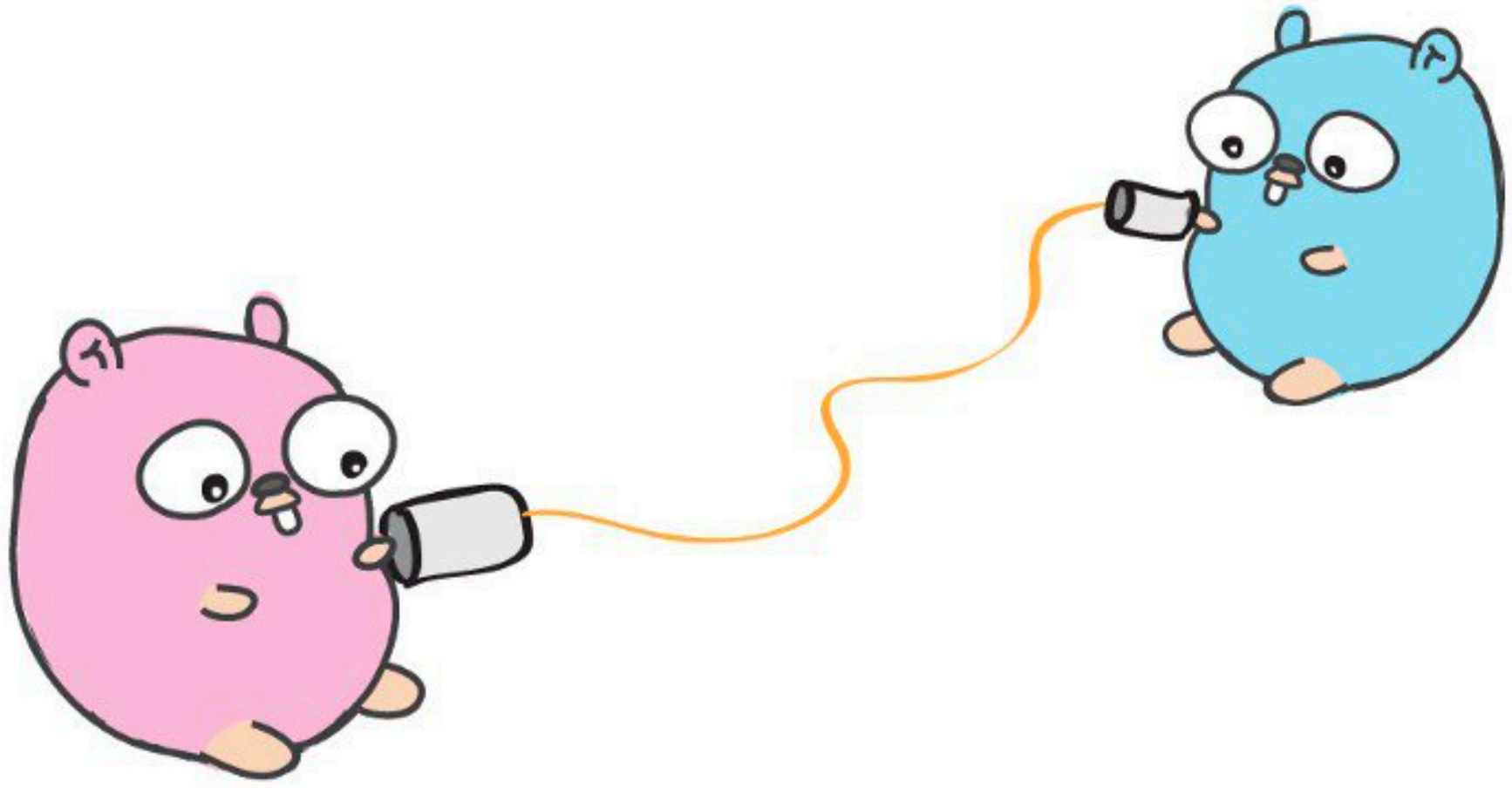


Go Concurrency Part 2

Jin Hyun Kim

Fall, 2019

Go Concurency

- select + Channel
- Channels
- Define Order of Routines
- Mutex

Channel Sync

```
package main

import "fmt"

func main() {
    done := make(chan bool)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(i)
        }
        done <- true
    }()

    // 위의 Go루틴이 끝날 때까지 대기
    <-done
}
```

