Concurrency in GO!

By Tanishq Saxena

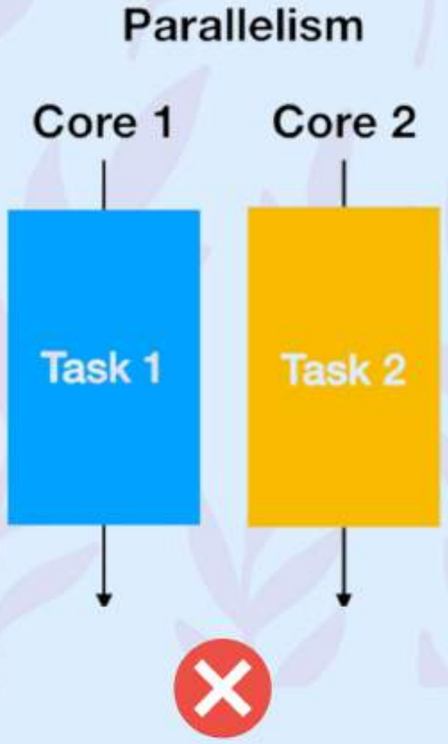
Now, what is Concurrency?

Suppose you're watching reels while your boring chemistry lecture. You can't focus on both reels and the lecture at the same time, so you're switching between them efficiently. This is concurrency. GO does it by "time-slicing" i.e. switching between the tasks to save time!

So overall we can say that concurrency is managing multiple tasks that can start, run, and complete in overlapping time periods. These tasks may not necessarily execute simultaneously. Note that in case of single core processors, concurrent tasks do not run at the exact same instant. But in case of multiple core processors, concurrent tasks can run in parallel as well.

Concurrency v/s Parallelism

People often confuse between concurrency and parallelism. Assume that you are now watching reels while eating. So, you're eating and watching at the exact same time. Both task are running in parallel. This is called "parallelism".



“Concurrency in GO”lang”

Having problem in GO syntax? ✅ Wait I'll explain

Goroutines

A goroutine is a lightweight thread managed by the Go runtime. Multiple Goroutines can run at the same time. You just add "go" keyword before a function to make it run as a Goroutine.

```go
package main

import (
 "fmt"
 "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

input

```
hello
world
world
hello
hello
world
world
hello
hello
```

output

Both functions running together !

Note that we are using time.Sleep() here
What if we don't use it?
Will the output be the same?
Let's See

Without time.Sleep() ?

```go
package main

import (
 "fmt"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        //time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
   go say("world")
   say("hello")
}
```

input

```
hello
hello
hello
hello
hello
```

output

Without time.Sleep() ?

```go
package main

import (
 "fmt"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        //time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
   go say("world")
   say("hello")
}
```

input

```
hello
hello
hello
hello
hello
```

output

So, what happened is that the say("hello") function was executed first and the code ended without even waiting for the say("world") function to do something. The main() function does not wait for the goroutine to finish

Solution? → "Waitgroups"

Waitgroups

A WaitGroup is used to wait for multiple goroutines to finish before moving on. Think of it like a counter that keeps track of how many goroutines are running, and when all are done, the program can continue. A WaitGroup helps pause execution until all goroutines are done.

How to use Waitgroups?

importing the required package →

declaring the variable “wg” as the waitgroup →

Adding something in the waitlist
(You can add more than I items in the list) →

Remove the item from the waitlist after the execution is done →

Tell the “main” function to wait for all the items in the waitlist →

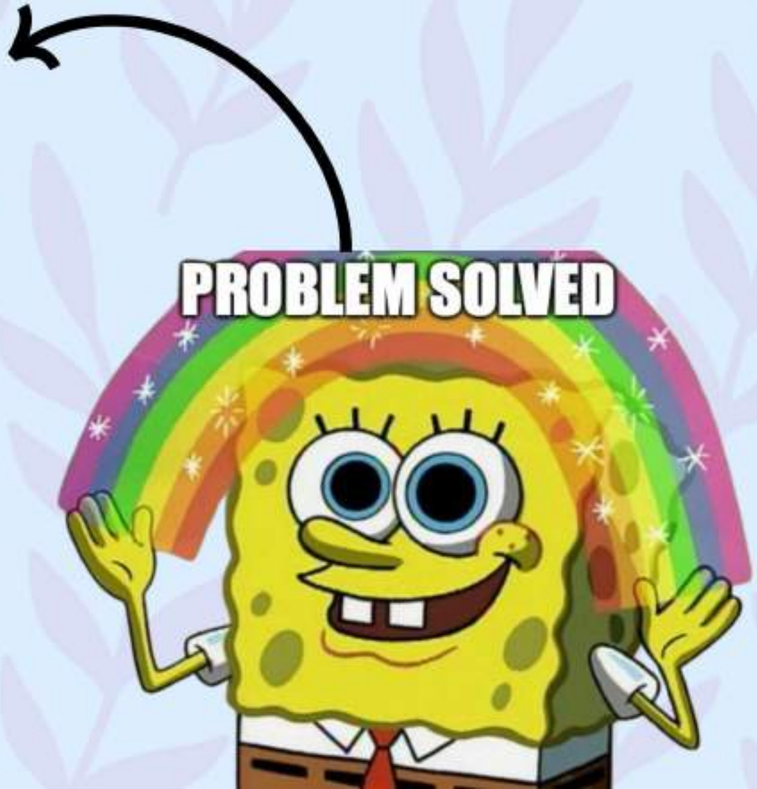
```
import "sync"

