dispatcher.go

```
for range d.workers {
  wg.Go(func() {
    for j := range jobs {
      select {
      case out <- deliver(ctx, j):
      case <-ctx.Done():
        return
      }
// happens-before, not hope
```

goroutines

chan · select

sync · atomic

context

memory model




DEEP DIVE FOR GO DEVELOPERS

# Go Concurrency in Depth

## Goroutines, channels, the memory model, and production patterns that hold up under load

Go 1.27 · scheduler · sync · atomic · context · errgroup
race detector · synctest · pprof · execution tracing

CORRECTNESS · CANCELLATION · PATTERNS · DEBUGGING

THENEXBOOKS

# Go Concurrency in Depth

*Goroutines, Channels, the Memory Model, and Production Patterns That Hold Up Under Load*

*Scheduler, sync, context, testing, profiling, and tracing — for Go 1.27*

First Edition

THENEXBOOKS

**Go Concurrency in Depth**
Goroutines, Channels, the Memory Model, and Production Patterns That Hold Up Under Load

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Go, its runtime, and the libraries discussed in this book evolve; always consult the official documentation for the versions you use. Timings and benchmark results shown in this book were measured on one test machine and will differ on yours.

Go and the Go gopher are trademarks of Google LLC. Windows is a trademark of Microsoft Corporation. Linux is a registered trademark of Linus Torvalds. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Google or the Go project.

Driftwire, Driftwire Relay, the tenants Harbor Bikes, Larch Ledger, and Nimbus Fitness, and the people in the running example are fictional, as are all events, endpoints, and data. Domains such as example.com and harbor.example are reserved for examples; no real webhook is ever called.

# Preface

Starting a goroutine takes two letters. Making a concurrent program *correct* — and keeping it correct as it grows, as load rises, and as the people who wrote it move on — takes much more. Most Go developers learn concurrency in a weekend: `go`, channels, a `WaitGroup`, maybe a mutex. Then production teaches the rest the hard way: a worker pool that leaks a goroutine per request, a shutdown that drops in-flight work, a counter that is wrong only under load, a deadlock that appears once a month at 3 a.m.

This book is the rest. It explains how the Go runtime actually schedules goroutines, what channels and `select` guarantee and what they do not, what the Go memory model promises and why "it works on my machine" means nothing for a racy program, and how every tool in `sync`, `sync/atomic`, and `context` fits a specific job. Then it turns that understanding into patterns — worker pools, pipelines, rate limiters, partitioned workers, event buses — and shows each one's failure modes before they reach production.

Just as important, it teaches you to *prove* concurrent code works. You will use the race detector, goroutine leak checks, `testing/synctest` virtual time, block and mutex profiles, and execution traces on one realistic system, and you will learn to read their output the way an experienced on-call engineer does.

## Who This Book Is For

- **Go developers** who already write Go comfortably and want to master concurrency rather than copy patterns they half understand.
- **Backend and platform engineers** who run Go services and need to reason about timeouts, cancellation, shutdown, backpressure, and leaks.
- **Developers from other concurrent languages** (Java, C#, Rust, Kotlin, C++) who want Go's exact rules instead of analogies.
- **Tech leads and reviewers** who want a checklist and a shared vocabulary for reviewing concurrent code.

We assume you know the Go language itself: types, slices, maps, structs, interfaces, errors, packages, modules, and basic testing. If you are new to Go, start with a beginner book and come back.

## What You Will Learn

- How goroutines are scheduled (G, M, and P; run queues; work stealing; preemption; `GOMAXPROCS`) and how to control their lifetimes.
- Exact channel and `select` semantics, timers and tickers, and the ownership rules that keep channel code from panicking or leaking.

- The Go memory model: happens-before, synchronized-before, data races, and what each synchronization primitive guarantees.
- Mutexes, read-write locks, atomics, `Once`, `Cond`, `Pool`, and `sync.Map` — and when each one is the wrong choice.
- `context` in depth, timeouts, retries with backoff and jitter, `errgroup`, `semaphore`, `singleflight`, and graceful shutdown.
- Production patterns: worker pools, pipelines, fan-out and fan-in, rate limiting, bounded queues and backpressure, per-key ordering, and pub/sub.
- Testing and debugging: the race detector, goroutine leak detection, `testing/synctest`, deadlock and starvation diagnosis, pprof, and execution tracing.
- Benchmarking and tuning concurrent code honestly, and building concurrent HTTP services that shed load instead of falling over.