https://brownbears.tistory.com/315

select

- Select a channel communication out of multiple channels simultaneously running
    - select문은 복수 채널들을 기다리면서 준비된 (데이타를 보내온) 채널을 실행하는 기능을 제공
    - select문은 여러 개의 case문에서 각각 다른 채널을 기다리다가 준비가 된 채널 case를 실행
    - select문은 case 채널들이 준비되지 않으면 계속 대기하게 되고, 가장 먼저 도착한 채널의 case를 실행
    - 만약 복수 채널에 신호가 오면, Go 런타임이 랜덤하게 그 중 한 개를 선택
    - select문에 default 문이 있으면, case문 채널이 준비되지 않더라도 계속 대기하지 않고 바로 default문을 실행

select

```go
package main

import "time"

func main() {
    done1 := make(chan bool)
    done2 := make(chan bool)

    go run1(done1)
    go run2(done2)

EXIT:
    for {
        select {
        case <-done1:
            println("run1 완료")

        case <-done2:
            println("run2 완료")
            break EXIT
        }
    }
}

func run1(done chan bool) {
    time.Sleep(1 * time.Second)
    done <- true
}

func run2(done chan bool) {
    time.Sleep(2 * time.Second)
    done <- true
}
```

http://golang.site/go/article

- 첫번째 run1()이 1초간 실행되고 done1 채널로부터 수신하여 해당 case를 실행

- 다시 for 루프를 돈다. for루프를 다시 돌면서 다시 select문이 실행되는데, 다음 run2()가 2초후에 실행되고 done2 채널로부터 수신하여 해당 case를 실행

- done2 채널 case문에 break EXIT 이 있는데, 이 문장으로 인해 for 루프를 빠져나와 EXIT 레이블로 이동

select

```go
package main

import "time"

func main() {
    done1 := make(chan bool)
    done2 := make(chan bool)

    go run1(done1)
    go run2(done2)

EXIT:
    for {
        select {
        case <-done1:
            println("run1 완료")

        case <-done2:
            println("run2 완료")
            break EXIT
        }
    }
}

func run1(done chan bool) {
    time.Sleep(1 * time.Second)
    done <- true
}

func run2(done chan bool) {
    time.Sleep(2 * time.Second)
    done <- true
}
```

http://golang.site/go/article

default

fmt.Println("default") }

- 위의 문장을 마지막 case 뒤에 삽입
- 실험

```
package main

import (
   "fmt"
   "time"
)
func process(ch chan string) {
   time.Sleep(10 * time.Second)
   ch <- "process successful"
}

func scheduling(){
   //do something
}
func main() {
   ch := make(chan string)
   go process(ch)
   for {
      time.Sleep(1 * time.Second)
      select {
      case v := <-ch:
         fmt.Println("received value: ", v)
         return
      default:
         fmt.Println("no value received")
      }

      scheduling()
   }
}
```

https://hamait.tistory.com/1017

- select: Producer and Consumer

```
package main

import (
"fmt"
"time"
)


func consuming (scheduler chan string){
  select {
  case <- scheduler:
        fmt.Println("이름을 입력받았습니다.")
  case <-time.After(5 * time.Second):
        fmt.Println("시간이 지났습니다.")
  }
}

func producing(scheduler chan string){
  var name string
  fmt.Print("이름:")
  fmt.Scanln(&name)
  scheduler <- name
}

func main() {
  scheduler := make(chan string)
  go consuming(scheduler)
  go producing(scheduler)

  time.Sleep(100 * time.Second)
}
```

