Understanding Threads & Processes

Introduction

Modern computing relies heavily on concurrency and multitasking to optimize system performance. While most people associate concurrency with multi-core processors, threads play a crucial role even on single-core CPUs. To understand why, we need to explore the concepts of processes, CPU scheduling, blocking operations, and how threads provide an efficient alternative to traditional multi-process approaches.

What is a Process?

A **process** is an instance of a running program. When you execute an application, the operating system creates a process to manage its execution. Each process has its own:

- **Process ID (PID):** A unique identifier.
- **Memory Space:** Code, data, heap, and stack.
- **Program Counter:** Keeps track of the next instruction.
- **Registers:** Store the process's execution state.
- **Open File Handles and I/O Resources.**

How the CPU Handles Multiple Processes

A single-core CPU can only execute **one instruction at a time**. However, modern operating systems create an illusion of simultaneous execution through **context switching**. The scheduler rapidly switches the CPU between different processes, allowing each process to make progress over time. This is known as **concurrency**.

Context Switching

Context switching is the process of storing the current state of a running process or thread and restoring the state of another process or thread so that execution can continue.

- When a process or thread is interrupted, the CPU saves its state (registers, program counter, etc.).
- The scheduler then loads another process/thread and restores its saved state.
- This allows the CPU to handle multiple tasks efficiently, even on a single-core processor.
- However, frequent context switches introduce **overhead**, leading to reduced performance.

Why Do Processes Block for I/O?

Not all operations require continuous CPU execution. Some operations, like reading a file from disk or waiting for user input, cause a process to **block**, meaning the CPU cannot execute any instructions until the operation is complete.

Examples of Blocking Operations:

1. **Disk I/O:** The CPU waits while the disk retrieves data.
2. **Network Communication:** A web server waits for a client request to be received.
3. **Peripheral Interaction:** A print job blocks until the printer is available.

To prevent the CPU from being idle, the operating system **switches execution to another process**, allowing useful work to continue while waiting for I/O operations to complete.

The Server Example: A Problem of Blocking Requests

Consider a web server that serves profile images when a client sends a request:

1. The server listens for requests on a port.
2. When a request arrives, it reads the image from disk.

3. Once the image is in memory, it sends it back to the client.

The Problem:

- The first request gets accepted and processed.
- Other requests must **wait** until the first request completes.
- The CPU sits idle during disk read operations, wasting valuable time.

This is known as the **blocking problem**, where useful computational time is wasted due to sequential execution.

Traditional Solution: Creating a New Process for Each Request

One approach to solving this problem is the **multi-process model**:

1. The main process (listener) accepts a request.
2. It spawns a **new process** to handle the request.
3. The listener remains free to accept new requests.

Drawbacks of Creating a New Process for Each Request

1. **High Memory Usage:** Each process has its own memory space, requiring significant system resources.
2. **Expensive Process Creation:** Spawning a new process is slow due to memory allocation and register setup.
3. **Inter-Process Communication (IPC) Overhead:** If processes need to share data, IPC mechanisms (pipes, shared memory, message queues) add complexity.
4. **Limited Scalability:** For thousands of concurrent requests, the system quickly runs out of resources.

A Better Approach: Using Threads Instead of Processes

Instead of creating a new process for each request, we can use **threads**, which allow concurrent execution within the same process.

What are Threads?

A **thread** is a lightweight unit of execution within a process. Unlike processes, threads share the same memory space but maintain their own execution state.

Each thread has:

- Its own **program counter** (tracks execution progress).
- Its own **CPU registers** (stores temporary data).
- Its own **stack** (for local function calls and variables).

However, threads within the same process share:

- **Code section** (executable program instructions).
- **Global variables and heap memory.**
- **File descriptors and I/O handles.**

Threads vs. Processes

Feature	Process	Thread
Memory Space	Separate for each process	Shared within the process
Creation Time	High (expensive)	Low (lightweight)
Communication	Uses IPC (complex)	Direct memory access (efficient)
Context Switching	Slower (more overhead)	Faster (low overhead)
Scalability	Limited by resource usage	More scalable

How Threads Solve the Blocking Problem

With a **multi-threaded server**:

1. The listener process accepts a request.
2. Instead of spawning a new process, it creates a **new thread**.
3. While one thread waits for disk I/O, another thread can process a different request.

This ensures the **CPU is utilized efficiently**, reducing idle time and improving response times.

Memory Management in Threads

Since threads share memory, they must manage access carefully:

- Each thread has its own **stack**, preventing data corruption.
- Shared **heap memory** requires synchronization to avoid race conditions.
- **Mutexes and semaphores** help prevent multiple threads from modifying shared data simultaneously.

OS-Level Implementation of Threads

Most operating systems implement threads within the **Process Control Block (PCB):**

- Linux represents both processes and threads using the **task structure**.
- Each process starts with a **main thread**; additional threads are created dynamically.
- The OS **schedules threads** instead of entire processes, improving efficiency.

Threads vs. Functions: A Key Distinction

- A **function** is a reusable block of code but **does not execute independently**.
- A **thread** is an **independent execution unit** that runs in parallel with other threads.
- Multiple threads can execute the **same function** simultaneously without conflicts.

Why Threads Are More Efficient Than Processes

1. **Faster Context Switching:** Switching between threads is much faster than switching between processes.
2. **Lower Memory Usage:** Threads avoid the overhead of duplicating memory spaces.
3. **Better Resource Sharing:** Threads can directly access shared data without IPC mechanisms.
4. **Higher Scalability:** A server can handle thousands of connections without excessive memory consumption.

Conclusion

Threads offer a powerful alternative to traditional **multi-process concurrency**, especially on **single-core processors**. They enable **efficient CPU utilization** by preventing idle time during I/O operations, reduce **memory overhead**, and allow **faster context switching**. While multi-threading introduces complexity, such as **synchronization challenges**, it remains the preferred method for building **scalable, high-performance applications**.