National Textile University

Department of Computer Science

Subject:

Operating System

Submitted to:

Sir Nasir

Submitted by:

Asbah Asif

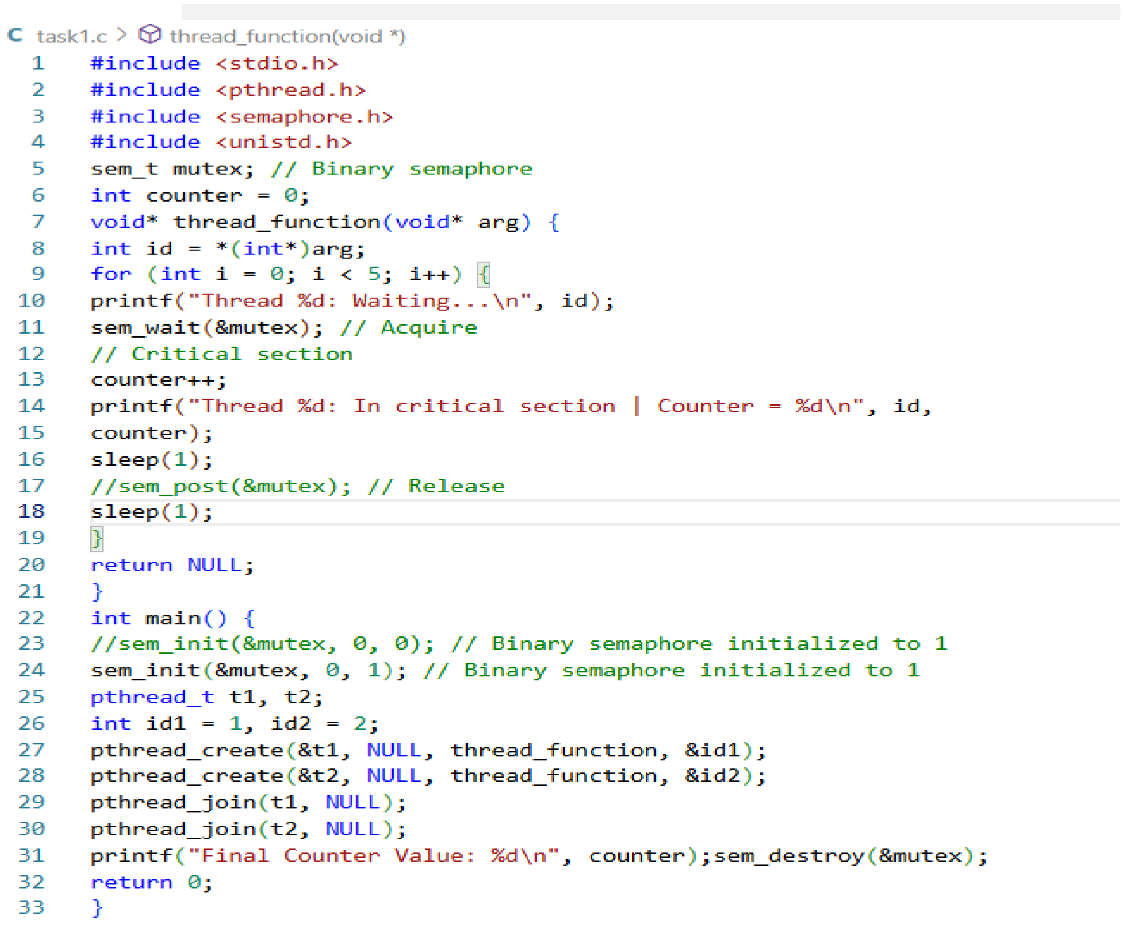
Reg number:

23-NTU-CS-1141

Lab no: 9

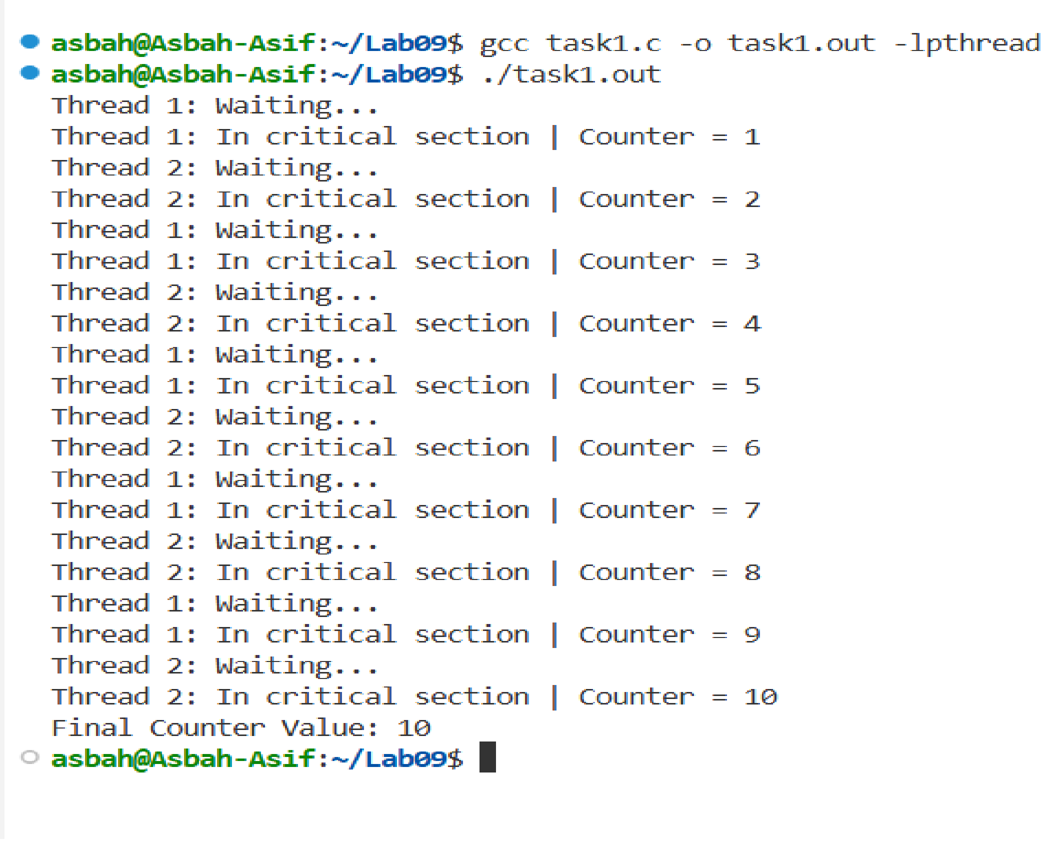
Semester: 5th

Task 1:



```c
#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 1
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;
}
```

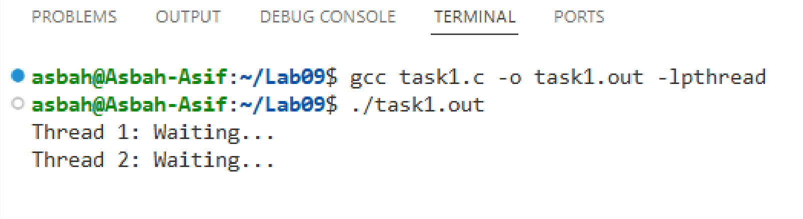
Output:



```
asbah@Asbah-Asif:~/Lab09$ gcc task1.c -o task1.out -lpthread
asbah@Asbah-Asif:~/Lab09$ ./task1.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
asbah@Asbah-Asif:~/Lab09$
```

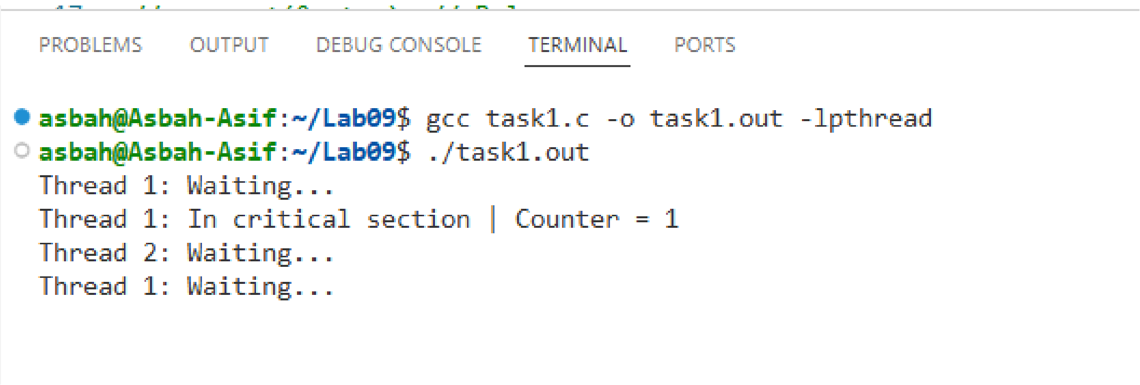
- When //sem_init(&mutex, 0, 0); // Binary semaphore initialized to 1