https://hamait.tistory.com/1017

- select: Producer and Consumer with timeout

```
package main

import (
"fmt"
"time"
)


func consuming (scheduler chan string){
  select {
  case <- scheduler:
        fmt.Println("이름을 입력받았습니다.")
  case <-time.After(5 * time.Second):
        fmt.Println("시간이 지났습니다.")
  }
}

func producing(scheduler chan string){
  var name string
  fmt.Print("이름:")
  fmt.Scanln(&name)
  scheduler <- name
}

func main() {
  scheduler := make(chan string)
  go consuming(scheduler)
  go producing(scheduler)

  time.Sleep(100 * time.Second)
}
```

```
package main

import (
"fmt"
"time"
)

var scheduler chan string

func consuming (prompt string){ fmt.Println("consuming 호출됨")
  select {
  case scheduler <- prompt:
        fmt.Println("이름을 입력받았습니다 : ", <- scheduler)
  case <-time.After(5 * time.Second):
         fmt.Println("시간이 지났습니다.")
  }
}

func producing (console chan string) {
  var name string
  fmt.Print("이름:")
  fmt.Scanln(&name)
  console <- name
}
func main() {
  console := make(chan string, 1)
  scheduler = make(chan string, 1)

  go func(){
  consuming(<-console)
  }()

  go producing(console)

  time.Sleep(100 * time.Second)
}
```

Buffered Channel

```
package main

import "fmt"

func main() {
  c := make(chan int)
  c <- 1   //수신루틴이 없으므로 데드락
  fmt.Println(<-c) //코멘트해도 데드락 (별도의 Go루틴없기 때문)
}
```

```
package main

import "fmt"

func main() {
    ch := make(chan int, 1)

    //수신자가 없더라도 보낼 수 있다.
    ch <- 101

    fmt.Println(<-ch)
}
```

Buffered Channel

```go
package main

import (
    "fmt"
)

func main() {
    numbers := make(chan int, 5)
    counter := 10

    for i := 0; i < counter; i++ {
        select {
        case numbers <- i:
        default:
            fmt.Println( a...: "Not enough space for", i)
        }
    }

    for i := 0; i < counter+5; i++ {
        select {
        case num := <-numbers:
            fmt.Println(num)
        default:
            fmt.Println( a...: "Nothing more to be done!")
            break
        }
    }
}

```

Terminate Go Routine

```go
package main

import ...

func main() {
    c1 := make(chan string)
    go func() {
        time.Sleep(time.Second * 3)
        c1 <- "c1 OK"
    }()

    select {
    case res := <-c1:
        fmt.Println(res)
    case <-time.After(time.Second * 1):
        fmt.Println( a...: "timeout c1")
    }

    c2 := make(chan string)
    go func() {
        time.Sleep(3 * time.Second)
        c2 <- "c2 OK"
    }()

    select {
    case res := <-c2:
        fmt.Println(res)
    case <-time.After(4 * time.Second):
        fmt.Println( a...: "timeout c2")
    }
}
```

Nil Channel

```go
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func add(c chan int) {
    sum := 0
    t := time.NewTimer(time.Second)

    for {
        select {
        case input := <-c:
            sum = sum + input
        case <-t.C:
            c = nil
            fmt.Println(sum)
        }
    }
}

func send(c chan int) {
    for {
        c <- rand.Intn( n: 10)
    }
}

func main() {
    c := make(chan int)
    go add(c)
    go send(c)

    time.Sleep(3 * time.Second)
}
```

