



# National Textile University

## Department of Computer Science

Subject:

Operating System

---

Submitted to:

Sir Nasir Mehmod

---

Submitted by:

Akasha Fatima

---

Reg. number:

23-NTU-CS-FL-1132

---

Semester:

5<sup>th</sup> - A

---

# **LAB\_09**

## **Task\_01: Actual Program**

**CODE:**

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

**Output:**

```
• akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$ gcc Task_01.c -lpthread
• akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$ ./Task_01.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
◦ akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$
```

## Task\_01\_A: Initializing Semaphores to the 0, 0: CODE:

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

## Output:

```
• akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$ gcc Task_01.c -lpthread
• akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$ ./Task_01.out
Thread 1: Waiting...
Thread 2: Waiting...
```

## Results:

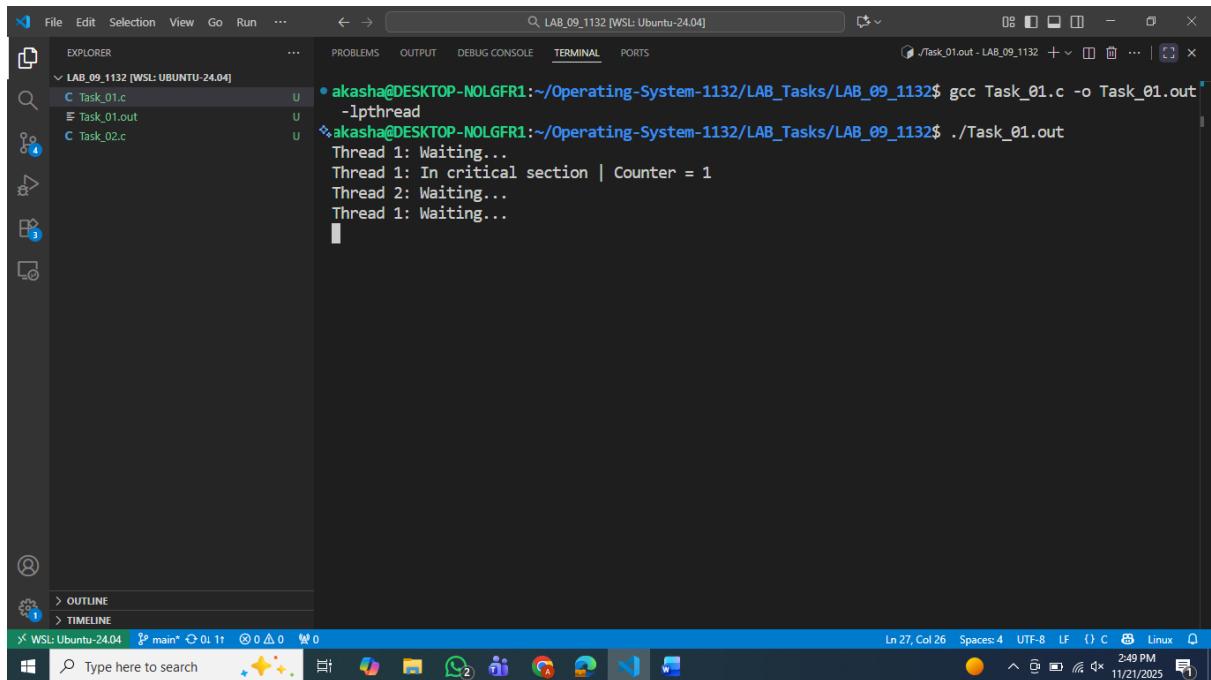
Since the binary semaphore initialized to 0,0 none of thread is executing, thus both threads are in waiting state.

### Task\_01\_B: Commenting the post

CODE:

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

## Output:



```
• akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$ gcc Task_01.c -lpthread
• akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$ ./Task_01.out
Thread 1: Waiting...
Thread 1: In critical section | Counter = 1
Thread 2: Waiting...
Thread 1: Waiting...
```

## Results:

By commenting the “**Post**” thread 1 has been executing critical section while the thread 2 is in waiting state.

## **Task\_01\_C: Commenting the wait and checking the working: CODE:**

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

## Output:

The screenshot shows the Visual Studio Code interface running in WSL (Ubuntu 24.04). The terminal window displays the execution of a C program named Task\_01.c. The program uses two threads (Thread 1 and Thread 2) to increment a shared counter. The output shows interleaved messages from both threads as they enter a critical section (marked by '| Counter = ...') and update the counter. The final output shows the counter has been incremented 10 times.

```
• akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$ gcc Task_01.c -lpthread
• akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$ ./Task_01.out
Thread 1: Waiting...
Thread 1: In critical section | Counter = 1
Thread 2: Waiting...
Thread 2: In critical section | Counter = 2
Thread 2: Waiting...
Thread 2: In critical section | Counter = 3
Thread 1: Waiting...
Thread 1: 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 2: Waiting...
Thread 2: In critical section | Counter = 7
Thread 1: Waiting...
Thread 1: In critical section | Counter = 8
Thread 2: Waiting...
Thread 2: In critical section | Counter = 9
Thread 1: Waiting...
Thread 1: In critical section | Counter = 10
Final Counter Value: 10
◦ akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$
```

## Results:

By commenting the “Wait”, the output will be same as the actual output but the change is that the existing numbers of threads will not provide any protection to the critical section.