## How This Book Is Organized

**Part I — How Go Runs Concurrent Code** (Chapters 1–5) covers goroutines and the scheduler, lifetimes and `WaitGroup`, channels, and `select` with timers. **Part II — Shared Memory and the Memory Model** (Chapters 6–10) covers the memory model, mutexes, atomics, the rest of the `sync` package, and the race detector. **Part III — Coordination and Cancellation** (Chapters 11–15) covers `context`, timeouts and retries, `errgroup` and friends, graceful shutdown, and goroutine leaks. **Part IV — Concurrency Patterns** (Chapters 16–20) builds worker pools, pipelines, rate limiters, partitioned workers, and an event bus. **Part V — Testing and Debugging Concurrent Code** (Chapters 21–24) covers deterministic tests, virtual time, a deadlock casebook, and profiling. **Part VI — Performance and Production** (Chapters 25–28) covers execution tracing, benchmarking, concurrent HTTP services, and the finished system. The appendices provide a cheat sheet, pattern templates, a production checklist, and a glossary.

Each chapter ends with **Try It Yourself** exercises and a **Key Takeaways** box for quick review.

## The Running Example: Driftwire Relay

Throughout the book we work on **Driftwire Relay**, the webhook delivery service of **Driftwire**, a fictional company. Customer applications — an online bike shop, an invoicing app, and a gym chain — publish events such as `order.created` and `invoice.paid`. Relay delivers every event to each subscribed webhook endpoint, retries failures with backoff, respects each endpoint's rate limit, and must shut down without losing work. Some endpoints are fast, one is flaky, one is retired, and one is too slow for its timeout: exactly the mix that makes concurrency necessary and difficult. Odile, the platform lead, asks what each design guarantees; Tariq, the on-call engineer, brings the incidents; and Juno, new to the team, makes the classic mistakes so you do not have to.

The complete reference project uses only the standard library plus three small, widely used modules (`golang.org/x/sync`, `golang.org/x/time`, and `go.uber.org/goleak`). Webhook endpoints are simulated in memory, so every example runs offline and every output in the book is reproducible.

## Versions and Conventions

Examples target **Go 1.27**. Most code also runs on Go 1.25 and later; where a feature needs Go 1.25, 1.26, or 1.27 — such as `sync.WaitGroup.Go`, `testing/synctest`, or `synctest.Sleep` — we say so. Runtime internals such as the scheduler are implementation details that can change between releases; we describe the current runtime and point to the official documentation where details may move. Timing and benchmark numbers come from one test machine and are labeled as such; your numbers will differ.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause races, deadlocks, leaks, or lost work.

Code appears in `monospace`. Long lines are wrapped to fit the page. File names above code blocks, such as `driftwire/relay/dispatcher.go`, tell you where the code lives. Commands work the same on Windows, macOS, and Linux unless noted.