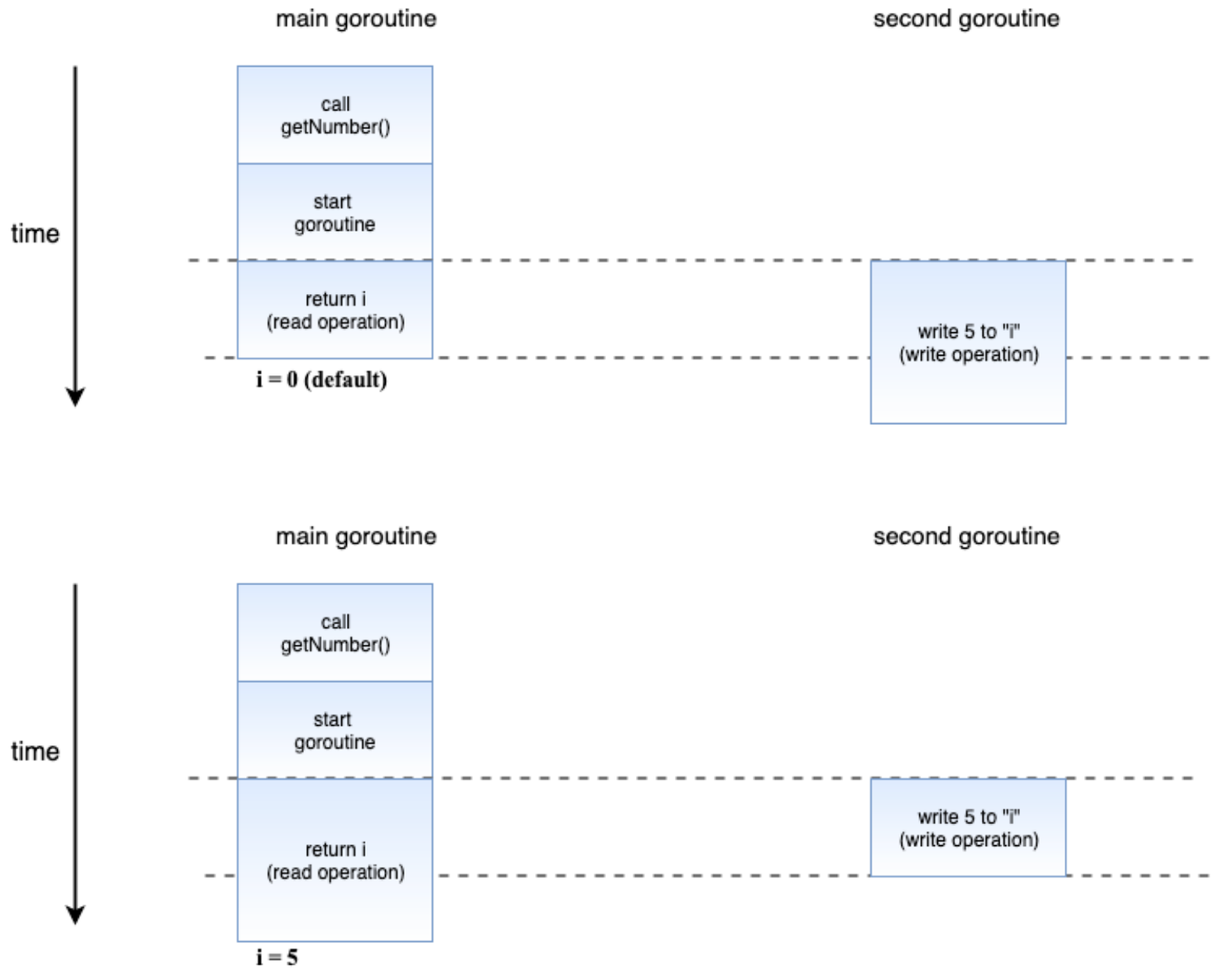
Define Order

```go
package main

import (
    "fmt"
    "time"
)

func A(a, b chan struct{}) {
    <-a
    fmt.Println( a...: "A()!")
    time.Sleep(time.Second)
    close(b)
}

func B(a, b chan struct{}) {
    <-a
    fmt.Println( a...: "B()!")
    close(b)
}

func C(a chan struct{}) {
    <-a
    fmt.Println( a...: "C()!")
}

func main() {
    x := make(chan struct{})
    y := make(chan struct{})
    z := make(chan struct{})

    go C(z)
    go A(x, y)
    go C(z)
    go B(y, z)
    go C(z)

    close(x)
    time.Sleep(3 * time.Second)
}
```

