alterra
academy

Concurrency in Golang

OUTLINE

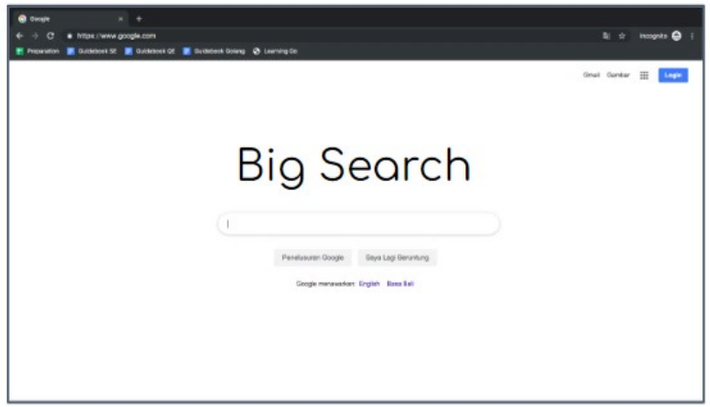
alterra academy

- “Big Search” Website
- Different Sequential, Parallel, and Concurrent
- Goroutines
- Channels and Select
- Race Condition
- Fixing Data Races
  - Channels Blocking
  - WaitGroups
  - Mutex

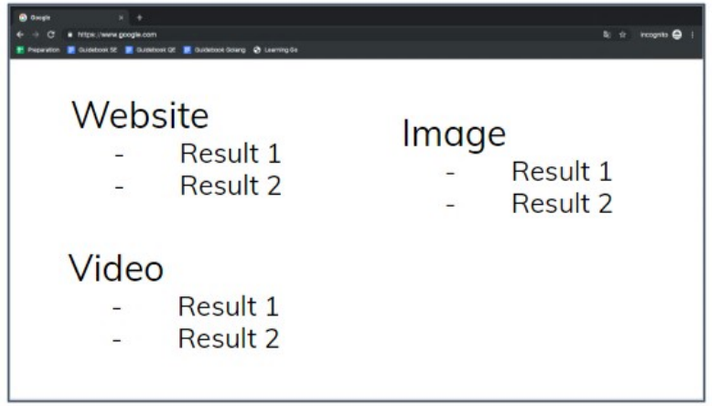
alterra academy

Let's get started!

"BIG SEARCH" WEBSITE



display result



This search engine will search the web for **websites, images, and videos.**

alterra
academy

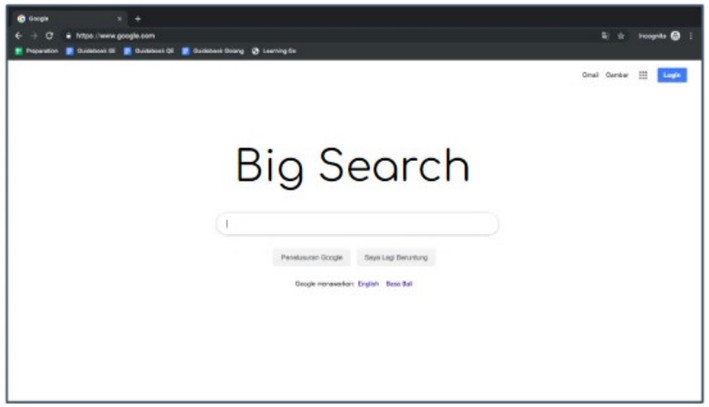
SEQUENTIAL VS PARALLEL VS CONCURRENT

alterra academy

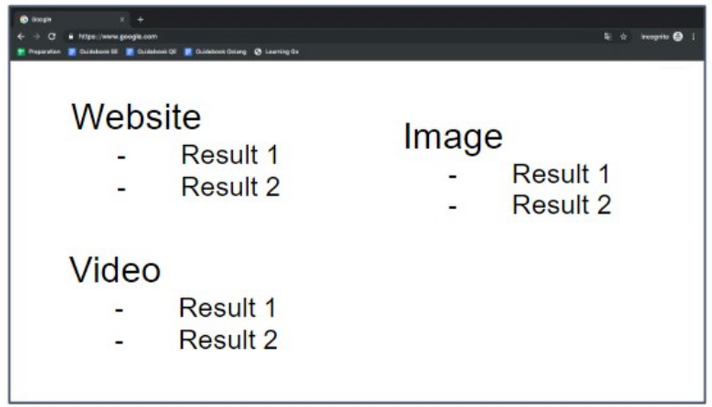
SEQUENTIAL PROGRAM

In sequential programs, before a new task starts, the previous one must finish.

