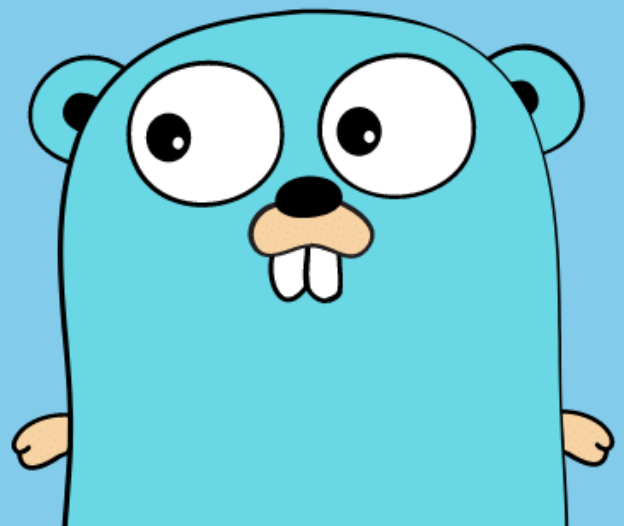
Nebenläufigkeit in Go

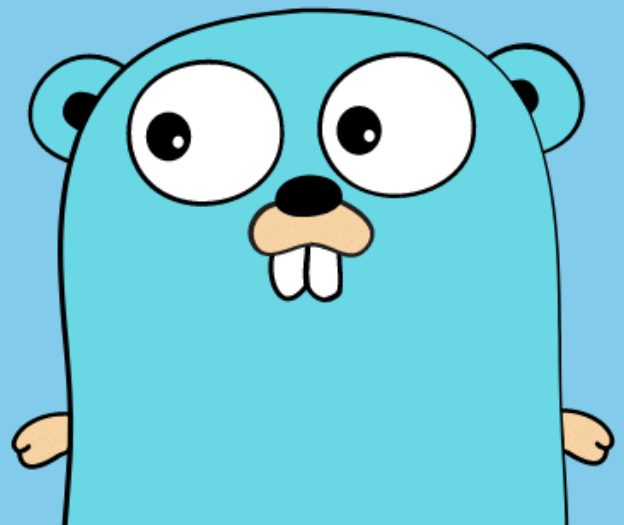
Till Burdorf



Agenda

1. Grundlagen der Nebenläufigkeit in Go
   - Go-Routinen
   - Channels
   - Select
   - Sync-Package
   - Waitgroups
2. Implementierung
   - Architektur
   - Tests & Benchmarks