- **close(a)** closes the channel a, that synchronizes with a routine waiting for the channel a

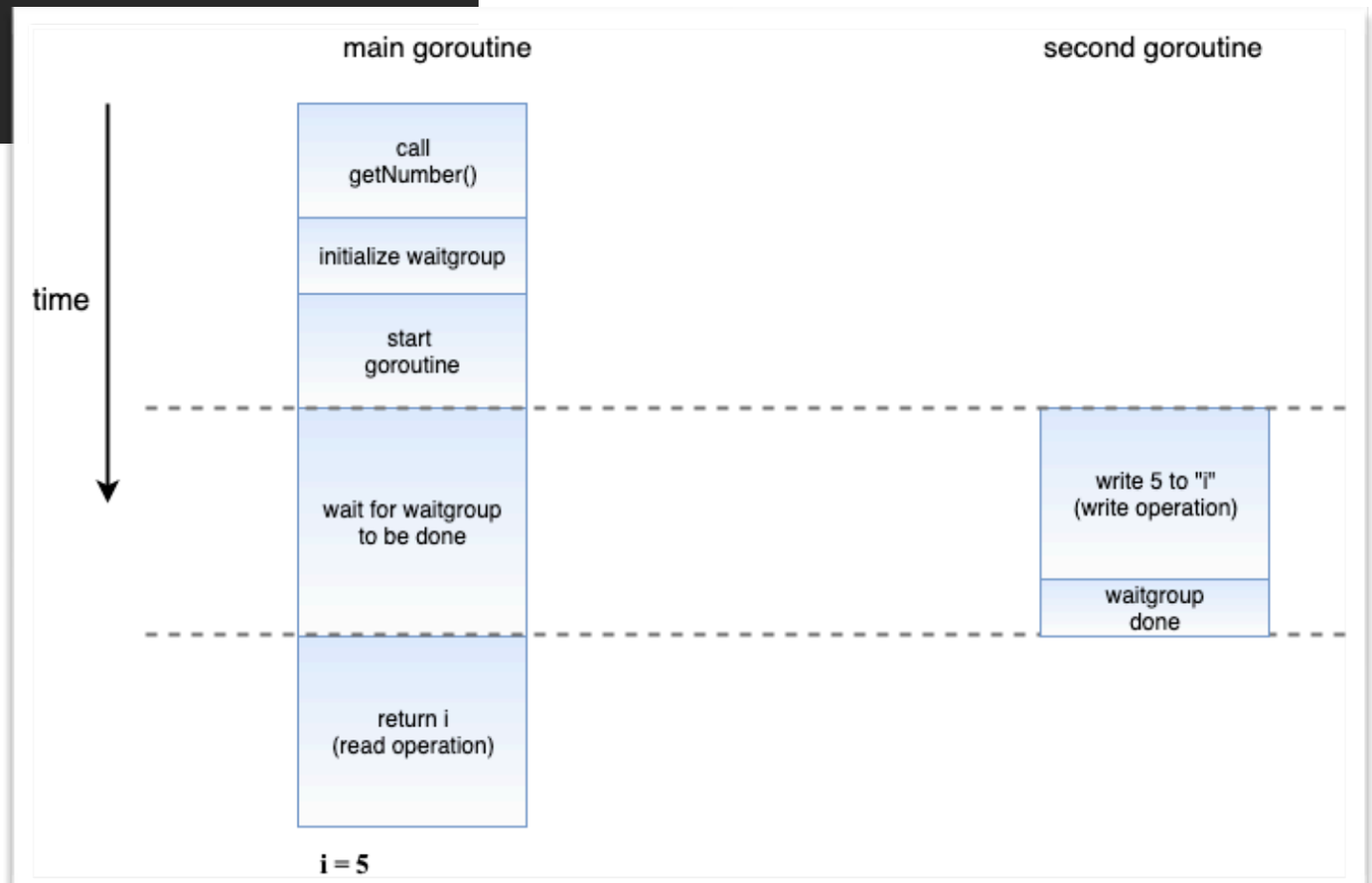
Data Race

```go
func main() {
    fmt.Println(getNumber())
}

func getNumber() int {
    var i int
    go func() {
        i = 5
    }()

    return i
}
```



