



National Textile University
Department of Computer Science

Subject: Operating System

Submitted to: Sir Nasir Mahmood

Submitted by: Nimra Tanveer

Reg number:23-NTU-CS-1201

Lab no .07

Task 1: 4.3. Binary Semaphore Example

Part a

- **Two threads run at the same time, but a binary semaphore makes sure only one thread enters the critical section at a time.**
- **Each thread increases to a shared variable counter safely without conflict.**
- **In the end, the program prints the final counter value after both threads finish.**



```
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  void* thread_function(void* arg) {
10     int id = *(int*)arg;
11
12     for (int i = 0; i < 5; i++) {
13         printf("Thread %d: Waiting...\n", id);
14         sem_wait(&mutex); // Acquire
15
16         // 4.4 Counting Semaphore Example
17         // A counting semaphore with initial value = 3 allows
18         // up to 3 threads to access a resource simultaneously.
19
20         // Critical section
21         counter++;
22         printf("Thread %d: In critical section | Counter = %d\n", id, counter);
23         sleep(1);
24
25         sem_post(&mutex); // Release
26         sleep(1);
27     }
28
29     return NULL;
30 }
31
32 int main() {
33     sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1
34
35     pthread_t t1, t2;
36     int id1 = 1, id2 = 2;
37
38     pthread_create(&t1, NULL, thread_function, &id1);
39     pthread_create(&t2, NULL, thread_function, &id2);
40
41     pthread_join(t1, NULL);
42     pthread_join(t2, NULL);
43
44     printf("Final Counter Value: %d\n", counter);
45
46     sem_destroy(&mutex);
47     return 0;
48 }
49
```

The screenshot shows a Visual Studio Code window with a C program named `task1_a.c` and its output in the terminal. The program uses a semaphore to coordinate two threads, `Thread 1` and `Thread 2`, which both increment a shared counter. The output shows that the threads execute in an interleaved manner, with each thread performing 10 iterations of entering the critical section and waiting.

```
#include <stdio.h>

// Semaphore
sem_t mutex;

// Thread 1
void* thread1(void*) {
    for (int i = 0; i < 10; i++) {
        sem_wait(&mutex);
        printf("Thread 1: In critical section | Counter = %d\n", counter);
        counter++;
        sem_post(&mutex);
        printf("Thread 1: Waiting...\n");
    }
}

// Thread 2
void* thread2(void*) {
    for (int i = 0; i < 10; i++) {
        sem_wait(&mutex);
        printf("Thread 2: In critical section | Counter = %d\n", counter);
        counter++;
        sem_post(&mutex);
        printf("Thread 2: Waiting...\n");
    }
}

int main() {
    sem_init(&mutex, 0, 1);
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Final Counter Value: %d\n", counter);
    return 0;
}
```

Terminal Output:

```
nimra@DESKTOP-8CMFJK1:~/OS-hometask1/lab7_05$ gcc task1_a.c -pthread -o task1_a.out
nimra@DESKTOP-8CMFJK1:~/OS-hometask1/lab7_05$ ./task1_a.out
Thread 1: In critical section | Counter = 1
Thread 2: Waiting...
Thread 1: Waiting...
Thread 2: In critical section | Counter = 2
Thread 1: In critical section | Counter = 3
Thread 2: Waiting...
Thread 1: Waiting...
Thread 2: In critical section | Counter = 4
Thread 1: In critical section | Counter = 5
Thread 2: Waiting...
Thread 1: Waiting...
Thread 2: In critical section | Counter = 6
Thread 1: In critical section | Counter = 7
Thread 2: Waiting...
Thread 1: Waiting...
Thread 2: In critical section | Counter = 8
Thread 1: In critical section | Counter = 9
Thread 2: Waiting...
Thread 1: In critical section | Counter = 10
Final Counter Value: 10
nimra@DESKTOP-8CMFJK1:~/OS-hometask1/lab7_05$
```

Task b: commenting `sem_t` mutex

- **Commenting on `sem_t` mutex means the semaphore does not exist, so a compile-time error will occur.**
- **The program will not run because there is no defined semaphore named `mutex`.**



