Agenda – Lab 2

- Goroutines in-depth
- Channels
- Concurrency control
- Mutexes

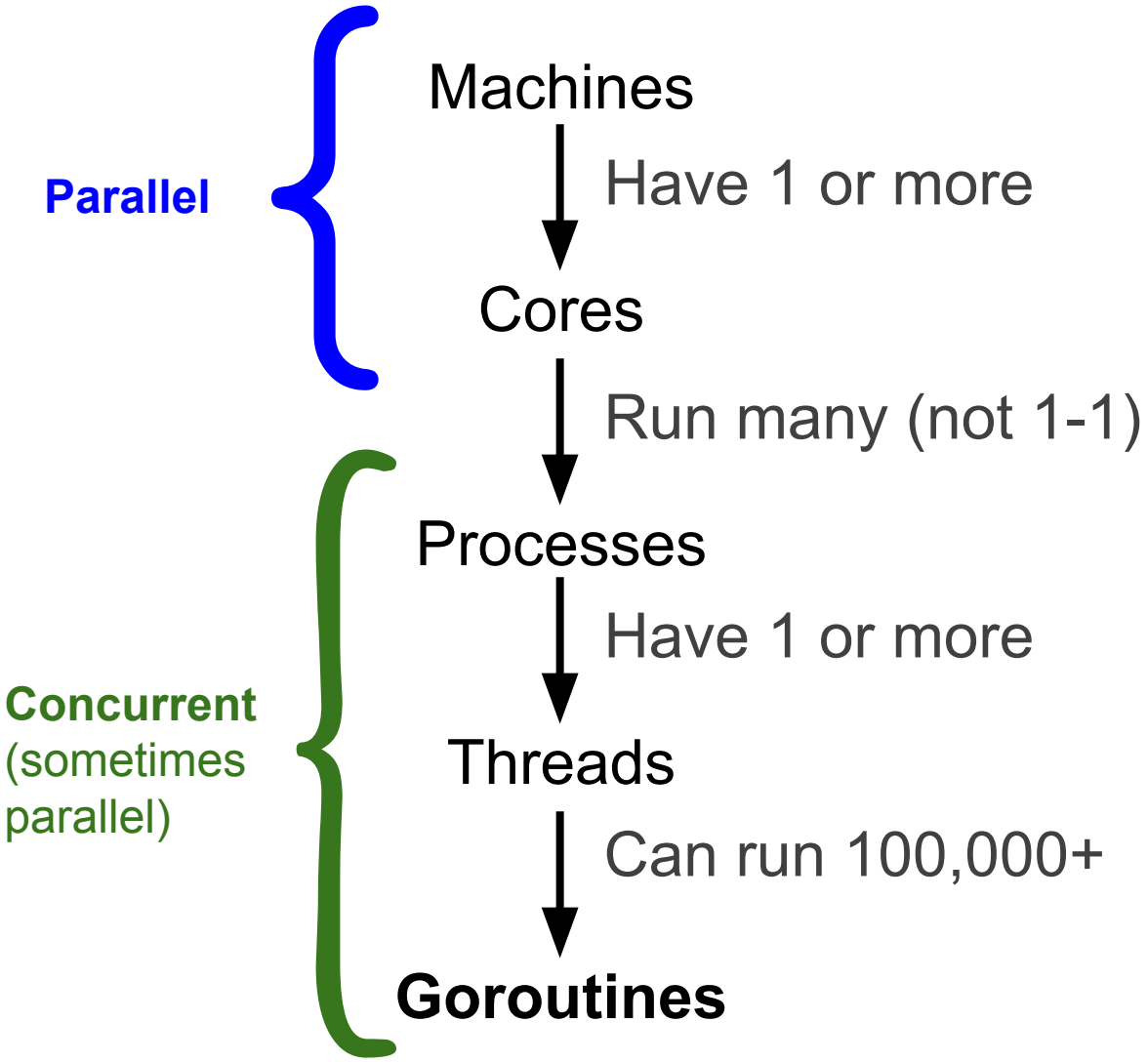
Once again, Go resources are your friends!

