Mutex

Let's learn about Mutex in Go and use it to solve race conditions.

WE'LL COVER THE FOLLOWING

> A mutex, or a mutual exclusion prevents other processes from entering a critical section of data while a process occupies it.

We import mutex from the `sync` package in Go. `sync.mutex` has two methods:

- `.Lock()` : acquires/holds the lock
- `.Unlock()`: releases the lock

```
var myMutex sync.Mutex

myMutex.Lock()

//critical section

myMutex.Unlock()
```

Critical Section

Let's continue on our example of deposit/withdraw balance from the lesson on race condition. Deposit and withdraw should not happen at the same time, i.e., the variable `balance` should not be read/modified simultaneously. Therefore, every time we read the variable `balance` or update the variable `balance`, that section of code is referred to as the critical section of data. In general, the section of code which is accessing or updating the shared resources is called the critical section. Mutexes provide us with a locking

mechanism which enables only one process to access the critical section at a time.

In Go, if one goroutine is in a critical section, then the mutex will not allow any other goroutine to enter the same critical section.

Example

Let’s look at the example below:

```
package main
import ( "fmt"
        "sync")

func deposit(balance *int,amount int, myMutex *sync.Mutex, myWaitGroup *sync.WaitGroup){
    myMutex.Lock()
    *balance += amount //add amount to balance
    myMutex.Unlock()
    myWaitGroup.Done()
}

func withdraw(balance *int, amount int, myMutex *sync.Mutex, myWaitGroup *sync.WaitGroup){
    myMutex.Lock()
    *balance -= amount //subtract amount from balance
    myMutex.Unlock()
    myWaitGroup.Done()
}

func main() {

    balance := 100
    var myWaitGroup sync.WaitGroup
    var myMutex sync.Mutex

    myWaitGroup.Add(2)
    go deposit(&balance,10, &myMutex, &myWaitGroup) //depositing 10
    withdraw(&balance, 50,&myMutex, &myWaitGroup) //withdrawing 50

    myWaitGroup.Wait()
    fmt.Println(balance)


}
```

The variable `balance` is a shared resource of the `deposit()` and `withdraw` functions. Therefore, whenever we encounter any operations that access and update `balance` such as:

```
*balance += amount
```

and

```
*balance -= amount
```

we surround that code with our locking mechanism. Once a lock has been acquired by a goroutine by `myMutex.Lock()`, no other goroutine is allowed to enter the critical section until and unless the critical section is executed and the lock is released through `myMutex.Unlock()`.

Let’s look at another example below:

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

func main() {
   myMutex := sync.Mutex{}
   myMutex.Lock()
   go func() {
    myMutex.Lock()
    fmt.Println("I am in the goroutine")
    myMutex.Unlock()
  }()
  fmt.Println("I am in main routine")
  myMutex.Unlock()
  time.Sleep(time.Second*1)

}
```

So first we acquire the lock on **line 9** and then we cannot hold another lock on **line 11**. The lock is released on **line 16** after printing `I am in main routine` on to the console. Only then, the goroutine is able to acquire the lock on **line 11** and print `I am in the goroutine` before releasing that lock on **line 13** and exiting the program.

Hope you understand mutexes better now. There is another type of mutex: RWMutex.

RWMutex

RWMutex stands for Reader/Writer mutual exclusion and is essentially the same as Mutex, but it gives the lock to more than one reading process or just a writing process.

RWMutex provides us with more control over memory. If you want to learn more about it, you can have a look at the documentation.

So, you just got familiar with another way of solving race conditions other than channels and waitgroups: Mutex.

Let’s see what else the sync Package has in store for us in the next lesson!