Understanding Concurrency

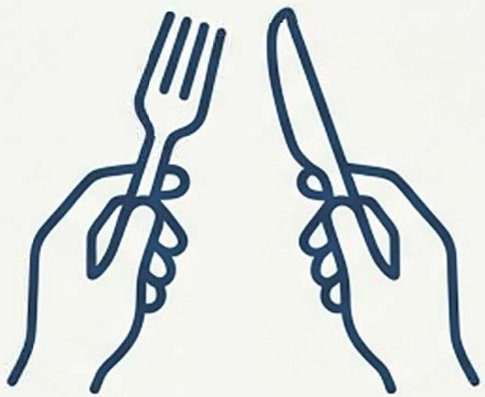
Dr. Talgat Turanbekuly

Lecture-1-1

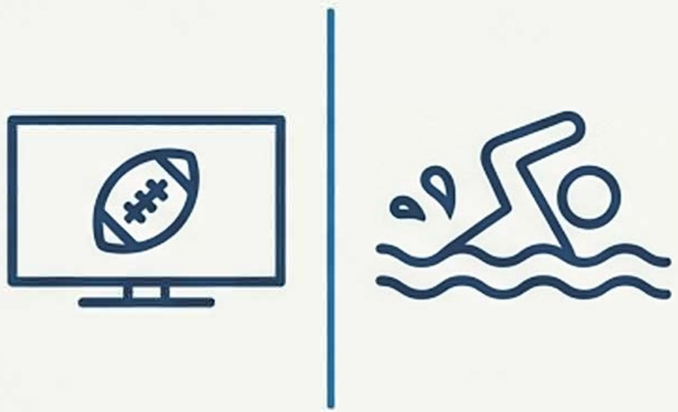
Concurrency is a natural part of life.

At its most basic level, concurrency is about two or more separate activities happening at the same time. We experience it constantly without a second thought.

Walking and talking simultaneously.



Using both hands to perform different actions.



One person watching football while another goes swimming.

In computers, concurrency means a single system performing multiple independent activities in parallel.

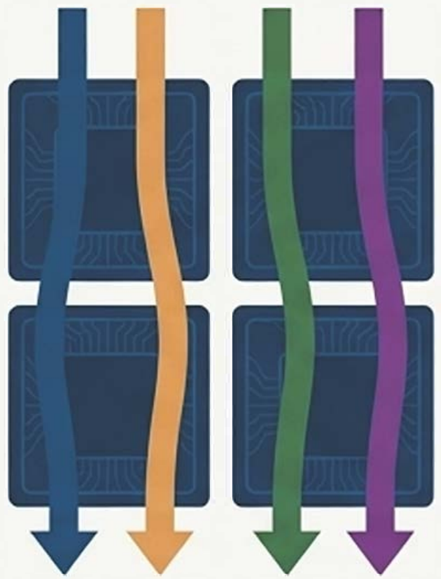
This isn't a new idea. Multitasking operating systems have long allowed a computer to run multiple applications at once. What has changed is the rise of multicore processors, which makes *true* hardware parallelism commonplace on everyday machines.

Past: Single Core CPU

Present: Multicore CPU

Computers achieve concurrency in two fundamental ways.

Hardware Concurrency (The Reality)

