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

Solutions for 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.

Exercice 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.

SOLUTION

1. Study the program

- The program creates a new thread using the POSIX threads (**pthreads**) API.
- The function **thread_1()** is executed by the new thread.
- the main() function runs in the main thread.
- Main thread prints: Before creating the thread.
- New thread prints: We are inside the thread.
- After the new thread finishes, main thread prints: After the creation of the thread.

2. Run the program several times

Because of pthread_join, the main thread waits for the created thread to finish before continuing. The output order is consistent and predictable. Even if you run it many times, you should always get the same order.

3. Remove pthread_join(thread1, NULL)

- Without pthread_join, the main thread does not wait for the new thread to finish.
- When pthread_join is removed, the main thread can terminate before the new thread runs.
- Once main() returns, the entire process exits — terminating all threads immediately.
- That's why the message *"We are inside the thread."* might not appear or might appear in a different order.

Conclusion

pthread_join() ensures synchronization — the main thread waits for the child thread to complete. Without it, threads may be terminated prematurely, leading to unpredictable output.

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)**

SOLUTION

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

#define N 10000   // Change to 100 for the first test

void* print_star(void* arg) {
```

```
    int i;
    for (i = 0; i < N; i++) {
        printf("*");
        fflush(stdout);  // ensure immediate output
    }
    return NULL;
}

void* print_hash(void* arg) {
    int i;
    for (i = 0; i < N; i++) {
        printf("#");
        fflush(stdout);
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // Create the two threads
    pthread_create(&t1, NULL, print_star, NULL);
    pthread_create(&t2, NULL, print_hash, NULL);

    // Wait for both to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("\n End of program.\n");
    return 0;
}
```

1. What you'll observe:

- For N = 100, the output may look like: **##*#*###***##

- For N = 10,000, the output becomes a chaotic mixture of * and #.

This happens because both threads write to the terminal at the same time, and their outputs interleave unpredictably.

Each run will likely produce a **different pattern**, showing the **non-deterministic** nature of thread **scheduling**.

2. Modification (All * First, Then All # — With Synchronization)

- **Mutex (lock)** ensures **mutual exclusion** while printing.

- When a thread locks the mutex, it prints **all its characters consecutively** before unlocking.

- The other thread waits until the mutex is unlocked, preventing interleaving.

```
#include <stdio.h>
#include <pthread.h>

#define NUM_PRINTS 10000

pthread_mutex_t lock;  // mutex to control printing
```

```c
void *print_star(void *arg) {
    pthread_mutex_lock(&lock);           // lock to ensure consecutive printing
    int i;
    for (i = 0; i < NUM_PRINTS; i++) {
        printf("*");
    }
    pthread_mutex_unlock(&lock);
    return NULL;
}

void *print_hash(void *arg) {
    pthread_mutex_lock(&lock);           // lock to ensure consecutive printing
    int i;
    for (i = 0; i < NUM_PRINTS; i++) {
        printf("#");
    }
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_mutex_init(&lock, NULL);

    // Create threads
    pthread_create(&t1, NULL, print_star, NULL);
    pthread_create(&t2, NULL, print_hash, NULL);

    // Wait for threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("\nDone.\n");

    pthread_mutex_destroy(&lock);
    return 0;
}
```

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.

SOLUTION

1. program without synchronization

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

int nb = 0; // shared variable

void *add_func(void *arg)
{
    int i;
    for (i = 0; i < 100; i++)
        nb = nb + 1;
    return NULL;
}

void *sub_func(void *arg)
{
    int i;
    for (i = 0; i < 100; i++)
        nb = nb - 1;
    return NULL;
}

int main(void)
{
    pthread_t t1, t2;

    // Create two threads
    pthread_create(&t1, NULL, add_func, NULL);
    pthread_create(&t2, NULL, sub_func, NULL);

    // Wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final value of nb = %d\n", nb);
    return 0;
}
```

Observation: Theoretically, the final value should be 0, but due to concurrent access, the actual result can vary (e.g., 2, –1, etc.). This happens because both threads access and modify nb at the same time — ***mutual exclusion problem***.

2- program with Mutex

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

int nb = 0;                    // shared variable
pthread_mutex_t lock;          // mutex to protect nb

void *add_func(void *arg)
{
int i;
    for (i = 0; i < 100; i++) {
        pthread_mutex_lock(&lock); // lock before updating
        nb = nb + 1;
```

```
        pthread_mutex_unlock(&lock); // unlock after updating
    }
    return NULL;
}

void *sub_func(void *arg)
{
int i;
    for (i = 0; i < 100; i++) {
        pthread_mutex_lock(&lock); // lock before updating
        nb = nb - 1;
        pthread_mutex_unlock(&lock); // unlock after updating
    }
    return NULL;
}

int main(void)
{
    pthread_t t1, t2;

    pthread_mutex_init(&lock, NULL); // initialize mutex

    // Create threads
    pthread_create(&t1, NULL, add_func, NULL);
    pthread_create(&t2, NULL, sub_func, NULL);

    // Wait for threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final value of nb = %d\n", nb);

    pthread_mutex_destroy(&lock); // destroy mutex
    return 0;
}
```

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.

SOLUTION

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

```c
#define NUM_CARS 4
#define CROSSES_PER_CAR 5

// Shared variable
int cars_passed = 0;

// Mutex for synchronization
pthread_mutex_t lock;

// Function for each car (thread)
void* car(void* arg) {
    int car_id = *((int*)arg);

    for (int i = 0; i < CROSSES_PER_CAR; i++) {
        // Lock before entering intersection
        pthread_mutex_lock(&lock);

        printf("Car %d is ENTERING the intersection (cross #%d)\n", car_id, i + 1);

        // Simulate time in intersection
        usleep(200000); // 200 ms

        cars_passed++;
        printf("Car %d is LEAVING the intersection | Total cars passed: %d\n",
               car_id, cars_passed);

        // Unlock after leaving
        pthread_mutex_unlock(&lock);

        // Simulate time before trying again
        usleep(100000); // 100 ms
    }

    return NULL;
}

int main() {
    pthread_t threads[NUM_CARS];
    int car_ids[NUM_CARS];

    // Initialize mutex
    pthread_mutex_init(&lock, NULL);

    // Create threads (cars)
    for (int i = 0; i < NUM_CARS; i++) {
        car_ids[i] = i + 1;
        if (pthread_create(&threads[i], NULL, car, &car_ids[i]) != 0) {
            perror("Failed to create thread");
            exit(1);
        }
    }

    // Wait for all threads to finish
    for (int i = 0; i < NUM_CARS; i++) {
        pthread_join(threads[i], NULL);
    }
```

```
    // Destroy mutex
    pthread_mutex_destroy(&lock);

    printf("\nAll cars have passed the intersection! Total crossings: %d\n",
cars_passed);

    return 0;
}
```