var wg sync.WaitGroup

wg.Add(1)

wg.Done()

wg.Wait()
```

Implementing the code

```go
package main

import (
    "fmt"
    "sync"
)

var wg sync.WaitGroup

func main() {
    go say("world")
    say("hello")
    wg.Wait()
}

func say(s string) {
    wg.Add(1)
    for i := 0; i < 5; i++ {
        fmt.Println(s)
    }
    wg.Done()
}
```

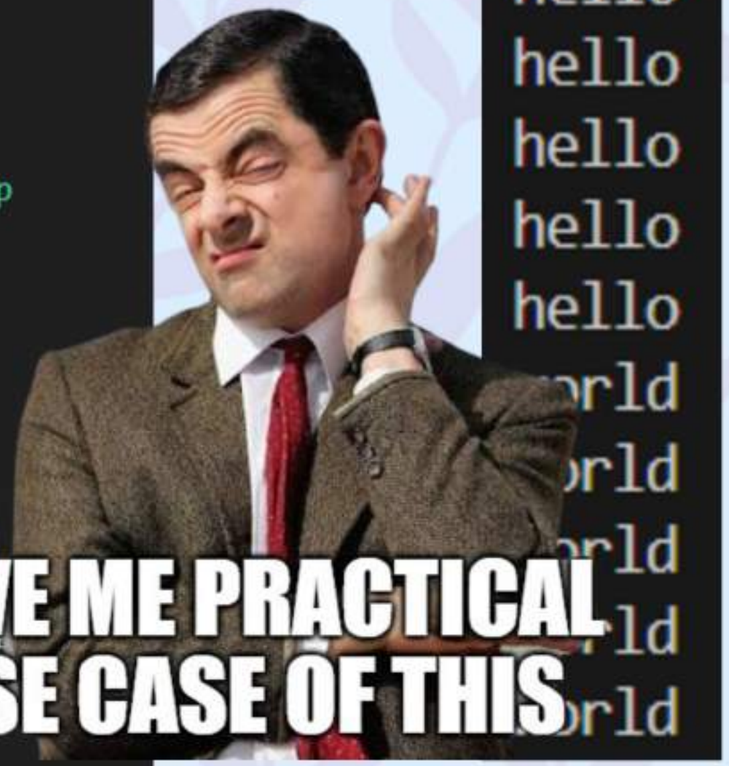
>>>