Grundlagen der Nebenläufigkeit in GO



Go-Routines

Definition

- leichte, nebenläufig ausgeführte Codeeinheiten
- nicht immer echt parallel

Verwaltung von Go-Routinen

- Durch GO Runtime und Scheduler

Erstellung von Go-Routinen

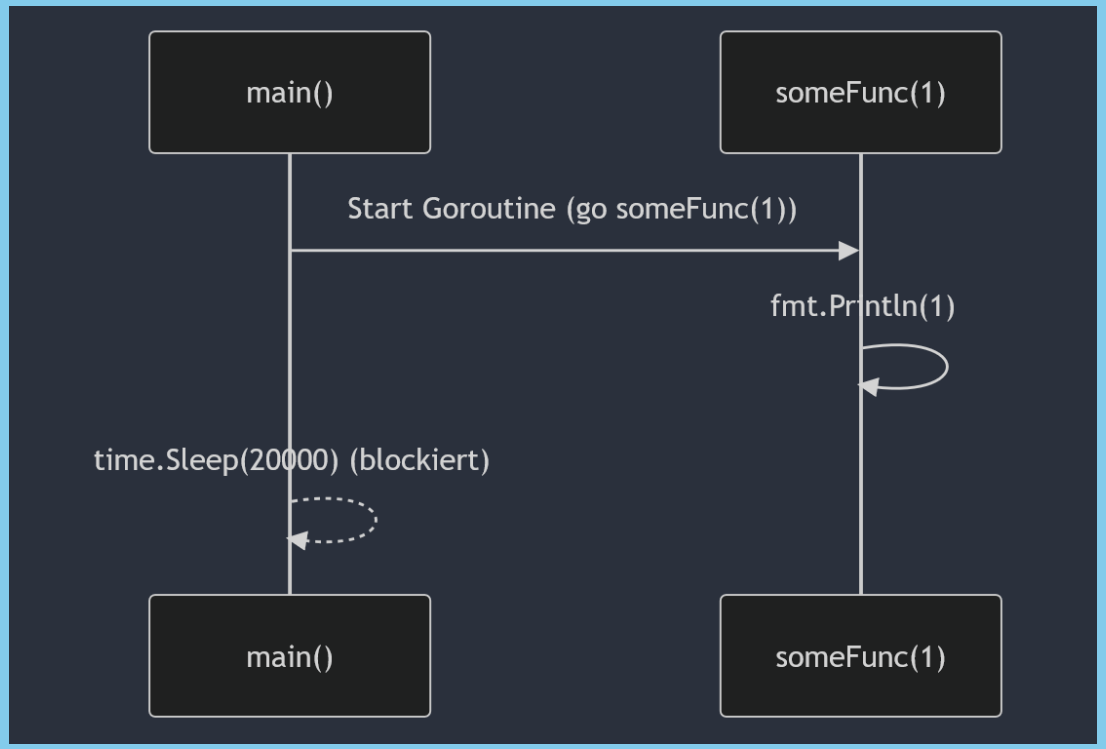
- Schlüsselwort: go
- asynchron

OS-Thread vs Go-Routine

	OS-Thread	Go-Routine
Stack Management	feste Größe	dynamisch
Scheduling	N:N scheduling	M:N scheduling
Concurrency Model	Verwendung von Shared Memory mit Locks und Mutex	Channels ermöglichen einfache Kommunikation und Synchronisation -> Vermeiden Deadlocks und Race Conditions
Performance	schwergewichtig, hoher Overhead durch Kontextwechsel, begrenzte Skalierbarkeit bei vielen Threads	leichtgewichtig, effiziente Planung durch Go-Runtime, hohe Skalierbarkeit und Ressourcennutzung

Go-Routines am Beispiel

```go
func someFunc(input int) {
    fmt.Println(input)
}

func main() {
    go someFunc(1)
    time.Sleep(20000)
}
```



„Don‘t communicate by sharing memory, share memory by communicating“

Channels

Definition

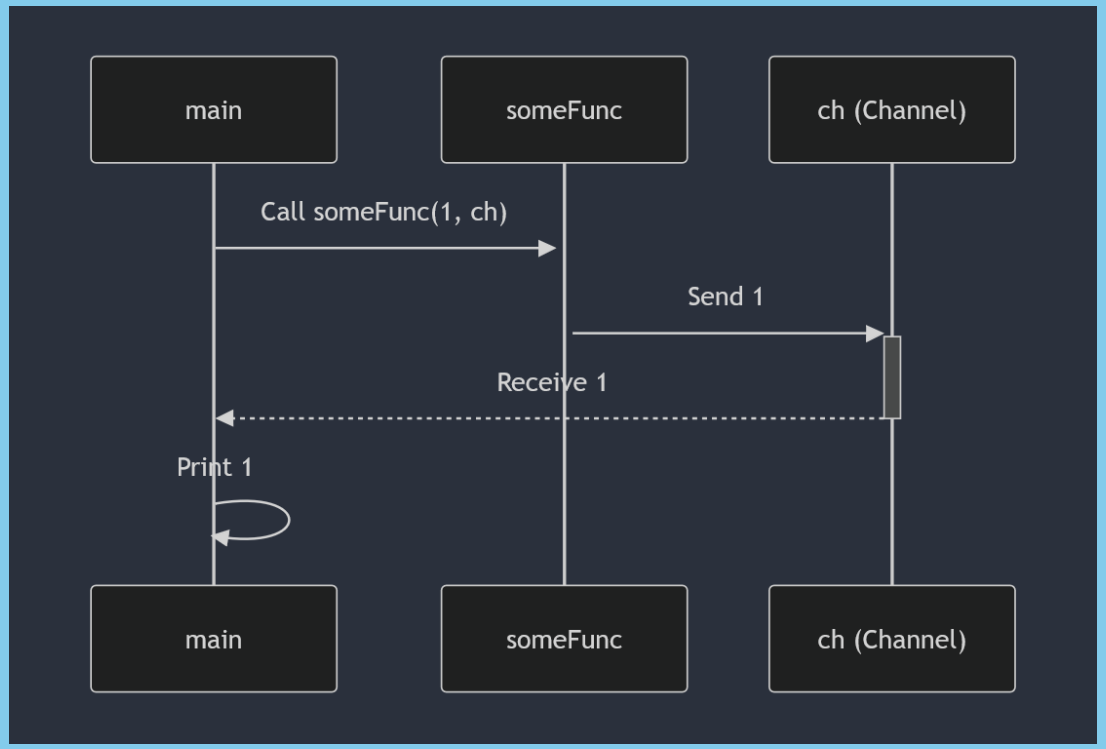
- Kommunikationsmittel, die es Go-Routinen ermöglichen, sicher und synchron Daten auszutauschen
- Kommunikation zwischen Go Routinen ohne Shared Memory