# Contents

Preface

PART I — HOW GO RUNS CONCURRENT CODE

1 Why Concurrency Is Hard — and What Go Gives You

2 Goroutines and the Go Scheduler

3 Goroutine Lifetimes, WaitGroup, and Structured Concurrency

4 Channels in Depth

5 select, Timers, and Tickers

PART II — SHARED MEMORY AND THE MEMORY MODEL

6 The Go Memory Model: Happens-Before Without the Mystery

7 Mutexes and RWMutex: Designing Critical Sections

8 sync/atomic: Counters, Flags, and Lock-Free Snapshots

9 Once, Cond, Pool, and sync.Map

10 The Race Detector

PART III — COORDINATION AND CANCELLATION

11 context in Depth: Cancellation, Deadlines, Causes, and Values

12 Timeouts, Retries, and Backoff

13 errgroup, semaphore, and singleflight

14 Graceful Shutdown

15 Goroutine Leaks: Prevention and Detection

PART IV — CONCURRENCY PATTERNS

16 Worker Pools: The Relay Dispatcher

17 Pipelines, Fan-Out, and Fan-In

18 Rate Limiting, Bounded Queues, and Backpressure

19 Per-Key Ordering and Partitioned Workers

20 Pub/Sub, Broadcast, and Event Buses

**PART V — TESTING AND DEBUGGING CONCURRENT CODE**

21 Testing Concurrent Code

22 Virtual Time with testing/synctest

23 Deadlocks, Livelocks, and Starvation: A Debugging Casebook

24 Profiling Concurrency: Goroutine, Block, and Mutex Profiles

**PART VI — PERFORMANCE AND PRODUCTION**

25 Execution Tracing and the Flight Recorder

26 Benchmarking and Tuning Concurrent Code

27 Concurrency in HTTP Services

28 The Production Relay: Putting It All Together

**APPENDICES**

A Concurrency Cheat Sheet

B Pattern Templates and API Reference

C Production Concurrency Checklist

D Glossary

PART I

# How Go Runs Concurrent Code

What concurrency buys Driftwire Relay and what it costs, how the runtime schedules goroutines onto threads, how to own a goroutine's lifetime, and the exact semantics of channels, select, timers, and tickers.

1 · Why Concurrency Is Hard — and What Go Gives You

2 · Goroutines and the Go Scheduler

3 · Goroutine Lifetimes, WaitGroup, and Structured Concurrency

4 · Channels in Depth

5 · select, Timers, and Tickers

CHAPTER 1 · PART I — HOW GO RUNS CONCURRENT CODE

# Why Concurrency Is Hard — and What Go Gives You

*Concurrent programs fail in ways sequential programs cannot: results that change from run to run, programs that freeze, memory that slowly fills with forgotten work. In this chapter we meet Driftwire Relay, run it sequentially and concurrently, watch the four classic failure modes happen in tiny programs, and map the tools Go gives us to tame each one.*

---

## A Webhook Relay at 3 a.m.

Driftwire is a fictional company that runs **Driftwire Relay**, a webhook delivery service. Its customers — we call them *tenants* — publish events such as "an order was paid" or "a member joined". Relay looks up which webhook endpoints each tenant has subscribed, sends every event to every interested endpoint, retries the failures with backoff, respects each endpoint's rate limit, and must be able to shut down without losing or duplicating a delivery.

Three people will keep us company throughout the book. **Odile** leads the platform team and owns Relay's design; her favorite question is "what does it guarantee?" **Tariq** is the on-call SRE, and he arrives with incidents: deliveries stuck for an hour, a goroutine count that climbs all week, a deadlock at 3 a.m. **Juno** recently joined the Relay team from another Go project. Juno writes good sequential Go and is about to learn, alongside you, every classic concurrency mistake — usually by making it first.

Relay has three tenants. `harbor` is Harbor Bikes, an online bike shop. `larch` is Larch Ledger, an invoicing app. `nimbus` is Nimbus Fitness, a gym chain. Between them they own five endpoints, and those endpoints behave the way real webhook receivers behave: most answer quickly, one has been retired and answers `410 Gone`, one is flaky and fails twice before it succeeds, and one is simply slow.

If every receiver answered in a millisecond, a simple loop would do. Receivers do not cooperate: a single slow endpoint can hold a sequential loop hostage while every other tenant waits behind it. Concurrency is how Relay keeps one tenant's slow server from becoming everyone's outage.

