GO language: fatal error: all goroutines are asleep - deadlock

Code below works fine with hard coded JSON data however doesn't work when I read JSON data from a file. I'm getting `fatal error: all goroutines are asleep - deadlock` error when using `sync.WaitGroup` .

WORKING EXAMPLE WITH HARD-CODED JSON DATA:

```
package main

import (
    "bytes"
    "fmt"
    "os/exec"
    "time"
)

func connect(host string) {
    cmd := exec.Command("ssh", host, "uptime")
    var out bytes.Buffer
    cmd.Stdout = &out
    err := cmd.Run()
    if err != nil {
        fmt.Println(err)
    }
    fmt.Printf("%s: %q\n", host, out.String())
    time.Sleep(time.Second * 2)
    fmt.Printf("%s: DONE\n", host)
}

func listener(c chan string) {
    for {
        host := <-c
        go connect(host)
    }
}

func main() {
    hosts := [2]string{"user1@111.79.154.111", "user2@111.79.190.222"}
    var c chan string = make(chan string)
    go listener(c)

    for i := 0; i < len(hosts); i++ {
        c <- hosts[i]
    }
    var input string
    fmt.Scanln(&input)
}
```

OUTPUT:

```
user@user-VirtualBox:~/go$ go run channel.go
user1@111.79.154.111: " 09:46:40 up 86 days, 18:16,  0 users,  load average: 5"
user2@111.79.190.222: " 09:46:40 up 86 days, 17:27,  1 user,  load average: 9"
user1@111.79.154.111: DONE
user2@111.79.190.222: DONE
```

NOT WORKING - EXAMPLE WITH READING JSON DATA FILE:

```
package main

import (
    "bytes"
    "fmt"
    "os/exec"
    "time"
    "encoding/json"
    "os"
    "sync"
)

func connect(host string) {
    cmd := exec.Command("ssh", host, "uptime")
    var out bytes.Buffer
    cmd.Stdout = &out
    err := cmd.Run()
    if err != nil {
        fmt.Println(err)
    }
    fmt.Printf("%s: %q\n", host, out.String())
```

```
    time.Sleep(time.Second * 2)
    fmt.Printf("%s: DONE\n", host)
}

func listener(c chan string) {
    for {
        host := <-c
        go connect(host)
    }
}

type Content struct {
    Username string `json:"username"`
    Ip       string `json:"ip"`
}

func main() {
    var wg sync.WaitGroup

    var source []Content
    var hosts []string
    data := json.NewDecoder(os.Stdin)
    data.Decode(&source)

    for _, value := range source {
        hosts = append(hosts, value.Username + "@" + value.Ip)
    }

    var c chan string = make(chan string)
    go listener(c)

    for i := 0; i < len(hosts); i++ {
        wg.Add(1)
        c <- hosts[i]
        defer wg.Done()
    }

    var input string
    fmt.Scanln(&input)

    wg.Wait()
}
```

OUTPUT

```
user@user-VirtualBox:~/go$ go run deploy.go < hosts.txt
user1@111.79.154.111: " 09:46:40 up 86 days, 18:16,  0 users,  load average: 5"
user2@111.79.190.222: " 09:46:40 up 86 days, 17:27,  1 user,  load average: 9"
user1@111.79.154.111 : DONE
user2@111.79.190.222: DONE
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc210000068)
    /usr/lib/go/src/pkg/runtime/sema.goc:199 +0x30
sync.(*WaitGroup).Wait(0xc210047020)
    /usr/lib/go/src/pkg/sync/waitgroup.go:127 +0x14b
main.main()
    /home/user/go/deploy.go:64 +0x45a

goroutine 3 [chan receive]:
main.listener(0xc210038060)
    /home/user/go/deploy.go:28 +0x30
created by main.main
    /home/user/go/deploy.go:53 +0x30b
exit status 2
user@user-VirtualBox:~/go$
```

HOSTS.TXT

```
[
   {
      "username":"user1",
      "ip":"111.79.154.111"
   },
   {
      "username":"user2",
      "ip":"111.79.190.222"
   }
]
```

go

edited Jan 22 at 10:35

asked Nov 14 '14 at 10:09

BentCoder
4,011 6 33 73

2 Answers

Go program ends when the main function ends.