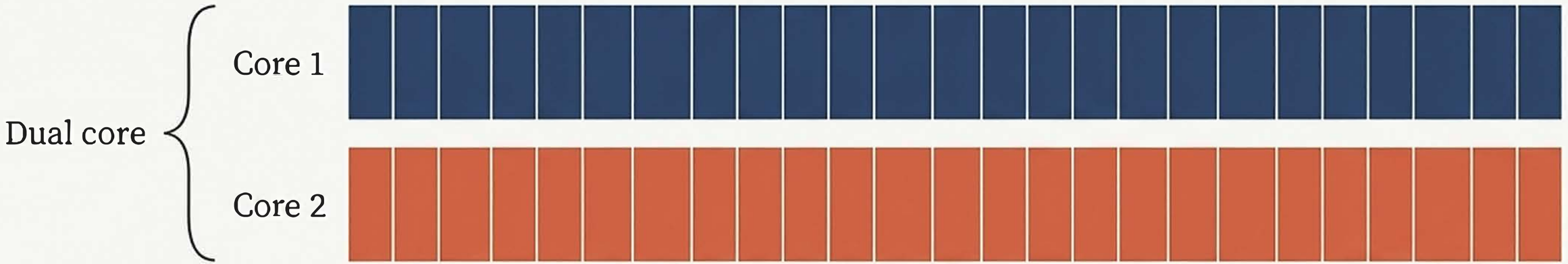


The system has multiple processor cores, genuinely running more than one task in parallel at the exact same time. The number of “hardware threads” determines how many tasks can run concurrently.

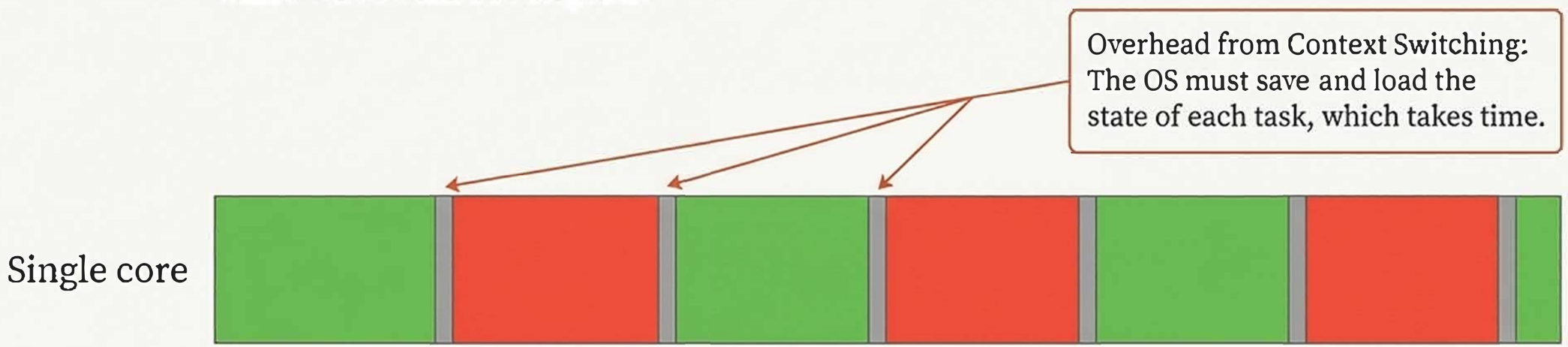
Task Switching (The Illusion)

A single processor core rapidly switches between tasks many times per second. By doing a small piece of each task in sequence, it creates the *appearance* of tasks happening at the same time.

Visualizing how two tasks execute on different hardware.



With two cores, each task runs on its own dedicated hardware in parallel.



Even with multiple cores, task switching is essential for modern systems.

A typical desktop computer has far more tasks running—often hundreds—than it has available cores. Task switching allows the system to juggle all background processes and user applications, ensuring everything runs smoothly.

A modern system juggling four distinct tasks (represented by colors) across two available cores. This combination of hardware concurrency and task switching is standard.

To understand software approaches to concurrency, imagine organizing a pair of developers.

The way we structure our code to handle multiple tasks mirrors two basic ways of structuring a work environment. We'll explore these two models: one focused on isolation, the other on collaboration.