## Concurrency Is Not Parallelism

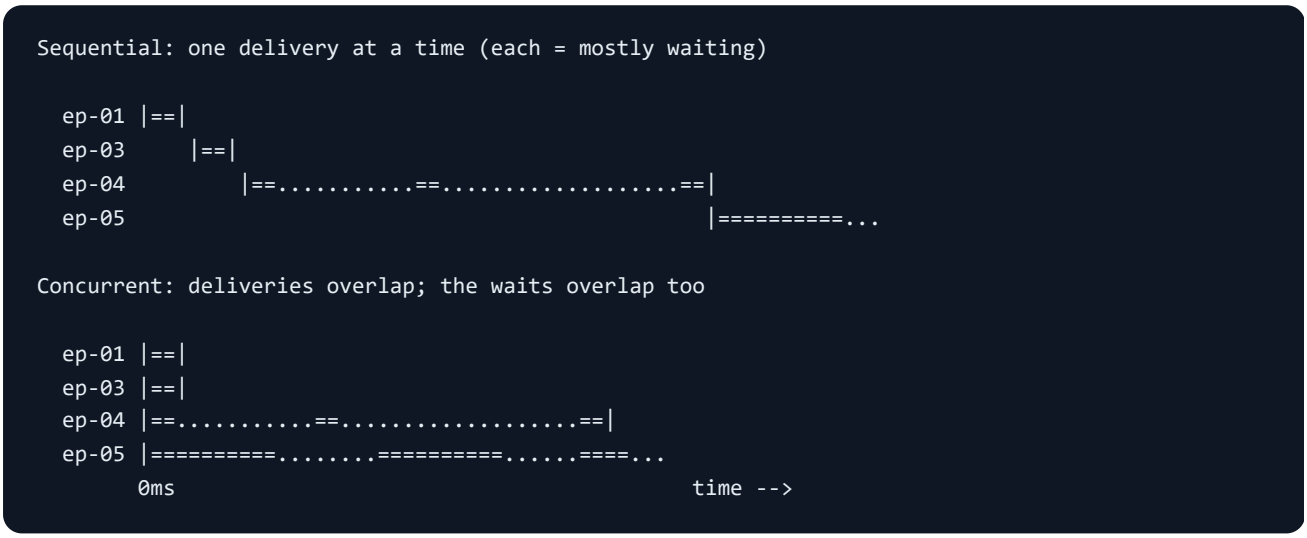
Two words get mixed up constantly, so we fix their meaning now.

- **Concurrency** is about *structure*: a program is concurrent when it is composed of independently executing activities whose lifetimes overlap. Several deliveries can be "in progress" at once even if only one of them is executing at any instant.

- **Parallelism** is about *execution*: two pieces of work are parallel when they literally run at the same instant on different CPU cores.

Concurrency is a property of the program's design; parallelism is a property of a particular run on particular hardware. A concurrent program can run on a single core, interleaving its activities, and it is still concurrent. A parallel program needs more than one core, but it does not need to be interesting: summing two halves of an array on two cores is parallel and barely concurrent.

The distinction matters because the two help with different problems. Parallelism helps when work is *CPU-bound*: more cores, more arithmetic per second. Concurrency helps when work is *waiting-bound*: while one delivery waits for a slow server, another delivery can make progress. Relay's work is almost entirely waiting. A webhook delivery is a few microseconds of encoding followed by tens or hundreds of milliseconds of waiting for the network.




In the concurrent picture nothing runs faster. Every individual delivery takes exactly as long as before. What changes is that the waits overlap, so the *total* time — the makespan — shrinks. This technique is called **latency hiding**, and it is the main reason network services are written concurrently. We will see in a moment that Relay gets almost all of its speedup even when Go is allowed to use only one CPU core, which is the clearest possible demonstration that concurrency, not parallelism, is doing the work.

## Where the time goes in a sequential Relay

Relay's sample data contains eight events, and routing them to the subscribed endpoints produces nine delivery jobs. The fake endpoints are scripted so the timing is exact. Under the default delivery policy — at most 3 attempts, a 200ms timeout per attempt, and a backoff of 100ms after the first failure and 200ms after the second — each job costs:

| Job | Endpoint behavior | Cost |
|---|---|---|
| evt-001 → ep-01 | OK after 20ms | 20ms |
| evt-002 → ep-03 | OK after 20ms | 20ms |

| | | |
|---|---|---|
| evt-003 → ep-04 | 503, 503, then OK; backoff 100ms + 200ms | 360ms |
| evt-003 → ep-05 | 300ms latency, each attempt times out at 200ms | 900ms |
| evt-004 → ep-01 | OK after 20ms | 20ms |
| evt-005 → ep-02 | 410 after 20ms, not retryable | 20ms |
| evt-006 → ep-03 | OK after 20ms | 20ms |
| evt-007 → ep-05 | times out three times | 900ms |
| evt-008 → ep-01 | OK after 20ms | 20ms |

The sum is 2,280ms. Two jobs to the slow endpoint `ep-05` account for 1,800ms of it, and during that time Relay's CPU does essentially nothing. The flaky `ep-04` adds 360ms, most of which is deliberate backoff sleep. A sequential Relay spends almost all of its life waiting.