This solution is described in documentation article Waiting for goroutines.

From the language specification

> Program execution begins by initializing the main package and then invoking the function main. When that function invocation returns, the program exits. It does not wait for other (non-main) goroutines to complete.

Therefore, you need to wait for your goroutines to finish. The common solution for this is to use sync.WaitGroup object.

The simplest possible code to synchronize goroutine:

```
package main

import "fmt"
import "sync"

var wg sync.WaitGroup // 1

func routine() {
    defer wg.Done() // 3
    fmt.Println("routine finished")
}

func main() {
    wg.Add(1) // 2
    go routine() // *
    wg.Wait() // 4
    fmt.Println("main finished")
}
```

And for synchronizing multiple goroutines

```
package main

import "fmt"
import "sync"

var wg sync.WaitGroup // 1

func routine(i int) {
    defer wg.Done() // 3
    fmt.Printf("routine %v finished\n", i)
}

func main() {
    for i := 0; i < 10; i++ {
        wg.Add(1) // 2
        go routine(i) // *
    }
    wg.Wait() // 4
    fmt.Println("main finished")
}
```

WaitGroup usage in order of execution.

1. Declaration of global variable. Making it global is the easiest way to make it visible to all functions and methods.
2. Increasing the counter. This must be done in main goroutine because there is no guarantee that newly started goroutine will execute before 4 due to memory model guarantees.
3. Decreasing the counter. This must be done at the exit of goroutine. Using deferred call, we make sure that it will be called whenever function ends no matter but no matter how it ends.
4. Waiting for the counter to reach 0. This must be done in main goroutine to prevent program exit.

* The actual parameters are evaluated before starting new gouroutine. Thus it is needed to evaluate them explicitly before `wg.Add(1)` so the possibly panicking code would not leave increased counter.

Use

```
param := f(x)
wg.Add(1)
go g(param)
```

instead of

```
wg.Add(1)
go g(f(x))
```

edited Jul 28 at 8:41

answered Nov 14 '14 at 11:07

Grzegorz Żur
18.1k 7 48 67

I've been using GO since yesterday (even less than a day) so I don't really know much although I've been reading the doc. The code above is result of 1 day of study/practice. Where exactly do I put sleep? I looked into the example. – BentCoder Nov 14 '14 at 11:13

@inanzzz Look at the first answer. – Grzegorz Żur Nov 14 '14 at 11:14

I think I'm putting things in wrong places. `fatal error: all goroutines are asleep - deadlock!` – BentCoder Nov 14 '14 at 11:18

@inanzzz then golang.org/doc/articles/race_detector.html might help – VonC Nov 14 '14 at 11:42

Guys I updated the post above because there is a progress at least however there is a `deadlock` this time. I moved `wg` lines arround but couldn't find the correct place. – BentCoder Nov 14 '14 at 11:49

Thanks for the very nice and detailed explanation Grzegorz Żur. One thing that I want to point it out that typically the func that needs to be threaded wont be in main(), so we would have something like this:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "io/ioutil"
    "math/rand"
    "os"
    "reflect"
    "regexp"
    "strings"
    "sync"
    "time"
)

var wg sync.WaitGroup    // VERY IMP to declare this globally, other wise one   //would
hit "fatal error: all goroutines are asleep - deadlock!"

func doSomething(arg1 arg1Type) {
     // cured cancer
}

func main() {
    r := rand.New(rand.NewSource(time.Now().UnixNano()))
    randTime := r.Intn(10)
    wg.Add(1)
    go doSomething(randTime)
    wg.Wait()
    fmt.Println("Waiting for all threads to finish")
}
```

The thing that I want to point it out is that global declation of wg is very crucial for all threads to finish before main()

answered May 14 at 2:04

Romeo
11 1