Threads

Lecture # 06

By
Ms. MARYAM ARSHAD

Threads

- A thread is a single sequence stream within a process. Threads are also called **lightweight processes** as they possess some of the properties of processes. Each thread belongs to exactly one process.
- In an operating system that supports multithreading, the process can consist of many threads. But threads can be effective only if the CPU is more than 1 otherwise two threads have to context switch for that single CPU.
- All threads belonging to the same process share – code section, data section, and OS resources (e.g. open files and signals)
- But each thread has its own (thread control block) – thread ID, program counter, register set, and a stack
- Any operating system process can execute a thread. we can say that single process can have multiple threads.

Why Do We Need Thread?

- Threads run in parallel improving the application performance. Each such thread has its own CPU state and stack, but they share the address space of the process and the environment.
- Threads can share common data so they do not need to use inter-process communication. Like the processes, threads also have states like ready, executing, blocked, etc.
- Priority can be assigned to the threads just like the process, and the highest priority thread is scheduled first.
- Each thread has its own Thread Control Block TCB. Like the process, a context switch occurs for the thread, and register contents are saved in (TCB). As threads share the same address space and resources, synchronization is also required for the various activities of the thread.

Components of Threads

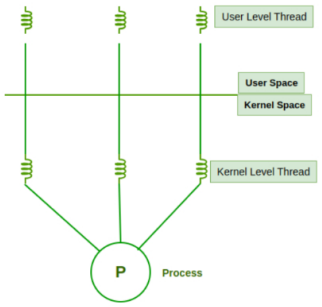
These are the basic components of the Operating System.

- **Stack Space:** Stores local variables, function calls, and return addresses specific to the thread.
- **Register Set:** Hold temporary data and intermediate results for the thread's execution.
- **Program Counter:** Tracks the current instruction being executed by the thread.

Types of Thread in Operating System

Threads are of two types. These are described below.

- User Level Thread
- Kernel Level Thread



User Level Thread

- User Level Thread is a type of thread that is not created using system calls. The kernel has no work in the management of user-level threads. User-level threads can be easily implemented by the user. In case when user-level threads are single-handed processes, kernel-level thread manages them. Let's look at the advantages and disadvantages of User-Level Thread.

Advantages of User-Level Threads

- Implementation of the User-Level Thread is easier than Kernel Level Thread.
- Context Switch Time is less in User Level Thread.
- User-Level Thread is more efficient than Kernel-Level Thread.
- Because of the presence of only Program Counter, Register Set, and Stack Space, it has a simple representation.

Disadvantages of User-Level Threads

- The operating system is unaware of user-level threads, so kernel-level optimizations, like load balancing across CPUs, are not utilized.
- If a user-level thread makes a blocking system call, the entire process (and all its threads) is blocked, reducing efficiency.
- User-level thread scheduling is managed by the application, which can become complex and may not be as optimized as kernel-level scheduling.

Kernel Level Threads

- A kernel Level Thread is a type of thread that can recognize the Operating system easily. Kernel Level Threads has its own thread table where it keeps track of the system. The operating System Kernel helps in managing threads. Kernel Threads have somehow longer context switching time. Kernel helps in the management of threads.

Advantages of Kernel-Level Threads

- Kernel-level threads can run on multiple processors or cores simultaneously, enabling better utilization of multicore systems.
- The kernel is aware of all threads, allowing it to manage and schedule them effectively across available resources.
- Applications that block frequency are to be handled by the Kernel-Level Threads.
- The kernel can distribute threads across CPUs, ensuring optimal load balancing and system performance.

Disadvantages of Kernel-Level threads

- Context switching between kernel-level threads is slower compared to user-level threads because it requires mode switching between user and kernel space.
- Managing kernel-level threads involves frequent system calls and kernel interactions, leading to increased CPU overhead.
- A large number of threads may overload the kernel scheduler, leading to potential performance degradation in systems with many threads.

- Implementation of this type of thread is a little more complex than a user-level thread.

Difference Between Process and Thread

- The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces. Threads are not independent of one another like processes are, and as a result, threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like a process, a thread has its own program counter (PC), register set, and stack space.

What is Multi-Threading?

- A thread is also known as a lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc. More advantages of multithreading are discussed below.
- Multithreading is a technique used in operating systems to improve the performance and responsiveness of computer systems. Multithreading allows multiple threads (i.e., lightweight processes) to share the same resources of a single process, such as the CPU, memory, and I/O devices.