- A Tour of Go: https://go.dev/tour/list
- Go by Example: https://gobyexample.com
- Effective Go: https://go.dev/doc/effective_go

Recall that…

- Concurrency vs. parallelism
  - Concurrency is the illusion of multiple tasks running in parallel via interleaving of instructions (i.e. the CPU rapidly switching between different tasks – **only one** task is being executed at any given moment)
  - Parallelism is multiple tasks running at the same moment
  - Concurrency is **single-core**, parallelism is **multi-core** (or multi-machine)

Concurrency in Go



- Cheaper abstractions as you move downwards
- Machines and cores are parallel
- Threads and goroutines are concurrent (and maybe parallel on multicore machines)

Goroutines

- **Goroutines** are lightweight threads managed by Go runtime – creating them is cheap, costing little more than stack allocation (which is initially small). Having tens, even hundreds of thousands of goroutines is very possible.
- A goroutine is a function executing concurrently with other goroutines in the same address space.
- To create a goroutine, use the keyword go followed by a **function invocation**.

```
func main() {
    // add function is called as goroutine, executing concurrently with calling one
    go add(20, 20)
    add(20, 10) //this…
    // … more work here in main
}

func add(x int, y int) {
    fmt.Println(x+y) //…is concurrent with this
}
```

https://go.dev/doc/effective_go#concurrency

Demo – why doesn't this print?

```
package main

import (
    "fmt"
    "time"
)

func hello() {
    fmt.Println("Hello world")
}

func goodbye() {
    fmt.Println("Goodbye world")
}

func main() {
    go hello()
    go goodbye()
}
```

Goroutines (cont'd)

- The goroutine will exit silently upon completion (similarly to the Unix shell's & notation for running a command in the background).
- Goroutines can also be started for anonymous function calls.

```
// Anonymous function that does not take parameter - called as Goroutine
go func() {
    fmt.Println("Another Goroutine")
}()    // last pair of parenthesis is the actual function invocation

// Anonymous function that takes parameters - called as Goroutine
go func(msg string) {
    fmt.Println("Hello " + msg)
}("World")    // last pair of parenthesis is the actual function invocation
```

- Goroutines are great, but so far not that practical without a way to signal completion and communicate – this is where **channels** come in.

