CAS CS 350
Lab 3

1. Type Composition/Embedding - struct
    a. Go does not have the typical, type-driven notion of subclassing, but it does have the ability to borrow pieces of an implementation by embedding types within a struct or interface
    b. Structs can have fields borrowed from another struct (parent)

```
type Human struct
     { Name string
     Address string
     Age int
}

type Employee struct {
     Human         // Embedded anonymous field that points to another struct
     EmployeeNo string
     Salary float32
     Designation string
}

human := Human {"Rob Pike", "USA", 45}
fmt.Println(human)                 // {Rob Pike USA 45}
```

```
emp := Employee{human, "3423434", 200456.78, "Chief Architect" }
fmt.Println(emp)                //  {{Rob Pike USA 45} 3423434 200456.78 Chief Architect}
```

    c.

2. Embed by value vs by ref
    a. Embedding could be by values or by ref

```
type Human struct
    { Name string
    Age int
}
```

`type Employee struct`
`{   Human      // Embed by value`
`    EmployeeNo string`
`    Designation string`
`}`
	`type Employee struct`
	`{   *Human     // Embed by ref`
	`    EmployeeNo string`
	`    Designation string`
	`}`

```
human := Human {"Rob Pike", 45}
fmt.Println(human)         // {Rob Pike
45}

emp := Employee{human, "3423434", "Chief Architect" }
fmt.Println(emp)           // {{Rob Pike 45} 3423434 Chief
Architect}
```

    b.

3. Type Composition/Embed - interface
    a. Embedding could be done for interface also

```
type CanWalk interface
    { Walk()
}
type CanFly interface
    { Fly()
}
type CanWalkAndFly interface
{
    CanWalk    // Embed
    CanFly     // Embed
}

type Human struct
        {
    Name string
}
func (human Human) Walk()
    { fmt.Println("Human
    walk")
}
func (human Human) Fly() {
    fmt.Println("Human
```

```
var h1 CanWalk = Human{"Aniruddha"}
h1.Walk()

var h2 CanFly = Human{"Aniruddha"}
h2.Fly()

var h3 CanWalkAndFly = Human{"Aniruddha"}
h3.Walk()
h3.Fly()
```

b.

4. Exporting from package
    a. Methods whose name start with Upper case are exported. Methods whose name do not start with upper case are not exported (private)

```
package library

import "fmt"

// Exported methods
func SayHello() {
        fmt.Println("Hello from Library")
}

func SayHello2() {
        fmt.Println("Hello2 from Library")
}

// Non exported method
func sayHelloPvt() {
        fmt.Println("sayHelloPvt from Library")
}
```

b.

5. Unique go features
    a. Returning multiple values from function
        i. Supports multiple return value from functions like tuples supported in other languages
        ii. Could be used to return a pair of values/return value and error code

```
func swap(x, y int) (int, int){
    return y, x
}

var x,y int = 10,20
var p,q int = swap(x,y)

fmt.Println(x,y)                 // 10 20
fmt.Println(p,q)                 // 20 10

func addSubMultiDiv(x,y int)     int, int,
    (int, return x+y, x-y, x*y,  int){
    x/y
}
fmt.Println(addSubMultiDiv(20,1  // 30 10 200
0))                              2
```

iii.

iv. More than two values could be returned

```
func getEmployee() (string, int, float32) { return
    "Bill", 50, 6789.50
}
func main() {
    name, age, salary := getEmployee()
    fmt.Println(name)  fmt.Println(age)
    fmt.Println(salary)
    fmt.Scanln()
}
```

v.

vi. Return values not required could be ignored by using _

```
func main() {
   name, _, salary := getEmployee()
   fmt.Println(name) fmt.Println(salary)
   fmt.Scanln()
}
```

vii.

b. Named return values

```
func concat(str1, str2 string) (res string)
    { res = str1 + " " + str2
    return
}

result := concat("Aniruddha","Chakrabarti")
fmt.Println(result)        // Aniruddha Chakrabarti

func getEmployee() (name string, age int, salary float32)
{
    name = "Bill" age
    = 50
    salary = 6789.50
    return
}

func main() {
    name, age, salary := getEmployee()
    fmt.Println(name)  fmt.Println(age)
    fmt.Println(salary)
    fmt.Scanln()
}
```

i.

