IIT KHARAGPUR

TITLE: GO CONCURRENCY

~ADITYA DEBNATH

CONTENTS :

Go Language

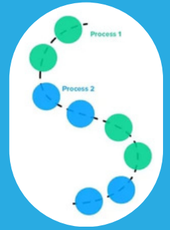
Developed by Google, 2007

Fast compilation and execution

Free and open source language

Easy in syntax

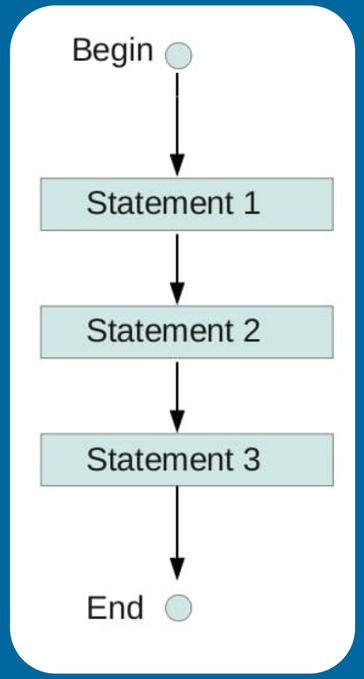
Compiled language



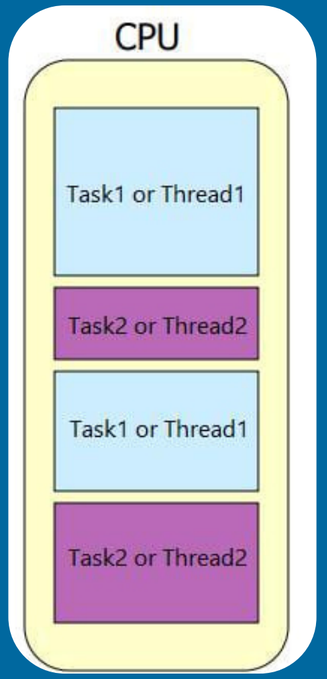
Supports concurrency

Sequential V/S Concurrent Programing

1. Single Execution Path
2. Deterministic Execution
3. Synchronous Execution
4. Limited Scalability
5. Readability
6. Ex. coding in C or C++



1. Parallel Executionh
2. Not always deterministic
3. Asynchronous Programming
4. Optimally scalable
5. Not easily readable
6. Ex. go routines func, in go



Single Execution Path

BENIFITS OF CONCURRENT PROGRAMMING

1. IMPROVED PERFORMANCE
2. Enhanced Responsiveness
3. Resource Utilization
4. Simplified Complex Systems:
5. Parallel Algorithms

How it's done then?

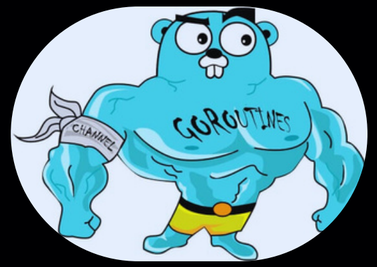
Go provides various fundamental blocks that help make concurrent programming user-friendly.

GO ROUTINES

CHANNELS

WAITGROUPS

MUTEXES



GO ROUTINES

- Goroutines are functions that run concurrently with other functions. They are lightweight threads managed by the Go runtime, allowing thousands or even millions of them to be created within a single Go program without significant overhead.

- Goroutines are created using the go keyword followed by a function call. When a function is called with go, it starts executing concurrently in its own goroutine, allowing the program to proceed with other tasks without waiting for it to finish.

- Goroutines are multiplexed onto a smaller number of OS threads, meaning that many goroutines can run on a single operating system thread, which makes them very efficient.

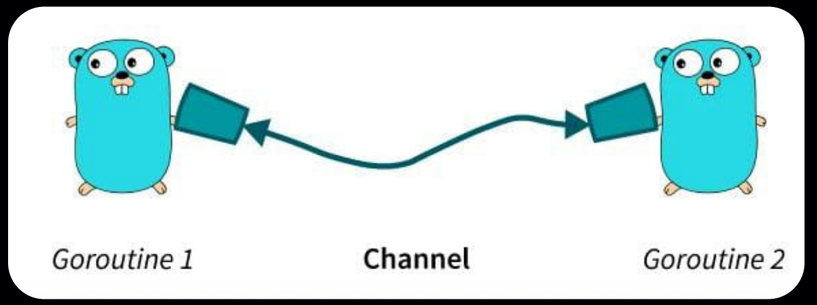
- Goroutines are a key feature of Go's approach to concurrency, enabling developers to write highly concurrent and efficient programs with ease.

```go
package main

import (
    "fmt"
    "time"
)

func sayHello() {
    fmt.Println("Hello from goroutine!")
}

func main() {
    // Start a new goroutine
    go sayHello()

    // Print from the main goroutine
    fmt.Println("Hello from main!")

    // Add a delay to allow the goroutine to execute
    time.Sleep(100 * time.Millisecond)
}
```

Run

```
go run /tmp/2JNAnGORjT.go
Hello from main!
Hello from goroutine!
```

CHANNELS



Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine.t

Create a new channel with make(chan val-type). Channels are typed by the values they convey.