## Touring the Reference Project

Every chapter in this book works against one small, real Go module, `example.com/driftwire`. It ships with the book in the `sample` folder and needs nothing beyond Go itself plus three small modules from the Go proxy (`golang.org/x/sync`, `golang.org/x/time`, and `go.uber.org/goleak`). There are no real network calls: webhook endpoints live under `.example` hosts and are simulated in memory by a `FakeSender`, so every run is reproducible.

Copy the sample to a folder of your own before you start editing — every exercise assumes you can change files freely and compare against the original. Then check that it builds and that the tests pass with the race detector:

```
go vet ./...
go test -race -count=1 ./...
```

The layout is small on purpose. The parts you will touch first are:

| Path | What it holds |
|---|---|
| `event/event.go` | `Event`, `Endpoint`, `Decode`, the sample data |
| `event/data/events.jsonl` | the eight canonical events, embedded |
| `relay/router.go` | `Router`, `Match`, `Job`, `Jobs` |
| `relay/sender.go` | `Sender`, `StatusError`, `FakeSender`, plans |
| `relay/deliver.go` | `Policy`, `Deliver`, `RunSequential`, `Summary` |
| `relay/dispatcher.go` | `Dispatcher`, `RunPool` (a worker pool) |
| `limiter/`, `ingest/` | rate limits and concurrent fetching |

| | |
|---|---|
| `cmd/relayd/main.go` | the relay daemon you run from the terminal |

An `Event` is one thing that happened in a tenant's application. An `Endpoint` is a webhook URL that wants some event types of one tenant. Here is the first line of the sample file, wrapped for the page (it is a single line on disk):

```
{"id":"evt-001","tenant":"harbor","type":"order.created","at":"2026-03-02T09:00:00Z",
"payload":{"order":"HB-1042","total_cents":12900}}
```

The router turns events into jobs. `Jobs` walks the events in order and, for each one, the matching endpoints in ID order, so the nine jobs always come out the same way: evt-001→ep-01, evt-002→ep-03, evt-003→ep-04, evt-003→ep-05, evt-004→ep-01, evt-005→ep-02, evt-006→ep-03, evt-007→ep-05, evt-008→ep-01.

The heart of the module is one function. Its signature is worth memorizing, because almost every chapter calls it:

driftwire/relay/deliver.go

```
func Deliver(ctx context.Context, s Sender, p Policy, ep event.Endpoint,
    ev event.Event) Result
```

`Deliver` makes up to `p.MaxAttempts` attempts, each with its own timeout derived from `ctx`, sleeps between attempts using exponential backoff, and returns a `Result` with a `Status` of `delivered`, `failed`, or `canceled`. It is a plain, blocking, sequential function. Nothing in it starts a goroutine. That is a deliberate design choice we will defend in Chapter 3: functions that do one thing synchronously are easy to compose concurrently, and the caller decides how much concurrency to apply.

> **NOTE**
> The book does not re-teach Go basics. We assume you are comfortable with slices, maps, structs, interfaces, modules, error wrapping with `%w`, and tests.

