**LAB REPORT**

**Lab Assignment 5: Process Synchronization**

**1. Objective**

* To implement **Peterson’s Solution** in C to achieve mutual exclusion between two processes.
* To simulate and resolve **race conditions** using **semaphores** in C.
* To understand how synchronization mechanisms help manage shared resources safely in concurrent programs.

**2. Theory**

**Process Synchronization** is a fundamental concept in operating systems used to coordinate access to shared resources by multiple processes or threads, preventing issues like data inconsistency or race conditions.

* **Race Condition**: Occurs when multiple processes access and manipulate shared data concurrently, and the final result depends on the sequence of execution.
* **Peterson’s Solution**: A classical software-based solution for achieving mutual exclusion between two processes. It uses two variables: a flag array to indicate interest in entering the critical section, and a turn variable to control which process can proceed.
* **Semaphores**: A synchronization primitive used to control access to shared resources. A **binary semaphore** acts like a lock (only 0 or 1), while **counting semaphores** can allow a limited number of accesses. In C, semaphores can be implemented using libraries like <semaphore.h> or manually with shared variables and wait() / signal() functions.

**3. Tools and Commands Used**

* **gcc** – GNU Compiler Collection for compiling C programs.
* **Terminal (Linux environment)** – For running and testing the programs.
* **C language libraries** – stdio.h, pthread.h, unistd.h, and optionally semaphore.h.

**4. Procedure**

1. **Peterson’s Solution Implementation**
   * Write a C program simulating two processes (or threads) competing for a critical section.
   * Use two shared variables: flag[2] and turn.
   * Implement entry and exit sections according to Peterson’s algorithm.
   * Use sleep/delay functions to simulate execution time and observe mutual exclusion in action.
2. **Race Condition with Semaphore**
   * Create a shared variable accessed by multiple threads.
   * Use a semaphore to protect the critical section where the variable is modified.
   * Initialize the semaphore to 1 (binary semaphore).
   * Use sem\_wait() (or manual wait) before entering the critical section, and sem\_post() after exiting.
   * Demonstrate that the race condition is avoided by synchronizing access.

**5. Program**

1. Write a program in C to simulate Peterson’s Solution.

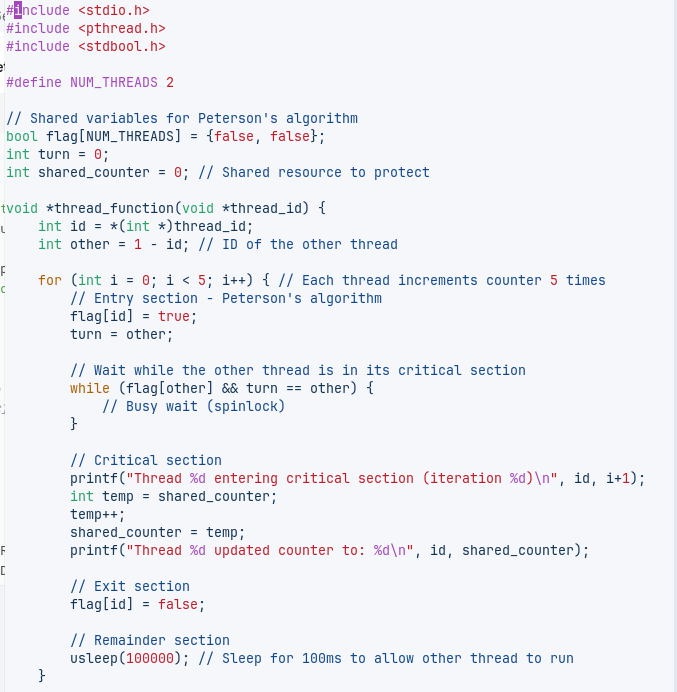
a.) Initiate the Directory.



b.) Create the C-Language file.



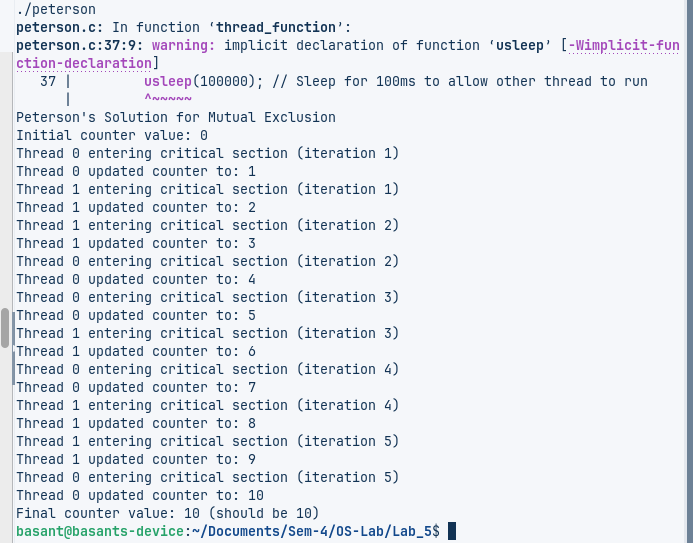
c.) Write the code of Peterson’s Solution.