SEQUENTIAL PROGRAMS



display result



Illustration

(Total 9 seconds)

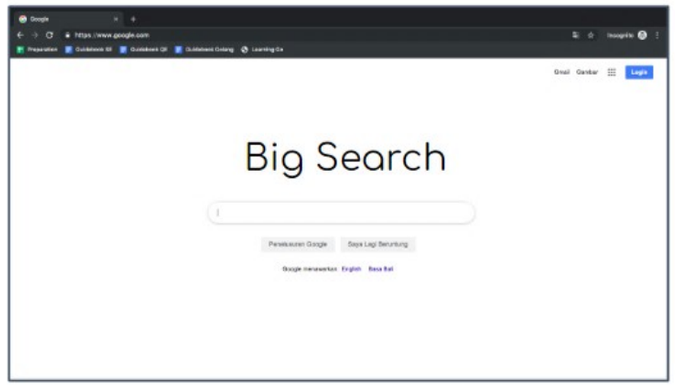
alterra academy

PARALLEL PROGRAM

*In parallel programs, multiple tasks can be executed **simultaneously**.*

(requires multi-core machines)

PARALLEL PROGRAMS



display result

Website
- Result 1
- Result 2

Image
- Result 1
- Result 2

Video
- Result 1
- Result 2

(Total 4 seconds)

Illustration multi-core machines

2 seconds
video

3 seconds
website

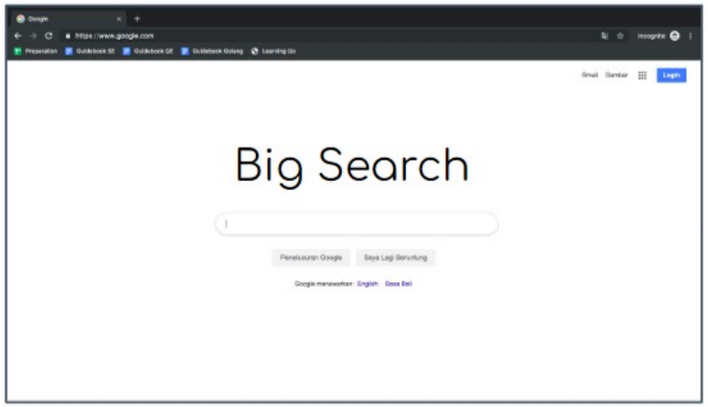
4 seconds
image

alterra academy

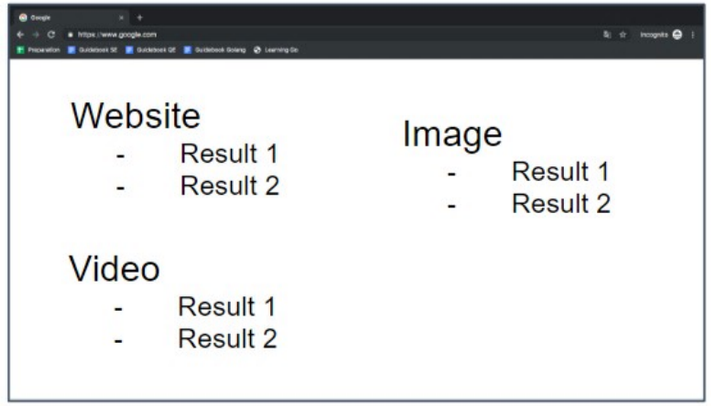
CONCURRENT PROGRAM

*In concurrent programs, multiple tasks can be executed **independently** and may **appear simultaneous**.*

CONCURRENT PROGRAMS



display result



Illustration

(Total 9 s)

