Practical Session N°4

Monitor-Based Synchronization in C on Linux (POSIX)

Objective: To practice monitor-based synchronization by implementing safe thread coordination in C using POSIX mutexes and condition variables on Linux.

POSIX Mutex and Condition Variable Operations:

Primitive / Operation	Description	Purpose
pthread_mutex_t myMutex;	Declare a mutex variable.	Prepare a mutex for protecting shared resources.
pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;	Static initialization of a mutex.	Ready-to-use mutex with default attributes.
pthread_mutex_init (&myMutex, NULL);	Dynamic initialization of a mutex.	Initialize a mutex at runtime with optional attributes.
pthread_mutex_destroy (&myMutex);	Destroy a mutex.	Release resources associated with the mutex.
pthread_cond_t myCondition;	Declare a condition variable.	Prepare a condition variable for signaling.
pthread_cond_t myCondition = PTHREAD_COND_INITIALIZER;	Static initialization of a condition variable.	Ready-to-use condition variable with default attributes.
pthread_cond_init (&myCondition, NULL);	Dynamic initialization of a condition variable.	Initialize a condition variable at runtime with optional attributes.
pthread_cond_destroy (&myCondition);	Destroy a condition variable.	Release resources associated with the condition variable.
pthread_mutex_lock (&mutex)	Lock the mutex for exclusive access.	Protect shared data from concurrent access.
pthread_mutex_unlock (&mutex)	Unlock the mutex to allow other threads to acquire it.	End of critical section; signal that shared resources are available.
pthread_cond_wait (&conditionVariable, &mutex)	Wait on the condition variable; unlocks mutex while waiting and re-acquires it on wake.	Suspend a thread until a condition is true.
pthread_cond_signal (&conditionVariable)	Wake up one waiting thread.	Notify a thread that the condition may now be satisfied.
pthread_cond_broadcast (&conditionVariable)	Wake up all waiting threads.	Notify all waiting threads.

Important Note:
When using `pthread_cond_wait()`, always wrap it in a `while` loop to recheck the condition after waking. This ensures correctness in case of spurious wakeups or multiple waiting threads. Using `if` alone may work in simple scenarios, but it is unsafe and not recommended.

Exercise 1: (Positive Shared Counter)

Consider a shared integer counter that is initially zero. Two types of processes can access the counter: incrementers, which increase its value, and decrementers, which decrease its value. A decrementer may only execute if the counter is greater than zero; otherwise, it must wait until the counter becomes positive. Implement a C program under POSIX that synchronize the two processes using monitors.

Exercise 2

For each of the following problems (covered in **chapter 2**), write a **C program** using **monitor principle**, to simulate its behavior:

1- The **Rendezvous Problem:** one RDV and N processes

2- Producers/Consumers: assume the program ends after a fixed number of iterations (10 productions)

GOOD LUCK