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

The screenshot shows a Visual Studio Code editor window with a C program named `task1_b.c` open. The program includes `<stdio.h>` and defines a semaphore `mutex` with an initial value of 0. It contains two threads, `thread_function` and `main`, both of which call `sem_wait(&mutex)`. The terminal output shows the program running successfully, with the counter increasing from 0 to 10. The compilation errors are as follows:

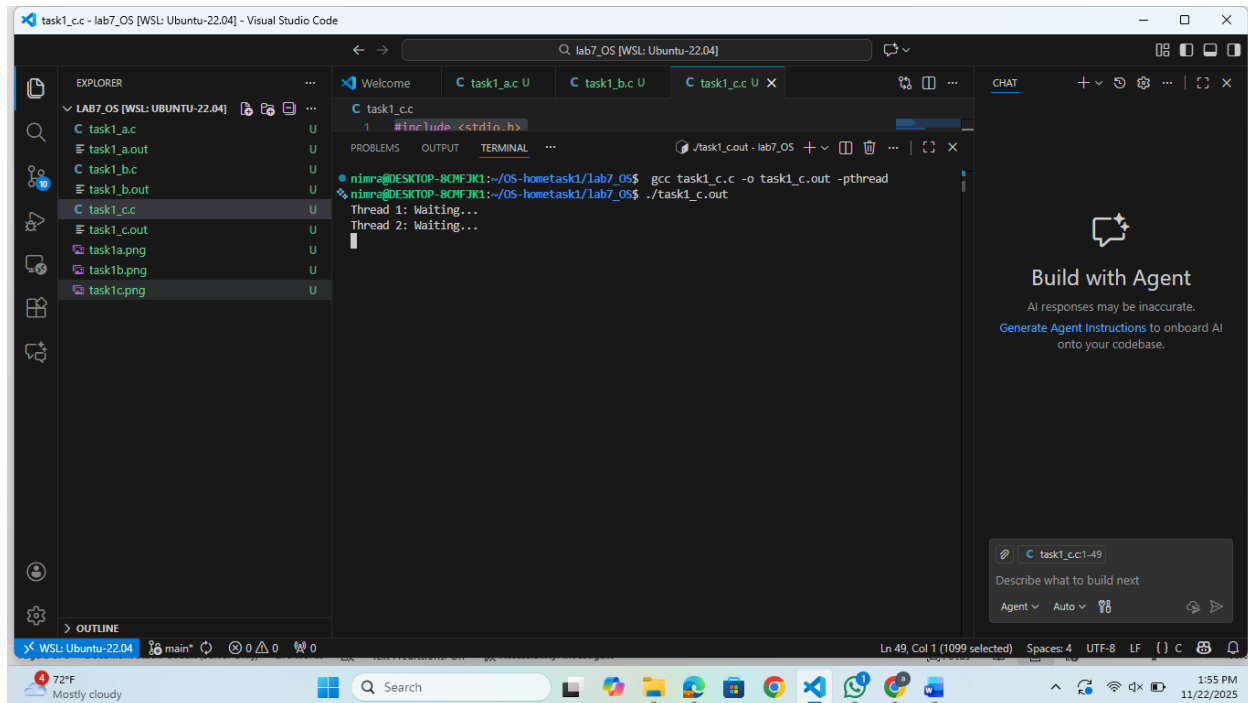
```
nimra@DESKTOP-8CMFJK1:~/OS-hometask1/lab7_OS$ gcc task1_b.c -pthread -o task1_b.out
task1_b.c: In function 'thread_function':
task1_b.c:11:11: error: 'mutex' undeclared (first use in this function)
11 | sem_wait(&mutex); // Acquire
   |           ^~~~~~
task1_b.c:11:11: note: each undeclared identifier is reported only once for each function it appears in
task1_b.c: In function 'main':
task1_b.c:24:11: error: 'mutex' undeclared (first use in this function)
24 | sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1
   |           ^~~~~~
```

Task c:

- With initial value 0, the semaphore is locked from the start.
- Both threads get stuck in `sem_wait()` and the program never continues.



```
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  void* thread_function(void* arg) {
10     int id = *(int*)arg;
11
12     for (int i = 0; i < 5; i++) {
13         printf("Thread %d: Waiting...\n", id);
14         sem_wait(&mutex); // Acquire
15
16         // 4.4 Counting Semaphore Example
17         // A counting semaphore with initial value = 3 allows
18         // up to 3 threads to access a resource simultaneously.
19
20         // Critical section
21         counter++;
22         printf("Thread %d: In critical section | Counter = %d\n", id, counter);
23         sleep(1);
24
25         sem_post(&mutex); // Release
26         sleep(1);
27     }
28
29     return NULL;
30 }
31
32 int main() {
33     sem_init(&mutex, 0, 0); // Binary semaphore initialized to 1
34
35     pthread_t t1, t2;
36     int id1 = 1, id2 = 2;
37
38     pthread_create(&t1, NULL, thread_function, &id1);
39     pthread_create(&t2, NULL, thread_function, &id2);
40
41     pthread_join(t1, NULL);
42     pthread_join(t2, NULL);
43
44     printf("Final Counter Value: %d\n", counter);
45
46     sem_destroy(&mutex);
47     return 0;
48 }
49
```



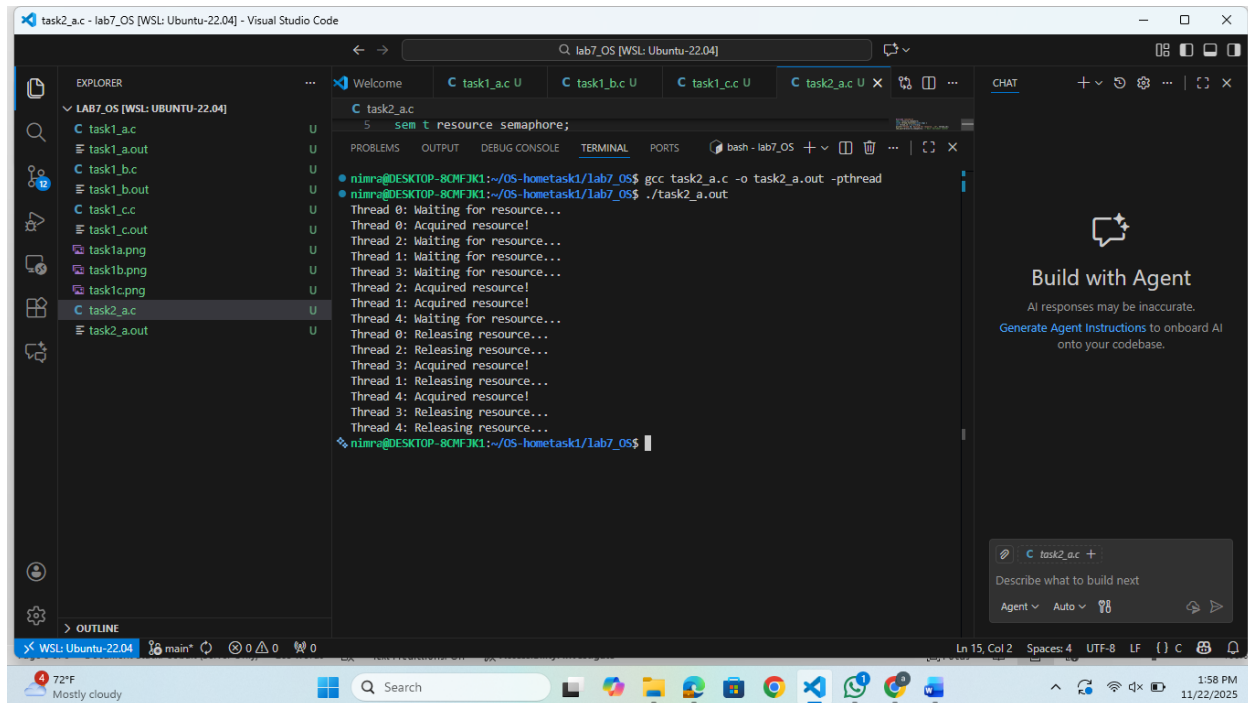
Task2: 4.4. Counting Semaphore Example

Task a:

- At the start, semaphore value is 3.
- So, Thread 0, Thread 1, Thread 2 will immediately enter.
- Thread 3 and Thread 4 will have to wait until one of the first threads releases the resource.
- This code allows only 3 threads to run inside the critical section at the same time using a counting semaphore.




