Practical Session N°3

Thread Synchronization with Semaphores (POSIX)

Objective: Understand and implement thread synchronization using POSIX semaphores to control execution order and coordination between concurrent threads in a Linux environment.

Basic Concepts:

- The POSIX semaphore manipulation services are provided in the **<semaphore.h>** library.
- The semaphore type is represented by the **sem_t** data type.
- This library provides the following functions:

Function	Description
int sem_init(sem_t *sem, int pshared, unsigned int value)	Initializes a semaphore. **sem** is a pointer to the semaphore to be initialized; **value** is the initial value of the semaphore; **pshared** specifies whether the semaphore is local to the process **(pshared = 0)** or shared between processes **(pshared ≠ 0)**. Currently, Linux does not support shared semaphores.
int sem_wait(sem_t *sem)	Equivalent to the **P** operation. Always returns **0**.
int sem_post(sem_t *sem)	Equivalent to the **V** operation. Returns **0** on success or **-1** on failure.
int sem_trywait(sem_t *sem)	Decrements the semaphore value if it is greater than 0; otherwise, it returns an error. This is an atomic (non-blocking) operation..
int sem_getvalue (sem_t* nom, int * sval)	Retrieves the current value of a semaphore. Always returns 0.
int sem_destroy (sem_t* nom)	Destroys a semaphore. Returns -1 if there are still threads waiting on it.

Exercise 1:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <sched.h>

sem_t mutex;
void* display(void* name) {
    int i, j;
    for (i = 0; i < 20; i++) {
        sem_wait(&mutex);
        for (j = 0; j < 5; j++)
            printf("%s ", (char*)name);
        sched_yield();

        for (j = 0; j < 5; j++)
            printf("%s ", (char*)name);

        printf("\n");
        sem_post(&mutex);
    }
    return NULL;
}
int main(void) {
    pthread_t threadA, threadB;

    if (sem_init(&mutex, 0, 1) != 0) {
        perror("sem_init");
        return 1;
    }

    if (pthread_create(&threadA, NULL, display, "AA") != 0) {
        perror("pthread_create threadA");
        return 1;
    }
    if (pthread_create(&threadB, NULL, display, "BB") != 0) {
        perror("pthread_create threadB");
        return 1;
    }

    pthread_join(threadA, NULL);
    pthread_join(threadB, NULL);

    printf("Parent finished\n");

    sem_destroy(&mutex);

    return 0;
}
```

1. Study this program, then run it several times and observe the results obtained.
2. Remove some important parts of the code (e.g., sem_post, pthread_join, …) and execute it multiple times to understand the purpose of each function.

Exercise 2: Alternation Problem Between Processes

Create a C program under POSIX that starts three threads: **ThreadA**, **ThreadB**, and **ThreadC**.

Each thread infinitely displays a character on the screen:

- **ThreadA:** while(true) { print("A"); }
- **ThreadB:** while(true) { print("B"); }
- **ThreadC:** while(true) { print("C"); }

Synchronize the three threads so that the output on the screen appears as follows:

1. ABCABCABCABC...
2. ABACABACABAC...

Exercise 3

For each of the following problems (covered in **chapter 2**), write a **C program** using **POSIX semaphores**, 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)

- **a)** Program for one producer and one customer,
- **b)** then generalize the program for multiple producers and customers

3) Readers/Writers:

- **a)** We must fix the number of readers and writers. The number of accesses must also be fixed (to avoid an infinite loop), for example `ACCES = 4`. The operations of reading the database, modifying the data, and writing to the database are simulated using `sleep(1)` for one second.
- **b)** Chapter 2's solution causes writer starvation; modify the program to ensure fairness

4) The Dining Philosophers Problem

GOOD LUCK