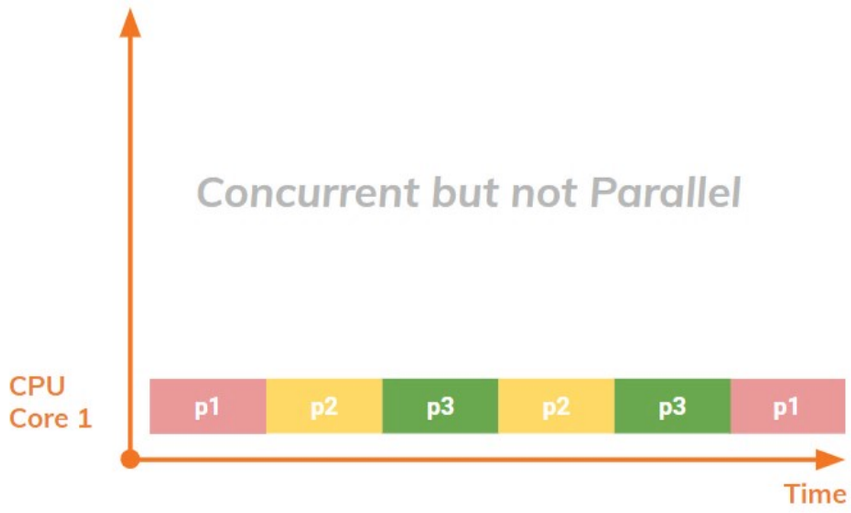
alterra academy

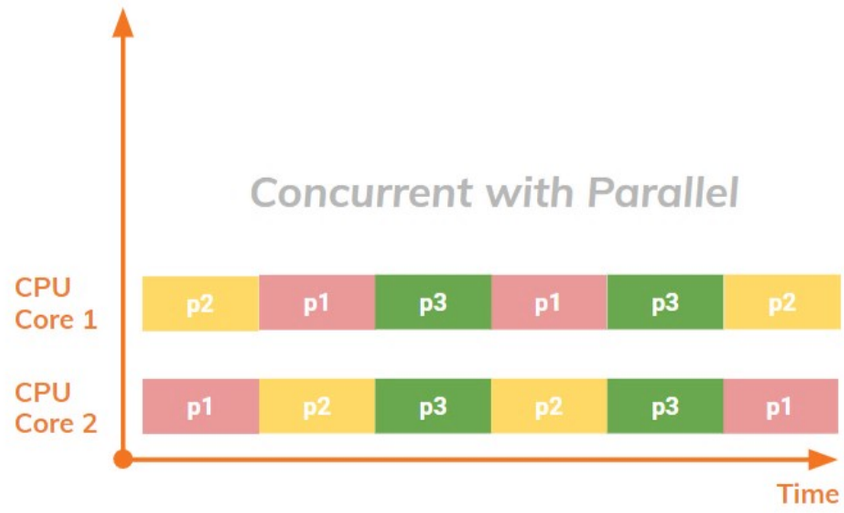
Parallel means
Simultaneous,
Concurrent means
Independent.

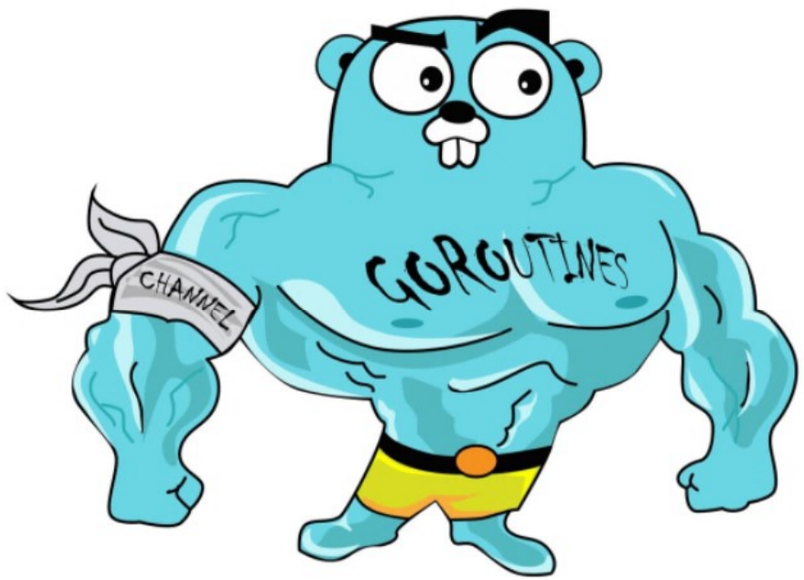
alterra academy

CONCURRENCY ALLOW PARALLELISM

Go's concurrency (goroutines) make it simple to build parallel programs that take advantage of machines with multiple processors (most machines today).







alterra
academy

Goroutines

alterra
academy

FEATURE GO

- Concurrent execution **(Goroutines)**.
- Synchronization and messaging **(Channels)**.
- Multi-way concurrent control **(Select)**.

alterra
academy

WHAT IS GOROUTINE?

Goroutines are functions or methods that ***run concurrently (independent) with other functions or methods****.*

The cost of creating a Goroutine is tiny when compared to a thread. A thread is a lightweight process, or in other words, a thread is a unit which executes the code under the program.

EXAMPLE
USING GOROUTINE

THE MAIN FUNCTION IS GOROUTINE

alterra academy

-code editor

```go
package main

import (
  "fmt"
  "time"
)

func hello() {
  fmt.Println("Hello world goroutine")
}

func main() {
  // go routine with keyword go
  go hello()
  fmt.Println("main function")
  time.Sleep(1 * time.Second)
}
```

alterra academy

USING GOROUTINE AND THREADS

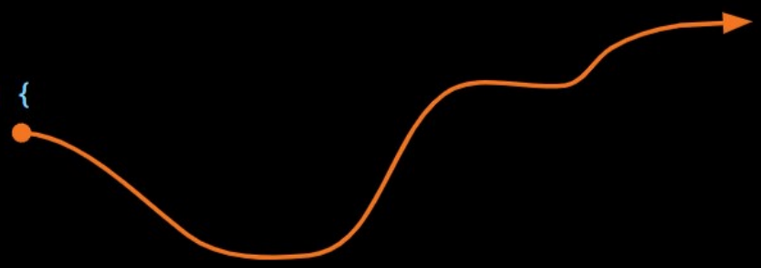
-code editor

```go
package main

import (
  "fmt"
  "time"
)

func hello(s string) {
  for i := 0; i < 5; i++ {
    time.Sleep(100 * time.Millisecond)
    fmt.Println(s)
  }
}

func main() {
  go hello("world") // using goroutine
  hello("hello ")   // using thread
}
```