c. Defer

i. A defer statement defers the execution of a function until the surrounding function returns

ii. The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns

iii. Defer is commonly used to simplify functions that perform various clean-up actions

```
package main

func main()

{

   defer println("World")
   println("Hello")
}

// prints Hello World
```

iv.

    v. Defer rules
        1. A deferred function’s arguments are evaluated when the defer statement is evaluated
        2. Deferred function calls are executed in Last In First Out order after the surrounding function returns
        3. Deferred functions may read and assign to the returning function’s named return values

d. Function closures

    i.

```
func swap(x, y int) (int, int){
    return y, x
}

var x,y int = 10,20
var p,q int = swap(x,y)
fmt.Println(x,y)          // 10 20
fmt.Println(p,q)          // 20 10



func addSubMultiDiv(x,y int)     int, int,
    (int, return x+y, x-y, x*y,  int){
    x/y
}
fmt.Println(addSubMultiDiv(20,1  // 30 10 200
0))                              2
```

e. Goroutines
    i. Goroutines are lightweight threads managed by Go runtime - since they are lightweight creating them is fast, and does not impact performance
    ii. A goroutine is a function that is capable of running concurrently with other functions
    iii. To create a goroutine, use the keyword go followed by a function invocation
    iv.

```
func main(){
    add(20, 10)
    // add function is called as goroutine - will execute concurrently with calling
    one
    go add(20, 10)
    fmt.Scanln()
}

func add(x int, y int)
    { fmt.Println(x+y)
}
```

    v. Goroutines could be started for anonymous function call also
    vi.

```
// 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
```

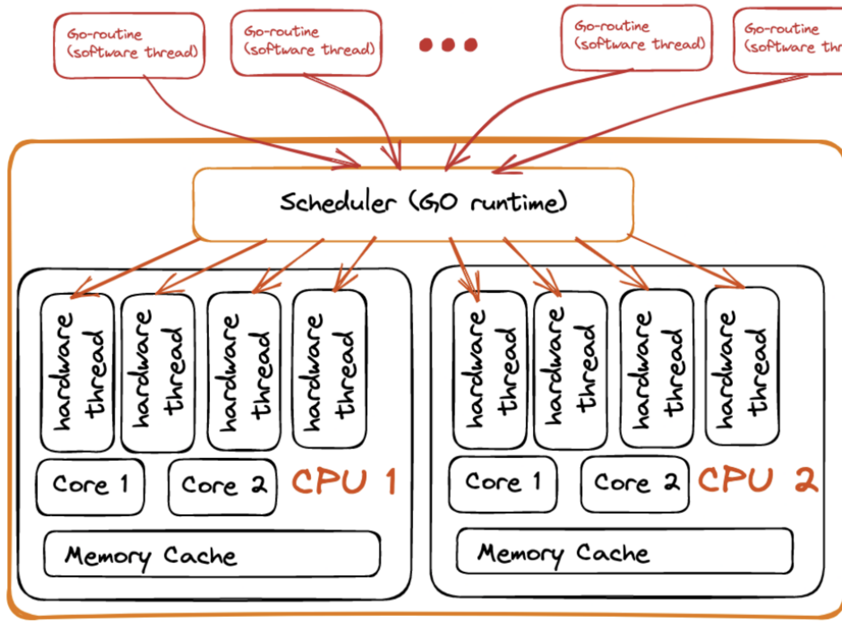
    vii. How Go manages its go-routines
        1. Goroutines itself are implemented using a stack, this allows them to be created and destroyed easily, reducing overhead. On the other hand, traditional thread-based concurrency models in other

languages typically require more resources to create and manage threads adding significant overhead

