National Textile University

Department of Computer Science

Subject: Operating System

Submitted to: Sir Nasir

Submitted by: Eisha Muzaffar

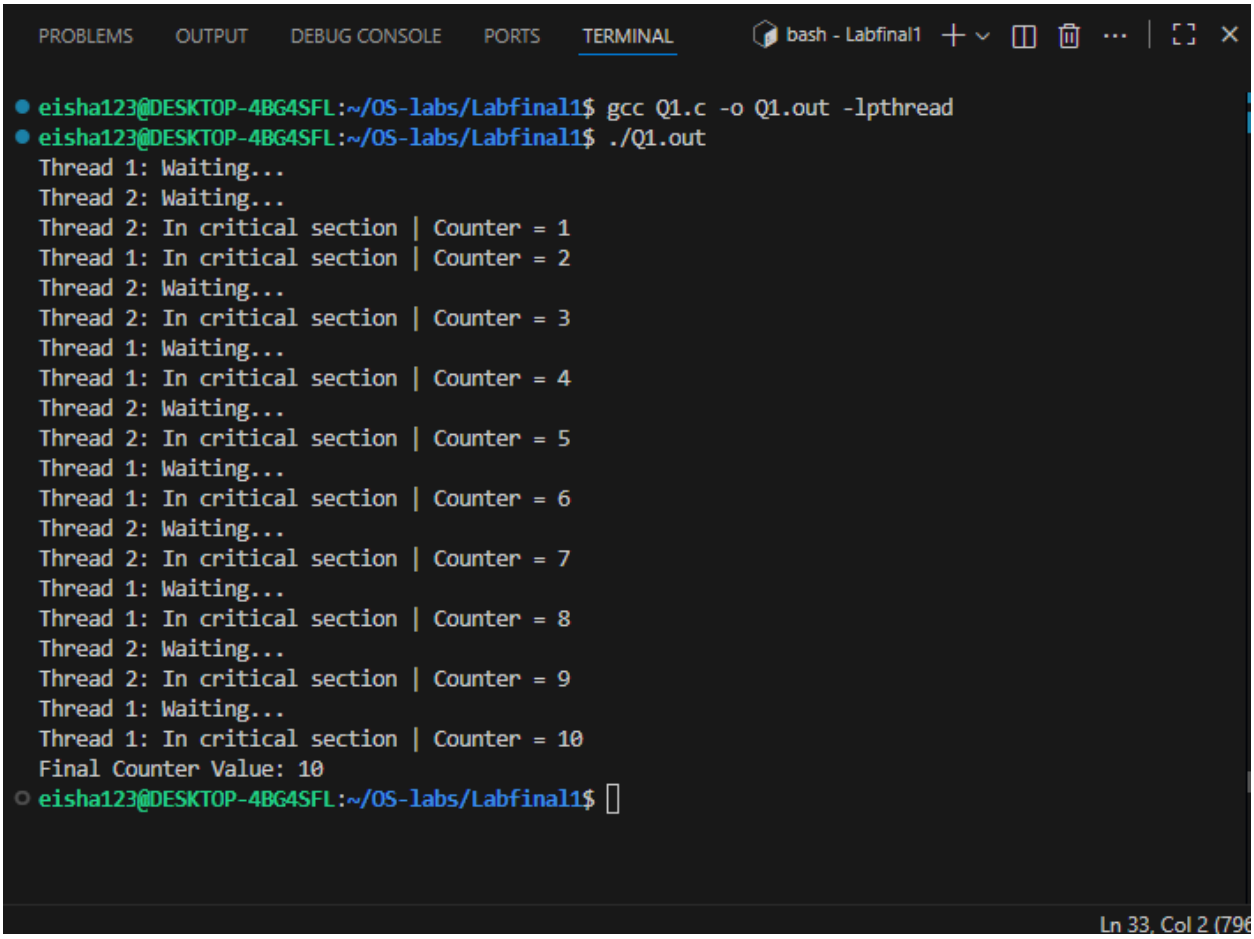
Reg. number: 23-NTU-CS-1147

Lab no.: Assignment

Semester:5th

```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, 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;
}
```

Terminal:



Initializing Semaphore with 0:

Explanation: As we have initialized with 0 the threads will be blocked ie both will keep waiting

```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
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;
}
```

Terminal:

```
eisha123@DESKTOP-4BG4SFL:~/OS-labs/Labfinal1$ gcc Q1sem.c -o Q1sem.out -lpthread
eisha123@DESKTOP-4BG4SFL:~/OS-labs/Labfinal1$ ./Q1sem.out
Thread 1: Waiting...
Thread 2: Waiting...
```