USING MULTIPLE GOROUTINE

alterra academy

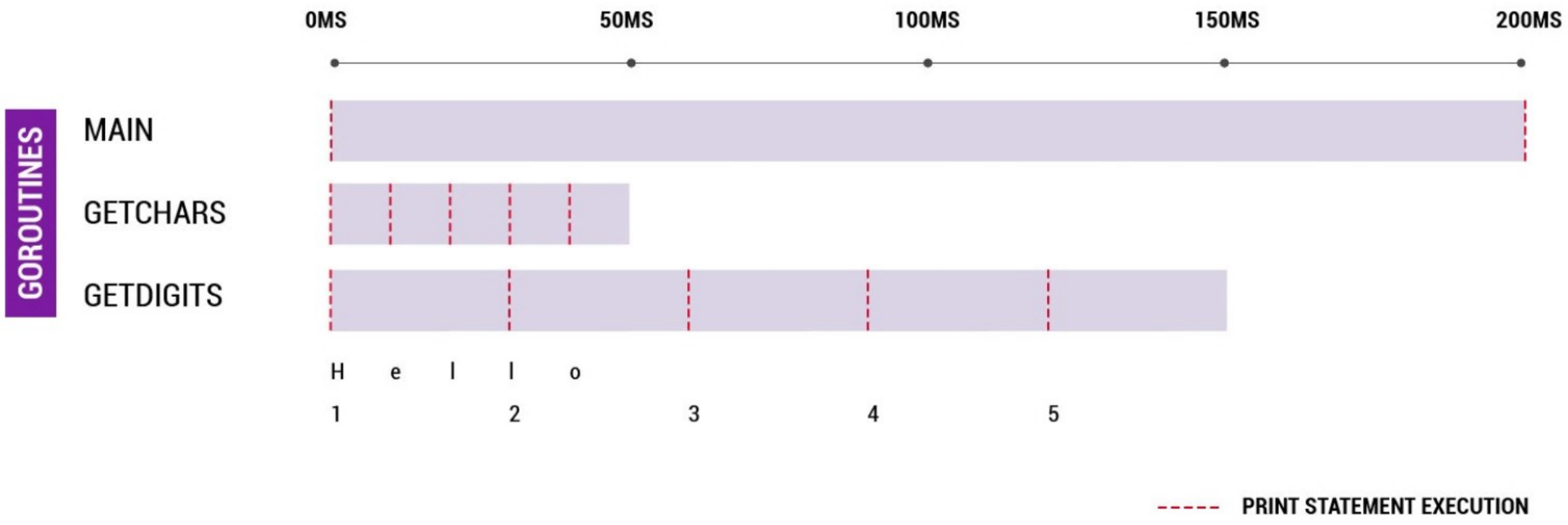
```go
package main

import (
  "fmt"
  "time"
)

var start time.Time

func init() {
  start = time.Now()
}

func getChars(sentence string) {
  for _, c := range sentence {
    fmt.Printf("%c at time %v\n", c, time.Since(start))
    time.Sleep(10 * time.Millisecond)
  }
}

func getDigits(digits []int) {
  for _, d := range digits {
    fmt.Printf("%d at time %v\n", d, time.Since(start))
    time.Sleep(30 * time.Millisecond)
  }
}

func main() {
  // code
}
```



```go
fmt.Println("main execution started at time",
time.Since(start))
fmt.Println()
// getChars goroutine
go getChars("Helo")
// getDigits goroutine
go getDigits([]int{1, 2, 3})
// schedule another goroutine
time.Sleep(200 * time.Millisecond)
fmt.Println("\nmain execution stopped at time",
time.Since(start))
```

alterra academy

MULTIPLE GOROUTINE EXECUTION



alterra academy

GOMAXPROCS

Gomaxprocs is used to **control the number of operating systemthreads** available for Goroutine execution to a particular go program.

-terminal

```
$ time GOMAXPROCS=1 go run main.go
```

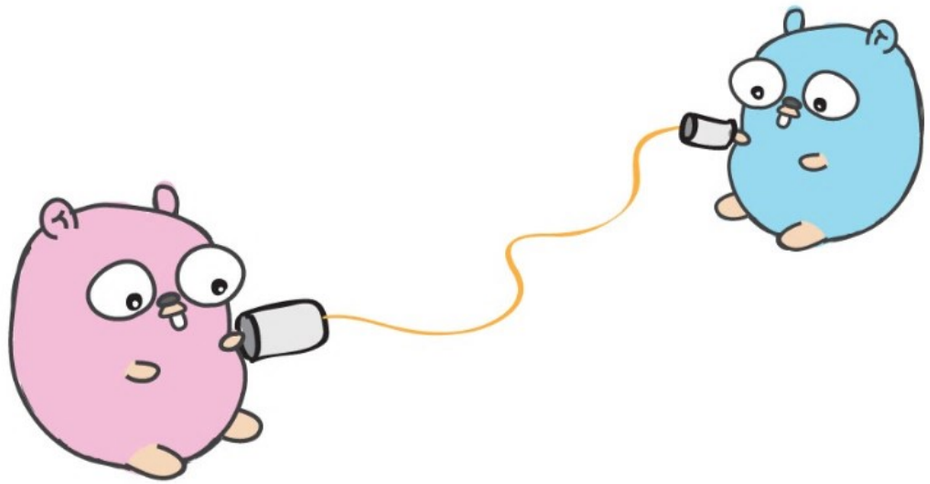
alterra
academy

Channel & Select

alterra
academy

CHANNEL

A channel is a communication object using which goroutines can communicate with each other.



