OPERATING SYSTEMS

LAB 5:

PROF SHAHID FARID

THREADS USING PTHREAD

What is a Thread?

A **thread** is the smallest unit of execution in a program. It represents a single sequence of instructions that can be managed independently by the operating system. A thread exists within a **process**, and a process can contain one or more threads. Each thread within a process shares the same memory and resources (like files and data), but they can execute independently.

In simpler terms, a **thread** is like a lightweight process. While a **process** is a complete instance of a program running on a computer, a thread is a smaller, more efficient way to perform tasks within that process.

- Threads are used in operating systems to achieve **concurrency**, **parallelism**, **resource sharing**, and **better performance**.
- They are widely used in multi-tasking applications such as web servers, GUIs, and real-time systems to improve responsiveness and efficiency.

What is a Race Condition?

A **race condition** occurs when two or more threads or processes try to access and modify shared data at the same time, leading to unexpected or incorrect results. The outcome of the program depends on the timing or sequence of the execution of these threads, making the behavior unpredictable.

In simpler terms, a race condition happens when multiple threads are "racing" to access or modify a shared resource (like a variable), and the final outcome depends on which thread finishes first. Since the operating system can switch between threads at any time, the results of the operations may vary on each run, leading to **bugs** and **inconsistent program behavior**.

What is a Mutex?

A **mutex** (short for **Mutual Exclusion**) is a synchronization primitive used to control access to a shared resource in a multi-threaded environment. A mutex allows only **one thread** to access the critical section (the part of the code that accesses shared resources) at a time, ensuring that no two threads can modify the same data simultaneously.

Mutexes are essentially **locks** that a thread must acquire before it can enter the critical section and work with shared resources. Once the thread has completed its task, it **unlocks** the mutex, allowing other threads to proceed

This code will help you with the syntax of creating threads as well.

```
int x = 0;  // shared variable

void* increment(void* arg) {
   x = x + 1;  // each thread increments x by 1
   return NULL;
}

int main() {
   pthread_t thread1, thread2;
   pthread_create(&thread1, NULL, increment, NULL);
   pthread_create(&thread2, NULL, increment, NULL);

   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

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

BASIC SYNTAX

1. Creating and starting a thread:

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void *(*start_routine) (void *), void *arg);

Parameters:

- pthread_t *thread: A pointer to a pthread_t variable that will hold the thread identifier.
- const pthread_attr_t *attr: Attributes for the thread (set to NULL for default attributes).
- void *(*start_routine)(void *): The function the thread will execute. This function must return void* and accept a void* argument.
- void *arg: The argument to pass to the thread's function (can be NULL if no arguments are needed).

2. **Waiting for a Thread to Finish** (pthread_join)

int pthread_join(pthread_t thread, void **retval);

Parameters:

- pthread_t thread: The thread to wait for (identified by the pthread_t variable returned by pthread_create).
- void **retval: A pointer to the return value of the thread (can be NULL if you don't need the return value).

3. **Mutex Initialization** (pthread_mutex_init)

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

Parameters:

- pthread_mutex_t *mutex: A pointer to the mutex object that you want to initialize.
- const pthread_mutexattr_t *attr: Mutex attributes (set to NULL for default attributes).

EXAMPLE:

pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);

- **Locking a mutex:**

int pthread_mutex_lock(pthread_mutex_t *mutex);

- **Unlocking a mutex:**

int pthread_mutex_unlock(pthread_mutex_t *mutex);

4.Semaphore Initialization (sem_init)

int sem_init(sem_t *sem, int pshared, unsigned int value);

Parameters:

- sem_t *sem: A pointer to the semaphore to be initialized.
- int pshared: Set to 0 if the semaphore is used between threads of a process; non-zero if between processes.
- unsigned int value: The initial value of the semaphore.

Semaphore wait:

int sem_wait(sem_t *sem);

Semaphore signal:

int sem_post(sem_t *sem);

Example code

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

int global_var = 0;           // Shared global variable
pthread_mutex_t mutex;        // Mutex for synchronization

void* increment(void* arg) {
   for (int i = 0; i < 1000; i++) {
      pthread_mutex_lock(&mutex);  // Lock the mutex
      global_var++;                // Increment the shared variable
      pthread_mutex_unlock(&mutex);  // Unlock the mutex
   }
   return NULL;
}

int main() {
   pthread_t thread1, thread2;

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

   // Create two threads
   pthread_create(&thread1, NULL, increment, NULL);
   pthread_create(&thread2, NULL, increment, NULL);

   // Wait for both threads to finish
   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

   // Print the final value of the global variable
   printf("Final value of global_var: %d\n", global_var);

   // Destroy the mutex
   pthread_mutex_destroy(&mutex);

   return 0;
}
```

Tasks:

Task 1: Create a Simple Thread

Objective: Understand how to create and run a basic thread.

1. Create a C program where the **main thread** creates a **new thread**.
2. The new thread should print "Hello from the new thread!".
3. The main thread should print "Hello from the main thread!".
4. Ensure the main thread waits for the new thread to finish before it exits.

Task 2: Create Multiple Threads

Objective: Learn how to create multiple threads.

1. Modify the previous program to create **three threads**.
2. Each thread should print its unique **thread number** (e.g., "Thread 1", "Thread 2", etc.).
3. The main thread should wait for all three threads to finish before it exits.

Task 3: Passing Arguments to Threads

Objective: Pass data to threads and retrieve results.

1. Create a program where each thread receives a unique integer number as an argument.
2. Each thread should print the number it received.
3. Modify the program so that each thread calculates the **square** of the number it received and prints it.

Task 4: Simple Thread Synchronization

Objective: Use a global variable and observe a race condition.

1. Create a **global variable** initialized to 0.
2. Create two threads. Each thread should increment the global variable **1000 times**.
3. After both threads have finished, print the final value of the global variable.
4. What is wrong with the final value? (This demonstrates a **race condition**)