Mohamed Boudiaf University – M'sila
Department of Computer Science
3rd year, Computer Science - ISIL

Practical Session N°1

Threads and Mutual Exclusion Manipulation in C Language under Linux

Objective: To learn how to create and manage threads in C under Linux using the **POSIX** API (**pthread.h**), and to understand the use of mutexes for **mutual exclusion** to prevent race conditions.

Basic Concepts of threads:

- **pthread_create:** creates a new thread (similar to *fork* for a process)
- **pthread_join:** waits for a thread to finish (similar to *wait* for processes)
- **pthread_exit:** terminates a thread (similar to *exit* for processes)

To compile and link, use the following command:

gcc -o your_program your_program.c -lpthread

Basic concepts of Mutex:

A mutex (short for mutual exclusion) ensures that only one thread at a time can enter a critical section of code where shared resources are modified. The Main POSIX Mutex Functions are:

- **pthread_mutex_init(&mutex, NULL):** Initializes a mutex before use.
- **pthread_mutex_lock(&mutex):** Locks the mutex, if it's already locked, the thread waits until it becomes available
- **pthread_mutex_unlock(&mutex):** Unlocks the mutex, allowing other threads to acquire it.
- **pthread_mutex_destroy(&mutex):** Destroys the mutex when it's no longer needed.

Exercise 1:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *thread_1(void *arg)
{
    printf("We are inside the thread.\n");
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread1;
    printf("Before creating the thread.\n");
    if(pthread_create(&thread1, NULL, thread_1, NULL) == -1) {
        perror("pthread_create");
        return EXIT_FAILURE;
    }
    pthread_join(thread1, NULL);
    printf("After the creation of the thread.\n");
    return 0;
}
```

1. Study this program.
2. Run this program several times and observe the results obtained.
3. Remove **pthread_join(thread1, NULL)** and see what happens.

Exercise 2

Write a C program that creates **two threads**; one thread prints the character *, and the other prints the character #.

1. Make each thread print its character **100 times**, then **10,000 times**. Run the program several times and observe how the output changes as the number of prints increases.
2. Modify the program so that **each thread completes all 10,000 prints without interruption** — that is, all the ***** are printed first, then all the **#**, or vice versa —**using synchronization mechanism (Mutex)**

Exercise 3 :

Write a C program that contains a variable nb initialized to 0, and that creates two threads. The first thread adds 1 to the variable nb one hundred times, and the second thread subtracts 1 from the variable nb one hundred times. Finally, display the value of the variable nb at the end.

1. Run this program several times and observe the results obtained.
2. Repeat the previous program, but this time use a **mutex** to protect the shared variable nb.

Exercise 4

Simulate four (**4)** cars crossing a single-lane intersection using threads in C. Represent each car as a thread, and have each car cross the intersection **5 times**. Use a shared variable **"cars_passed"** to track how many cars have crossed, and ensure that updates and prints are done safely so that only one car is in the intersection at a time. Each car should print messages when it enters and leaves the intersection. After all cars have crossed, print a final summary stating that all cars have passed. Use **"usleep()"** to simulate crossing times.

Good Luck