alterra academy

DECLARATION AND USING CHANNEL

-code editor

```go
package main

import "fmt"

func greet(c chan string) {
  data := <-c //  storing data from channel c
  fmt.Println("Hello " + data + "!")
}

func main() {
  fmt.Println("main() started")
  c := make(chan string)

  go greet(c)

  c <- "John" // push or write data to the channel c
  fmt.Println("main() stopped")
}
```

Example using channel

```
# OUTPUT
Miner: Received ore1 from finder
Miner: Received ore2 from finder
Miner: Received ore3 from finder
```

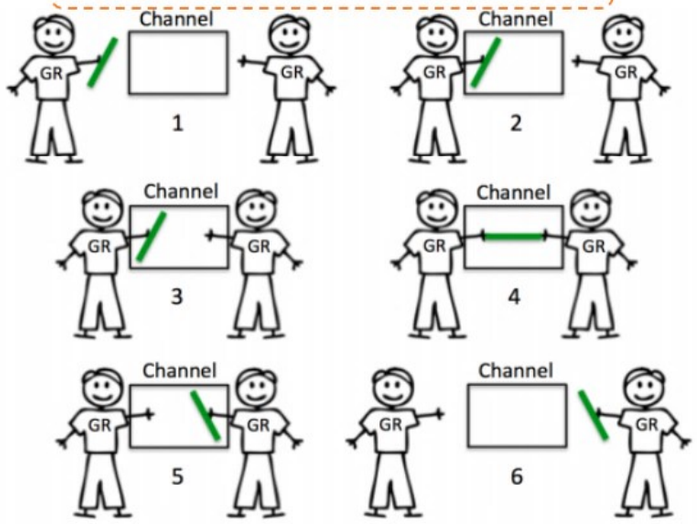
```go
package main

import (
  "fmt"
  "time"
)

func main() {
  theMine := [5]string{"ore1", "ore2", "ore3"}
  oreChan := make(chan string)

  // Finder
  go func(mine [5]string) {
    for _, item := range mine {
      oreChan <- item //send
    }
  }(theMine)

  // Ore Breaker
  go func() {
    for i := 0; i < 3; i++ {
      foundOre := <-oreChan //receive
      fmt.Println("Miner: Received " + foundOre + "from finder")
    }
  }()
  <-time.After(time.Second * 5) // Again, ignore this for now
}
```

alterra academy

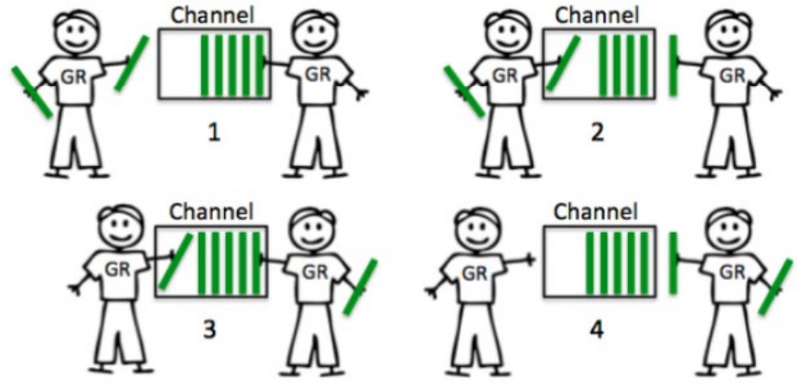
alterra academy

UNBUFFERED AND BUFFERED CHANNELS

```
c := make(chan string)
```




```
c := make(chan string, 42)
```



alterra academy

EXAMPLE BUFFERED CHANNELS

-code editor

```go
package main

import "fmt"

func main() {
  messages := make(chan string, 2) // buffered channel

  messages <- "buffered"
  messages <- "channel"

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

-terminal

```
$ go run channel-buffering.go
buffered
channel
```

alterra
academy

SELECT

Select makes it easier to control data communication through one or many channels.

SELECT EXAMPLE

alterra academy

```go
package main

import "fmt"
import "runtime"

func getAverage(numbers []int, ch chan float64) {
  var sum = 0
  for _, e := range numbers {
    sum += e
  }
  ch <- float64(sum) / float64(len(numbers))
}

func getMax(numbers []int, ch chan int) {
  var max = numbers[0]
  for _, e := range numbers {
    if max < e {
      max = e
    }
  }
  ch <- max
}

func main() {
  // code
}
```

```go
runtime.GOMAXPROCS(2)
var numbers = []int{3, 4, 3, 5, 6, 3, 2, 2, 3, 4, 6, 3}
fmt.Println("numbers :", numbers)
var ch1 = make(chan float64)
go getAverage(numbers, ch1)

var ch2 = make(chan int)
go getMax(numbers, ch2)

