What is Concurrency?

Definition (Precise)

Concurrency is the ability of a program to handle multiple tasks by structuring it into independently executing components that can make progress without waiting for each other to complete.

Concurrency is NOT parallelism. (We'll clarify this crucial distinction in the next document.)

Concurrency is about **structure** and **composition**. It's about decomposing a program into pieces that *could* execute independently, whether or not they actually do.

Mental Model

Think of concurrency as **interleaved execution**:

```
Task A: ████░░░░░░░░████░░░░█████
Task B: ░░░░██████░░░░░░████░░░░░
Task C: ░░░░░░░░░░███░░░░░░░░░███
Time:   ─────────────────────────>
```

Tasks A, B, and C are **concurrent**—they're making progress over the same time period, but not necessarily at the exact same instant.

The scheduler (Go runtime in our case) decides which task runs when, potentially switching between them hundreds of times per second.

Why Concurrency Exists

Concurrency exists to solve three fundamental problems:

1. I/O Wait is Wasteful

When your program waits for network, disk, or user input, the CPU sits idle. Concurrency lets other tasks make progress during these waits.

```go
// Sequential: Total time = 1s + 1s + 1s = 3 seconds
func fetchUsers() {
    user1 := fetchFromDB("user1") // 1 second
    user2 := fetchFromDB("user2") // 1 second
    user3 := fetchFromDB("user3") // 1 second
}

// Concurrent: Total time ≈ 1 second (limited by slowest fetch)
func fetchUsersConcurrent() {
    ch := make(chan User, 3)

    go func() { ch <- fetchFromDB("user1") }()
    go func() { ch <- fetchFromDB("user2") }()
    go func() { ch <- fetchFromDB("user3") }()
    // Collect results...
}
```

2. Responsiveness Requires Non-Blocking

UIs, servers, and real-time systems must respond to events while doing background work.

```
// A server MUST handle new requests while processing existing ones
http.HandleFunc("/process", func(w http.ResponseWriter, r *http.Request) {
    // If this blocks, the entire server stops responding
    // Concurrency (goroutine per request) solves this
})
```

3. Decomposition of Complex Systems

Some problems are naturally concurrent: a web crawler visits multiple pages, a video encoder processes frames, a server handles multiple clients.

Concurrency lets you model these as independent components.

What Concurrency Is NOT

❌ It's NOT about making things faster

Concurrency can improve throughput, but it often doesn't. Concurrent code can be slower due to coordination overhead.

When concurrency helps:

- I/O-bound tasks (network, disk, databases)
- Tasks that naturally wait (event-driven systems)

When concurrency doesn't help:

- CPU-bound algorithms on a single core
- Tasks that require sequential steps
- Tight loops with no I/O

❌ It's NOT parallelism

- **Concurrency**: Structure—tasks can make progress independently
- **Parallelism**: Execution—tasks literally execute at the same instant on different cores

You can have concurrency without parallelism (one core, many tasks). You cannot have parallelism without concurrency.

❌ It's NOT free

Goroutines are cheap (2KB initial stack), but not zero-cost:

- Memory overhead (stack, scheduler state)
- Context switching overhead
- Coordination overhead (mutexes, channels, atomics)
- **Mental overhead** (debugging is much harder)

❌ It's NOT a silver bullet

Many problems should NOT use concurrency:

- Simple scripts
- CPU-bound single-core tasks
- Code where correctness is more important than performance
- Anything you don't deeply understand

The Core Challenge: Shared State

Concurrency becomes hard when tasks interact:

```go
// Two goroutines, one variable
var counter int

go func() {
    counter++ // Read counter, add 1, write counter
}()

go func() {
    counter++ // Read counter, add 1, write counter
}()
```

Without synchronization, this is a data race.

