Concurrency

Concurrency

Concurrency is not Parallelism

- Concurrency is about dealing with a lot of things at once.
- Parallelism is about doing a lot of things at once.

Goroutines

- A goroutine is a lightweight thread managed by the Go runtime.
- A goroutine is a function capable of running concurrently with other functions.
- Goroutines are not threads. They are small, lightweight resources that require only a few bytes of memory.
- It's not uncommon to have hundreds, if not thousands, of goroutines.
- In contrast to system threads, which have a fixed stack size and therefore a finite limit to the number of threads you can run, you can run millions of goroutines

Synchronization Between Goroutines

Since goroutines run in the same address space, access to shared memory must be synchronized.

The `sync` package provides useful primitives to accomplish this, but there are also other primitives such as channels as well.

Your First Goroutine

Run the following program:

```
package main

import "fmt"

func main() {
    go sayHello()
}

func sayHello() {
    fmt.Println("hello")
}
```

Ø

What Went Wrong?

Concurrency & Goroutines

Goroutines execute concurrently. In this case, `main()` and `sayHello()` were running at the same time. `main()` finished and `sayHello()` never got a chance to run!

To solve this problem you need to use some sort of communication, or synchronization device, to signal that a goroutine is complete.

We typically do that with channels, or waitgroups, depending on whether you need to return a value from the function you're running.

Waitgroups

WaitGroups are defined in the `sync` package in the standard library. They're a simple, and powerful, way to ensure that all of the goroutines you launch have completed before you move on to do other things.

`waitgroup` documentation

Waitgroup Example

```
package main

import (
    "fmt"
    "sync"
)

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

func sayHello(wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Println("hello")
}
```

Ø

WaitGroups are also especially powerful when combined with closures (anonymous functions).

Closure Example

```
func main() {
    var wg sync.WaitGroup
    var urls = []string{
        "http://www.gopherguides.com/",
        "http://www.golang.org/",
        "http://www.google.com/",
    }
    // Increment the WaitGroup counter.
    wg.Add(len(urls))
    for i, url := range urls {
        // Launch a goroutine to fetch the URL.
        go func(i int, url string) {
            // Decrement the counter when the goroutine completes.
            defer wg.Done()
            begin := time.Now()
            // Fetch the URL.
            resp, _ := http.Get(url)
            fmt.Printf("%d) Site %q took %s to retrieve with status code of %d.\n", i, url, time.Si
        }(i, url)
    }
    // Wait for all HTTP fetches to complete.
    wg.Wait()
}
                                                                                              Ø
```

BUG TIME!

Closure Bug

```go
package main

import "sync"

func main() {
    var wg sync.WaitGroup
    wg.Add(5)
    for i := 0; i <5; i++ {
        go func() {
            println(i)
            wg.Done()
        }()
    }
    wg.Wait()
}
```

Ø

Can you spot the bug?

run on go playground

Closure Bug Solution

```go
package main

import "sync"

func main() {
    var wg sync.WaitGroup
    wg.Add(5)
    for i := 0; i <5; i++ {
        go func(i int) {
            println(i)
            wg.Done()
        }(i)
    }
    wg.Wait()
}
```

Ø

run on go playground

Closure Bug Explanation

The first example has the scope of `i` at the outer scope. Since the goroutines are being added to the execution stack, but not run, by the time they run, and access that memory space, `i` has already been changed.

Sending `i` in as an argument creates a new scope and copies the value to a new memory location to preserve the proper value.

Mutex

A mutex allows you to synchronize goroutines for safe access to the same shared memory. There are two types of mutexes, a full lock, and a read/write lock.

```
var m sync.Mutex // Make a new mutex

m.Lock() // lock the mutex

// Code in here that is protected by this mutex is safe to
// read or write for all go routines observing this mutex.

m.Unlock() // unlock the mutex
```

mutex documentation

Simple Mutex Example - Setup

```
var mu sync.Mutex
var counter int

set := func(i int) {
    mu.Lock()
    defer mu.Unlock()
    counter = i
}

get := func() int {
    mu.Lock()
    defer mu.Unlock()
    return counter
}
```

Ø

Simple Mutex Example - Routines

```
    go func() {
        for {
            time.Sleep(500 * time.Millisecond)
            i := get()
            fmt.Printf("counter: %d\n", i)
        }
    }()

    var wg sync.WaitGroup

    for i := 0; i <= 10; i++ {
        wg.Add(1)
        go func(i int) {
            set(i)
        }(i)
        time.Sleep(750 * time.Millisecond)
    }

    wg.Done()
```

Ø

Using A RW Mutex

Using `sync.RWMutex` we can lock access to data based on how that data is being used.

```
    quit := make(chan struct{})
    var mu sync.RWMutex
    var counter int

    set := func(i int) {
        mu.Lock()
        defer mu.Unlock()
        counter = i
    }

    get := func() int {
        mu.RLock()
        defer mu.RUnlock()
        return counter
    }
```

Ø

BUG TIME!

Mutex Bug

```
var wg sync.WaitGroup
wg.Add(1)
var mu sync.Mutex

var count int

go func() {
    for i := 0; i < 5; i++ {
        mu.Lock()
        defer mu.Unlock()
        count = i
        println(i)
    }
    wg.Done()
}()

wg.Wait()
```

Ø

run on go playground

Mutex Bug Solution

```
var wg sync.WaitGroup
wg.Add(1)
var mu sync.Mutex

var count int

go func() {
    for i := 0; i < 5; i++ {
        mu.Lock()
        count = i
        println(i)
        mu.Unlock()
    }
    wg.Done()
}()

wg.Wait()
```

Ø

run on go playground

Mutex Bug Explanation

The `defer` keyword only executes one it exits the scope it was called in. Because it is in a loop, the `defer` calls get put on the execution stack, but never get called because the loop doesn't exit scope.