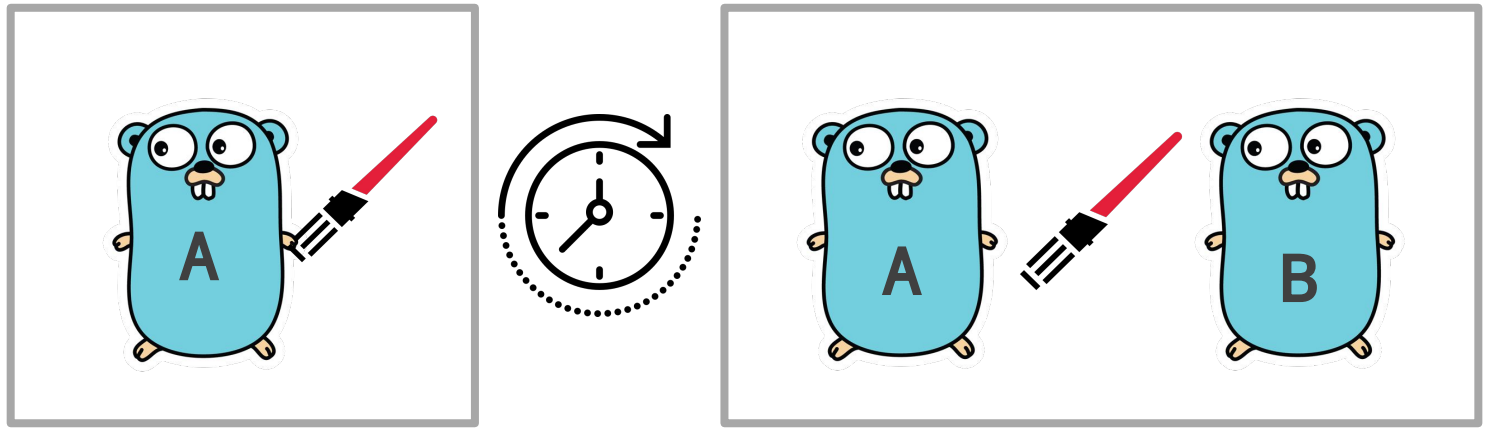
Channels

- Channels are a **typed conduit** through which you can send and receive values – think of them as communication pipes that connect concurrently executing goroutines.
- By default, sends and receives **block** until the other side is ready. Sends to a buffered channel block only when the buffer is full.
- Channels are used for **communication** and **synchronization**

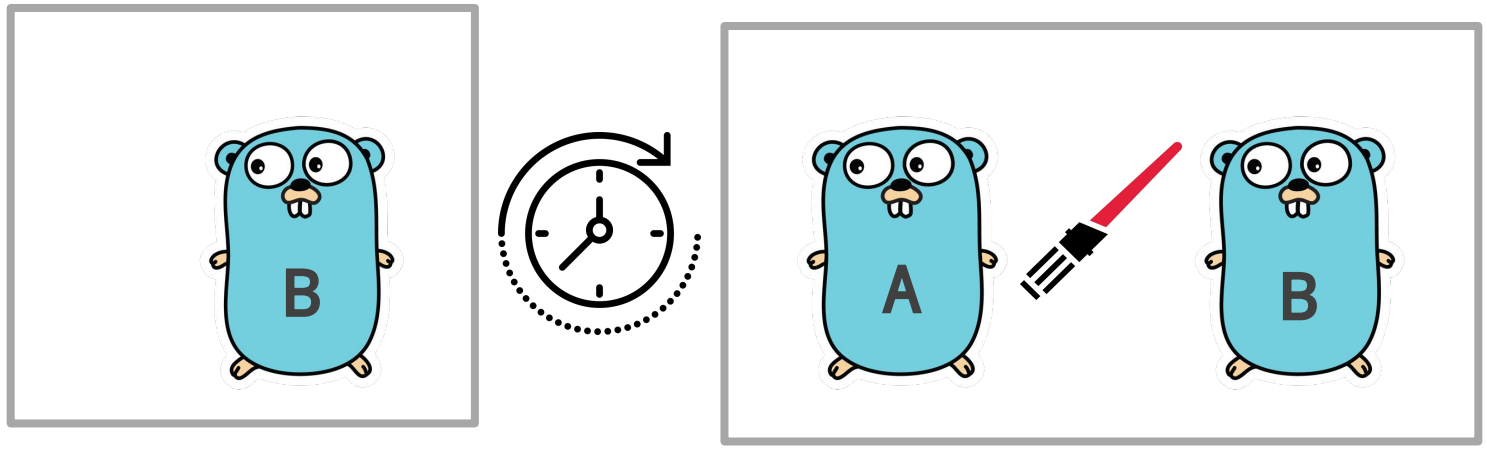
```
ch := make(chan int)      // default unbuffered channel of integers
ch := make(chan int, 10) // buffered channel of integers with capacity=10
```

