Chapter 5: Goroutines — Concurrency Made Composable

> “Don’t communicate by sharing memory; share memory by communicating.” — Effective Go

One of Go’s defining features is its built-in concurrency model. Unlike traditional thread-based concurrency in Java or async/await models in Rust, Python, or JavaScript, Go uses **goroutines**: lightweight, user-space threads that make concurrency feel natural and composable.

What Is a Goroutine?

A goroutine is a function executing independently, managed by the Go runtime. To launch a goroutine use the go keyword:

```go
func sayHello() {
    fmt.Println("Hello from goroutine")
}

func main() {
    go sayHello() // launch it
    // give it time to finish
    // or it will return immediately and end the program
    time.Sleep(time.Second)
}
```

Unlike threads, goroutines:

- Have tiny initial stack sizes (~2 KB)
- Grow/shrink dynamically
- Are multiplexed onto system threads by the Go scheduler (M:N model)

You can easily spawn **thousands** of them.

The anatomy of a go keyword call is the following:

```go
go func() { // a method, an anonymous or named function
    time.Sleep(time.Second) // the anonymous function body
}() // the () guarantees immediate invocation, otherwise it is just created
↪   and not called
```

Understanding Go's Scheduler: The M:P:G Model

Go's concurrency model is built on a lightweight, user-space scheduler that enables efficient multitasking without relying heavily on operating system threads. It follows a preemptive scheduling model, allowing the Go runtime to interrupt long-running goroutines to maintain fairness and responsiveness.

The scheduler revolves around three core components:

- G (Goroutine) — Represents a lightweight, independent unit of execution. Goroutines are cheaper than threads and can number in the hundreds of thousands within a single process.
- M (Machine) — Maps to an actual OS thread. It's the entity that executes Go code on a CPU core.
- P (Processor) — Holds the run queue of runnable goroutines and manages context switching. A P is required for an M to execute Go code, and each P has its own local queue of Gs.

At runtime:

- Each P is bound to one M, forming an execution unit.
- G instances (goroutines) are scheduled onto available Ps, which are then executed by Ms.
- If a goroutine performs a blocking syscall, the associated M is parked and a new M is assigned to the P to avoid stalling the system.

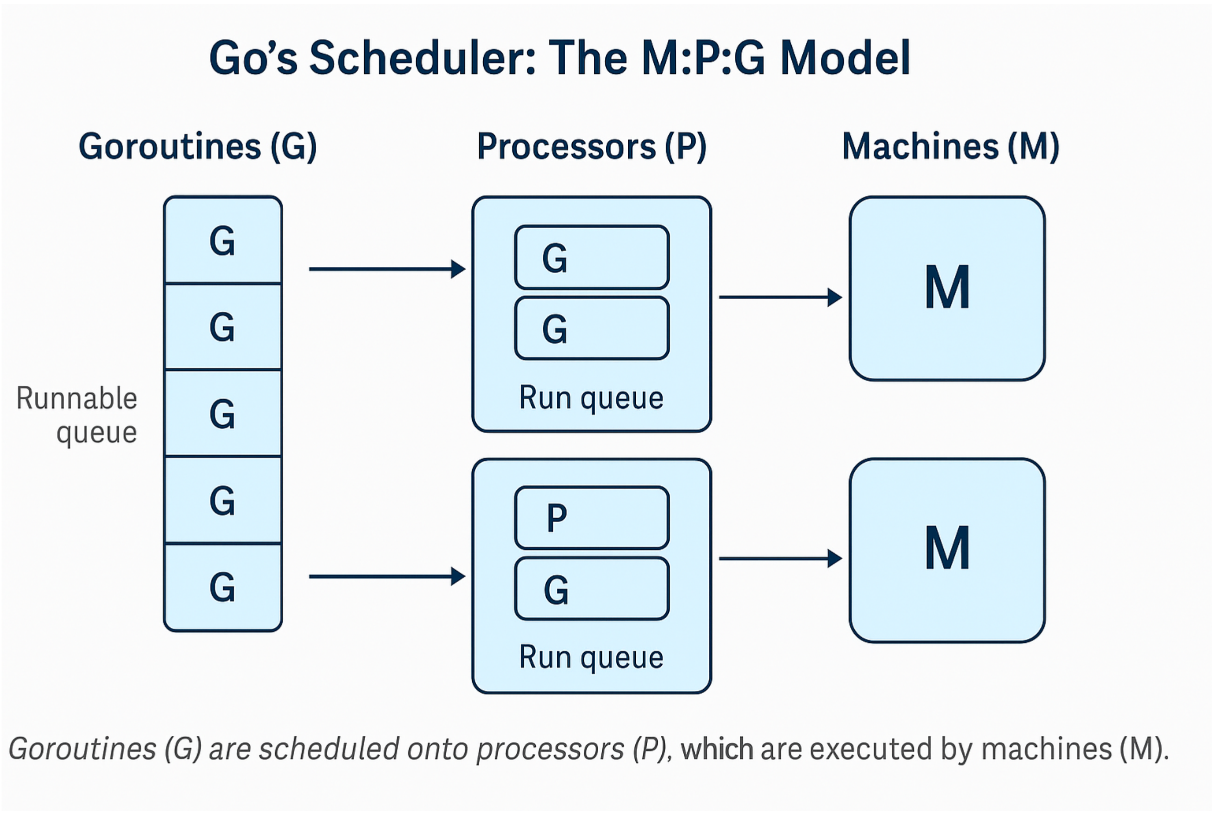
Go's Scheduler: The M:P:G Model
Goroutines (G)
Processors (P)
Machines (M)
Runnable queue
G
G
G
G
G
G
G
Run queue
P
G
Run queue
M
M
Goroutines (G) are scheduled onto processors (P), which are executed by machines (M).

