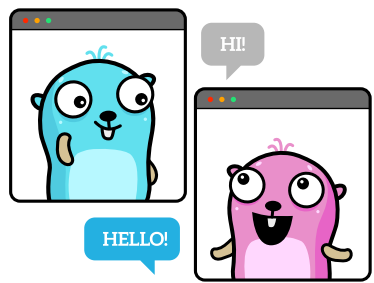
CS3211 Tutorial 5

Goroutines and Channels

(AY 25/26 Semester 2)

March 12, 2026



(Compiled by Benson, thanks to all past and current TAs!)

Contents

Contents

Welcome to the Go World!

C++ (50%)

- Threads
- Synchronization
- Atomics
- Memory Ordering
- Debugging
- Lock-Free Programming

Go (25%)

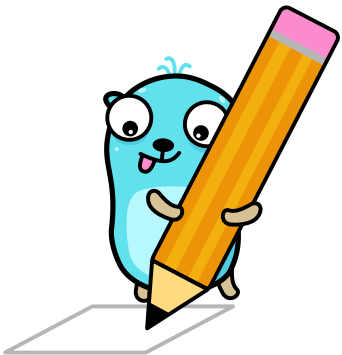
- Lightweight Co-routines (Goroutines)
- Channels and Message Passing
- Etc

Rust (25%)

- Compile-time safety checking
- Safe futures / async
- Safe data parallelism
- Etc

Why Go?

- In Go, sharing resources is done by **explicit message passing (Channels)**.
  - Idea: No shared memory $\Rightarrow$ No data race.



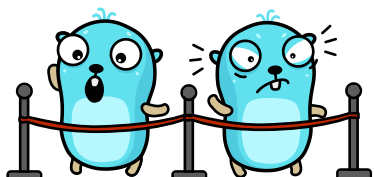
Let's play with Channels!

What is the output of this program?

```
1  package main
2  import "fmt"
3
4  func main() {
5    c := make(chan int)
6    c <- 1
7    fmt.Println(<-c)
8  }
```

Link to godbolt

Deadlock. How to fix?

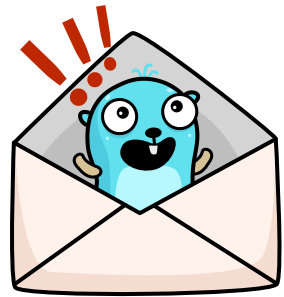


Let's play with Channels!

What is the output of this program?

```go
package main
import "fmt"

func main() {
  ch := make(chan int)
  go func() {
    ch <- 1
    ch <- 1729
    close(ch)
  }
  fmt.Println(<-ch)
  fmt.Println(<-ch)
  fmt.Println(<-ch)
  fmt.Println(<-ch)
}
```

Link to godbolt



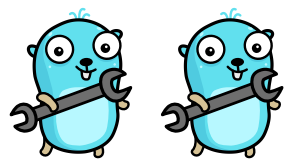
Always 1 1729 0 0.

Let's play with Channels!

What is the output of this program?

```go
package main
import "fmt"

func producer(val int, c chan<- int) {
  c <- val
  c <- val
}

func main() {
  c := make(chan int)

  go producer(1, c)
  go producer(7, c)

  fmt.Println(<-c + <-c)
}
```

Link to godbolt



Could be $1 + 1 = 2$,
$1 + 7 = 8$ or $7 + 7 = 14$.

Let's play with Channels!

`select` statement:

- Without default case, it might block.
- With default case, it never blocks.
- Commonly wrapped in a `for {}` block.

```
1  func ooga(eventA, eventB chan int) {
2    select {
3      case a := <-eventA:
4        fmt.Println("Received eventA")
5      case b := <-eventB:
6        fmt.Println("Received eventB")
7    }
8  }
```

Let's play with Channels!

`for...range` statement:

- Keep reading values from the channel.
- Exits when the channel is closed.

```
1  sum := 0
2  for value := range c {
3    sum += value
4  }
5  fmt.Println(sum)
```

Contents

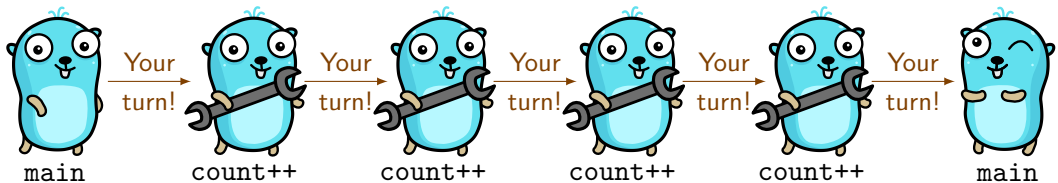
Concurrent Counter

```go
package main

import "fmt"

func main() {
  count := 0

  for i := 0; i < 1000; i++ {
    go func() {
       count++
    }()
  }

  fmt.Println("Count: ", count)
}
```

Link to godbolt

Concurrent Counter

How to synchronize?



Concurrent Counter

How did a deadlock appear?