```
hello
hello
hello
hello
hello
world
world
world
world
world
```



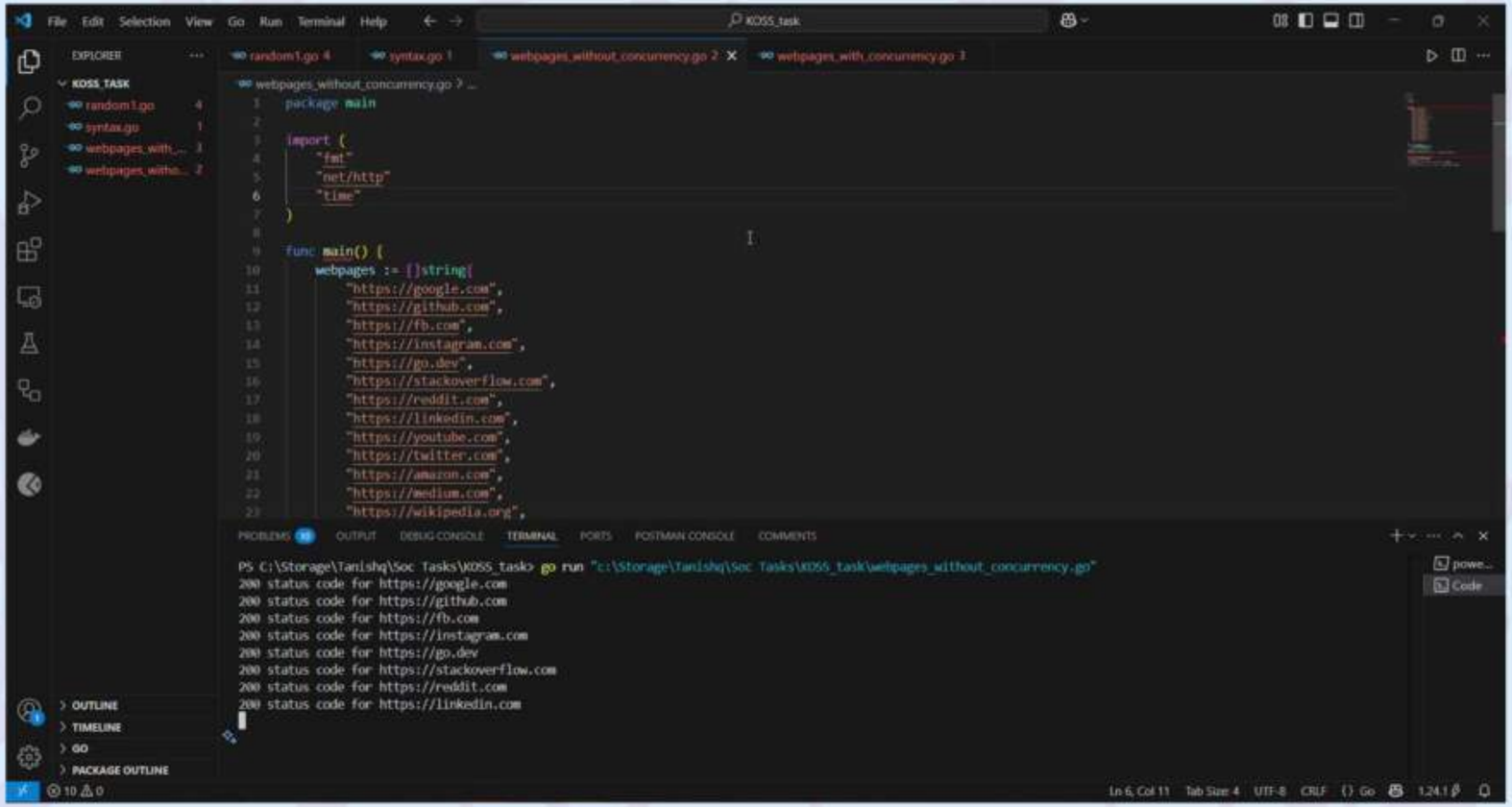
Implementing the code

```go
package main

import (
    "fmt"
    "sync"
)

