Race Conditions in Golang

Created	@May 19, 2024 4:57 PM
Class	Go Specialization - UCI
Type	Lecture
Reviewed	☐

In this code, we have two goroutines (`increment`) that are concurrently modifying a shared global variable `counter`. The `increment` function iterates 1000 times and increments the `counter` variable by 1 each time. However, the statement `counter++` is not atomic, meaning it consists of multiple steps (read, increment, write) and is therefore susceptible to a race condition.

A race condition occurs when the outcome of a program depends on the non-deterministic timing of multiple concurrent operations. In this example, both goroutines are reading the value of `counter`, incrementing it, and writing it back to memory. However, because these operations are not synchronized, it's possible for one goroutine to read the `counter` value before the other goroutine has finished updating it. This can lead to incorrect results because the changes made by one goroutine may be overwritten by the other goroutine before they are properly accounted for.

To prevent the race condition, we can use synchronization mechanisms such as mutexes or channels to ensure that only one goroutine can access the shared variable `counter` at a time. This way, we can guarantee that the operations on `counter` are performed atomically and the race condition is avoided.

```
package main

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

```go
var counter = 0
var wg sync.WaitGroup

func increment() {
    for i := 0; i < 1000; i++ {
        counter++ // This statement is not atomic
    }
    wg.Done()
}

func main() {
    wg.Add(2)
    go increment()
    go increment()
    wg.Wait()
    fmt.Println("Final counter value:", counter)
}
```