for i := 0; i < 2; i++ {
  select {
  case avg := <-ch1:
    fmt.Printf("Avg \t: %.2f \n", avg)
  case max := <-ch2:
    fmt.Printf("Max \t: %d \n", max)
  }
}
```

alterra academy

Race Condition

alterra
academy

RACE CONDITION

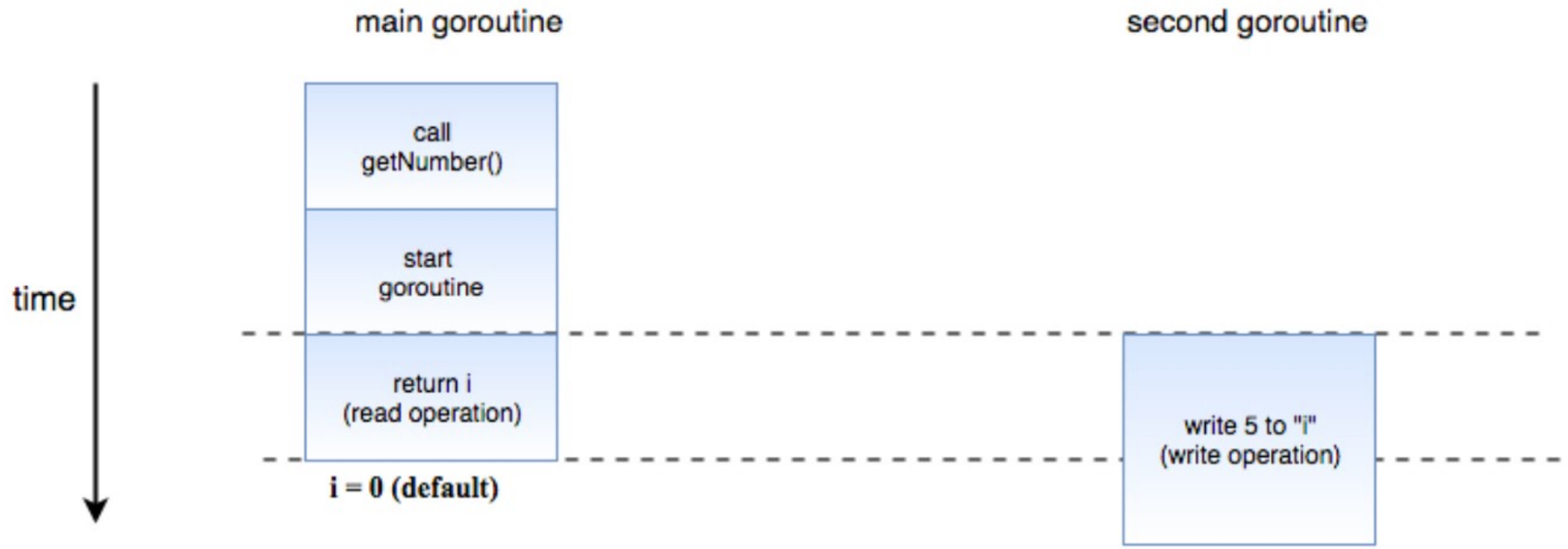
Race conditions are where 2 threads are accessing memory at the same time, one of which is writing. Race conditions occur because of unsynchronized access to shared memory.

Example Race Condition

```go
package main

import "fmt"

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

  return i
}

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

-terminal

```
$ go run -race main.go
```


main goroutine
second goroutine
time
call
getNumber()
start
goroutine
return i
(read operation)
write 5 to "i"
(write operation)
i = 0 (default)