var wg sync.WaitGroup

func main() {
    go say("world")
    say("hello")
    wg.Wait()
}

func say(s string) {
    wg.Add(1)
    for i := 0; i < 5; i++ {
        fmt.Println(s)
    }
    wg.Done()
}
```



Practical use case

Now, let's have a look at the real life use of concurrency in GO! Suppose we have a list of webpages and our backend server in GO wants to access them (Yes! GO is used for backend dev as well). Let us try doing that with and without using concurrency!



Results?

Without concurrency, the code took around 17 seconds to execute. But when we used concurrency, it took only 1.5 seconds ! So, concurrency is actually helpful in optimizing real life problems!

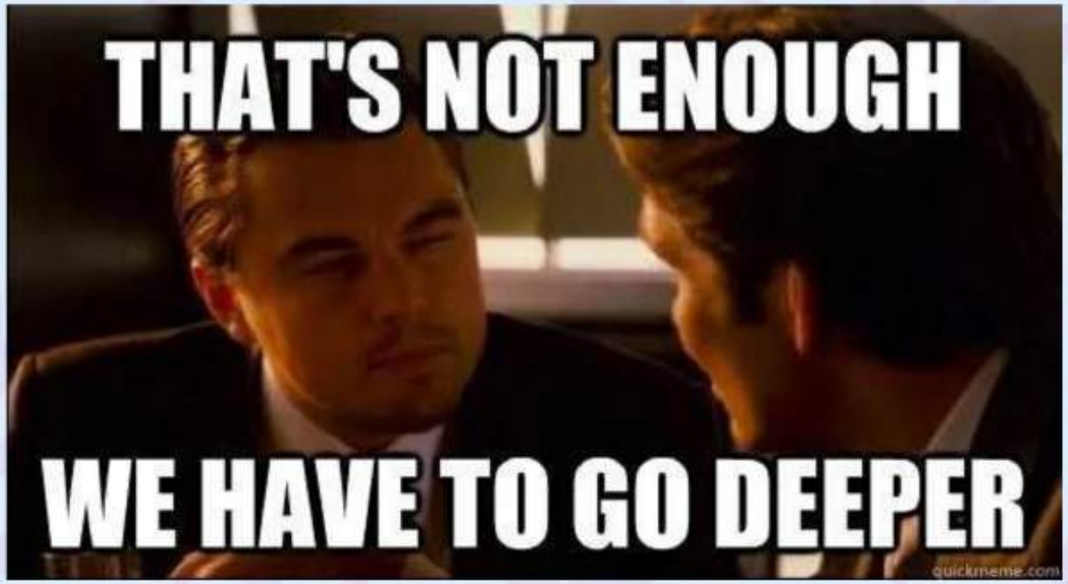
SURPRISED?

Results?

Without concurrency, the code took around 17 seconds to execute. But when we used concurrency, it took only 1.5 seconds ! So, concurrency is actually helpful in optimizing real life problems!

SURPRISED?

But,



Channels

A channel is a way for goroutines to communicate safely. It helps send and receive data between goroutines. Think of it like a "pipe" where one goroutine puts data in, and another takes it out.

Syntax?

Initializing a channel which stores "int" value type, you may use any other type like strings, bool, float64, etc.

Sending something to the channel

Receiving data from the channel

Closing a channel

```
myChannel = make(chan int)
myChannel <- 5
fmt.Print(<-myChannel)
close(myChannel)
```

Example code for channels

Basic Syntax

```go
package main
import (
    "fmt"
    "sync"
)
var wg sync.WaitGroup

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

    wg.Add(2)

    go func() {
        fmt.Println(<-myCh)
        wg.Done()
    }()

    go func() {
        myCh <- "Yo wassup?"
        wg.Done()
    }()
    wg.Wait()
}
```

A function used to receive data from a channel

A function used to send data to a channel
(Note that the order of calling the functions doesn't matter cause we are using concurrency anyways)

Yo wassup?
output

What do you think?

Let's write a simple code which uses 1000 goroutines to add +1 in the "count" variable at the same time. Do you think that the value of count will be 1000 or something else?

```go
package main

import (
    "fmt"
    "sync"
)

var count int
var wg sync.WaitGroup

func increment() {
    count++
    wg.Done()
}
```

```go
func main() {
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go increment()
    }