Unbuffered vs Buffered

- Unbuffered: Blockierend
- Buffered: Blockierend bei vollem Puffer

Channels am Beispiel

```go
func someFunc(input int, ch chan int) {
    // write input into the Channel
    ch <- input
}

func main() {
    // initate a Channel
    ch := make(chan int)

    // start Go Routine
    go someFunc(1, ch)
    // read the data from the channel into chContent
    chContent := <- ch
    fmt.Println(chContent)
}
```



Select

- Definition
  - ermöglicht es gleichzeitig auf mehrere Kanaloperationen zu warten
  - blockiert bis eine Operation ausgeführt wird

- Deadlocks
  - Timeout und Default können festgelegt werden
  - verhindert die Blockierung des Programmes

Select am Beispiel

```go
func someFunc(input int, ch chan int){
    ch <- input
}

func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)

    go someFunc(1, ch1)
    go someFunc(2, ch2)


    select {
    case msg1 := <-ch1:
            fmt.Println(msg1)
    case msg2 := <-ch2:
        fmt.Println(msg2)
    case <-time.After(3 * time.Second):
        fmt.Println("Timeout reached!")
    }
}
```

- someFunc sendet an unterschiedliche Channels
- select führt den Fall aus, der zuerst eintritt
- zufällige Wahl bei gleichzeitiger Bereitstellung
- timeout nach 3 Sekunden

sync–package

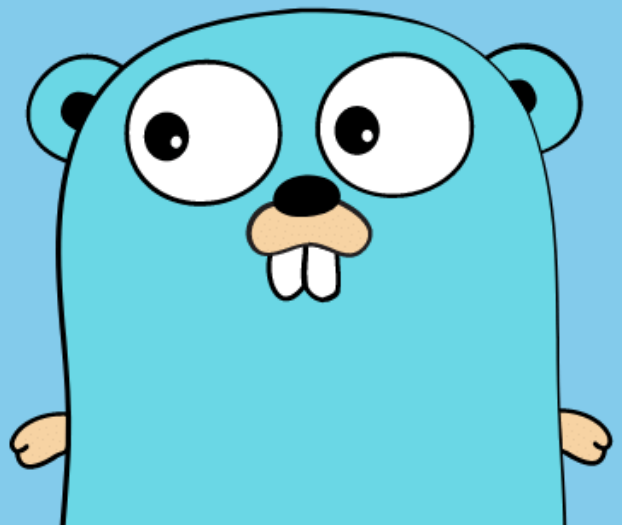
- **sync** bietet grundlegende Synchronisationsprimitiven
- enthält Mechanismen wie z.b. Mutex
- die Typen **Once** und **WaitGroup** sind enthalten.
- die meisten Synchronisationsmechanismen sind für niedrigschwellige Bibliotheksroutinen gedacht
- höherstufige Synchronisation sollte über Kanäle und Kommunikation erfolgen

