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:

5th - A

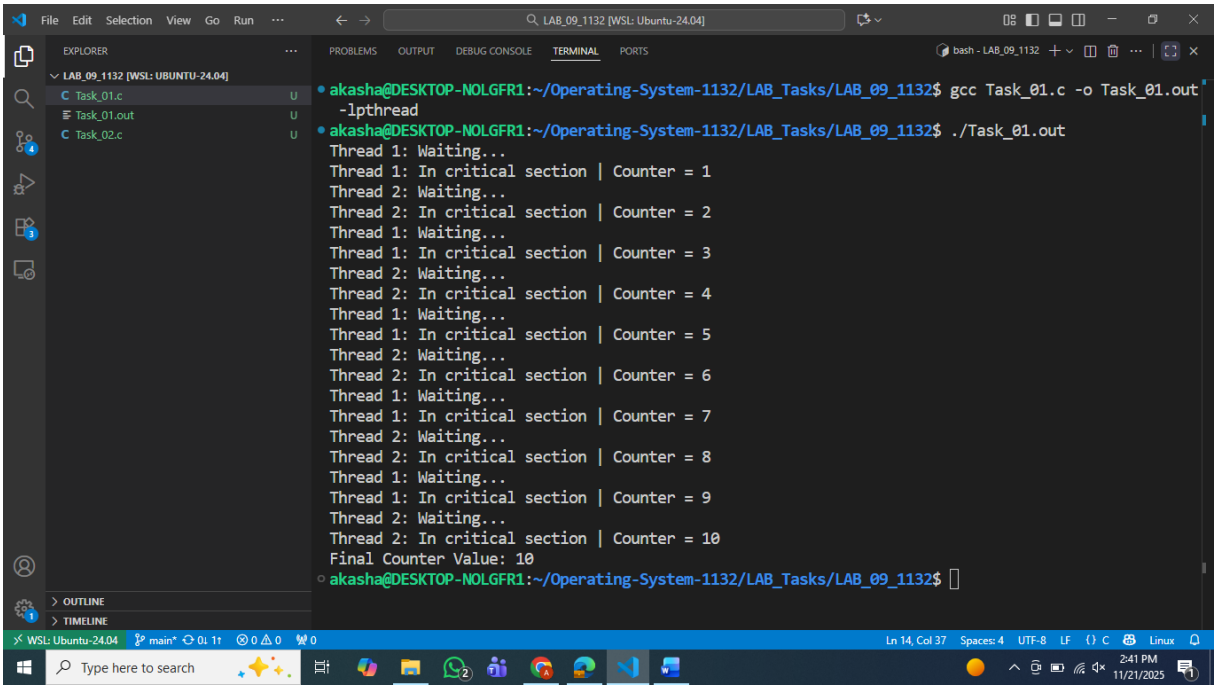
LAB_09

Task_01: Actual Program

CODE:

```c
// Binary Semaphore Example

#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:

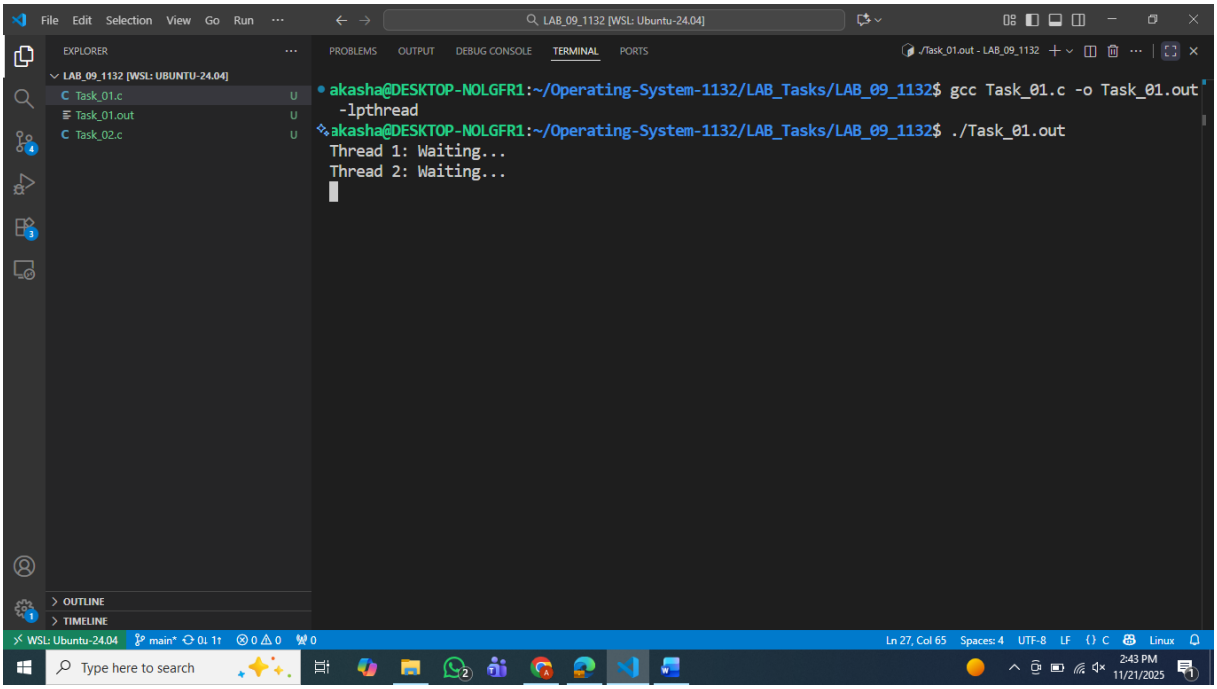


Task_01_A: Initializing Semaphores to the 0, 0:

CODE:

```c
// Binary Semaphore Example

#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, 0); // Binary semaphore initialized to 0
    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:



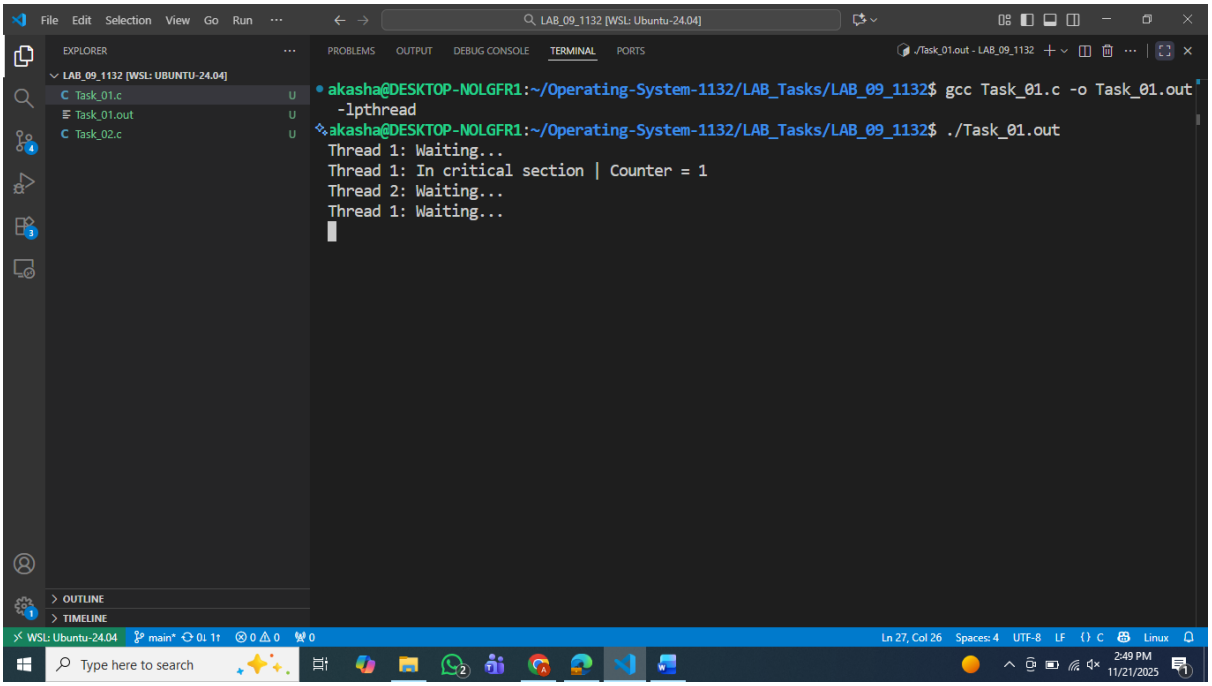
Results:
Since the binary semaphore initialized to 0,0 none of thread is executing, thus both threads are in waiting state.