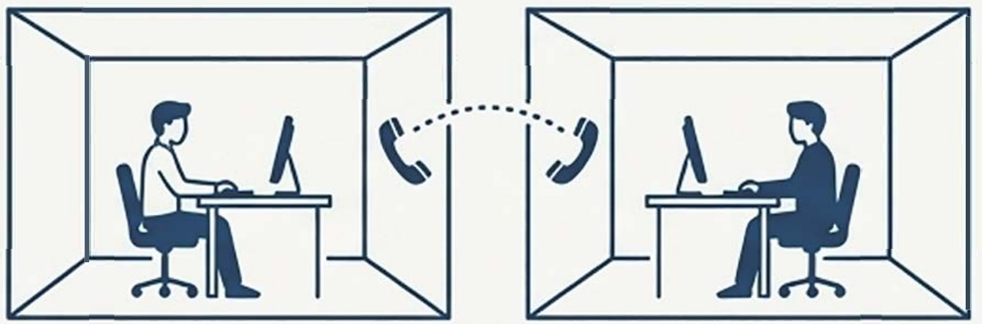
Separate Offices

Shared Office

Approach 1: Multiple Processes are like developers in separate offices.

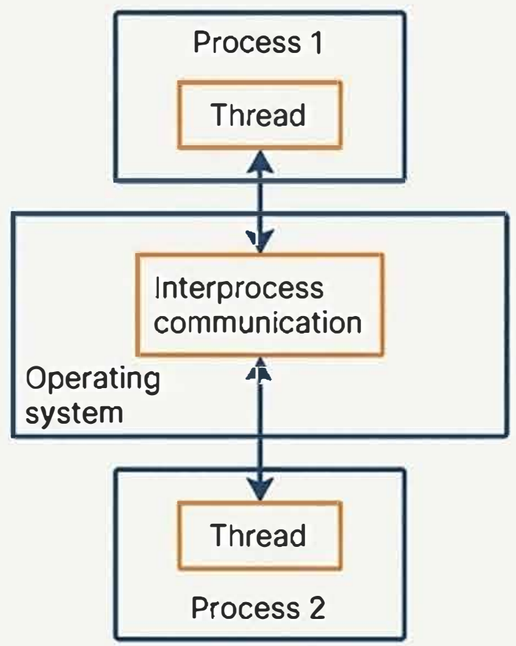
Analogy

Each developer works in their own secure office. They are protected from each other but communication is formal and slow.



Technical Explanation

- **How it works:** Each process is a separate application with its own protected memory space.
- **Communication:** Slow and complex. The OS manages communication through channels like signals, sockets, files, or pipes.
- **Pros:** High degree of protection (one process can't easily crash another), makes it easier to write 'safe' code, can be run on different machines across a network.
- **Cons:** High overhead to start and manage processes; communication between them is slow.



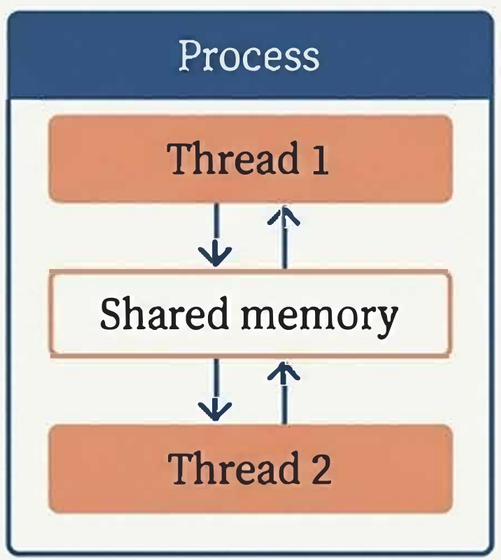
Approach 2: Multiple Threads are like developers in a shared office.

Analogy

Developers work in the same room. They can talk freely and share a whiteboard and bookshelf, making collaboration fast, but they can also distract each other other or misplace shared resources.

Technical Explanation

- **How it works:** All threads exist within a single process and share the same memory address space.
- **Communication:** Extremely fast and direct, by reading and writing to shared memory.
- **Pros:** Low overhead to create and manage; fast communication; efficient resource use.
- **Cons:** Dangerous if not managed carefully. One thread can corrupt data used by another. Requires careful synchronization to prevent problems like, “Where’s the reference manual gone now?”