alterra
academy

FIXING DATA RACE

alterra academy

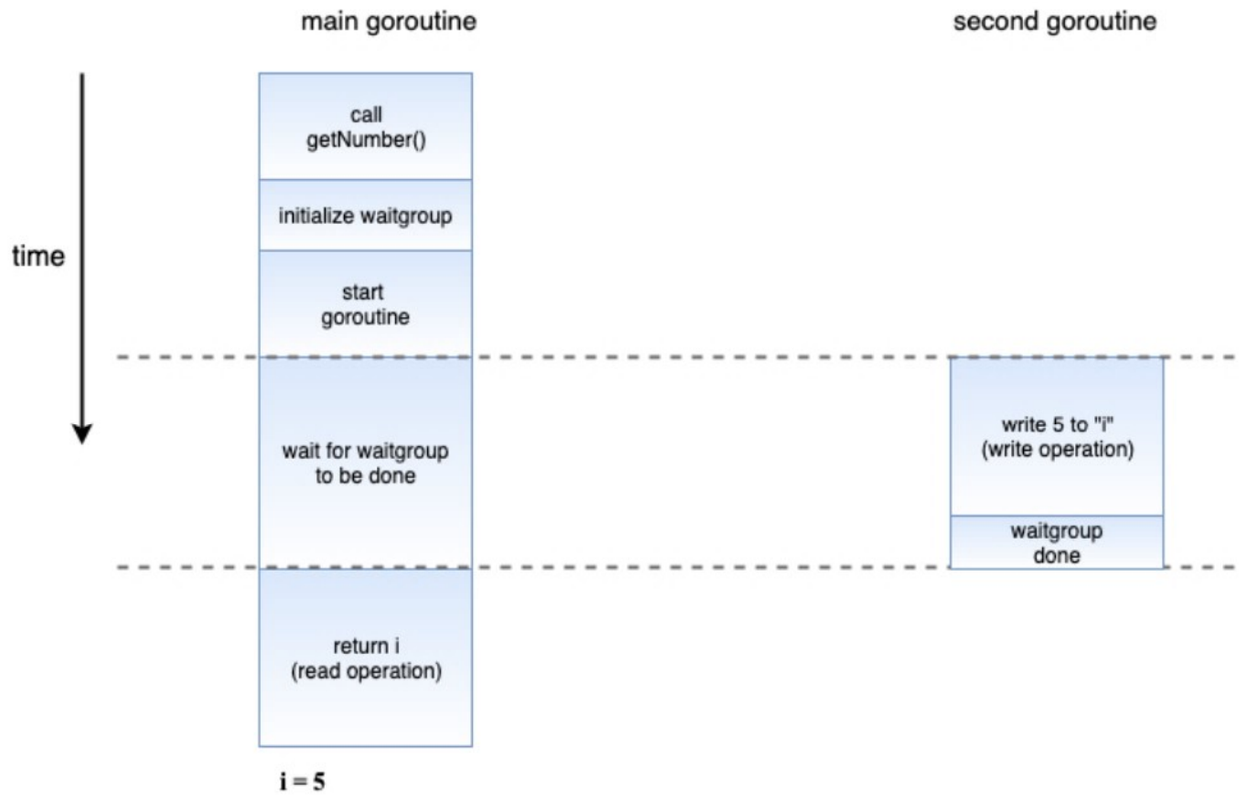
BLOCKING WITH WAITGROUPS

The most straightforward way of solving a data race, is to block read access until the write operation has been completed.

WAITGROUPS EXAMPLE

alterra academy

```go
package main

import (
  "fmt"
  "sync"
)

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

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 waiting fo
    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
}
```


main goroutine
second goroutine
time
call
getNumber()
initialize waitgroup
start
goroutine
wait for waitgroup
to be done
return i
(read operation)
write 5 to "i"
(write operation)
waitgroup
done
i = 5

alterra academy

CHANNEL BLOCKING

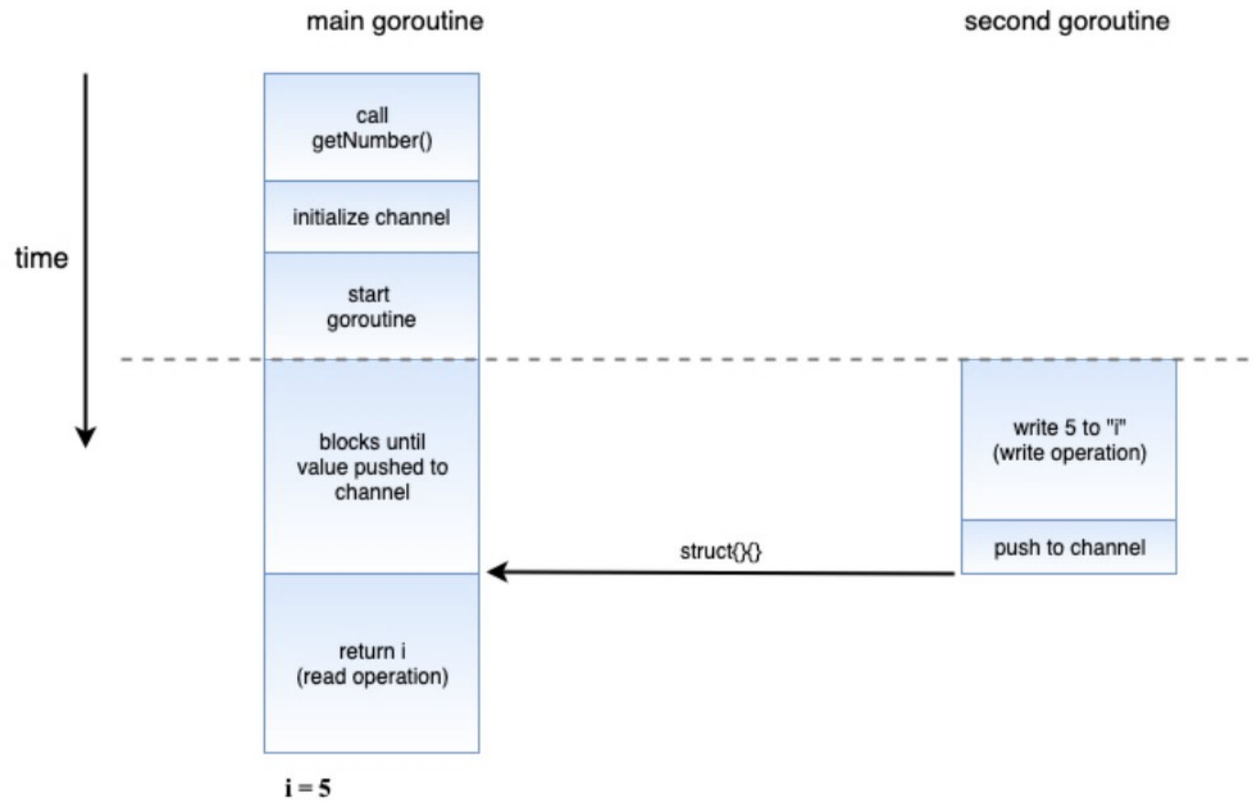
This allows our goroutines to sync up with each other for a moment.