Goroutine 1 **Channel** *Goroutine 2*

Unbuffered channels are blocking

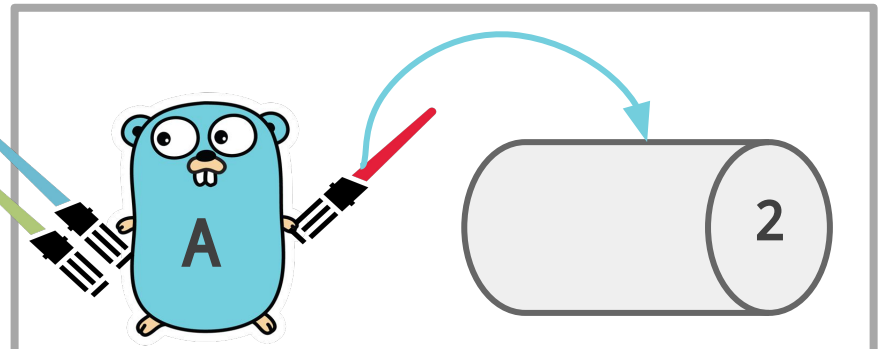


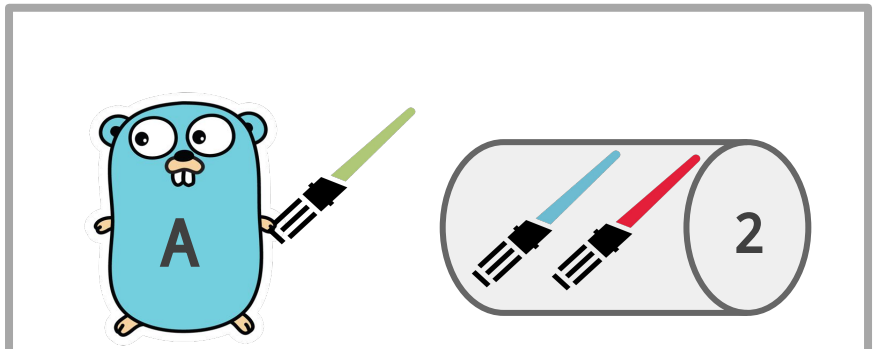
Gopher A tries to send to the channel but gopher B isn't there to read, so gopher A has to wait.

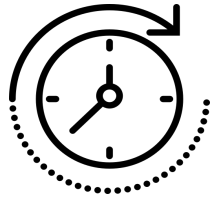


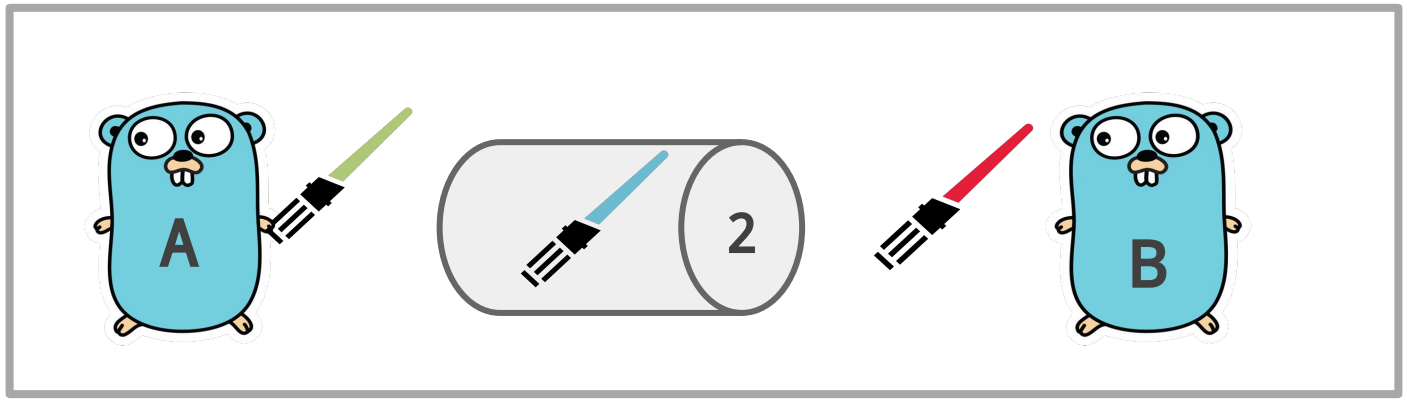
Gopher B tries to read from the channel but gopher A isn't there to send, so gopher B has to wait.