## Task\_02\_A:

CODE:

```
1 // Counting Semaphore Example
2
3 #include <stdio.h>
4 #include <pthread.h>
5 #include <semaphore.h>
6 #include <unistd.h>
7
8 sem_t resource_semaphore;
9 void* thread_function(void* arg) {
10     int thread_id = *(int*)arg;
11     printf("Thread %d: Waiting for resource...\n", thread_id);
12     sem_wait(&resource_semaphore); // Wait: decrement counter
13     printf("Thread %d: Acquired resource!\n", thread_id);
14     sleep(2); // Use resource
15     printf("Thread %d: Releasing resource...\n", thread_id);
16     sem_post(&resource_semaphore); // Signal: increment counter
17     return NULL;
18 }
19
20 int main() {
21     sem_init(&resource_semaphore, 0, 3); // Allow 3 concurrent threads
22     pthread_t threads[5];
23     int ids[5];
24     for (int i = 0; i < 5; i++) {
25         ids[i] = i;
26         pthread_create(&threads[i], NULL, thread_function, &ids[i]);
27     }
28
29     for (int i = 0; i < 5; i++) {
30         pthread_join(threads[i], NULL);
31     }
32
33     sem_destroy(&resource_semaphore);
34     return 0;
35 }
```

## Output:

The screenshot shows a Windows desktop environment with the Visual Studio Code (VS Code) application open. The terminal window displays the output of a C program running in WSL (Ubuntu-24.04). The program uses multiple threads to access a shared resource, specifically a mutex. The output shows alternating messages from different threads: some threads are waiting for the resource, while others have acquired it and are performing operations. The terminal window has tabs for 'bash - LAB\_09\_1132' and 'Task\_02.out'. The status bar at the bottom of the terminal window shows the current line (Ln 35), column (Col 1), spaces (Spaces: 4), and encoding (UTF-8). The date and time (11/21/2025, 3:00 PM) are also visible.

```
• akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$ gcc Task_02.c -o Task_02.out
• akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$ ./Task_02.out
Thread 0: Waiting for resource...
Thread 1: Waiting for resource...
Thread 0: Acquired resource!
Thread 2: Waiting for resource...
Thread 2: Acquired resource!
Thread 3: Waiting for resource...
Thread 1: Acquired resource!
Thread 4: Waiting for resource...
Thread 0: Releasing resource...
Thread 2: Releasing resource...
Thread 3: Acquired resource!
Thread 1: Releasing resource...
Thread 4: Acquired resource!
Thread 3: Releasing resource...
Thread 4: Releasing resource...
○ akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$
```

## Task\_02\_B:

CODE:

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4 #include <unistd.h>
5
6 sem_t mutex; // Binary semaphore
7 int counter = 0;
8
9 // Thread that increments counter
10 void* increment_thread(void* arg) {
11     int id = *(int*)arg;
12     for (int i = 0; i < 5; i++) {
13         printf("Thread %d: Waiting to increment...\n", id);
14         sem_wait(&mutex); // acquire
15         counter++;
16         printf("Thread %d: Incremented | Counter = %d\n", id, counter);
17         sleep(1);
18         sem_post(&mutex); // release
19         sleep(1);
20     }
21     return NULL;
22 }
23
24 // Thread that decrements counter
25 void* decrement_thread(void* arg) {
26     int id = *(int*)arg;
27     for (int i = 0; i < 5; i++) {
28         printf("Thread %d: Waiting to decrement...\n", id);
29         sem_wait(&mutex); // acquire
30         counter--;
31         printf("Thread %d: Decrement | Counter = %d\n", id, counter);
32         sleep(1);
33         sem_post(&mutex); // release
34         sleep(1);
35     }
36     return NULL;
37 }
38
39 int main() {
40     sem_init(&mutex, 0, 1); // semaphore = 1
41     pthread_t t1, t2;
42     int id1 = 1, id2 = 2;
43     pthread_create(&t1, NULL, increment_thread, &id1);
44     pthread_create(&t2, NULL, decrement_thread, &id2);
45
46     pthread_join(t1, NULL);
47     pthread_join(t2, NULL);
48
49     printf("Final Counter Value: %d\n", counter);
50     sem_destroy(&mutex);
51     return 0;
52 }
```

## Output:

```
akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$ gcc Task_02.c -o Task_02.out -lpthread
akasha@DESKTOP-NOLGFR1:~/Operating-System-1132/LAB_Tasks/LAB_09_1132$ ./Task_02.out
Thread 1: Waiting to increment...
Thread 2: Waiting to decrement...
Thread 1: Incremented | Counter = 1
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 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
Final Counter Value: 0
```

## Task\_03: Comparison Between Mutex and Semaphores:

### Similarities

#### 1. Both control access to shared resources

They make sure multiple threads don't mess up a shared variable or memory area.

#### 2. Both prevent race conditions

They stop two threads from entering a critical part of code at the same time.

#### 3. Both may cause threads to wait

If the resource is not available, the thread pauses.

#### 4. Both use wait and release concepts

Mutex → lock / unlock

Semaphore → wait (P) / signal (V)

### Difference

The difference between the mutex and semaphore is as follows

Feature	Mutex	Semaphore
<b>Basic Idea</b>	A lock used by only one thread at a time	A counter that allows multiple threads depending on availability

<b>Ownership</b>	Has an owner — only the locking thread can unlock	No owner — any thread can signal (increase) it
<b>Number of Threads Allowed</b>	Only <b>1</b> thread	<b>Many</b> threads depending on counter value
<b>Counter Value</b>	Always <b>0 or 1</b>	Can be <b>0, 1, 2, ... N</b>
<b>Main Purpose</b>	Mutual exclusion (protect a critical section)	Control access to multiple resources or synchronize threads
<b>If Resource Is Busy</b>	Thread waits until mutex is unlocked	Thread waits only if counter is 0
<b>Concept Type</b>	Locking mechanism	Signaling + resource counting mechanism
<b>Best Used For</b>	Protecting shared variables / one-at-a-time tasks	Managing limited resources (connections, buffers, threads)