d.) Compile the C-file



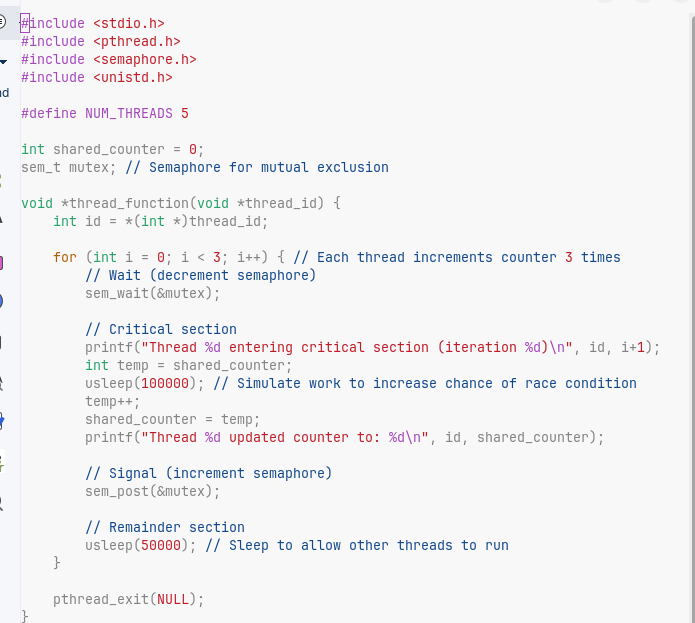
e.) Execute the File.

:

2. Write a program in C to solve the problem of race conditions using a Semaphore.

a.) Create the C-Language file.

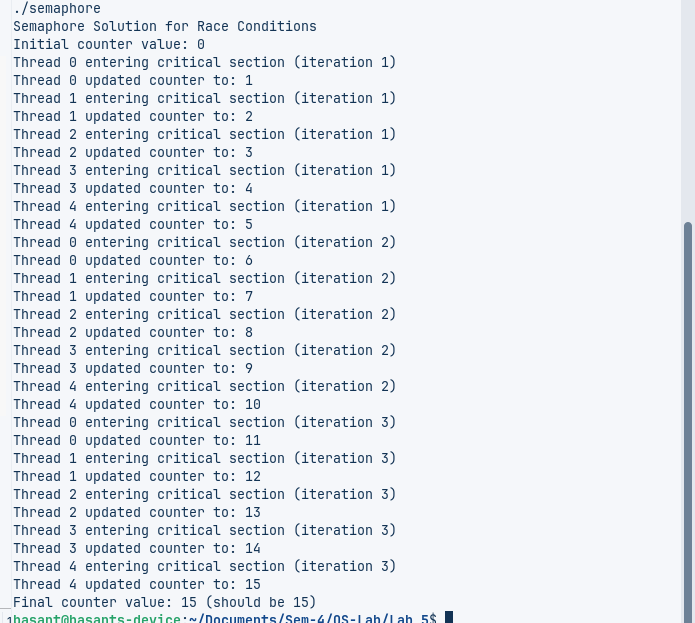


b.) Write the Semaphore Solution.  


c.) Compile the C-file.



d.) Execute the File.



e.) Determine the Race-Condition problem in Semaphore Solution.

In the initial semaphore-based solution, a race condition arises due to improper synchronization when multiple processes attempt to access a shared resource simultaneously. Although semaphores are designed to enforce mutual exclusion, incorrect usage—such as failing to enclose the entire critical section between wait() and signal() operations, or mismanaging semaphore values—can allow concurrent access, leading to inconsistent or corrupted data. This flaw highlights the need for careful placement and atomic handling of semaphore operations. The improved semaphore solution corrects these issues by ensuring strict mutual exclusion, thereby eliminating the race condition.

**6. Conclusion**

This lab focused on solving concurrency issues in multi-process environments using synchronization techniques. By implementing Peterson’s Solution, we understood how software-based mutual exclusion can be achieved. The use of semaphores further demonstrated an effective and scalable method to prevent race conditions and ensure safe access to shared resources. These concepts are vital for designing reliable and consistent concurrent programs.