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

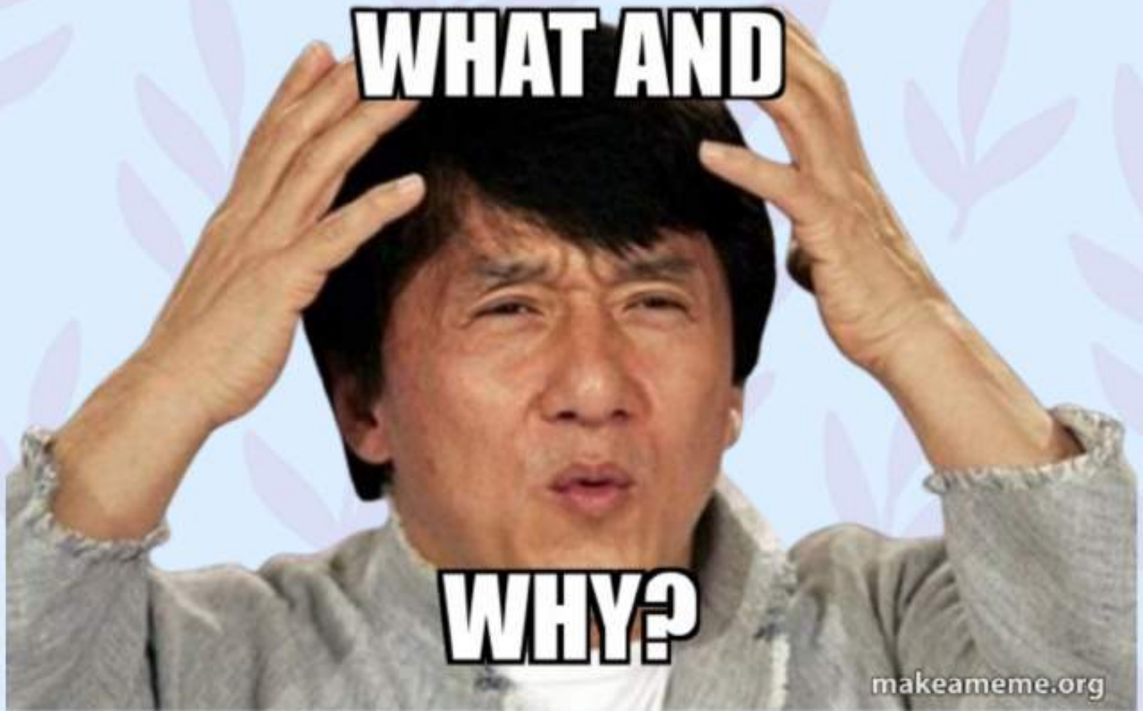
The output?

Well, a few times the output was 1000 but most of the time the output was a little less than 1000. For example -

```
Final Count:  988
```

```
Final Count:  996
```

```
Final Count:  989
```



Bruh! Why?

This was because of "Race Condition"

A race condition happens when multiple goroutines (or processes) try to change shared data at the same time. This can lead to unexpected or incorrect behavior.

> "Imagine two people writing on the same whiteboard at the same time. If both try to write different numbers, the final result depends on who finishes last, and the other person's work might be lost."

Mutex

Solution of “Race condition” is “Mutex”
A Mutex (short for Mutual Exclusion) is used to prevent multiple goroutines from accessing shared data at the same time, avoiding race conditions. Mutex does it by locking the shared resource, using it and then unlocking it.

Syntax?

Initializing the mutex pointer

Blocking other goroutines to make changes to the data

Unblocking other goroutines after making changes to the shared resource

```
var mut sync.Mutex
mut.Lock()
// Make changes to shared variable
mut.Unlock()
```

Mutex

Solution of “Race condition” is “Mutex”
A Mutex (short for Mutual Exclusion) is used to prevent multiple goroutines from accessing shared data at the same time, avoiding race conditions. Mutex does it by locking the shared resource, using it and then unlocking it.

Syntax?

Initializing the mutex pointer

Blocking other goroutines to make changes to the data

Unblocking other goroutines after making changes to the shared resource

Using mutex!

```go
package main
import "fmt"
import "sync"

var count int
var wg sync.WaitGroup
var mut sync.Mutex

func increment() {
    mut.Lock()
    count++
    mut.Unlock()
    wg.Done()
}
```

```go
func main() {
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go increment()
    }

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

Output was 1000 everytime!

Why only GO?

Concurrency is used in other languages like Python but still GO is preferred because -

- Go uses goroutines, which are super lightweight and can run in huge numbers without using much memory while Python uses threads, which are heavier, limiting how many can run at once.

- Go uses channels, making it easy and safe for goroutines to talk to each other while Python uses locks or shared memory, which can lead to errors if not handled carefully.

"

Congrats!
Now you can flex on your friends that you know concurrency!

"

Thanks!