National University of Computer and Emerging Sciences

Lab Manual 14
Operating Systems Lab

Course Instructor	Mr. Mubashar Hussain
Lab Instructor(s)	Mr. Hassan Raza
Section	BSE 5B
Semester	Fall 2024

Department of Software Engineering
FAST-NU, Lahore, Pakistan

A semaphore is a synchronization primitive that controls access to shared resources by multiple processes or threads. It maintains a counter and supports two fundamental operations:

- **Wait (P) Operation**: Decrements the semaphore value. If the semaphore value is non-negative, the decrement proceeds, and the process continues. If the semaphore value becomes negative (i.e., no more resources available), the process is blocked until another process increments the semaphore.
- **Signal (V) Operation**: Increments the semaphore value. If any processes were blocked waiting for this semaphore, one of them is allowed to proceed.

POSIX Semaphores

POSIX (Portable Operating System Interface) semaphores are a standardized form of semaphores available on Unix and Linux systems. They provide a way to synchronize processes (including unrelated processes) using named semaphores. POSIX semaphores are part of the POSIX Threads (pthreads) library (**libpthread**).

Key Functions for POSIX Semaphores

1. sem_open

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
```

- Opens or creates a named semaphore.
- **name**: Name of the semaphore (must start with a **/** character).
- **oflag**: Flags indicating the mode of operation (**O_CREAT** for creating if not existing, **O_EXCL** to ensure creation fails if the semaphore already exists).
- **mode**: Permissions for the semaphore if created (**0644** is commonly used).
- **value**: Initial value of the semaphore.

2. sem_wait

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

- Waits (P operation) on the semaphore.
- Decrements the semaphore value.
- Blocks if the semaphore value is zero (no resources available).

3. sem_post

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

- Signals (V operation) on the semaphore.
- Increments the semaphore value.

- Unblocks one of the waiting processes (if any).

4. sem_close

#include <semaphore.h>

int sem_close(sem_t *sem);

- Closes the named semaphore.
- Releases the associated resources.
- After closing, the semaphore can no longer be used by the process.

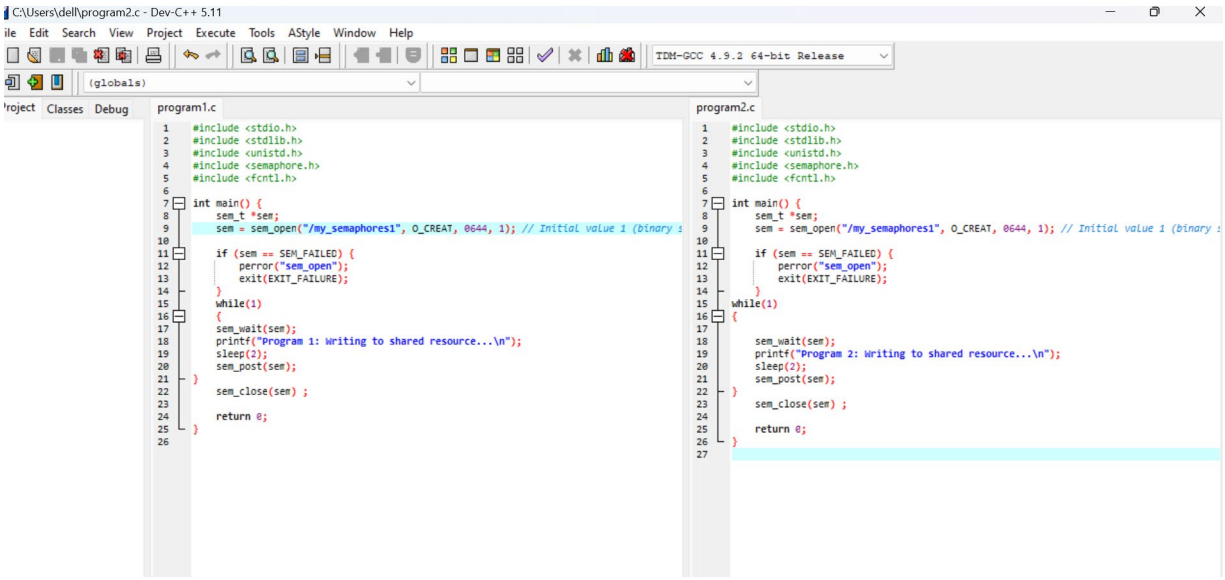
5. sem_unlink

#include <semaphore.h>

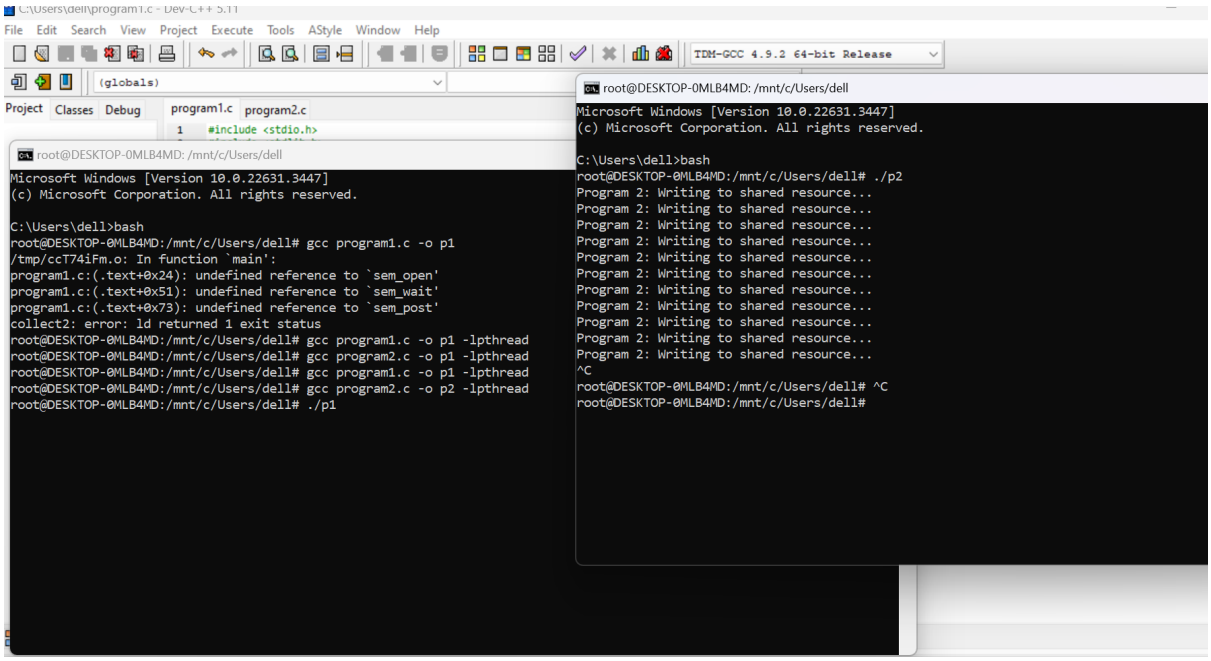
int sem_unlink(const char *name);

- Removes a named semaphore from the system.
- The semaphore can no longer be opened or used after unlinking.
- This is typically done after all processes using the semaphore have finished.

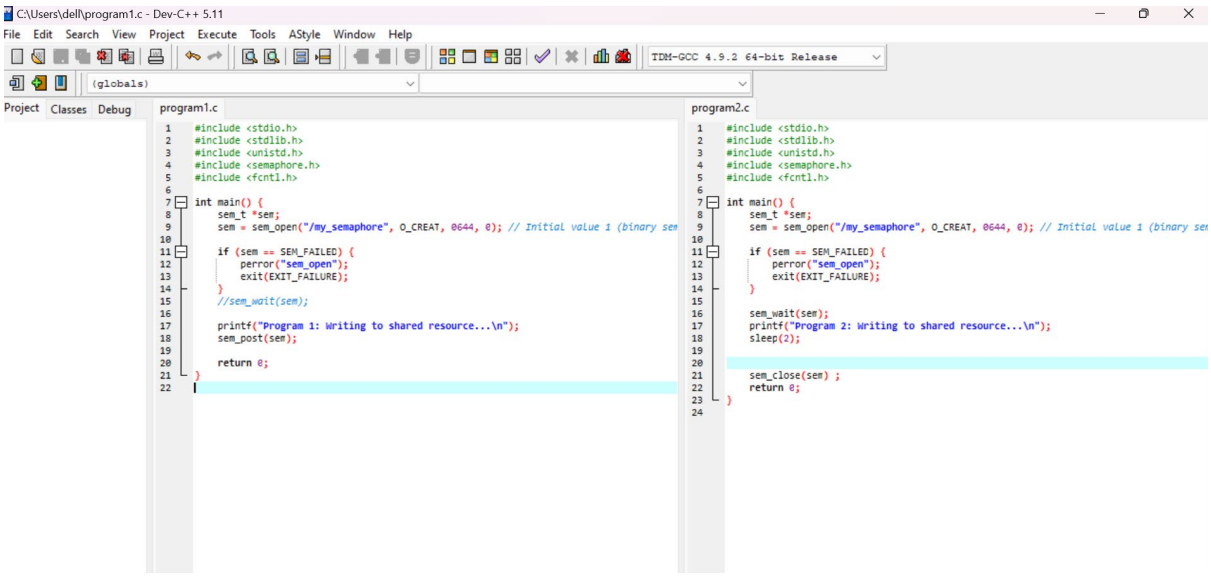
Example 1:



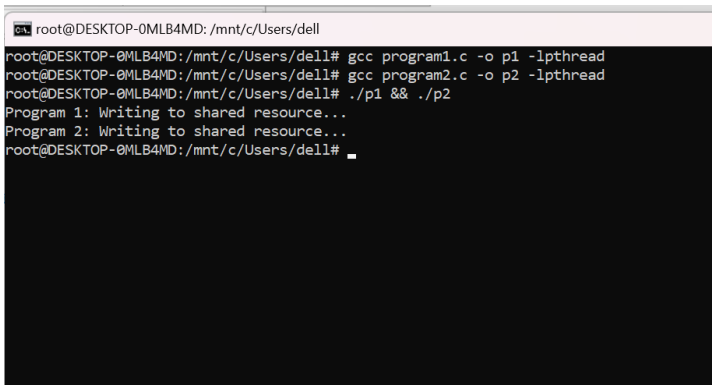
Output:



Example 2:



Output 2:



Question 1: Consider a scenario where three processes (**P1**, **P2**, **P3**) need to execute in a specific order while sharing a common resource (file) that requires mutual exclusion. Use semaphores to ensure that the processes execute in the sequence **P1 -> P2 -> P3** and have exclusive access to the shared resource when needed.

- P1 opens a file having all integers and calculate their sum and write it into the same file.
- After that P2 counts the integers and also write it in the same file.
- P3 reads sum and count from this file calculated by P1 and P2 and calculates average and print it on the screen.

Question 2: You are tasked with designing a system where three independent services (S1, S2, S3) process orders in an e-commerce platform. These services must execute in the sequence S1 -> S2 -> S3 to ensure data consistency. Each service uses a shared log file to record its activity, which requires mutual exclusion.

1. **S1 (Order Placement):** S1 receives and writes the order details (order ID and customer information) to the shared log file.
2. **S2 (Payment Processing):** Once the order is recorded, S2 reads the order details from the log, processes the payment, and updates the log with the payment status.
3. **S3 (Order Confirmation):** After the payment is processed, S3 reads the updated log, confirms the order, and writes the confirmation status along with a timestamp into the log.

Task:
Write the pseudocode to synchronize these services using semaphores to ensure:

- **Sequential Execution:** S1 completes before S2, and S2 completes before S3.
- **Mutual Exclusion:** Only one service can write to the shared log at a time.