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



Task b: commenting mutex

- If you comment out `sem_t resource_semaphore` then the variable no longer exists, so every place that uses it `sem_wait` gives an undeclared error.
- Because the semaphore is removed but still used in the code, the program cannot compile and therefore cannot run.



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



```

1  #include <stdio.h>
2  #include <pthread.h>
3  #include <semaphore.h>
4  #include <unistd.h>
5
6  sem_t resource_semaphore;
7
8  // FIRST THREAD FUNCTION
9  void* thread_function1(void* arg) {
10     int thread_id = *(int*)arg;
11
12     printf("Thread %d (F1): Waiting for resource...\n", thread_id);
13     sem_wait(&resource_semaphore);
14
15     printf("Thread %d (F1): Acquired resource!\n", thread_id);
16     sleep(2);
17
18     printf("Thread %d (F1): Releasing resource...\n", thread_id);
19     sem_post(&resource_semaphore);
20
21     return NULL;
22 }
23
24 // SECOND THREAD FUNCTION (same style)
25 void* thread_function2(void* arg) {
26     int thread_id = *(int*)arg;
27
28     printf("Thread %d (F2): Waiting for resource...\n", thread_id);
29     sem_wait(&resource_semaphore);
30
31     printf("Thread %d (F2): Acquired resource!\n", thread_id);
32     sleep(2);
33
34     printf("Thread %d (F2): Releasing resource...\n", thread_id);
35     sem_post(&resource_semaphore);
36
37     return NULL;
38 }
39
40 int main() {
41     sem_init(&resource_semaphore, 0, 3);
42
43     pthread_t t1, t2;
44     int id1 = 1, id2 = 2;
45
46     // FIRST function thread
47     pthread_create(&t1, NULL, thread_function1, &id1);
48
49     // SECOND function thread
50     pthread_create(&t2, NULL, thread_function2, &id2);
51
52     pthread_join(t1, NULL);
53     pthread_join(t2, NULL);
54
55     sem_destroy(&resource_semaphore);
56
57     return 0;
58 }
59

```

```
task2_c.c - lab7_OS [WSL: Ubuntu-22.04] - Visual Studio Code
lab7_OS [WSL: Ubuntu-22.04]
task2_c.c
1 #include <stdio.h>
...
nirm@DESKTOP-8CMFJK1:~/OS-hometask1/lab7_OS$ gcc task2_c.c -o task2_c.out -pthread
nirm@DESKTOP-8CMFJK1:~/OS-hometask1/lab7_OS$ ./task2_c.out
Thread 1 (F1): Waiting for resource...
Thread 1 (F1): Acquired resource!
Thread 2 (F2): Waiting for resource...
Thread 2 (F2): Acquired resource!
Thread 2 (F2): Releasing resource...
Thread 1 (F1): Releasing resource...
```

Task 3: Difference between Semaphore and Mutex

Feature	Semaphore	Mutex
Definition	A signaling mechanism that uses a counter to control access to multiple resources.	A locking mechanism that allows only one thread or process to access a resource at a time.
Type	Can be Counting or Binary	Always Binary
Resource Access	Can allow multiple threads at the same time (based on counter).	Allows only ONE thread at a time.

Ownership	No ownership, any thread can release a semaphore.	Ownership exists, only the thread that locked the mutex can unlock it.
Usage	Used for resource pools, limiting access (e.g., 5 database connections).	Used for mutual exclusion on shared data.
Risk	If released incorrectly, resource inconsistency.	If not released by the owner, deadlock can occur.
Value Range	Value can be 0 to N	Only 0 or 1 (locked or unlocked).