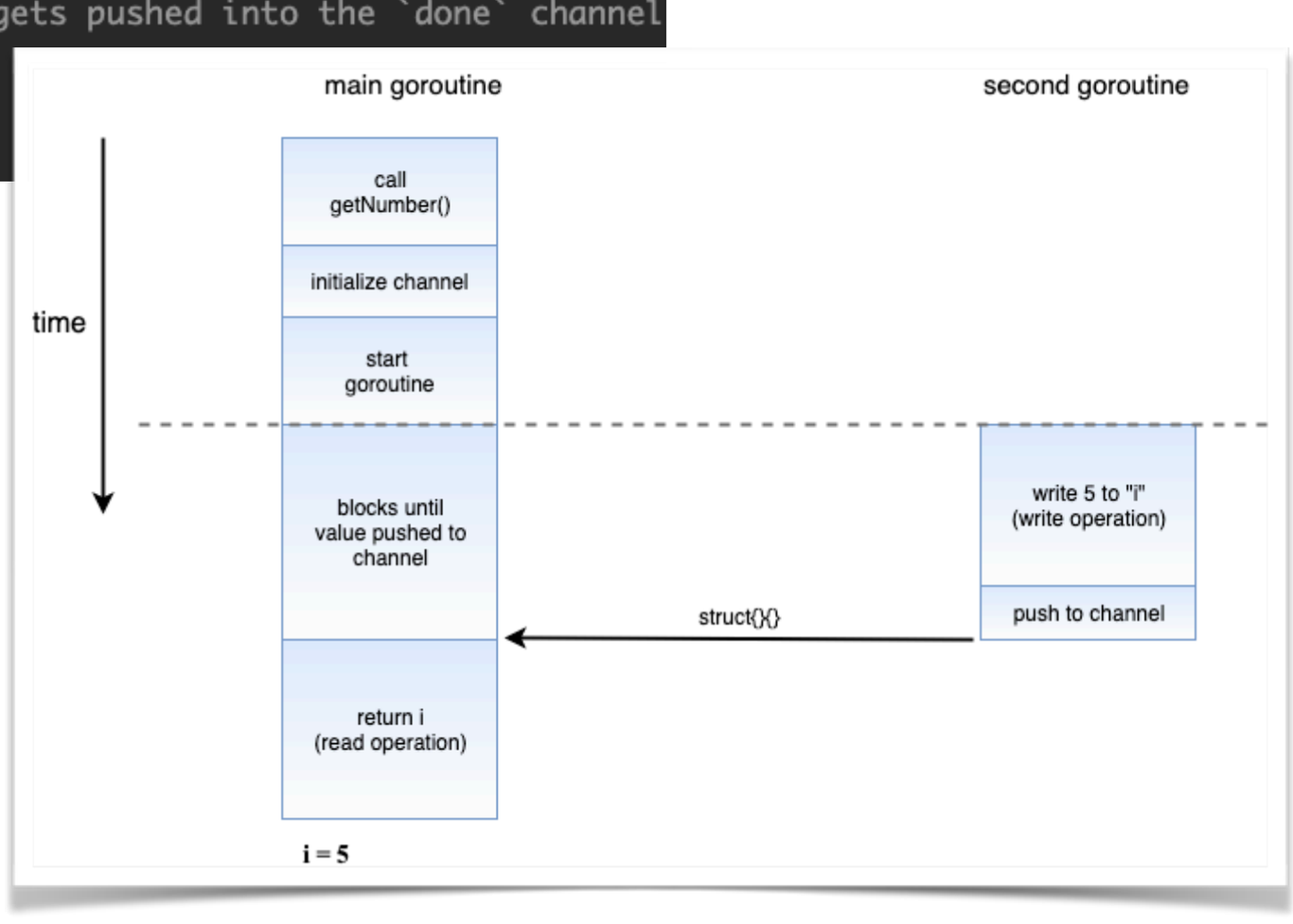
First Solution

```go
func getNumber() int {
    var i int
    // Initialize a waitgroup variable
    var wg sync.WaitGroup
    // `Add(1) signifies that there is 1 task that we need to wait for
    wg.Add(1)
    go func() {
        i = 5
        // Calling `wg.Done` indicates that we are done with the task we are wait
        wg.Done()
    }()
    // `wg.Wait` blocks until `wg.Done` is called the same number of times
    // as the amount of tasks we have (in this case, 1 time)
    wg.Wait()
    return i
}
```