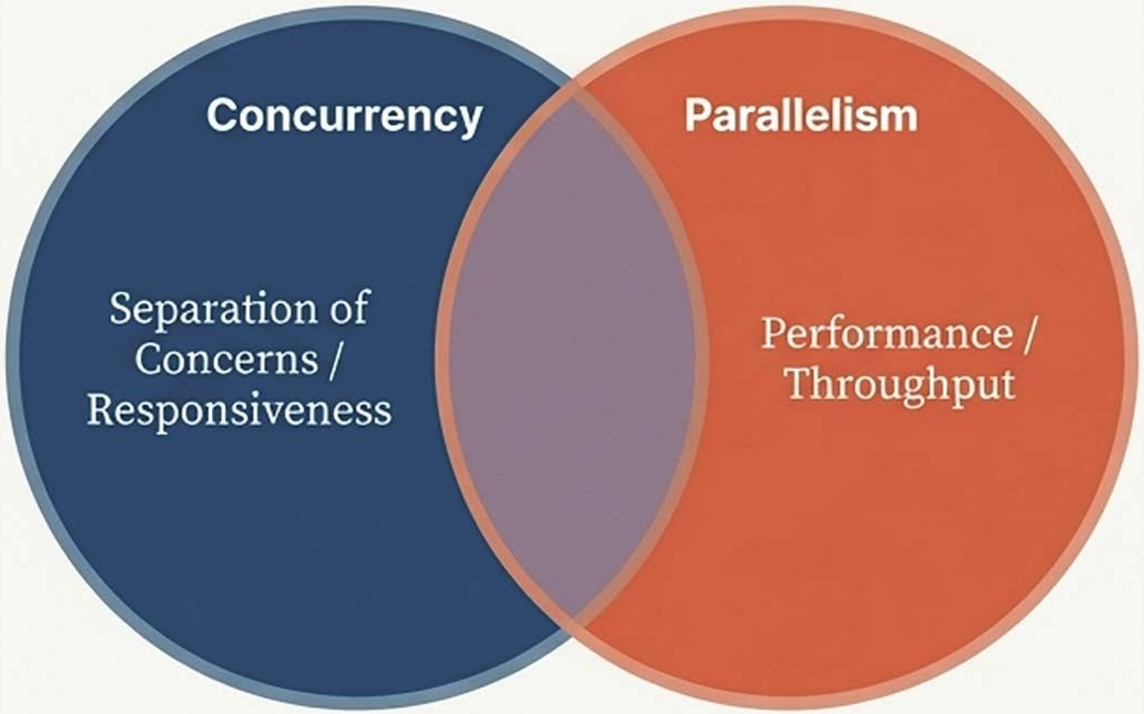


Choosing Your Concurrency Model: A Summary of the Trade-Offs

	Multiple Processes	Multiple Threads
Memory Space	 Separate address spaces.	 Single shared address space.
Communication Speed	 Slow, managed by the OS.	 Very fast, via shared memory.
Overhead	 High. Slow to create and manage.	 Low. Fast and lightweight.
Safety & Protection	 High. OS provides strong protection.	 Low. Requires careful programmer synchronization.

Concurrency vs. Parallelism: The difference is intent.

The terms are related and often used interchangeably, but they have different focuses. Both involve running multiple tasks simultaneously.



Focus: Managing multiple tasks at once.