## Running Relay Sequentially and Concurrently

The daemon `relayd` has two delivery modes. `-mode seq` calls `RunSequential`, a plain loop over the jobs. The default, `-mode pool`, calls `RunPool`, which hands the jobs to a fixed number of worker goroutines (four unless you pass `-workers`). The `-timing` flag prints the elapsed wall-clock time to standard error. Start with the sequential baseline:

```
go run ./cmd/relayd -mode seq -timing
```

On our test machine it printed:

```
evt-001 -> ep-01 delivered attempts=1
evt-002 -> ep-03 delivered attempts=1
evt-003 -> ep-04 delivered attempts=3
evt-003 -> ep-05 failed attempts=3 err=context deadline exceeded
evt-004 -> ep-01 delivered attempts=1
evt-005 -> ep-02 failed attempts=1 err=endpoint returned 410
evt-006 -> ep-03 delivered attempts=1
evt-007 -> ep-05 failed attempts=3 err=context deadline exceeded
evt-008 -> ep-01 delivered attempts=1
summary: deliveries=9 delivered=6 failed=3 canceled=0 attempts=15
mode=seq elapsed=2.293s
```

Now the default pool:

```
go run ./cmd/relayd -timing
```

The nine result lines and the summary are byte-for-byte identical, so we show only the last two lines:

```
summary: deliveries=9 delivered=6 failed=3 canceled=0 attempts=15
mode=pool elapsed=1.052s
```

These elapsed times are from our test machine (an AMD Ryzen 9 5950X running Windows 11 and Go 1.27.1); your numbers will differ by some milliseconds, and they will vary from run to run on the same machine. The shape will not change: four workers cut the run from about 2.3 seconds to about 1 second.

### Reading the output

Each result line says which event went to which endpoint, the final status, how many attempts were made, and the last error if there was one. Read it line by line against the endpoint behaviors:

- `evt-003 -> ep-04 delivered attempts=3` — the flaky endpoint answered 503 twice; both were retryable, and the third attempt succeeded.
- `evt-003 -> ep-05 failed attempts=3 err=context deadline exceeded` — every attempt to the slow endpoint ran into the 200ms per-attempt timeout. A timeout is retryable, so `Deliver` tried three times and then gave up.
- `evt-005 -> ep-02 failed attempts=1 err=endpoint returned 410` — a 410 is permanent, so there was no point retrying.
- The summary adds it up: 9 deliveries, 6 delivered, 3 failed, none canceled, 15 attempts in total (six single attempts, plus 3 + 3 + 3 for the three multi-attempt jobs).

Notice what is *not* in the output: any sign of which order the pool actually ran the deliveries in. Workers finish in whatever order the endpoints answer, so `RunPool` collects the results and sorts them by event ID, then endpoint ID, before returning. That sorting is why the output is identical in both

modes, and it is a habit worth adopting: make concurrent programs produce deterministic output so you can compare runs, write golden tests, and spot regressions. We will lean on it constantly.

### Why four workers take about 940ms

Wall-clock timings wobble, so the reference tests measure the same runs in *virtual time* using `testing/synctest`, a standard-library package that gives a test a fake clock which advances only when every goroutine in the test is blocked (Chapter 22 covers it fully). In virtual time the sequential run takes exactly 2,280ms and the four-worker pool exactly 940ms. Here is why, worker by worker:

```
        0    20   40   60   80            360               900  940 ms
W1  [ev1 ep-01][ev4 ep-01][ev6 ep-03][ev8 ep-01]
W2  [ev2 ep-03][ev5 ep-02][ev7 ep-05 ......................................]
W3  [ev3 ep-04 ............................]
W4  [ev3 ep-05 ..........................................................]
```

(Event IDs are shortened to `ev1` for `evt-001`; bars are not to scale, and which worker takes which job can vary.) At time zero the four workers take the first four jobs. The two quick ones finish at 20ms and those workers pick up the next jobs, which also finish quickly. At 40ms worker W2 picks up the second `ep-05` job, which takes 900ms and therefore ends at 940ms. Everything else is done long before, so the slowest single chain decides the makespan.

Two more runs on our test machine complete the picture. With nine workers, every job starts at time zero, and the run is bounded by the slowest single delivery, 900ms in virtual time:

```
go run ./cmd/relayd -workers 9 -timing
```

```
mode=pool elapsed=998ms
```

And with the Go runtime restricted to a single CPU core — set the `GOMAXPROCS` environment variable to `1` before running (in PowerShell, `$env:GOMAXPROCS = "1"`; in a Unix shell, prefix the command with `GOMAXPROCS=1`) — the four-worker pool still finished on our test machine in:

```
mode=pool elapsed=944ms
```

One core, same speedup. There was no parallelism in that run at all, only concurrency: while one goroutine waited for a timer, another ran. This single experiment is the most important intuition in Part I. For waiting-bound work, the number of things you can have *in flight* matters far more than the number of cores. Chapter 2 explains how the Go scheduler makes that possible.