Waitgroups

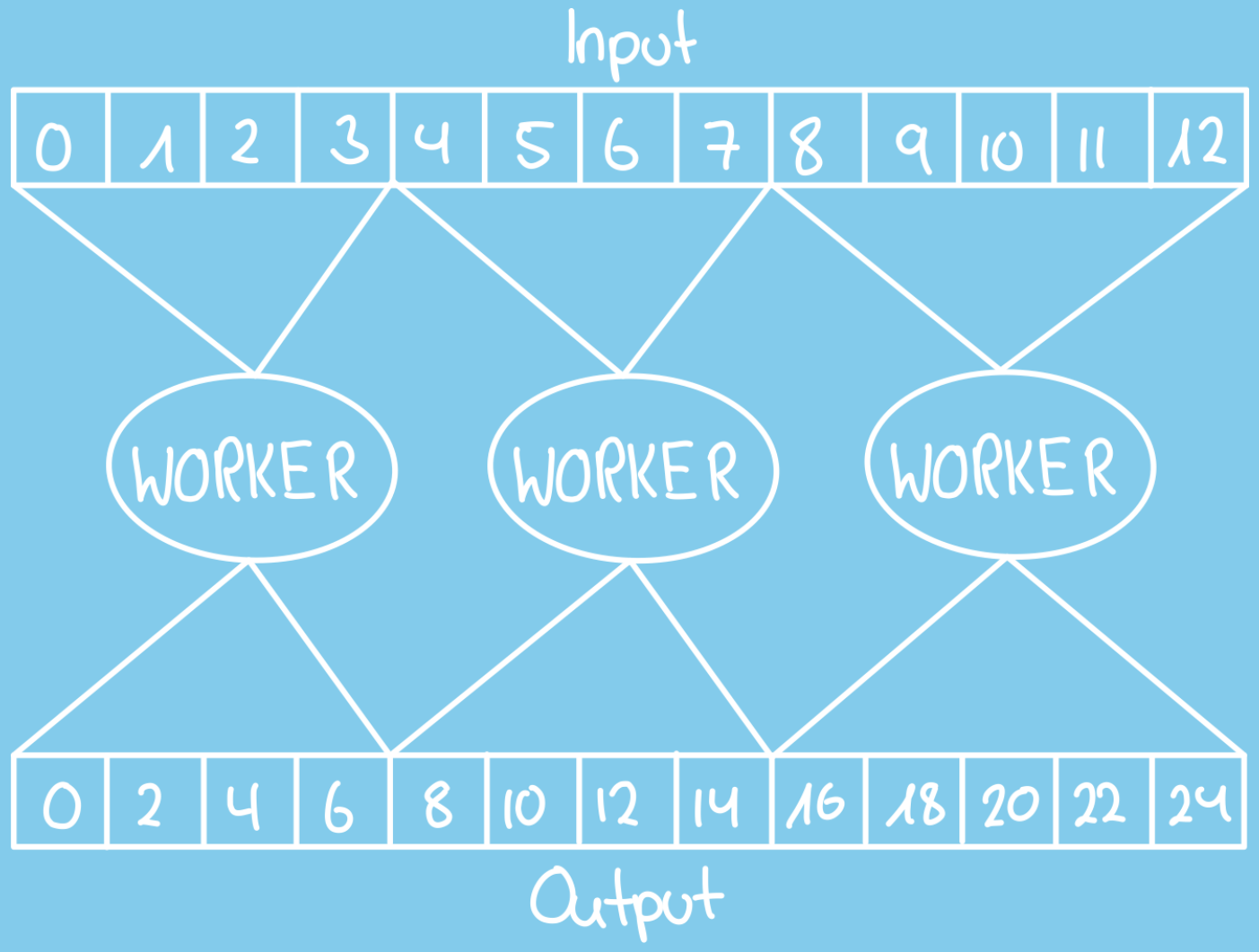
```go
func someFunc(input int, wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Println(input)
}

func main() {
    var wg sync.WaitGroup
    wg.Add(1)
    go someFunc(1, &wg)
    wg.Wait()
}
```

- erstellt eine Waitgroup
- wg.Add(1) erhöht wg
- defer wg.Done() reduziert wg
- wg.Wait wartet bis wg 0 ist

Implementierung



Architektur: parallelMap()



Architektur: parallelReduce()