alterra academy

CHANNEL BLOCKING

-code editor

```go
package main

import "fmt"

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
}

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


main goroutine
second goroutine
time
call
getNumber()
initialize channel
start
goroutine
blocks until
value pushed to
channel
return i
(read operation)
write 5 to "i"
(write operation)
push to channel
struct{}{}
i = 5

alterra academy

MUTEX

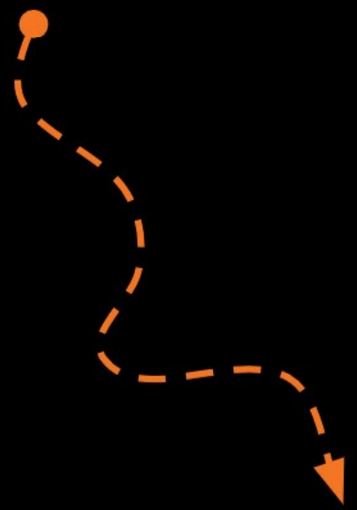
Where we don't care about the order of reads and writes, we only require that they do not occur simultaneously.

If this sounds like your use case, then we should consider using a mutex.

MUTEX EXAMPLE

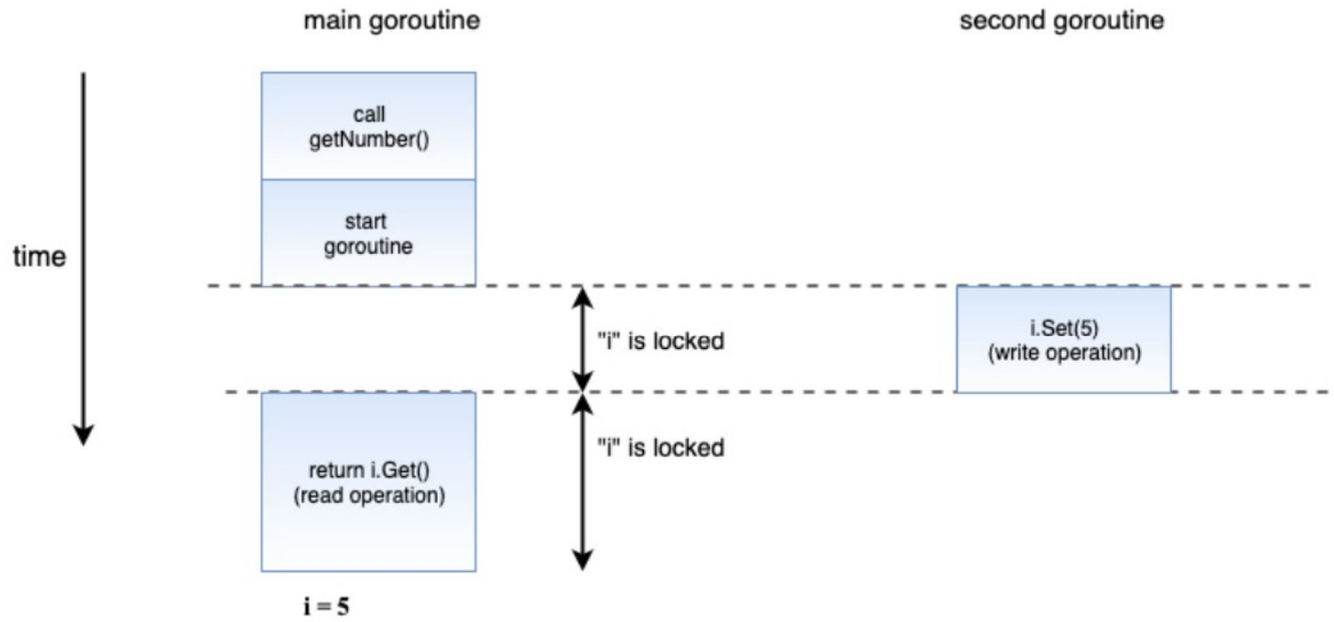
alterra academy

```go
package main

import (
  "fmt"
  "sync"
  "time"
)

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



alterra academy

```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 vice versa
  go func() {
    i.Set(5)
  }()
  time.Sleep(time.Second)
  return i.Get()
}
```


main goroutine
second goroutine
time
call
getNumber()
start
goroutine
"i" is locked
i.Set(5)
(write operation)
"i" is locked
return i.Get()
(read operation)
i = 5

alterra academy

“Make it work, make it right, make it fast.”

- Kent Beck -