Figure 1: go-scheduler

This design decouples goroutines from OS threads, enabling Go to multiplex thousands of goroutines over a limited number of threads, scaling efficiently across CPU cores.

You can influence the level of parallelism using runtime.GOMAXPROCS(), which controls how many Ps (and thus logical parallel executions) can run simultaneously. Since Go 1.5, this defaults to the number of available CPU cores.

There is a great article about the scheduler which illustrate its mechanics.

Goroutine Lifecycle and Pitfalls

Lifecycle

1. Goroutines are created via go keyword.
2. It is then scheduled and run by the Go runtime
3. Exits when the function returns

Pitfalls

- **Goroutine leaks**: created but never exited (e.g., waiting forever on a channel)
- **Unbounded spawning**: creating thousands per event/request without bounds, with high cost to the scheduler
- **Access to shared memory** without synchronization, leading to race conditions and data corruption
- **Homemade expiration** blocking time-based controls

Whenever we have to spawn a group of goroutines, there may be a point where we need to wait for them to finish and move on. You will find the `wait` verb in many languages and libraries.

One of the ways of not doing it, which seems counter intuitive is to really wait over a period of time.

```go
func main() {
    for i := 0; i < 1000; i++ {
        go func() {
            time.Sleep(time.Second)
        }()
    }
    // naive attempt to wait all goroutines to finish
    time.Sleep(2 * time.Second)
}
```

And what will happen if we need any measure of control of our goroutines ? Something like this using a mess of channels.

To help clear this up, the standard library offers:

- Use `sync.WaitGroup` to manage a collection of goroutines and wait them to finish.
- Use `context.Context` for cancelation and control from outside and within goroutines.

We will cover both in the next chapters in detail

5.4 Goroutines vs Threads vs Async

Feature	Goroutines	OS Threads	Async/Await (Rust/JS)
Stack Size	Tiny (~2 KB)	Large (1MB+)	N/A

Feature	Goroutines	OS Threads	Async/Await (Rust/JS)
Launch Overhead	Very low	High	Low
Syntax Simplicity	Very simple	Moderate	Moderate to High
Scheduling	User-level	OS-level	Cooperative
Communication	Channels	Shared Memory	Futures + State

Go's syntax and concurrency model feels like synchronous code while being highly concurrent. No callbacks, `.await`, Futures or Promises.

Goroutines in practice

Concurrent HTTP Server

In Go, each incoming HTTP request is handled by its own goroutine. When a server receives a request, it launches a new goroutine to execute the handler function associated with the request's URL path.

This concurrency model allows the server to handle multiple requests simultaneously without blocking. Usually each request is self contained or end up storing or transforming data.

```go
package main

import (
    "fmt"
    "log"
    "net/http"
    "time"
)

func handler(w http.ResponseWriter, r *http.Request) {
    id := time.Now().UnixNano()
    fmt.Printf("Start handling request %d\n", id)
    time.Sleep(2 * time.Second) // simulate work
    fmt.Fprintf(w, "Handled by goroutine: %d\n", id)
    fmt.Printf("Finished handling request %d\n", id)
}
```

```go
func main() {
    http.HandleFunc("/", handler)
    fmt.Println("Server running at http://localhost:8080")
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

then on a terminal type:

```
curl http://localhost:8080 &
curl http://localhost:8080 &
curl http://localhost:8080 &
```

You'll see logs indicating that multiple requests are being handled concurrently in separate goroutines. In the book code examples you will find a complete example which tracks memory usage and total goroutine count.

Concurrent Web Client Requests

Speaking of HTTP there is a common pattern in the client side of a HTTP request, for crawlers where the return of a given request is sent through a single channel to be processed.

Note that there is no explicit limit or control of number of concurrent goroutines and no blocking requests after spawning them for each item on the urls list. The goroutines ends when the client request ends, there is no mechanism to interfere with them, it is unidirectional.