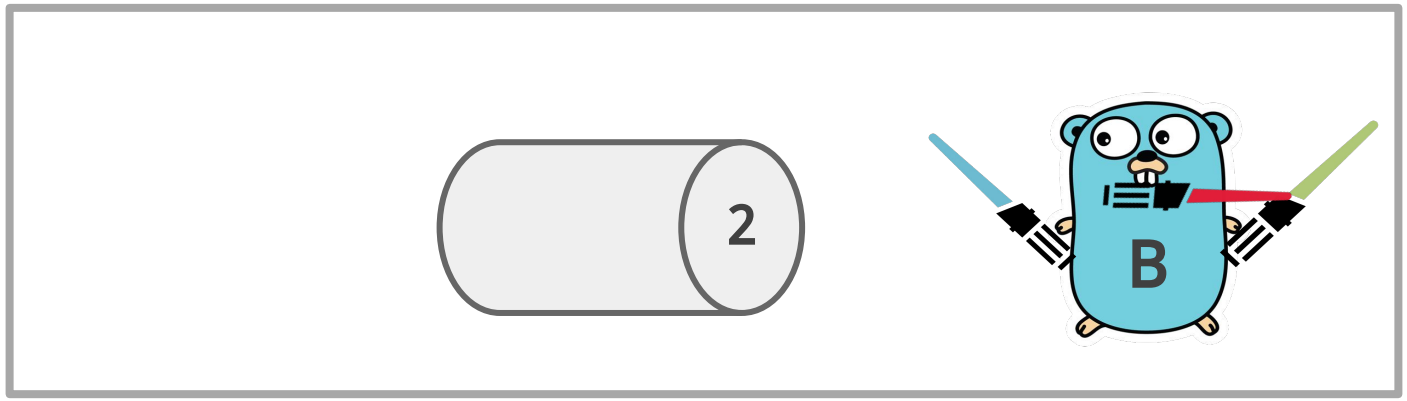
Buffered Channels











Sending to the buffered channel will block when the buffer is full.

The buffer has a capacity of 2 so gopher A has to wait for gopher B to free up room in the channel.

Channels

- Like maps, channels are allocated with `Make` and the resulting value acts as a reference to the underlying data structure. An optional integer parameter sets the buffer size of the channel.

```
ch := make(chan int)      // default unbuffered channel of integers
ch := make(chan int, 10) // buffered channel of integers
```

- Unbuffered (synchronous) channels combine communication with synchronization – guarantees that two goroutines are in a known state (without explicit locks or condition variables).

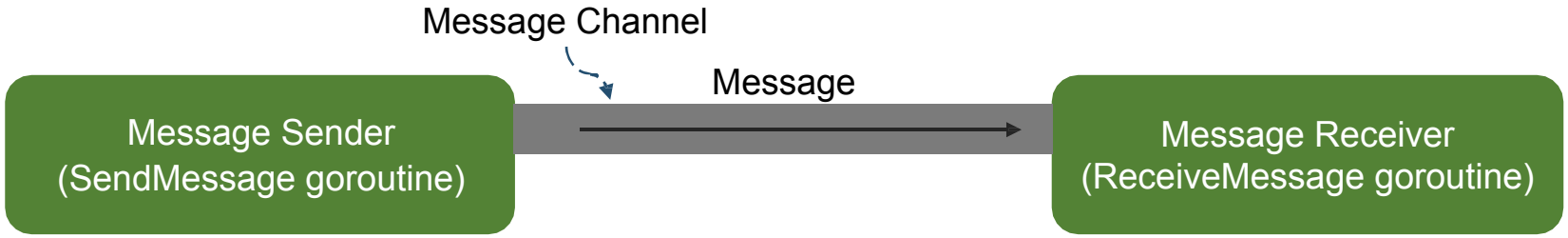
"Do not communicate by sharing memory; instead, share memory by communicating."

