12.1-What are Threads?

Introduction to Threads and Multithreading

When we run an application on a computer device, such as a laptop or desktop, it operates within the **Operating System (OS)**. The OS, in turn, interacts with the hardware layer, which includes components like:

- **CPU (Central Processing Unit)**: Responsible for executing instructions and performing arithmetic and logical operations.
- **RAM (Random Access Memory)**: Stores data temporarily for quick access.
- **ROM (Read-Only Memory)**: ROM is a type of non-volatile memory, meaning it retains its contents even when the power is turned off. It is an essential component in computer systems and embedded devices for storing critical data and instructions also Often include firmwares which are needed while booting.

The **CPU** is primarily responsible for executing tasks, such as performing arithmetic operations. For example, if we write a program to add two numbers, the CPU executes this operation.

- **Understanding Time Sharing and Multi-Tasking**

Modern operating systems support **multi-tasking**, which means multiple tasks can run simultaneously. When multiple tasks are executed on the OS at the same time, they follow a principle called **time sharing**. Time sharing means that each task receives a specific amount of CPU time to run, process, and execute before switching to another task.

This time-sharing mechanism ensures that all tasks are processed efficiently, making the system more responsive.

What is a Thread?

The smallest and lightest unit of execution within a process is called a **Thread**.

A process constitutes a program that is currently executing. It denotes an active instance of a program executing on a computer. Upon executing a program, the operating system generates a process to oversee and execute it. Each process

functions autonomously and serves as the fundamental unit of resource management in contemporary operating systems.

A **thread** is a single sequence of execution within a program. It is sometimes referred to as a **lightweight process** because it shares the same resources as the main process but operates independently.

Each thread belongs to a single process, and in an operating system that supports **multithreading**, a process can have multiple threads running simultaneously.

Key Points About Threads:

- Threads perform multiple tasks by dividing them into smaller units, allowing for smooth program execution.
- A thread is a single, sequential stream within a process.
- Each thread has its own CPU state and stack, but they share the address space of the process and the environment.
- Threads are useful in systems with multiple CPUs because they can run independently on different cores. However, on a single-core system, threads need to context switch, which can cause overhead.

Why Do We Need Threads?

Threads are essential for the following reasons:

- **Parallel execution**: Threads allow a process to run multiple tasks simultaneously, improving the performance and responsiveness of an application.
- **Shared Data**: Threads share common data and resources, reducing the need for complex inter-process communication.
- **Reduced Resource Consumption**: Threads are lightweight compared to full processes, as they share the same memory and resources.
- **Efficient Utilization**: Threads make efficient use of system resources, as multiple threads can share the same memory space, making efficient use of system resources.

Example: Multi-Threading in Real Life

In a web browser, each tab can be considered a separate thread. Similarly, in an application like MS Word, multiple threads may be running simultaneously:

- **Thread 1**: Formatting the text.
- **Thread 2**: Processing user inputs.
- **Thread 3**: Checking spelling and grammar in real-time.

Each of these threads runs independently, making the application responsive and efficient.

What is Multi-Threading?

Multi-threading is a technique that allows a process to be divided into multiple threads, enabling parallelism and efficient utilization of CPU resources. This technique helps in achieving **concurrent execution** of multiple parts of a program.

Key Features of Multi-Threading:

- **Parallelism**: Multiple threads can run simultaneously, allowing tasks to be performed faster.
- **Resource Sharing**: All threads within a process share the same memory and resources.
- **Improved Responsiveness**: Multi-threading makes applications more responsive, especially when performing tasks like file I/O, network communication, and user interactions.

Example: Multi-Threading in a Video Game

Consider a video game like a soccer match. In the game:

- One thread handles the **player's movements**.
- Another thread manages the **background audio**.
- A separate thread controls the **graphics rendering**.

- Additional threads handle the **audience cheering** and **ball physics**.

Each of these tasks operates as a separate thread, allowing the game to run smoothly and respond quickly to user inputs.

By using threads, the operating system achieves a **multi-tasking environment**, where multiple tasks are executed efficiently without noticeable lag.

Conclusion

Threads and multi-threading are essential concepts for improving application performance and responsiveness. They allow processes to perform multiple tasks in parallel by breaking them into smaller, independent units. By understanding and implementing threads effectively, developers can build applications that are fast, efficient, and capable of handling complex operations simultaneously.