Channels

Channels are a typed conduit through which you can send and receive values.

> *I can't tell you how many times I start with channels, and by the time I'm done, I've completely optimized them out. -- Cory LaNou*

Channels

The key to understanding how channels work is quite simple:

- When does a channel block?
- When does a channel unblock?

Simple Channel Example

```
package main

func main() {
    messages := make(chan string)

    // this go routine launches immediately
    go func() {
        // This line blocks until the channel is read from
        messages <- "hello!"
    }()

    // this line blocks until someone writes to the channel
    msg := <-messages

    println(msg)
}
```

Ø

Buffered Channels

By default channels are unbuffered -- sends on a channel will block until there is something ready to receive from it.

You can create a buffered channel - which allows you to control how many items can sit in a channel unread.

```
package main

func main() {
    // Adding a second argument to the make function creates a buffered channel
    messages := make(chan string, 2)

    // The program is no longer blocked on writing to a channel,
    // as it has capacity to write
    messages <- "hello!"
    messages <- "hello again!"

    // Reads are no longer blocked as there is already somethign to read from
    println(<-messages)
    println(<-messages)
}
```

Ø

Buffered Channels

Use buffered channels cautiously. They do not guarantee delivery of the message.

```
messages := make(chan string, 10)

go func() {
    for i := 0; i < 10; i++ {
        msg := fmt.Sprintf("message %d", i)
        messages <- msg
        fmt.Printf("sent: %s\n", msg)
    }
}()

m := <-messages
fmt.Printf("received: %s\n", m)
os.Exit(0)
// sent: message 0
// sent: message 1
// sent: message 2
// received: message 0
// sent: message 3
// sent: message 4
```

Ø

Closing Channels

You can signal that there are no more values in a channel by closing it. Receivers can test to see if a channel is closed during a read.

```
package main

func echo(s string, c chan string) {
    for i := 0; i < cap(c); i++ {
        // write the string down the channel
        c <- s
    }
    close(c)
}

func main() {
    // make a channel with a capacity of 10
    c := make(chan string, 10)

    // launch the goroutine
    go echo("hello!", c)

    for s := range c {
        println(s)
    }
}
                                                                                Ø
```

Using Select

You can use a `select` statement to allow a goroutine to operate on multiple communication operations:

```
c := make(chan string)
quit := make(chan struct{})

go func(messages []string) {
    for _, s := range messages {
        c <- s
    }
    close(quit)
}([]string{"hi", "bye"})

for {
    select {
    case message := <-c:
        println(message)
    case <-quit:
        println("shutting down")
        os.Exit(0)
    }
}
```

Ø

Select With Default

If no operations inside a `select` statement can process, the `select` will block. To work around this problem, you can add a `default` statement to your select:

```
package main

import "time"

func main() {
    tick := time.Tick(100 * time.Millisecond)
    done := time.After(500 * time.Millisecond)
    for {
        select {
        case <-tick:
            print("tick")
        case <-done:
            print(".done")
            return
        default:
            print(".")
            time.Sleep(10 * time.Millisecond)
        }
    }
}
```

Ø

The Wise Words Of Dave Cheney

Read Dave Cheney's article on some of the more interesting properties of channels.

Curious Channels - https://dave.cheney.net/2013/04/30/curious-channels

Goroutine Exercise

Create a program that does the following:

- has a function that takes three arguments
  - count
  - prefix
  - a wait group
- function should count up to the count provided, and print it out with the prefix provided.
- launch the function twice both in their own go routine.
- don't let main exit until both functions have completed

Goroutine Template

```go
package main

import "sync"

func count(prefix string, count int, wg *sync.WaitGroup) {
    // print out `prefix` `count` times
}

func main() {
    // Define the wait group

    // Increment wait group

    // Call count with a goroutine, first argument "first", second argument 50,
    // third argument the wait group

    // Call count with a goroutine, first argument "second", second argument 50,
    // third argument the wait group

    // Wait...
}
```

Ø

Go Routine Solution

```go
package main

import (
    "fmt"
    "sync"
)

func count(prefix string, count int, wg *sync.WaitGroup) {
    defer wg.Done()
    for i := 0; i < count; i++ {
        fmt.Println(prefix, i)
    }
}

func main() {
    var wg sync.WaitGroup
    wg.Add(2)
    go count("first: ", 50, &wg)
    go count("second: ", 50, &wg)
    wg.Wait()
}
```

Ø

Channel Exercise

Create a program that does the following:

- has a buffered channel of type `string` with a capacity of `5`
- add 5 messages to the channel
- have a function that ranges through the channel printing them out
- the function should close a `signal` channel to allow the program to exit
- sending an empty string to the messages channel will close the goroutine

Channel Template

```
package main

func process(messages chan string, quit chan struct{}) {
    // Loop through your messages
    for m := range messages {
        // Check to see if they sent a blank string, if so , quit

        // print the message
    }
}

func main() {
    // declare the messages channel of type string and capacity of 5

    // declare a signal channel

    // launch process in a goroutine

    // declare 5 fruits in a []string
    for _, f := range fruits {
    }

    // loop through the fruits and send them to the messages channel

    // write an empty string to the messages channel

    // wait for everything to finish

    println("done")
}
```

Ø

Channel Solution

```
package main

func process(messages chan string, quit chan struct{}) {
    for m := range messages {
        if m == "" {
            break
        }
        println(m)
    }

    close(quit)
}

func main() {
    messages := make(chan string, 5)
    quit := make(chan struct{})

    go process(messages, quit)

    fruits := []string{"apple", "plum", "peach", "pear", "grape", ""}
    for _, s := range fruits {
        messages <- s
    }

    <-quit
    println("done")
}
                                                                    Ø
```