Channels – synchronization

- `time.Ticker` is an example of using channels for synchronization instead of communication.

```
// Return a channel that emits signals at regular intervals specified by d.
func heartbeatTicker(d time.Duration) chan struct{} {
    ch := make(chan struct{})
    go func() {
        for {
            time.Sleep(d)
             ch <- struct{}{}
        }
    }()
    return ch
}
```

Channels – communication




```
msgCh := Make(chan string)

go SendMessage(msgCg)
go ReceiveMessage(msgCg)
```

```
// goroutine that sends the message
// msgCh chan string is a channel of strings
func SendMessage(msgCh chan string) {
   msgCh <- "sending @" + time.Now().String()
}
```

```
// goroutine that receives the message
func ReceiveMessage(msgCh chan string) {
   message := <- msgCh
   fmt.Println(message)
}
```

Channels (cont'd)

- A sender can `close` a channel to signal that no more values will be sent.

```
func doStuff(ch chan int) {
    // send values to channel until done
    close(ch)
}
```

- Receivers can test whether a channel has been closed.

```
value, ok := <-ch
```

- The loop `for i := range c` will receive values from a channel repeatedly until it is closed.

```
func main() {
    ch := make(chan int, 10)
    go doStuff(ch)
    for i := range ch {
        fmt.Println(i)
    }
}
```

Channels – select

- The `select` statement lets a goroutine wait on multiple communication operations.

- A `select` blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

```
func process(data chan []byte, command chan int) {
    for {
        select {
        case d := <-data:
            // process the byte slice 'd'
            fmt.Println("Received data:", d)
        case cmd := <-command:
            fmt.Println("Received command:", cmd)
            return
        }
    }
}
```

Default select case

- The `default` case in a `select` is run if no other case is ready.

```
// Return a channel that sends the current time at rate 'd', dropping ticks
// if the reader falls behind.
func latestTimeTicker(d time.Duration) chan time.Time {
    ch := make(chan time.Time, 1) // 1-element time buffer
    go func() {
        for {
            time.Sleep(d)
             select {
             case ch <- time.Now():
             default: // Drop ticks if the reader falls behind.
             }
        }
    }()
    return ch
}
```

Mutex

We've seen how channels are great for communication among goroutines.

But what if we don't need communication? What if we just want to make sure only one goroutine can access a variable at a time to avoid conflicts?

This concept is called *mutual exclusion*, and the conventional name for the data structure that provides it is *mutex*.

You might have also heard this called a *lock*.

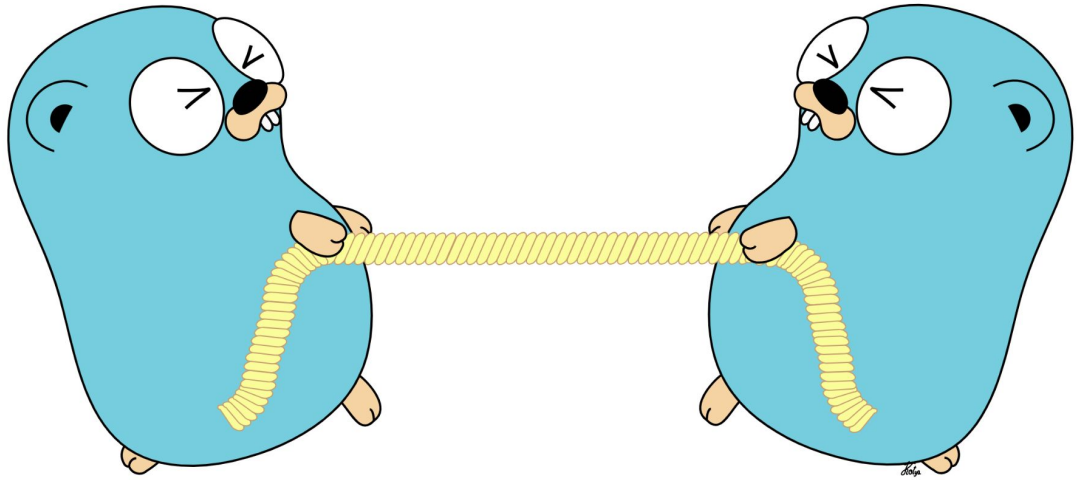
Go's standard library provides mutual exclusion with sync.Mutex and its two methods:

- `Lock`
- `Unlock`

Why do we need mutexes?