Output be like:



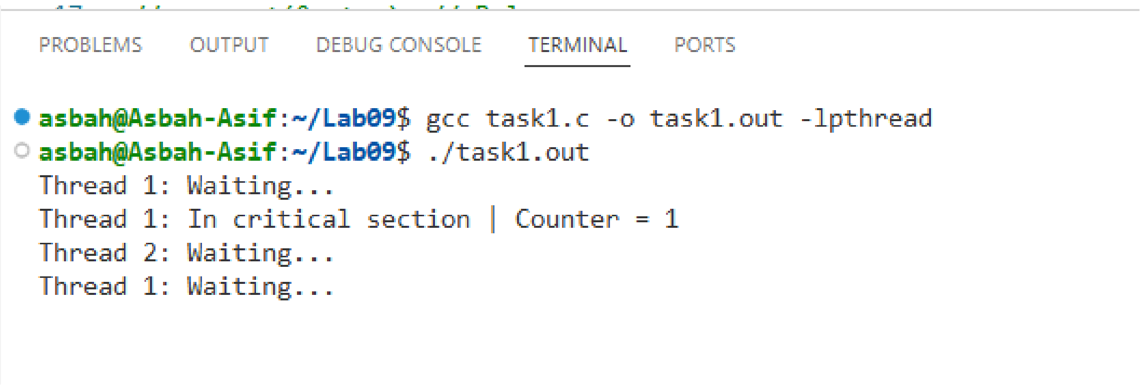
```
● asbah@Asbah-Asif:~/Lab09$ gcc task1.c -o task1.out -lpthread
○ asbah@Asbah-Asif:~/Lab09$ ./task1.out
  Thread 1: Waiting...
  Thread 2: Waiting...
```

- When Post is Committed:



```
● asbah@Asbah-Asif:~/Lab09$ gcc task1.c -o task1.out -lpthread
○ asbah@Asbah-Asif:~/Lab09$ ./task1.out
  Thread 1: Waiting...
  Thread 1: In critical section | Counter = 1
  Thread 2: Waiting...
  Thread 1: Waiting...
```

- When wait is committed:



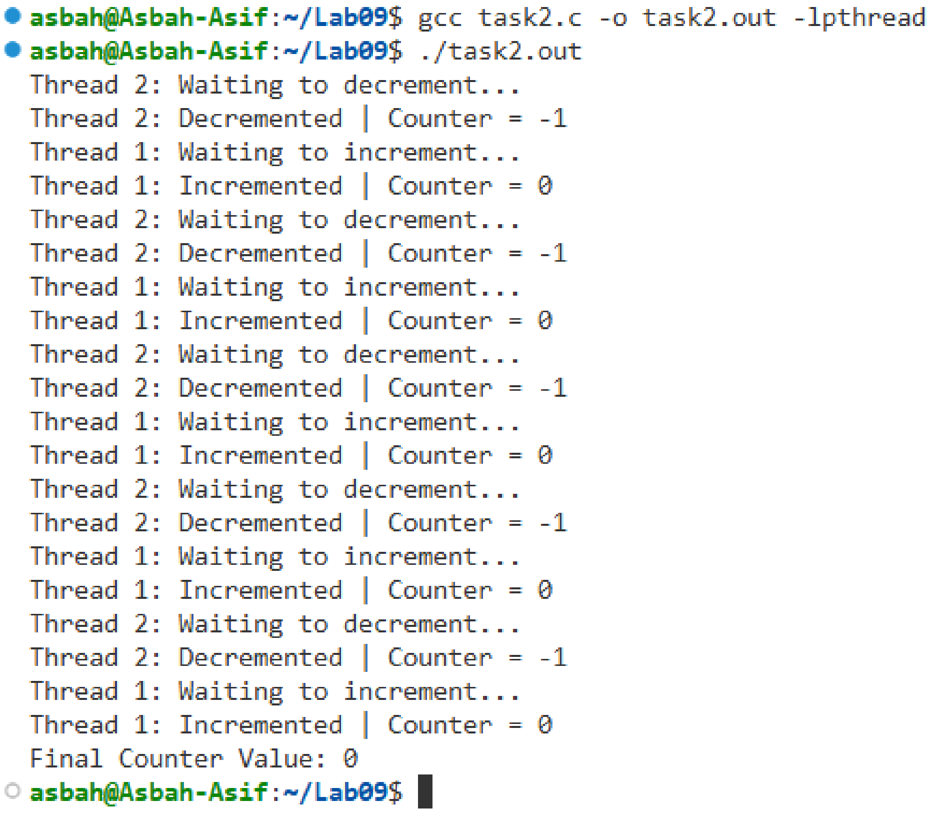
```
● asbah@Asbah-Asif:~/Lab09$ gcc task1.c -o task1.out -lpthread
○ asbah@Asbah-Asif:~/Lab09$ ./task1.out
  Thread 1: Waiting...
  Thread 1: In critical section | Counter = 1
  Thread 2: Waiting...
  Thread 1: Waiting...
```