Send a value into a channel using the channel <- syntax. Here we send "ping" to the messages channel we made above, from a new goroutine.

The <-channel syntax receives a value from the channel. Here we'll receive the "ping" message we sent above and print it out.

By default sends and receives block until both the sender and receiver are ready. This property allowed us to wait at the end of our program for the "ping" message without having to use any other synchronization

```
package main

import "fmt"

func main() {

    messages := make(chan
string)

    go func() { messages <-
"ping" }()

    msg := <-messages
    fmt.Println(msg)
}
```

~ BUFFERED CHANNELS

By default channels are *unbuffered*, meaning that they will only accept sends (chan <-) if there is a corresponding receive (<- chan) ready to receive the sent value. *Buffered channels* accept a limited number of values without a corresponding receiver for those values.

Here we make a channel of strings buffering up to 2 values

Because this channel is buffered, we can send these values into the channel without a corresponding concurrent receive.

```
package main

import "fmt"

func main() {

    messages := make(chan string, 2)

    messages <- "buffered"
    messages <- "channel"

    fmt.Println(<-messages)
    fmt.Println(<-messages)
}.
```

WAITGROUPS

This WaitGroup is used to wait for all the goroutines launched here to finish. Note: if a WaitGroup is explicitly passed into functions, it should be done by pointer.

Launch several goroutines and increment the WaitGroup counter for each.

Block until the WaitGroup counter goes back to 0; all the workers notified they're done.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func worker(id int) {
    fmt.Printf("Worker %d starting\n", id)

    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\n", id)
}

func main() {

    var wg sync.WaitGroup

    for i := 1; i <= 5; i++ {
        wg.Add(1)

        go func() {
            defer wg.Done()
            worker(i)
        }()
    }

    wg.Wait()

}
```

MUTEXES

In Go, a mutex (short for mutual exclusion) is a synchronization primitive used to protect shared resources from being accessed concurrently by multiple goroutines. Mutexes ensure that only one goroutine can access the shared resource at any given time, preventing race conditions and data races.

Mutexes work by locking and unlocking. When a goroutine wants to access a shared resource, it locks the mutex. If the mutex is already locked by another goroutine, the goroutine will be blocked until the mutex becomes available. Once the goroutine finishes its work with the shared resource, it unlocks the mutex to allow other goroutines to access it

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

var (
    counter int
    mutex   sync.Mutex
)

func increment() {
    mutex.Lock()
    defer mutex.Unlock()
    counter++
    fmt.Println("Incrementing counter to", counter)
}

func main() {
    for i := 0; i < 5; i++ {
        go increment()
    }

    time.Sleep(time.Second) // Waiting for goroutines to finish
    fmt.Println("Final counter value:", counter)
}
```

Run

//code time </>

```go
package main

import (
    "fmt"
    "sync"
)

func worker(id int, jobs <-chan int, results chan<- int, wg *sync.WaitGroup, mu *sync.Mutex) {
    defer wg.Done()
    for j := range jobs {
        fmt.Printf("Worker %d processing job %d\n", id, j)
        // Simulating some work
        // Protecting shared resource with mutex
        mu.Lock()
        results <- j * 2
        mu.Unlock()
    }
}

func main() {
    // Number of workers
    numWorkers := 3
    // Number of jobs
    numJobs := 5

    // Create channels
    jobs := make(chan int, numJobs)
    results := make(chan int, numJobs)

    // Create wait group and mutex
    var wg sync.WaitGroup
    var mu sync.Mutex

    // Add workers to wait group
    for i := 1; i <= numWorkers; i++ {
        wg.Add(1)
        go worker(i, jobs, results, &wg, &mu)
    }

    // Send jobs to workers
    for j := 1; j <= numJobs; j++ {
        jobs <- j
    }
    close(jobs)

    // Wait for all workers to finish
    wg.Wait()

    // Close results channel after all workers are done
    close(results)

    // Collect and print results
    for r := range results {
        fmt.Println("Result:", r)
    }
}
```

Run

main.go Outp

```
go run /tmp/d18cMumM9O.go
Worker 3 processing job 1
Worker 3 processing job 2
Worker 3 processing job 3
Worker 3 processing job 4
Worker 1 processing job 5
Result: 2
Result: 4
Result: 6
Result: 8
Result: 10
```


THANK YOU