Second Solution

```go
func getNumber() int {
    var i int
    // Create a channel to push an empty struct to once we're done
    done := make(chan struct{})
    go func() {
        i = 5
        // Push an empty struct once we're done
        done <- struct{}{}
    }()
    // This statement blocks until something gets pushed into the `done` channel
    <-done
    return i
}
```



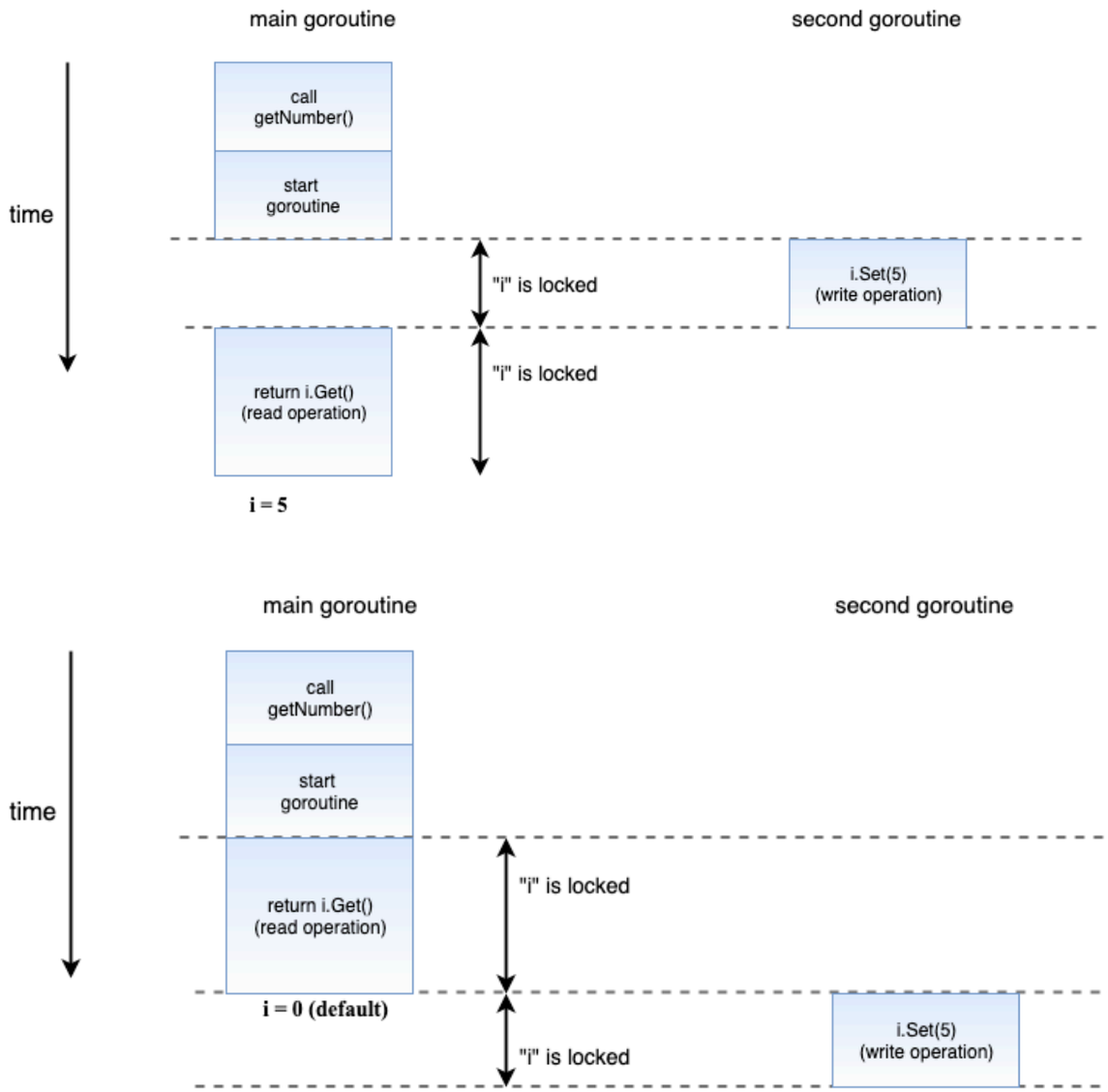
Mutex

```go
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)

var (
    m  sync.Mutex
    v1 int
)

func change(i int) {
    m.Lock()
    time.Sleep(time.Second)
    v1 = v1 + 1
    if v1%10 == 0 {
        v1 = v1 - 10*i
    }
    m.Unlock()
}

func read() int {
    m.Lock()
    a := v1
    m.Unlock()
    return a
}
```

```go
func main() {
    if len(os.Args) != 2 {
        fmt.Println( a...: "Please give me an integer!")
        return
    }

    numGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    var waitGroup sync.WaitGroup

    fmt.Printf( format: "%d ", read())
    for i := 0; i < numGR; i++ {
        waitGroup.Add( delta: 1)
        go func(i int) {
            defer waitGroup.Done()
            change(i)
            fmt.Printf( format: "-> %d", read())
        }(i)
    }

    waitGroup.Wait()
    fmt.Printf( format: "-> %d\n", read())
}
```

