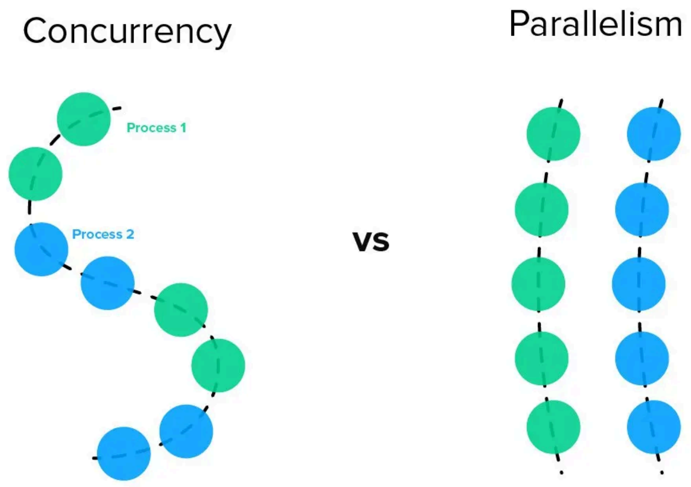
⚙ The Nature of Concurrency

> “Concurrency isn’t chaos — it’s *managed unpredictability*.”

- Multiple tasks **appear** to run at once.
- Sometimes they **actually** do (parallelism).
- Always about **coordination**, **resource sharing**, and **timing**.



🧩 Parallel vs. Concurrent Execution

Concept	Meaning	Example
Parallel	Tasks truly run at the same time on multiple cores or CPUs.	Two threads crunching math on different cores.
Concurrent	Tasks *overlap* in time — they make progress independently, but not necessarily simultaneously.	A web server switching rapidly among requests.

✅ **Parallel** = *simultaneous*

✅ **Concurrent** = *interleaved*

🚫 Confusing them = 3 AM debugging sadness.

🧠 It's Stupid Analogy Time

- “**Parallelism** is like having multiple chefs cooking separate dishes.”

- “**Concurrency** is one chef juggling several dishes without burning anything.”

Both aim to get dinner done, but the mental model and approach differs.

🧮 Shared Resources & Nondeterminism

- When two or more processes share:
  - CPU time
  - Memory (shared variables)
  - I/O devices
- Execution order becomes **nondeterministic**:

> “Who gets the resource first?”
> “What happens if two threads update the same value?”

🧮 Shared Resources & Nondeterminism

Nondeterminism leads to:

- **Race conditions**
- **Deadlocks**
- **Inconsistent states**
- **Occasional student tears**

💥 Example: A Classic Race Condition

```
// Two threads both incrementing the same variable:
counter = counter + 1;
```

If interleaved like this:

Thread A	Thread B
Reads counter (5)	Reads counter (5)
Adds 1 → 6	Adds 1 → 6
Writes 6	Writes 6

```
Expected: 7
Actual  : 6 → You just lost an increment
```

🧰 Fixing It: Synchronization Primitives

Tool	What It Does	Example
Mutex	Mutual exclusion — one thread at a time.	`pthread_mutex_lock()`
Semaphore	Counter-based lock (n threads allowed).	Producer-consumer queues
Monitor	Combines locks and condition variables.	Java `synchronized` block
Atomic Ops	Hardware-level single-step updates.	`std::atomic<int>`

Each protects shared resources — at the cost of some performance.

🌐 Real-World Concurrency

Domain	Example	Why It Matters
Web servers	Handling thousands of simultaneous client requests.	Keeps user experiences smooth and scalable.
Operating systems	Scheduling processes and threads.	Maximizes CPU utilization and fairness.
Databases	Multiple users reading/writing simultaneously.	Ensures consistency with transactions and locks.
IoT / Robotics	Sensors and actuators acting in parallel.	Real-time response and coordination.

⏱️ Schedulers and Interleaving

The OS scheduler decides **who runs next**:

- CPU-bound vs I/O-bound trade-offs.
- Context switching introduces overhead.
- Preemption ensures fairness.

You can simulate concurrency *even with one core* by slicing time:

```
T1:  ████----████----
T2:  ----████----████
```

👆 “Looks and feels parallel” — but it’s actually concurrent.

🧩 Determinism vs. Nondeterminism

Type	Description	Example
Deterministic	Same input → same output every time.	Pure functions, single-threaded code.
Nondeterministic	Output depends on timing/order of events.	Multithreaded updates to shared memory.

> Testing concurrent programs feels like herding cats — sometimes the bug just sleeps.

🧠 Concurrency: The OS’s Middle Name

- The OS is the **referee**:
  - Prevents conflicts (mutexes, semaphores).
  - Allocates time (scheduler).
  - Keeps everyone fair and fed (resource management).
- But even the OS itself has to play by concurrency’s rules.

🎓 Key Takeaways

- Concurrency ≠ Parallelism — but they often coexist.
- Shared state introduces nondeterminism.
- Synchronization primitives tame chaos.
- The OS is the ultimate concurrency controller.

🏁 Next Up: Synchronization Strategies

> “Locks, semaphores, and monitors: controlling critical access to all threads across the globe.”

We’ll explore:

- Critical sections
- Deadlocks & starvation
- Real-world scheduling examples