Possible outcomes:

- `counter == 2` (correct)
- `counter == 1` (one increment lost)
- Program crashes (corrupted memory)

The fundamental theorem of concurrent programming:

> *When multiple tasks access shared state, and at least one modifies it, you MUST coordinate access or you have undefined behavior.*

Concurrency in Go: The Go Way

Go's concurrency model is based on **CSP (Communicating Sequential Processes)**:

> *Don't communicate by sharing memory; share memory by communicating.*

This means:

- Prefer channels (communication) over mutexes (shared memory)
- Use goroutines (lightweight threads managed by the Go runtime)
- Embrace structured concurrency (start goroutines, ensure they finish)

But this is **philosophy, not law**. Sometimes mutexes are correct. Sometimes atomics are correct. We'll learn when.

Real-World Failure: Cloudflare's 2020 Incident

What happened: Cloudflare's edge servers experienced CPU spikes causing global outages.

Root cause: A poorly managed concurrent cleanup routine. Engineers spawned goroutines for each cache eviction without rate limiting. During a traffic spike, millions of goroutines were created, exhausting CPU scheduling capacity.

Lessons:

1. Unbounded concurrency is dangerous
2. Goroutines are cheap but not free
3. Always bound your concurrency (we'll cover this in patterns)

Cost: Estimated $20M+ impact plus reputation damage.

When to Use Concurrency

✅ **Use concurrency when:**

- Handling multiple independent I/O operations
- Building servers/services that must remain responsive
- Processing naturally parallel data (map-reduce style)
- You need to wait for multiple operations simultaneously
- The problem naturally decomposes into concurrent tasks

❌ **Avoid concurrency when:**

- The sequential version is simple and fast enough
- You're processing data with dependencies (each step needs the previous)
- You can't reason about all possible interleavings
- Debugging is more costly than any performance gain
- You're on a deadline and haven't mastered concurrency

How Concurrency Fails in Real Systems

1. Silent Data Loss

Two goroutines update a map without locking → random panics weeks later in production.

2. Goroutine Leaks

Start a goroutine, forget to stop it → memory grows unbounded over days.

3. Deadlocks

Two goroutines wait for each other → entire service hangs.

4. Race Conditions

Code passes tests but fails intermittently in production under load.

5. Context Ignored

Spawn goroutine without context → operations continue after client disconnects, wasting resources.

We will study each of these in depth.

Interview Traps

Trap 1: "Concurrency makes things faster"

Correct answer: "Concurrency enables better resource utilization in I/O-bound scenarios. It doesn't automatically make CPU-bound work faster on a single core. Parallelism requires multiple cores."

Trap 2: "Goroutines are threads"

Correct answer: "Goroutines are user-space constructs managed by the Go runtime. Multiple goroutines multiplex onto a smaller number of OS threads. They're much lighter (2KB vs 1-2MB for threads)."

Trap 3: "This mutex makes my code concurrent"

Correct answer: "Mutexes coordinate concurrent access; they don't create concurrency. You need goroutines for concurrency. Mutexes prevent data races when goroutines access shared state."

Trap 4: "Race-free code is correct code"

Correct answer: "A program can be free of data races but still incorrect due to logical bugs in ordering or synchronization. Race freedom is necessary but not sufficient for correctness."

Key Takeaways

1. **Concurrency is structure**, not execution
2. **Shared state** is the root of all concurrent evil
3. Go provides primitives (goroutines, channels, mutexes), not magic
4. Concurrency makes programs **harder to reason about**
5. Always ask: "Do I need concurrency?" before adding it
6. Every concurrent program must answer: "How do I coordinate access to shared state?"

What You Should Be Thinking Now

- "How does Go schedule goroutines?"
- "What's the difference between concurrency and parallelism exactly?"
- "How do I know if my code has a data race?"
- "When should I use channels vs mutexes?"

Next: concurrency-vs-parallelism.md - We'll answer the first two questions.

Exercises (Do These Before Moving On)

1. **Identify**: Find 3 places in code you've written where concurrency could help (I/O waits, blocking operations).
2. **Analyze**: For each, write down: "What would happen if I used concurrency here incorrectly?"
3. **Reflect**: Have you ever written code with a data race? What broke?

Don't move forward until you can explain: "Concurrency is about structure; parallelism is about execution" in your own words.