- A **race condition** occurs when two or more goroutines can access shared data and they try to change it at the same time – because the go runtime can swap between goroutines at any time, you don't know the order in which the goroutines will attempt to access the shared data.
- The result of the change in data is dependent on the order they get scheduled in, i.e. both threads are "racing" to access/change the data.

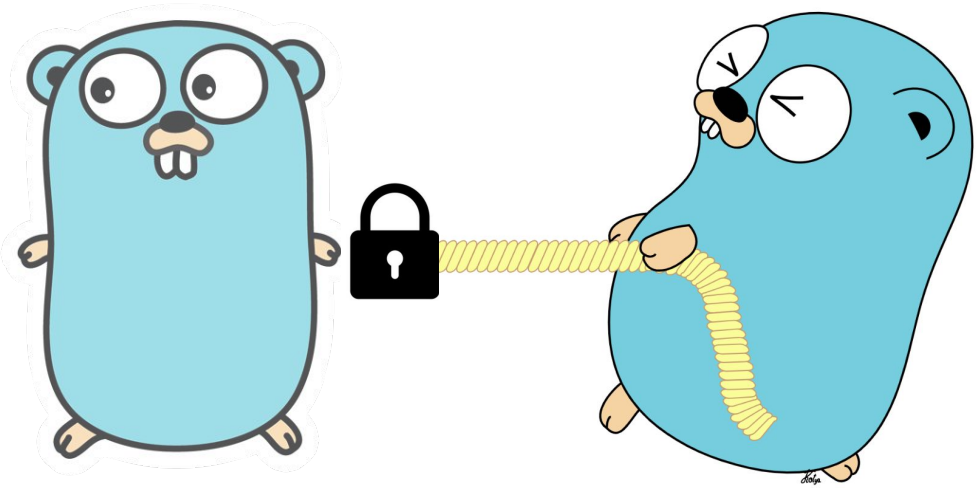
```
x := 10
y := x // Read from x.
y = y + 1
x = y // Set x to 11… right? What if another go routine tries to write to x in between when we read and write?
```



Why do we need mutexes?

- We can define a block of code to be executed in mutual exclusion by surrounding it with a call to `Lock` and `Unlock`.
- If a routine bypasses the lock, or forgets to release it, you run into trouble.

```
mutex := sync.Mutex{}
x := 10

mutex.Lock()

y := x // Read from x
y = y + 1

// The other goroutine tries to acquire the
// lock here to change x but cannot.

x = y // Set x to 11 like we expect. We know it
hasn't changed since we acquired the lock.

mutex.Unlock()
```