```go
func fetch(url string, ch chan<- string) {
    resp, _ := http.Get(url)
    ch <- resp.Status
}

func main() {
    urls := []string{"https://example.com", "https://golang.org"}
    ch := make(chan string)

    for _, u := range urls {
        go fetch(u, ch) // named function with parameters
    }

    for range urls {
        fmt.Println(<-ch)
    }
}
```

This example fetches multiple pages and pipe their HTTP return code through a channel. That could be the website body and code for later processing.

The runtime of each goroutine depends on latency and size of the page fetched and when it is over the only signal is the data through the channel. We can't interrupt them as there is no bidirectional communication.

Background Workers

A background worker may navigate across many states that communicates across the system. It also may be required to finish following a command or condition that is not time or error based.

For coordination, it may live within a tree of processes that are required to fail or stop all along.

Hence, an external way to communicate with the goroutine is shown below. Remember contexts ? Before explore them, let's see an application:

```go
func backgroundWork(ctx context.Context) { // always receive a Context
    ticker := time.NewTicker(time.Second)
    defer ticker.Stop()
    for {
        select {
        // the Context's has a Done() method
        // which returns a channel
        case <-ctx.Done():
            return
        case t := <-ticker.C:
            fmt.Println("Tick at", t)
        }
    }
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    go backgroundWork(ctx)
    time.Sleep(3 * time.Second)
    cancel() // stop the goroutine using its Context.
}
```

These basic building blocks allow for more complex and robust coordination and concurrent architecture. But concurrency is tricky and we will see some common pitfalls below.

Race conditions and shared memory

Because goroutines run concurrently and can share memory, race conditions are a real risk.

Example: Race Condition

```go
var counter int // global variable

func increment() {
    for i := 0; i < 1000; i++ {
        counter++
    }
    fmt.Println(counter)
}

func main() {
    go increment()
    go increment()
    time.Sleep(time.Second)
    fmt.Println("Counter:", counter) // inconsistent result
}
```

Run this for some times and see that each time the order of `fmt.Println(counter)` executed will change. The memory is not being changed in the order that it seems in the code, first to 1000 then to 2000. The end result is the same but that hides this complexity. That could mean data corruption or loss in a real world setting.

To fix it we need to employ synchronization functions. They don't mean to slow down the code but to surround and protect regions in a way that only one read and write is done at time.

Fix: Use `sync.Mutex`

```go
var mu sync.Mutex // Mutual Exclusion locks - mutex - lives in the sync
↪ package of the standard library
var counter int

func increment() {
    for i := 0; i < 1000; i++ {
        mu.Lock() // only one goroutine at time, the others will wait
        counter++
        mu.Unlock() // from here on the lock is free
    }
}
```

Our program now looks like this:

```go
package main

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

var mu sync.Mutex
var counter int

func increment() {
  for i := 0; i < 1000; i++ {
    mu.Lock()
    counter++
    mu.Unlock()
  }
  fmt.Println(counter)
}

func main() {
  go increment()
  go increment()
  time.Sleep(time.Second)
  fmt.Println("Counter:", counter)
}
```

Synchronization primitives when multiple goroutines access shared data. There are primitives for Read and Write locks like `RWMutex` worth exploring when you need more granularity than just a full lock - as in if a data can be always read but has to be sequentially written.

Debugging and Tooling

Concurrency bugs can be subtle, non-deterministic, and hard to reproduce. Go provides robust tooling to help catch these issues early and keep systems observable in production.

Detecting Leaks and Concurrency Bugs

1. **Race Detector**

```
go run -race main.go
```

Detects unsynchronized memory access between goroutines at runtime. Essential during development and CI.

2. **Goroutine Dumps**

```
import "runtime"
fmt.Println("Goroutines:", runtime.NumGoroutine())
```

Use to monitor for leaks or runaway spawns. Stack traces via:

```
curl http://localhost:6060/debug/pprof/goroutine?debug=2
```

3. **pprof**

```
import _ "net/http/pprof"
```

Automatically exposes `/debug/pprof` endpoints for inspecting heap, goroutines, mutex contention, and more. Integrates with go tool pprof for offline analysis.

4. **Execution tracing**

```
go run -race main.go
curl http://localhost:6060/debug/pprof/goroutine?debug=2
```

Use runtime/trace or pprof/trace for deep event timelines and scheduling behavior.

Design for Observability

Debugging concurrent programs is not just about tools—it's about designing systems that are easy to observe.

Log contextually, propagate trace IDs, expose metrics and goroutine counts, and isolate state changes behind clear synchronization primitives. Many seasoned Go teams evolve their designs after their first real goroutine leak or race condition.

Observability isn't a feature you add later—it's something you build into the concurrency model from day one.