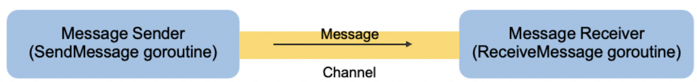
2. Golang manages its go-routines using the Go runtime scheduler, the scheduler is responsible for managing the execution of go-routines on a limited number of hardware threads, this includes allocating go-routines to available processors and switching between go-routines as needed.
3. The scheduler uses a work-stealing algorithm. When a processor becomes idle, it will look for the go-routines that are ready to run and steal work from them — this ensures that go-routines are evenly distributed across processes and that idle processors are used to their full potential.
4. Additionally, the golang garbage collector also helps remove go routines that are no longer being used. In the context of go-routines, when a goroutines is no longer being referenced by any other object, it is garbage collected.

viii. Software vs Hardware threads

1. A software thread, otherwise referred to as a logical thread or lightweight thread is managed entirely by software. Typically, implemented using a stackbased design and scheduled and managed during runtime and can run concurrently on a single processor as well, not just across multiple processors.
2. A hardware thread, otherwise referred to as a physical thread or a heavyweight thread, is entirely managed by hardware. Typically, they are implemented with dedicated hardware resources and managed by the operating system.
    a. Best understood as a physical CPU or core. For example, a 4 core CPU can support 4 hardware threads.
    b. One hardware thread can run many software threads.

ix. 

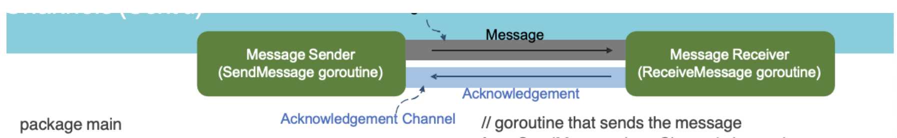
f. Channels




```
package main

import (
	"fmt"
	"time“
)

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

	go SendMessage(channel)
	go ReceiveMessage(channel)

	fmt.Scanln();
}
```

```
// goroutine that sends the message
func SendMessage(channel chan string) {
	for {
		channel <- "sending message @" +
time.Now().String()
		time.Sleep(5 * time.Second)
	}
}


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

i.




```
package main

import (
	"fmt"
	"time"
)

func main(){
	// Channel to send message from
sender to receiver
	msgChnl := make(chan string)

	// Channel to acknowledge message receipt
by receiver
	ackChnl := make(chan string)

	go SendMessage(msgChnl, ackChnl)
	go ReceiveMessage(msgChnl, ackChnl)

	fmt.Scanln();
}
```

```
// goroutine that sends the message
func SendMessage(msgChannel chan string,
ackChannel chan string) {
	for {
		msgChannel <- "sending message @" +
time.Now().String()
		time.Sleep(2 * time.Second) ack :=
		<- ackChannel fmt.Println(ack)
	}
}

// goroutine that receives the message  func
ReceiveMessage(msgChannel chan string, ackChannel
chan string) {
	for {
		message := <- msgChannel
		fmt.Println(message)
		ackChannel <- "message received @" +
time.Now().String()
	}
```

ii.

g. Mutex
   i. 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?
   ii. This concept is called mutual exclusion, and the conventional name for the data structure that provides it is mutex
   iii. Go's standard library provides mutual exclusion with sync.Mutex and its two methods
      1. Lock
      2. Unlock
   iv. We can define a block of code to be executed in mutual exclusion by surrounding it with a call to Lock and Unlock as shown on the Inc method
   v. We can also use defer to ensure the mutex will be unlocked as in the Value method

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

// SafeCounter is safe to use concurrently.
type SafeCounter struct {
	mu sync.Mutex
	v  map[string]int
}

// Inc increments the counter for the given key.
func (c *SafeCounter) Inc(key string) {
	c.mu.Lock()
	// Lock so only one goroutine at a time can access the map c.v.
	c.v[key]++
	c.mu.Unlock()
}

// Value returns the current value of the counter for the given key.
func (c *SafeCounter) Value(key string) int {
	c.mu.Lock()
	// Lock so only one goroutine at a time can access the map c.v.
	defer c.mu.Unlock()
	return c.v[key]
}

func main() {
	c := SafeCounter{v: make(map[string]int)}
	for i := 0; i < 1000; i++ {
		go c.Inc("somekey")
	}

	time.Sleep(time.Second)
	fmt.Println(c.Value("somekey"))
}
```

   vi.