment the code the other one waits and vice versa.

- **Different cases:**  
`//sem_post(&mutex):`

Because of this one thread gets locked in the semaphore forever and other thread gets to wait forever and never prints the final counter value.

**Sem\_init(&mutex, 0,0):**

We initialized the semaphore with zero, this means that both threads will be blocked and never gets to enter the &sem\_wait(&mutex) and the program freezes here.

**//sem\_wait(&mutex)**

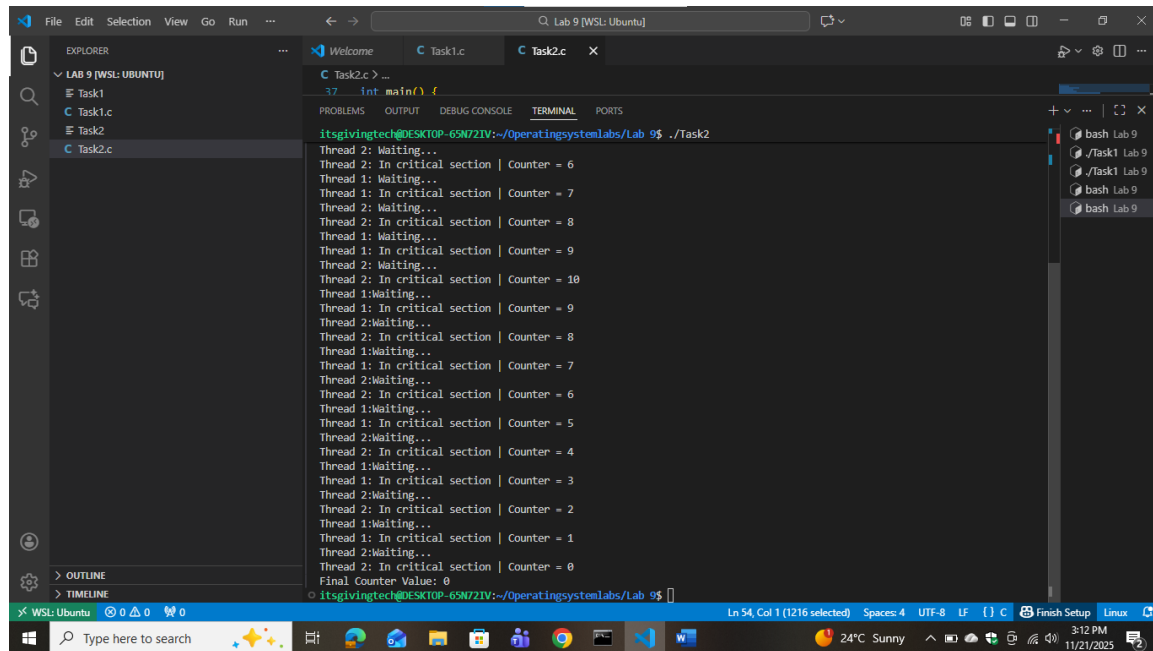
Thread 1 and 2 are blocked forever here.

**Task2**

**Code:**

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <semaphore.h>
4  #include <unistd.h>
5  sem_t mutex;
6  int counter = 0;
7  void* thread_function(void* arg) {
8  int id = *(int*)arg;
9  for (int i = 0; i < 5; i++) {
10 printf("Thread %d: Waiting...\n", id);
11 sem_wait(&mutex);
12
13
14
15 counter++;
16 printf("Thread %d: In critical section | Counter = %d\n", id,
17 counter);
18 sleep(1);
19 sem_post(&mutex);
20 sleep(1);
21 }
22 return NULL;
23 }
24 void* thread_function2(void* arg){
25 int id=*(int*)arg;
26 for(int i=0;i<5;i++){
27 printf("Thread %d:Waiting...\n",id);
28 sem_wait(&mutex);
29 counter--;
30 printf("Thread %d: In critical section | Counter = %d\n", id,counter);
31 sleep(1);
32 sem_post(&mutex);
33 sleep(1);
34 }
35 return NULL;
36 }
37 int main() {
38 sem_init(&mutex, 0, 1);
39 pthread_t t1, t2;
40 int id1 = 1, id2 = 2;
41 pthread_create(&t1, NULL, thread_function, &id1);
42 pthread_create(&t2, NULL, thread_function2, &id2);
43 pthread_join(t1, NULL);
44 pthread_join(t2, NULL);
45
46 printf("Final Counter Value: %d\n", counter);
47 sem_destroy(&mutex);
48 return 0;
49 }
50
```

## Output:



```
itsgivingtech@DESKTOP-65N72IV:~/OperatingSystemLabs/Lab 9$ ./Task2
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
Thread 1: Waiting...
Thread 1: In critical section | Counter = 9
Thread 2: Waiting...
Thread 2: In critical section | Counter = 8
Thread 1: Waiting...
Thread 1: In critical section | Counter = 7
Thread 2: Waiting...
Thread 2: In critical section | Counter = 6
Thread 1: Waiting...
Thread 1: In critical section | Counter = 5
Thread 2: Waiting...
Thread 2: In critical section | Counter = 4
Thread 1: Waiting...
Thread 1: In critical section | Counter = 3
Thread 2: Waiting...
Thread 2: In critical section | Counter = 2
Thread 1: Waiting...
Thread 1: In critical section | Counter = 1
Thread 2: Waiting...
Thread 2: In critical section | Counter = 0
Final Counter Value: 0
itsgivingtech@DESKTOP-65N72IV:~/OperatingSystemLabs/Lab 9$
```

## Remarks:

Here I've created two functions for each thread where one function increments the counter and the other one decrements the counter.

- **Cases:**

**//sem\_post(&mutex):**

Because of this the one thread gets locked in the semaphore forever and other thread gets to wait forever and never prints the final counter value

**Sem\_init(&mutex,0,0):**

Both threads are blocked to enter the sem\_wait(&mutex)

**//sem\_wait(&mutex)**

Thread 1 and 2 waits forever here

### Task3:

#### Difference between mutex lock and binary semaphore:

Feature	Binary Semaphore (your code)	Mutex Lock (if you used <code>pthread_mutex_t</code> )
Type	<code>sem_t</code> mutex initialized to 1	<code>pthread_mutex_t</code> mutex
Ownership	No ownership for semaphore any thread can call <code>sem_post()</code> even if it didn't call <code>sem_wait()</code>	Ownership is enforced which means only the thread that locked it can unlock it
Use in above codes	<code>sem_wait(&amp;mutex)</code> locks the counter and <code>sem_post(&amp;mutex)</code> unlock the counter	We can use <code>pthread_mutex_lock(&amp;mutex)</code> and <code>pthread_mutex_unlock(&amp;mutex)</code> unlocks the counter
Can be used for signaling?	Yes, semaphores can signal other threads/processes	No, mutex is strictly for mutual exclusion
Error checking	No error if another thread calls <code>sem_post()</code> incorrectly	It shows error if a thread that doesn't own the mutex tries to unlock it.
Blocking behavior	The threads wait if the semaphore value is 0	Threads wait if the mutex is locked by another thread
Initialization	<code>sem_init(&amp;mutex, 0, 1)</code>	<code>pthread_mutex_init(&amp;mutex, NULL)</code>