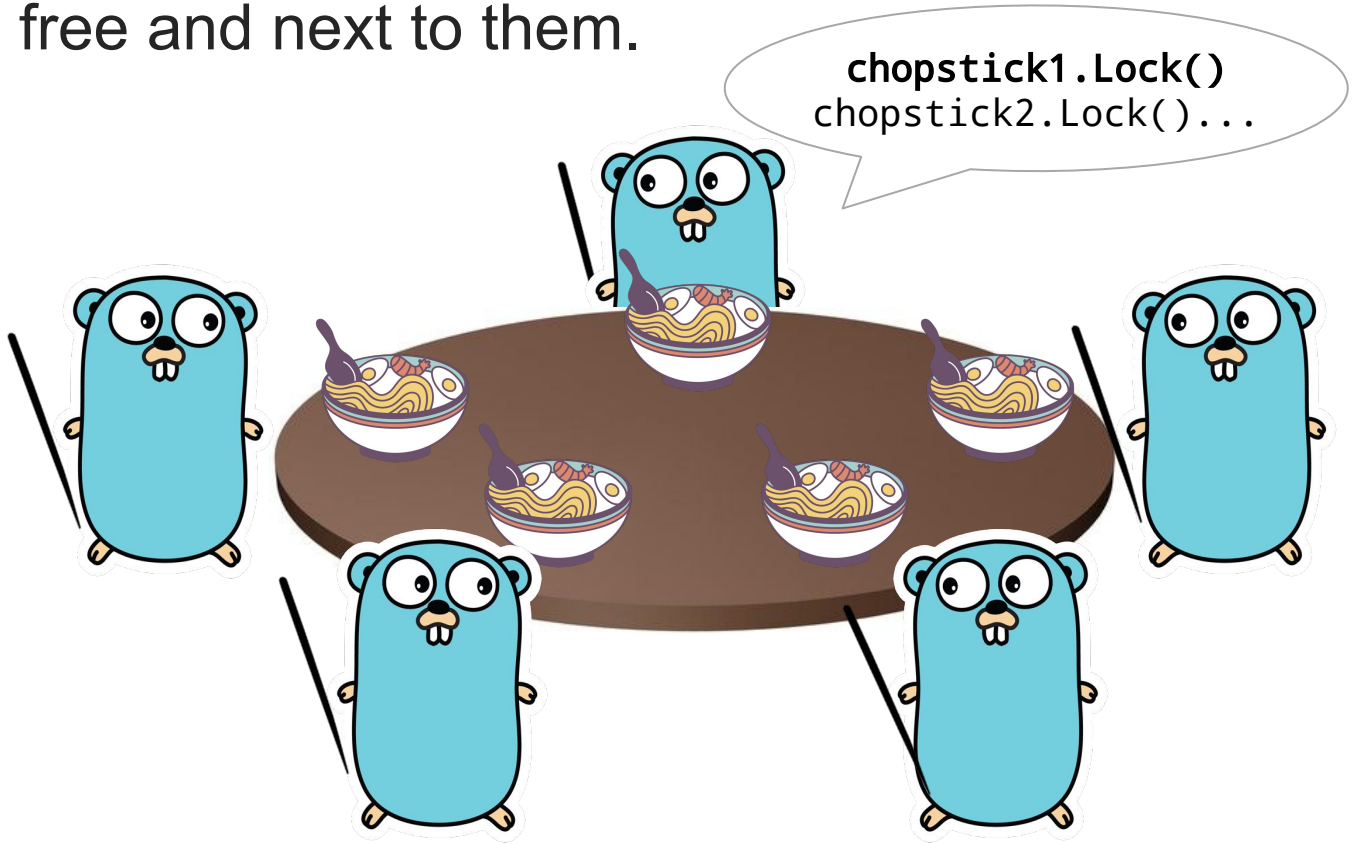
Race condition demo

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    x := 0
    for i := 0; i < 100_000; i++ {
        go func() {
            y := x
            y = y + 1
            x = y
        }()
    }
    time.Sleep(1 * time.Second)
    fmt.Println(x) //hmmmmmm
}
```

Deadlocks

- Deadlocks can occur because two (or more) routines are holding onto resources the other needs before they can proceed, and neither is willing to release their own resource first.
- Dining Gophers: Each gopher needs two chopsticks to eat the bowl of noodles, but can only grab a chopstick if it is free and next to them.

Each gopher immediately grabs the chopstick to their left but there aren't enough chopsticks, and thus all the gophers starve.



https://en.wikipedia.org/wiki/Dining_philosophers_problem

Deadlocks demo

- Let’s look at some actual code
- We’ve seen the problem, what are some ideas for how to solve it?

https://en.wikipedia.org/wiki/Dining_philosophers_problem