MUTEX : MUTUAL EXCLUSION

RACE : when two or more than two processes try to access the same resource and try to modify it then the modification or operation don't remain atomic .
In case of multithreading [using go routines] we might encounter this situation because things happens concurrently

Ex : On social media you don't wait for other people to first they like the post then you, multiple people like the post simultaneously

```go
// mimic post of socials
type post struct {
    views int
}

// when you call inc functin views get incremented
func (p *post) inc() {
    p.views += 1
}
```

* here we have a datatype struct for the post which contains the likes
* so when you call the inc function then likes count increases ..

```go
func main() {

    myPost := post{views: 0} //default 0
    myPost.inc()             //1
    myPost.inc()             //2
```

* here there is no problem as one finish then next is going or if you want to do multiple then you can do using the loop

* suppose we want to run this fun for 100 times

```go
    for i := 0; i < 100; i++ {
        myPost.inc()
    }

    fmt.Println(myPost.views)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS   bash

shivam@shivam:~/Development/Golang$ go run 24_mutex/mutex.go
100
```

- till now there is no problem because things are happening one by one
- as it is running synchronous once it is increment then next iteration is going on

- to make it concurrent we can use go routines

```
for i := 0; i < 100; i++ {
    go myPost.inc()
}
```

* since our main function will end first so to hold it we will use the wait groups

```
func (p *post) inc(wg *sync.WaitGroup) {
    defer func() {
        wg.Done()
    }()
    p.views += 1
```

```
var wg sync.WaitGroup

for i := 0; i < 100; i++ {
    wg.Add(1)

    go myPost.inc(&wg)
```

* so this is how you can see that we are using the wait groups

NOTE : we are not using the defer and wg where we are calling inc with go because that is now non blocking so it will end immediately that's why we are passing the wg in the inc function

* now notice the output …

```
shivam@shivam:~/Development/Golang$ go run 24_mutex/mutex.go
100
shivam@shivam:~/Development/Golang$ go run 24_mutex/mutex.go
91
shivam@shivam:~/Development/Golang$ go run 24_mutex/mutex.go
89
shivam@shivam:~/Development/Golang$ go run 24_mutex/mutex.go
95
```

- here we are getting 91, 89, 95, 100
- so it is not fixed now because things are going concurrently so there is no consistency

** multiple lightweight threads are now accessing the same views and incrementing them in in the same time

** *I found something or derived something* **
- here 100 is very small number that's why things are going very fast such that very few lightweight threads are able to write concurrently that's why we are seeing very less difference[compared to 100] ..

- but to see a large concurrency let’s increase the number

```
34    for i := 0; i < 100000; i++ {
35        wg.Add(1)
36
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  bash

shivam@shivam:~/Development/Golang$ go run 24_mutex/mutex.go
78267
shivam@shivam:~/Development/Golang$ go run 24_mutex/mutex.go
74535
shivam@shivam:~/Development/Golang$ go run 24_mutex/mutex.go
76627
```

- now you can see the gap or difference when no is big then you can see multiple threads in action

---

** so now you see threads are overriding the changes of another
- to avoid this we can use mutex

** so we will lock that process of changing , like when views is getting modified by one process then at that time only that routine will be able to modify

[... modifying ...] ** locked ** other routines are waiting
[... done ...] ** free

---

* we can mutex also globally but it’s good thing that we should keep the things to that entity [here post struct] whom it is related ..

```
type post struct {
    views int
    mu    sync.Mutex
}
```

```
func (p *post) inc(wg *sync.WaitGroup) {
    defer func() {
        wg.Done()
    }()

    //since here modification is going on so
    p.mu.Lock()
    p.views += 1
    p.mu.Unlock()
```

* so where the modification is going on lock that and unlock that
- now you will get 100, 100, 100 .. .. ..
- no RACE condition is there

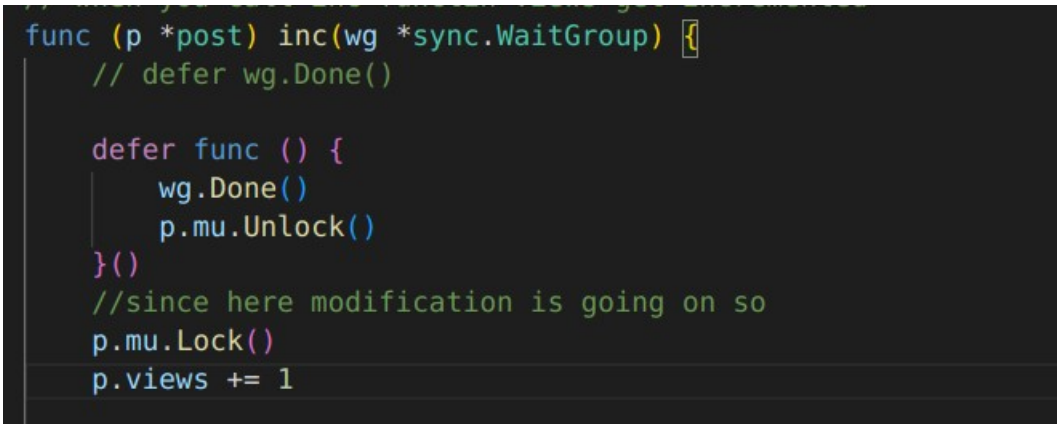
** BEST PRACTICE **

- since only one thing [increment] is going on
- but in real case there might be multiple things so in that case if some error occurs then it will never reach to unlock so it will be locked for forever

- so we will keep unlock in the defer function

[*keep in mind that Its not the single line is incrementing, for each increment function is getting called so keeping defer will not affect the increment logic*]

- as defer get executed at-least once after execution of that function so that if it cause error for one routine/process then other routines/process can work without getting affected



```
func (p *post) inc(wg *sync.WaitGroup) {
    // defer wg.Done()

    defer func () {
        wg.Done()
        p.mu.Unlock()
    }()
    //since here modification is going on so
    p.mu.Lock()
    p.views += 1
```

---

* using mutex is a good thing but
* locking may become bottleneck in some cases as go routines have to wait there

** so in real life there could be multiple things going on so lock only that thing which is necessary [modification is going on]
=> so only that part becomes only bottleneck rest things works fine ...