Increment Goroutine

```
1  count := <-ch
2  count++
3  ch <- count
4  wg.Done()
```

Increment Goroutine

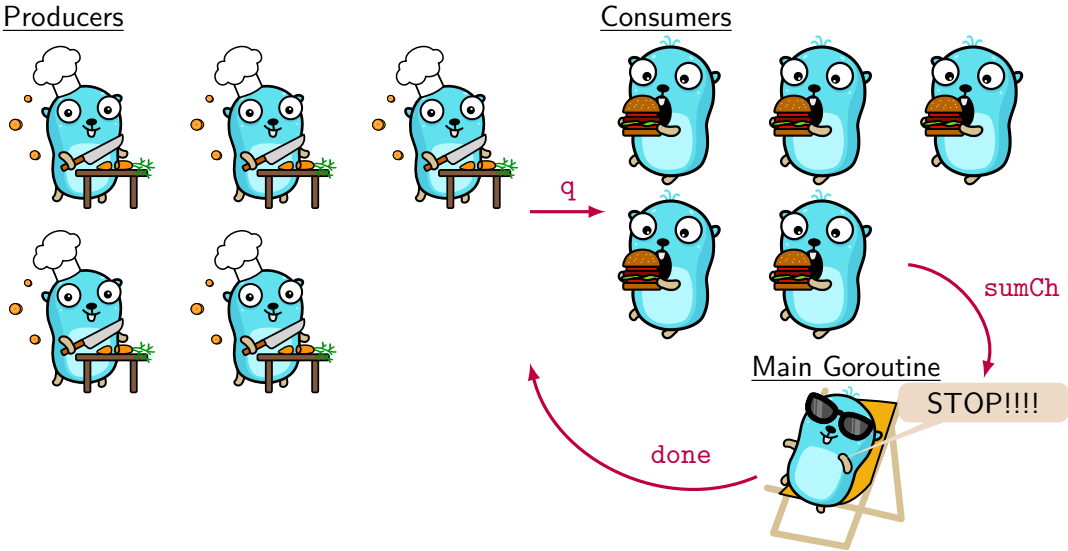
```
1  count := <-ch
2  count++
3  ch <- count
4  wg.Done()
```

Main Goroutine

```
1  ch <- 0
2  wg.Wait()
3  fmt.Println("Count: ", <-ch)
```

Contents

Multi-Producer Multi-Consumer Queue



Multi-Producer Multi-Consumer Queue

```go
func producer(done chan struct{}, q chan<- int) {
  for {
    select {
    case q <- 1: // keeps incrementing...
    case <-done: // until stopped
      return
    }
  }
}

func main() {
  done := make(chan struct{})
  q := make(chan int)
  // TODO spawn producers and consumers
  time.Sleep(time.Second)
  close(done) // stop all producers and consumers
  // TODO merge into global sum
  sum := 0
  fmt.Println("Sum: ", sum)
}
```

Link to godbolt

Can you also create a non-blocking version of the queue?

Multi-Producer Multi-Consumer Queue

Operation	Channel state	Result
Read	`nil`	Block
	Open and Not Empty	Value
	Open and Empty	Block
	Closed	<default value>, false
	Write Only	Compilation Error
Write	`nil`	Block
	Open and Full	Block
	Open and Not Full	Write Value
	Closed	**panic**
	Receive Only	Compilation Error
`close`	`nil`	**panic**
	Open and Not Empty	Closes Channel; reads succeed until channel is drained, then reads produce default value
	Open and Empty	Closes Channel; reads produces default value
	Closed	**panic**
	Receive Only	Compilation Error

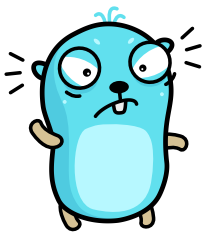
```
1  c := make(chan int)
2  go func() {
3    c <- 1
4    close(c)
5  }
6  go func() {
7    c <- 1
8    close(c)
9  }
```

panic(err)

Contents

The context package

```go
// cancel (2nd return value) can be used to manually cancel() before the timeout
ctx, cancel := context.WithTimeout(context.Background(), time.Second)

// from producers and consumers
<-ctx.Done()

// from main goroutine
cancel()
```



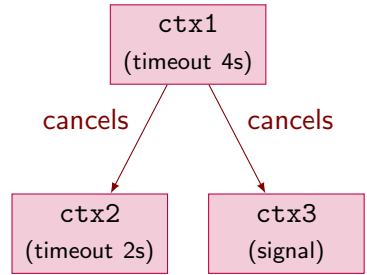
Exit Conditions:

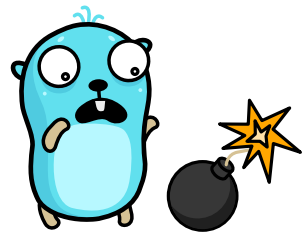
- The spawner of the goroutines receives an external signal to terminate.
- There are two dependant goroutines that are running concurrently but one fails / is cancelled.
- Some timeout has been reached / a deadline has passed (e.g., deadline while waiting for some response).

The context package

```
1  func main() {
2    ctx, _ := context.WithTimeout(
3        context.Background(), 4 * time.Second)
4    ctx2, _ := context.WithTimeout(ctx, 2 * time.Second)
5    go func() {
6      <-ctx2.Done()
7    }()
8    ctx3, cancel := context.WithCancel(ctx)
9    go func() {
10     <-ctx3.Done()
11   }()
12   go func() {
13     <-handleSigs()
14     cancel()
15   }
16   <-ctx.Done()
17 }
```

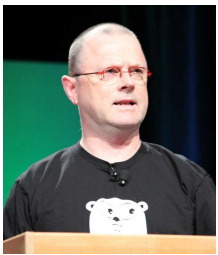
Link to godbolt





Summary

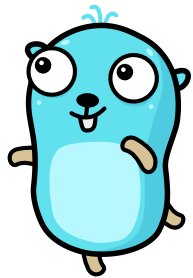
- A taste of concurrency in Go.
  - Wait Groups, Channels, Select, Context.



"Don't communicate by sharing memory; share memory by communicating." — Rob Pike

Attendance

Session ID: 1940155
CS3211 Tutorial 5
12 March 2026 14:00-16:00
BIZ2-02-02 - SEMINAR ROOM 2-2



That's it!

Anonymous Feedback (throughout the semester):

https://forms.gle/6a9T4t88wNwYxG4G8

The link to the tutorial slides will be posted in telegram.