Third Solution

```go
// First, create a struct that contains the value we want to return
// along with a mutex instance
type SafeNumber struct {
    val int
    m   sync.Mutex
}

func (i *SafeNumber) Get() int {
    // The `Lock` method of the mutex blocks if it is already locked
    // if not, then it blocks other calls until the `Unlock` method is called
    i.m.Lock()
    // Defer `Unlock` until this method returns
    defer i.m.Unlock()
    // Return the value
    return i.val
}

func (i *SafeNumber) Set(val int) {
    // Similar to the `Get` method, except we Lock until we are done
    // writing to `i.val`
    i.m.Lock()
    defer i.m.Unlock()
    i.val = val
}

func getNumber() int {
    // Create an instance of `SafeNumber`
    i := &SafeNumber{}
    // Use `Set` and `Get` instead of regular assignments and reads
    // We can now be sure that we can read only if the write has completed, or vi
    go func() {
        i.Set(5)
    }()
    return i.Get()
}
```



Conclusions

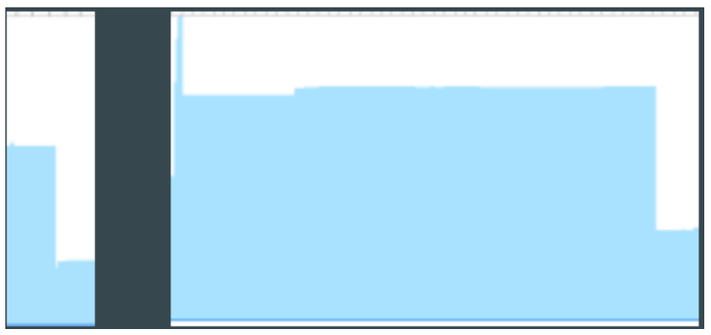
- Go Routine - LightweightThread

<Speed>

	Thread (Java)	Goroutine
개수	100000	100000
처리속도	5.902s	226.750829ms

<CPU>

<Memory>



Conclusions

- Channel - Communication mechanism for go routines to exchange data with each other

- Mutex - Synchronization mechanism for go rountines to interact with each other