Multithreaded Programming

- **Multithreaded Programming** is a programming technique that allows multiple threads to execute concurrently within a single process. A **thread** is the smallest unit of execution, and multithreading enables a program to perform multiple tasks at the same time, improving efficiency and responsiveness.

Benefits:

- **Improved Performance** – Utilizes CPU resources effectively.
- **Responsiveness** – UI applications remain responsive while handling background tasks.
- **Efficient Resource Sharing** – Threads share memory space, reducing overhead.
- **Better Scalability** – Suitable for multi-core processors.

Multithreading

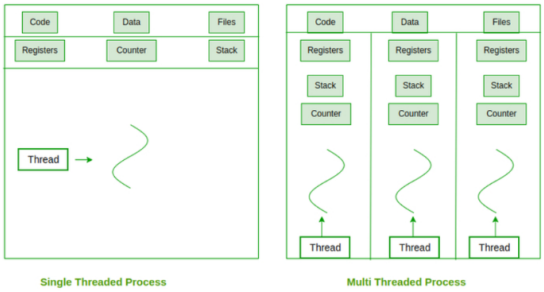
Challenges of Multithreading:

- **Complex Debugging** – Harder to trace and fix concurrency issues.
- **Race Conditions** – Multiple threads modifying shared data simultaneously.
- **Deadlocks** – Threads waiting indefinitely for resources.
- **Increased Overhead** – Managing threads consumes system resources.

How multithreading created?

- **Multithreading can be done without OS support**, as seen in Java's multithreading model. In Java, threads are implemented using the Java Virtual Machine (JVM), which provides its own thread management. These threads, also called user-level threads, are managed independently of the underlying operating system.
- Application itself manages the creation, scheduling, and execution of threads without relying on the operating system's kernel. The application contains a threading library that handles thread creation, scheduling, and context switching. The operating system is unaware of User-Level threads and treats the entire process as a single-threaded entity.

Difference between single & multithreaded process



Single Threaded Process Multi Threaded Process

Types of Multithreading Models:

1. Many-to-One Model

- **Definition**: Multiple user threads are mapped to a single kernel thread.
- **Implementation**: Thread management is done in **user space** (without kernel intervention).
- **Pros**:
  Fast thread switching (no kernel involvement).
  Efficient in single-processor systems.
- **Cons**:
  If one thread blocks, all threads in the process are blocked.
  Cannot utilize multi-core processors effectively.

Types of Multithreading Models:

2. One-to-One Model

- **Definition**: Each user thread is mapped to a separate kernel thread.
- **Implementation**: The OS manages threads at the kernel level.
- **Pros**:
  True parallel execution on multi-core systems.
  If one thread blocks, others keep running.
- **Cons**:
  Creating too many threads can cause high overhead.
  More system resource consumption.

3. Many-to-Many Model

- **Definition**: Multiple user threads are mapped to multiple kernel threads, allowing a flexible relationship.
- **Implementation**: The OS schedules kernel threads dynamically.
- **Pros**:
  Combines the benefits of both previous models.
  Can use multiple cores efficiently.
  Avoids excessive kernel thread creation.
- **Cons**:
  Complex implementation and management.

Example:

Example 2: A Task with 10% Sequential Execution ($S = 0.1$) on 8 Cores

Given:

- 10% of the program is sequential ($S = 0.1$).
- The rest (90%) can be parallelized ($1 - S = 0.9$).
- Running on **8 processors** ($N = 8$).

Speedup Calculation:

$$\text{Speedup} \leq \frac{1}{0.1 + \frac{0.9}{8}}$$

$$\text{Speedup} \leq \frac{1}{0.1 + 0.1125} = \frac{1}{0.2125} \approx 4.7$$

Even with **8 cores**, we get only **4.7× speedup** instead of 8×.

Benefits of Thread in Operating System

- **Responsiveness:** If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
- **Faster context switch:** Context switch time between threads is lower compared to the process context switch. Process context switching requires more overhead from the CPU.
- **Effective utilization of multiprocessor system:** If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors. This will make process execution faster.
- **Resource sharing:** Resources like code, data, and files can be shared among all threads within a process. Note: Stacks and registers can't be shared among the threads. Each thread has its own stack and registers.
- **Communication:** Communication between multiple threads is easier, as the threads share a common address space. while in the process we have to follow some specific communication techniques for communication between the two processes.
- **Enhanced throughput of the system:** If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.