Task_01_B: Commenting the post

CODE:

```c
// Binary Semaphore Example

#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:



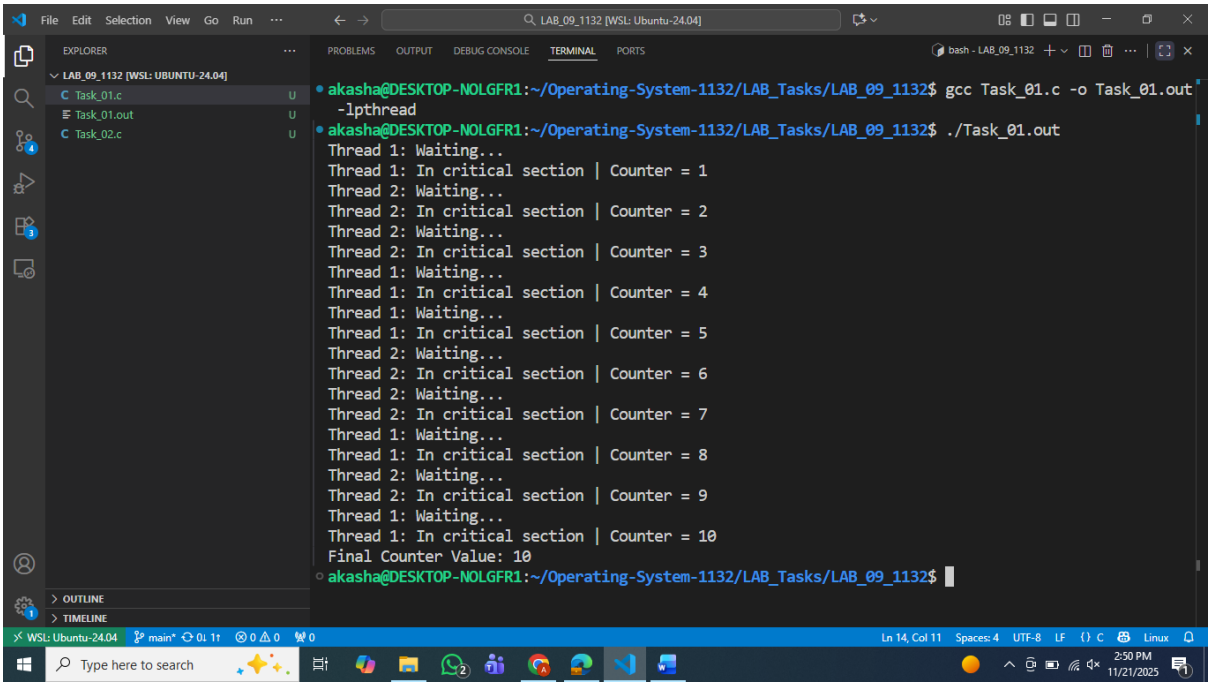
Results:
By commenting the "**Post**" thread 1 has been executing critical section while the thread 2 is in waiting state.

Task_01_C: Commenting the wait and checking the working:

CODE:

```c
// Binary Semaphore Example

#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:



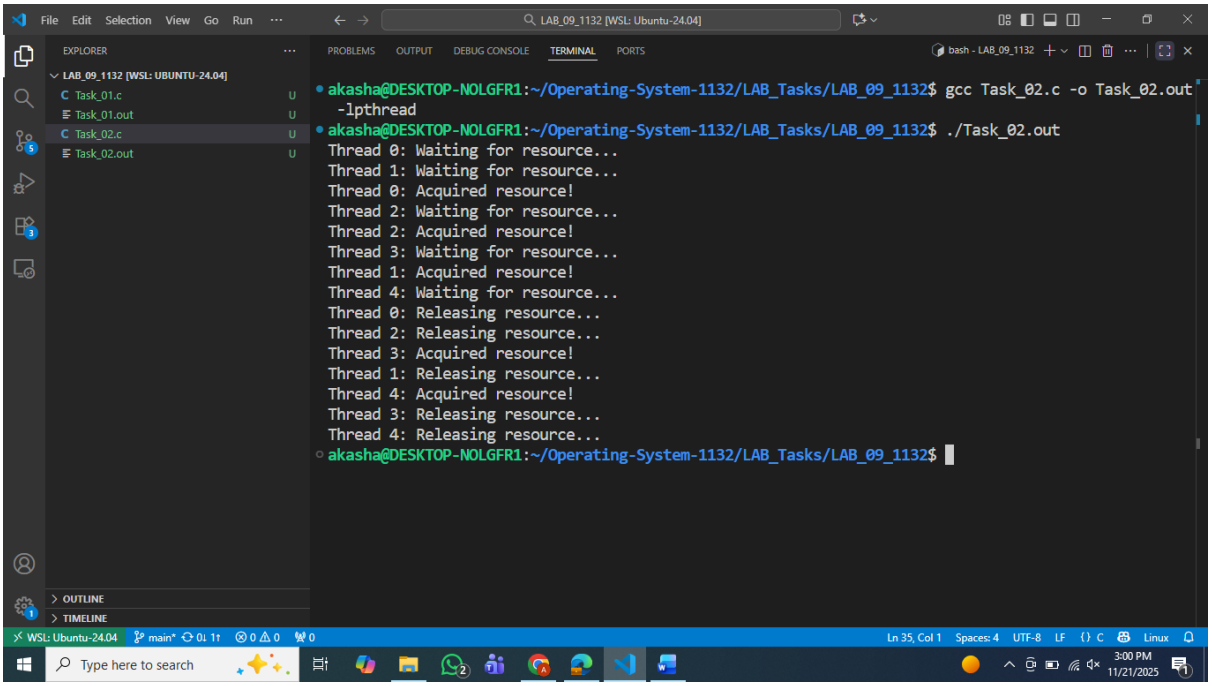
Results:
By commenting the "**Wait",** the output will be same as the actual output but the change is that the existing numbers of threads will not provide any protection to the critical section.