Task 2:

Code:

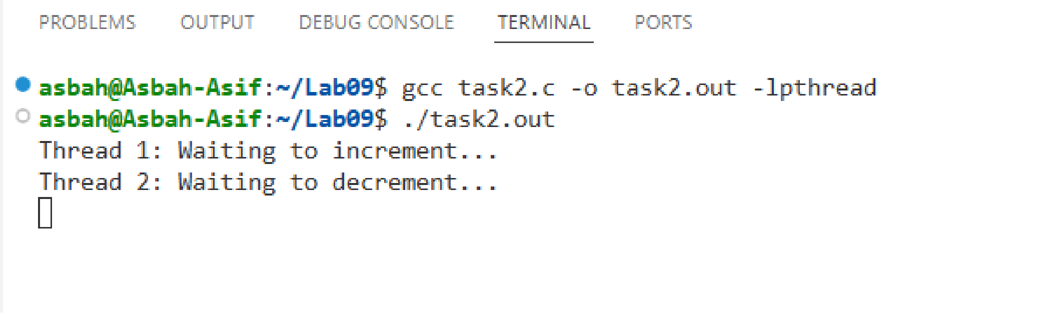
```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

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

// Thread that 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:



```
● asbah@Asbah-Asif:~/Lab09$ gcc task2.c -o task2.out -lpthread
● asbah@Asbah-Asif:~/Lab09$ ./task2.out
  Thread 2: Waiting to decrement...
  Thread 2: Decremented | Counter = -1
  Thread 1: Waiting to increment...
  Thread 1: Incremented | Counter = 0
  Thread 2: Waiting to decrement...
  Thread 2: Decremented | Counter = -1
  Thread 1: Waiting to increment...
  Thread 1: Incremented | Counter = 0
  Thread 2: Waiting to decrement...
  Thread 2: Decremented | Counter = -1
  Thread 1: Waiting to increment...
  Thread 1: Incremented | Counter = 0
  Thread 2: Waiting to decrement...
  Thread 2: Decremented | Counter = -1
  Thread 1: Waiting to increment...
  Thread 1: Incremented | Counter = 0
  Thread 2: Waiting to decrement...
  Thread 2: Decremented | Counter = -1
  Thread 1: Waiting to increment...
  Thread 1: Incremented | Counter = 0
  Final Counter Value: 0
○ asbah@Asbah-Asif:~/Lab09$ █
```

- When Semaphore is initialized to 0 sem_init(&mutex, 0, 0);   // semaphore = 0



```
● asbah@Asbah-Asif:~/Lab09$ gcc task2.c -o task2.out -lpthread
○ asbah@Asbah-Asif:~/Lab09$ ./task2.out
  Thread 1: Waiting to increment...
  Thread 2: Waiting to decrement...
  ▯
```

Task 3:
Technical Difference between Mutex and Semaphore

Feature	**Semaphore**	**Mutex**
Type	Signalling mechanism	Locking mechanism
Value	Integer value (≥ 0). Can be 0, 1, or more	Only 0 or 1 (binary lock)
Ownership	**No ownership** — any thread can signal (post)	**Has ownership** — only locking thread can unlock
Used For	Managing **multiple resources**	Protecting **single shared resource**
Blocking Behaviour	Sem _ wait () decrements; if value < 0 → thread blocks	Pthread _mutex _lock () blocks until lock is free
Unlocking	sem_ post () increments and may wake waiting threads	Pthread_ mutex _ unlock () releases lock
Type Variants	Binary semaphore, Counting semaphore	Only one type (mutex)
Multiple Access	Allows **N threads** to access resource if semaphore value is N	Allows **only 1 thread** at a time
Typical Usage	Producer-Consumer, Reader Limits, Resource Pool	Critical sections, shared variable updates
Kernel / User Level	Supported in both	Supported in both
Synchronization Level	Lightweight, used for signalling and resource count	Strict exclusion mechanism
Race Condition Protection	Yes (with care), but depends on correct use	Strong protection for critical sections
Deadlock Possibility	Yes	Yes
Example	sem_init (&sem, 0, 3); (3 resources)	pthread_mutex_lock(&m);