> **TIP**
>
> When a speedup looks too good or too bad, rerun with `GOMAXPROCS=1`. If the program is just as fast on one core, its speedup comes from overlapping waits (concurrency). If it slows down a lot, it really is using multiple cores (parallelism). The answer tells you which tuning knobs matter.

## What Can Go Wrong: Four Failure Modes

Concurrency bought Relay a speedup of more than two times with a dozen lines of code. It also opened the door to four families of bugs that sequential code cannot have. Every incident Tariq brings to the team over the course of this book belongs to one of them. We will meet each now in a tiny program so you can recognize it later in a large one.

### 1. Data races

A **data race** happens when two goroutines access the same memory location at the same time, at least one of the accesses is a write, and nothing orders them. Juno's first idea for counting deliveries across goroutines was the obvious one:

racecount/main.go

```
// Command racecount shows a data race: many goroutines increment one int.
package main

import (
    "fmt"
    "sync"
)

func main() {
    var delivered int // BUG: shared without synchronization
    var wg sync.WaitGroup
    for range 1000 {
        wg.Go(func() {
            delivered++ // BUG: read-modify-write races with other goroutines
        })
    }
    wg.Wait()
    fmt.Println("delivered:", delivered)
}
```

(`wg.Go` starts a goroutine and tracks it in the `WaitGroup`; it is new in Go 1.25 and Chapter 3 covers it.) Three runs on our test machine printed:

```
delivered: 995
delivered: 985
delivered: 989
```

A thousand increments, and none of the three runs counted them all. `delivered++` is not one operation: it loads the value, adds one, and stores the result. Two goroutines can both load 41, both compute 42, and both store 42, and one increment disappears. Worse, the program's answer changes from run to run, and on another machine or on a quiet day it may print 1000 and pass every test you have.

Go ships a tool that finds this bug for you. Add `-race`:

```
go run -race ./racecount
```

```
==================
WARNING: DATA RACE
Read at 0x00c00000c138 by goroutine 48:
  main.main.func1()
      .../racecount/main.go:14 +0x2e

Previous write at 0x00c00000c138 by goroutine 91:
  main.main.func1()
      .../racecount/main.go:14 +0x44

Goroutine 48 (running) created at:
  sync.(*WaitGroup).Go()
      .../src/sync/waitgroup.go:238 +0x72
...
==================
delivered: 924
Found 2 data race(s)
exit status 66
```

The report names the memory address, the two conflicting accesses (a read and a previous write, both at line 14), and where each goroutine was created. Chapter 10 teaches you to read these reports fluently. For now, take away two facts. First, the race detector found a bug that three ordinary runs only hinted at. Second, a racy Go program is not "slightly wrong": the Go memory model (Chapter 6) gives racy programs very weak guarantees, and races on values such as slices, strings, interfaces, and maps can corrupt memory and crash the program. The fixes — a mutex, an atomic counter, or not sharing the variable at all — come in Chapters 7, 8, and 3.

## 2. Deadlocks

A **deadlock** is a state in which goroutines wait for each other in a way that can never be resolved. The smallest possible example is one goroutine waiting for itself:

deadlock/main.go

```
// Command deadlock blocks forever: nobody receives from results.
package main

import "fmt"
```

```
func main() {
    results := make(chan string)
    results <- "evt-001 -> ep-01 delivered" // BUG: no receiver, main blocks
    fmt.Println(<-results)
}
```

A send on an unbuffered channel waits until another goroutine receives. The only goroutine that could receive is `main` itself, and it is stuck in the send. The runtime notices that no goroutine can ever make progress and stops the program:

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
    .../deadlock/main.go:8 +0x36
exit status 2
```

The message is helpful, and it is also a trap. The runtime reports a deadlock only when *every* goroutine is blocked. In a real service, some goroutines are always alive — an HTTP server waiting for connections, a ticker loop, a signal handler — so a partial deadlock, where only the delivery workers are stuck, produces no message at all. The process simply stops making progress while looking perfectly healthy. Tariq's 3 a.m. incident in Chapter 23 is exactly that, and finding it takes goroutine dumps rather than a friendly fatal error.

### 3. Goroutine leaks

A **goroutine leak** is a goroutine that was started and will never finish. Each one holds its stack and everything its stack references. Leaks rarely crash anything quickly; they accumulate, and memory use climbs for days until something gives. Here Juno asks three mirrors for an answer and takes whichever comes back first:

leak/main.go

```
// Command leak abandons goroutines that block forever on a send.
package main

import (
    "fmt"
    "runtime"
    "time"
)

// firstAnswer asks three mirrors and returns the first answer.
func firstAnswer() string {
    answers := make(chan string) // BUG: unbuffered; the losers block forever
    for _, mirror := range []string{"a", "b", "c"} {
        go func() {
            time.Sleep(10 * time.Millisecond)
            answers <- "answer from " + mirror
        }()
```

```
    }
    return <-answers
}

func main() {
    for range 5 {
        firstAnswer()
    }
    time.Sleep(50 * time.Millisecond) // let the losers reach their send
    fmt.Println("goroutines still alive:", runtime.NumGoroutine())
}
```

```
goroutines still alive: 11
```

Five calls, two losers per call, ten goroutines blocked forever on a send that no one will ever receive, plus `main`. The function returned the right answer every time, and every test of `firstAnswer` passes. The bug is invisible until you count goroutines. (The one-character fix is to give the channel a buffer of three, so every mirror can send and exit; Chapter 4 explains why that works and Chapter 15 is devoted to leaks.)

### 4. Starvation

**Starvation** means a goroutine is runnable, or waiting for a resource, but rarely gets it because others keep winning. In this program one goroutine holds a mutex while doing a little work and grabs it again immediately; another just wants the lock briefly from time to time:

starve/main.go

```
// Command starve shows a greedy goroutine hogging a mutex.
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    var mu sync.Mutex
    var greedy, polite int
    done := make(chan struct{})
    var wg sync.WaitGroup
    wg.Go(func() { // takes the lock back immediately, over and over
        for {
            select {
            case <-done:
                return
            default:
            }
            mu.Lock()
            greedy++
            time.Sleep(100 * time.Microsecond) // "work" while holding the lock
```

```
            mu.Unlock()
        }
    })
    wg.Go(func() { // wants the lock now and then
        for {
            select {
            case <-done:
                return
            default:
            }
            mu.Lock()
            polite++
            mu.Unlock()
            time.Sleep(100 * time.Microsecond)
        }
    })
    time.Sleep(time.Second)
    close(done)
    wg.Wait()
    fmt.Println("greedy:", greedy, "polite:", polite)
}
```

Two runs on our test machine printed the following; your numbers will differ:

```
greedy: 1818 polite: 461
greedy: 1554 polite: 382
```

Both goroutines asked for the lock at similar rates, yet the greedy one held it roughly four times as often. Nothing is broken in the sense of a crash or a wrong value; the system is just unfair. In Relay, the same shape appears when one tenant's slow endpoint occupies every worker and the other tenants wait (Chapter 23), or when a slow call holds a shared lock (Chapter 7). Go's `sync.Mutex` has a built-in starvation mode that limits how unfair it can become, which Chapter 7 describes; it bounds the damage, but it does not make bad lock design good.

> **WARNING**
>
> None of these four bugs is reliably caught by running your program and checking the output. The racy counter often prints the right number, the leaky function always returns the right answer, and a partial deadlock produces no error at all. Concurrency bugs need dedicated tools — the race detector, goroutine counts and dumps, leak checks, profiles — and those tools only see what actually runs. Plan for them from the first day, not after the first incident.

## Why These Bugs Are So Hard

It helps to name exactly what makes concurrent code harder than sequential code, because each source of difficulty points at a defense.