Task_02_A:

CODE:

```c
// Counting Semaphore Example

#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t resource_semaphore;
void* thread_function(void* arg) {
    int thread_id = *(int*)arg;
    printf("Thread %d: Waiting for resource...\n", thread_id);
    sem_wait(&resource_semaphore); // Wait: decrement counter
    printf("Thread %d: Acquired resource!\n", thread_id);
    sleep(2); // Use resource
    printf("Thread %d: Releasing resource...\n", thread_id);
    sem_post(&resource_semaphore); // Signal: increment counter
    return NULL;
}

int main() {
    sem_init(&resource_semaphore, 0, 3); // Allow 3 concurrent threads
    pthread_t threads[5];
    int ids[5];
    for (int i = 0; i < 5; i++) {
        ids[i] = i;
        pthread_create(&threads[i], NULL, thread_function, &ids[i]);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }

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

Output:

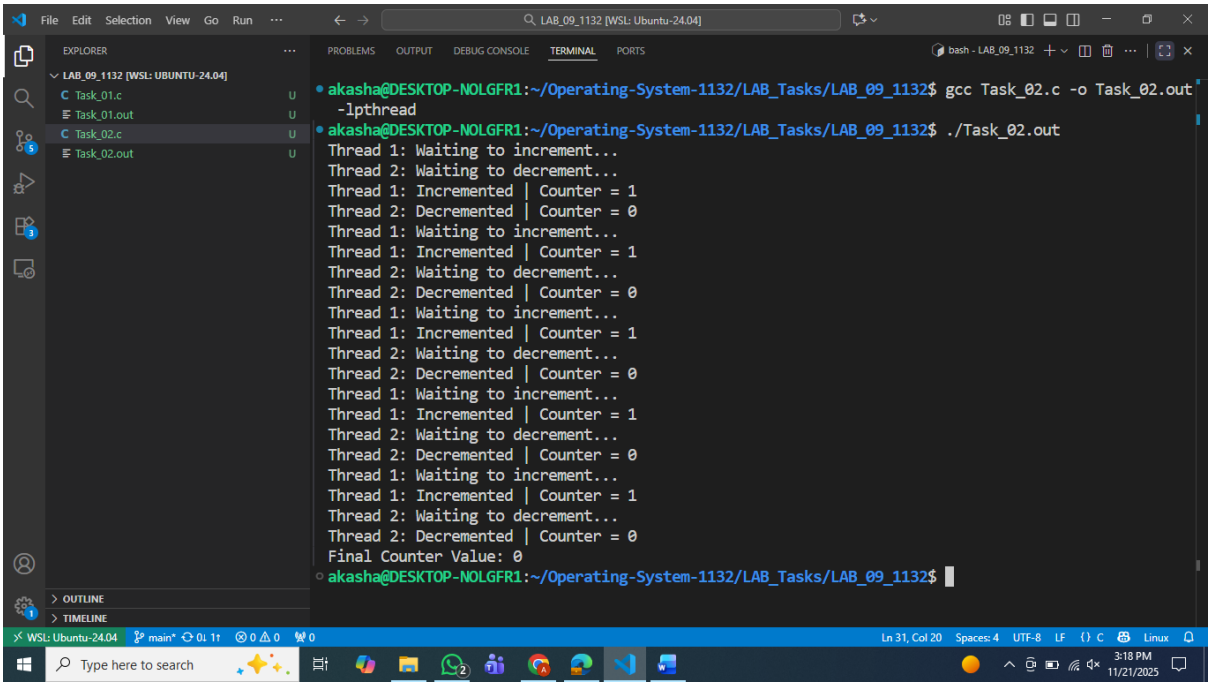


Task_02_B:

CODE:

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex; // Binary semaphore
int counter = 0;

// Thread that increments counter
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 decrements counter
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;
    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:



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
Basic Idea	A lock used by only one thread at a time	A counter that allows multiple threads depending on availability

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