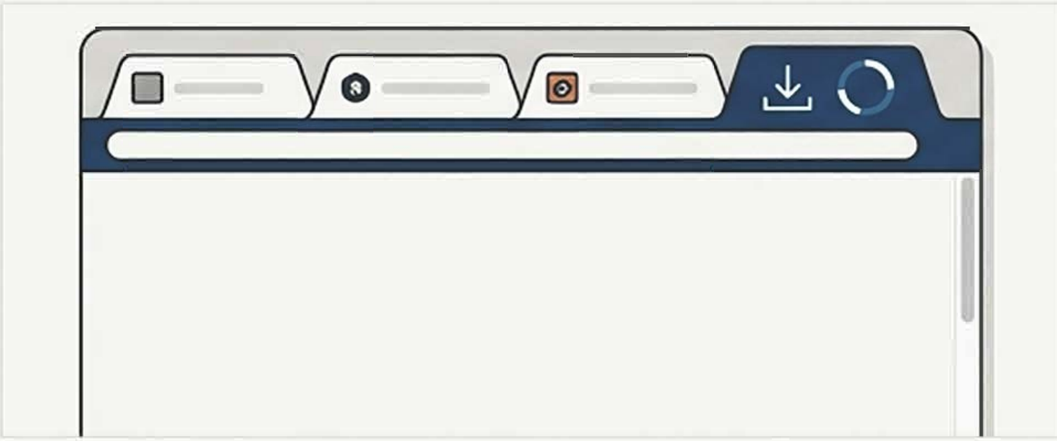
Intent: Separation of concerns or improving responsiveness. For example, keeping a user interface from freezing while a background task runs.

Focus: Running multiple parts of a task simultaneously to finish faster.

Intent: Performance. Taking advantage of hardware to increase the throughput of bulk data processing.

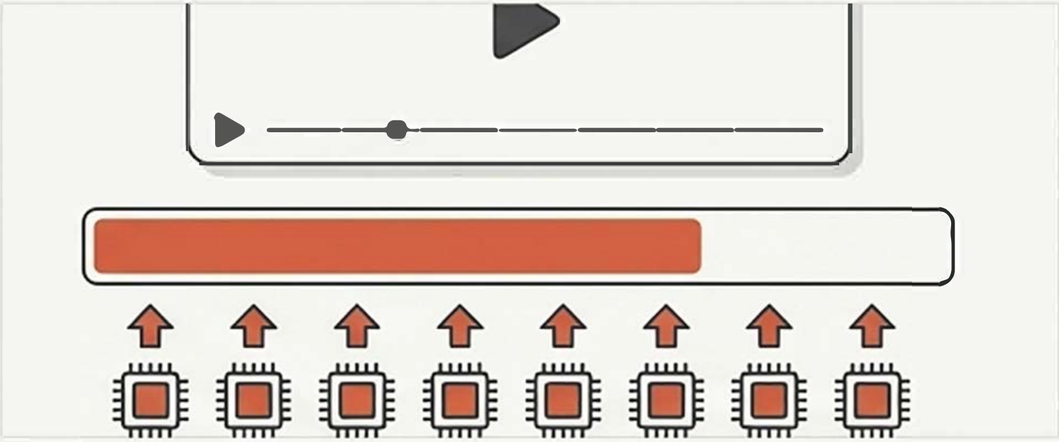
Concurrency is about structure, Parallelism is about execution.

A Modern Web Browser



Concurrency in Action

My browser uses **concurrency** to manage separate tasks: I can scroll through one tab while a large file downloads in another.



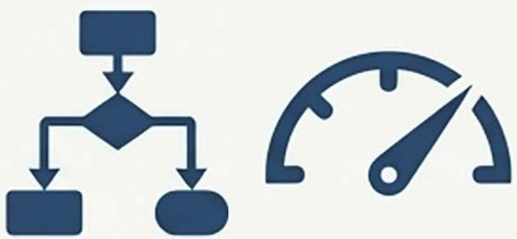
Parallelism in Action

My browser uses **parallelism** to execute a single, heavy task faster: it uses all 8 of my CPU cores to render a complex video.

Key Takeaways

Concurrency in hardware is achieved through **true parallelism** on multiple cores or the **illusion of parallelism** via rapid task-switching.

In software, concurrency is implemented using isolated **Processes** (safe but slow communication) or shared-memory **Threads** (fast but requires careful synchronization).



Concurrency is about structuring an application to manage multiple tasks, while **Parallelism** is about using hardware to execute tasks faster for performance.

A Foundation for Mor Modern Software

Understanding the models and trade-offs of concurrency is no longer optional. It is fundamental to designing the efficient, responsive, and powerful applications that users expect today. These core concepts are the starting point for tackling more advanced topics like memory models, performance tuning, and designing concurrent code.