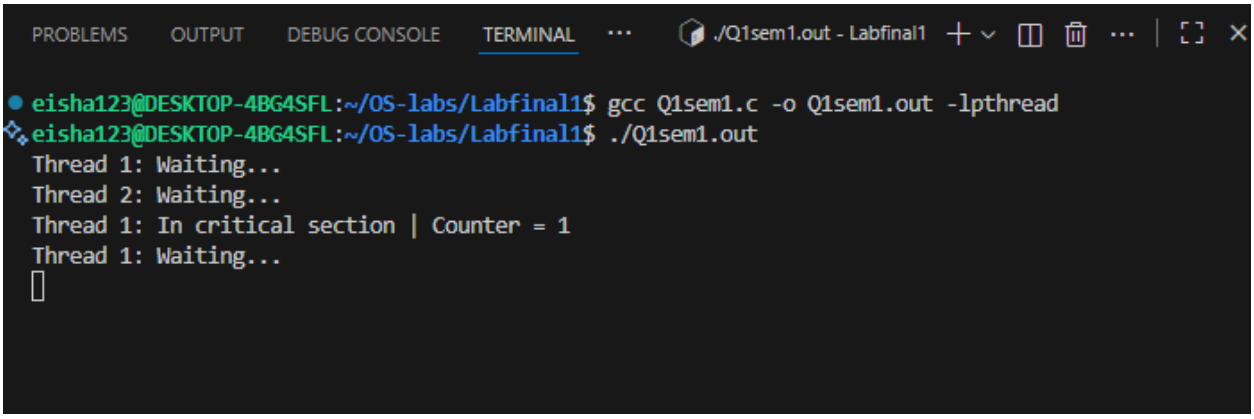
Commenting Post:

If you comment out `sem_post()`, the thread **never releases the semaphore**, so the second thread can **never enter the critical section**.
This causes the program to **freeze (deadlock)** forever.

```
#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;
```

```c
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;
}
```

Terminal:



Commenting Wait:

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex; // Binary semaphore
```

```c
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;
}
```

Terminal

```
TERMINAL
eisha123@DESKTOP-4BG4SFL:~/OS-labs/Labfinal1$ gcc Q1sem1.c -o Q1sem1.out -lpthread
eisha123@DESKTOP-4BG4SFL:~/OS-labs/Labfinal1$ ./Q1sem1.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
eisha123@DESKTOP-4BG4SFL:~/OS-labs/Labfinal1$
```

Creating Two thread Functions:

```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;
}
```

```
TERMINAL                                                  bash - Labfinal
eisha123@DESKTOP-4BG4SFL:~/OS-labs/Labfinal1$ gcc Q2.c -o Q2.out -lpthread
eisha123@DESKTOP-4BG4SFL:~/OS-labs/Labfinal1$ ./Q2.out
Thread 1: Waiting to increment...
Thread 1: Incremented | Counter = 1
Thread 2: Waiting to decrement...
Thread 2: Decremented | Counter = 0
Thread 1: Waiting to increment...
Thread 1: Incremented | Counter = 1
Thread 2: Waiting to decrement...
Thread 2: Decremented | Counter = 0
Thread 1: Waiting to increment...
Thread 1: Incremented | Counter = 1
Thread 2: Waiting to decrement...
Thread 2: Decremented | Counter = 0
Thread 1: Waiting to increment...
Thread 1: Incremented | Counter = 1
Thread 2: Waiting to decrement...
Thread 2: Decremented | Counter = 0
Thread 1: Waiting to increment...
Thread 1: Incremented | Counter = 1
Thread 2: Waiting to decrement...
Thread 2: Decremented | Counter = 0
Final Counter Value: 0
eisha123@DESKTOP-4BG4SFL:~/OS-labs/Labfinal1$
```

Task3: Difference between mutex and semaphore:

Feature	Mutex	Semaphore
Definition	A locking mechanism to provide **mutual exclusion** for shared resources	A signaling mechanism to control access to resources or coordinate threads
Value	Binary (locked/unlocked)	Counting (can be 0,1,...N) or binary (0/1)
Ownership	Has ownership — only the thread that locks can unlock	No ownership — any thread can post (release)
Purpose	Primarily for **mutual exclusion**	Can be used for mutual exclusion **and** signaling between threads
Blocking behavior	Thread waits if mutex is already locked	Thread waits if semaphore value ≤ 0
Who can release/unlock	Only the owning thread	Any thread can perform `sem_post()`

Feature	Mutex	Semaphore
Use case	Protect critical sections	Control multiple resources, producer-consumer problems, or event signaling
Deadlock possibility	If thread forgets to unlock → deadlock	Same, but more flexible; misusing post can allow race conditions