Level 5–3

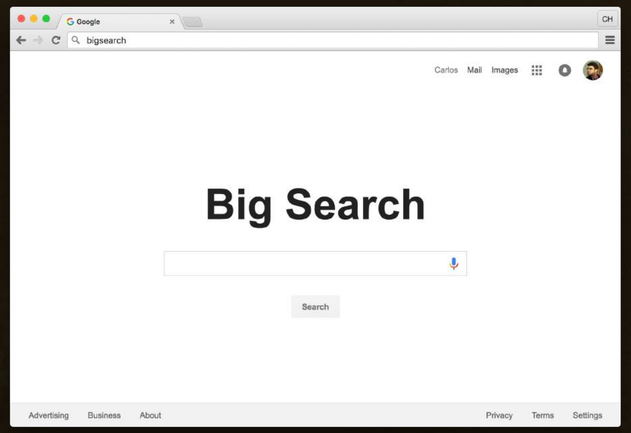
Gophers & Friends

goroutines

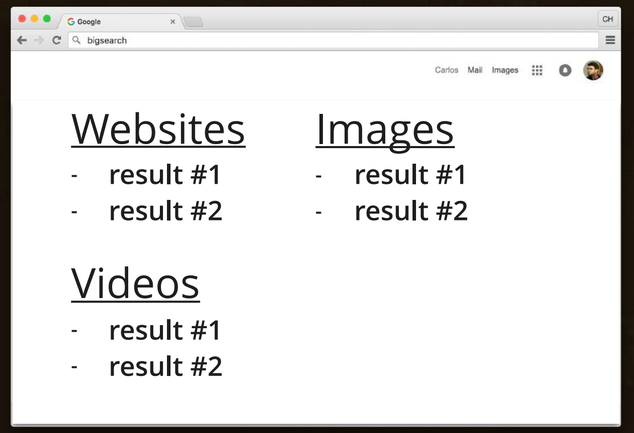
ON TRACK with GOLANG

The “Big Search” Website

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

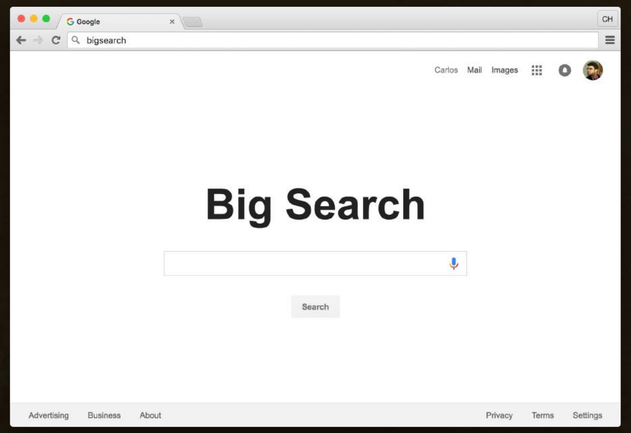


Display results for a search

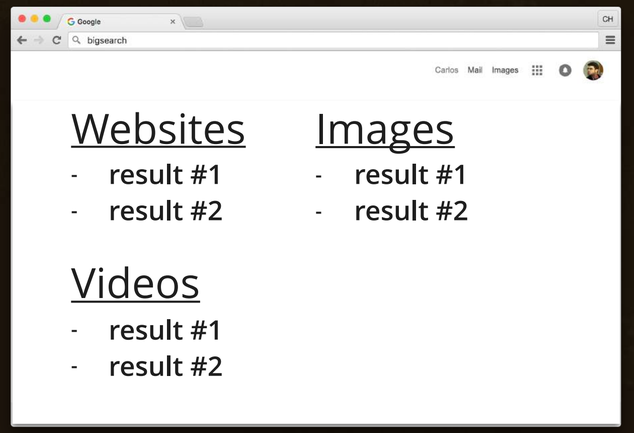


Sequential Programs

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



Searches websites, images, and videos one at a time



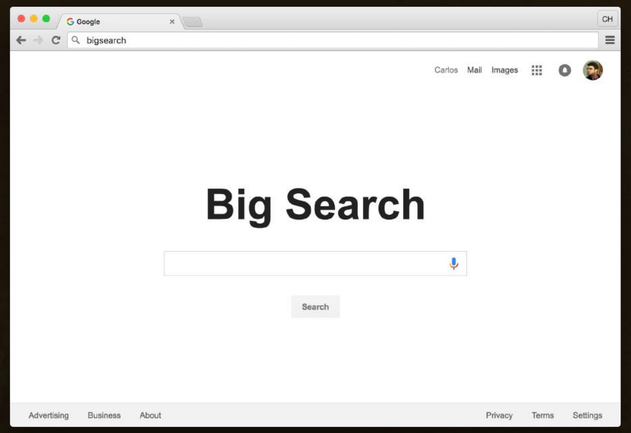
(Total: 9 seconds)

Websites *Images* *Videos*

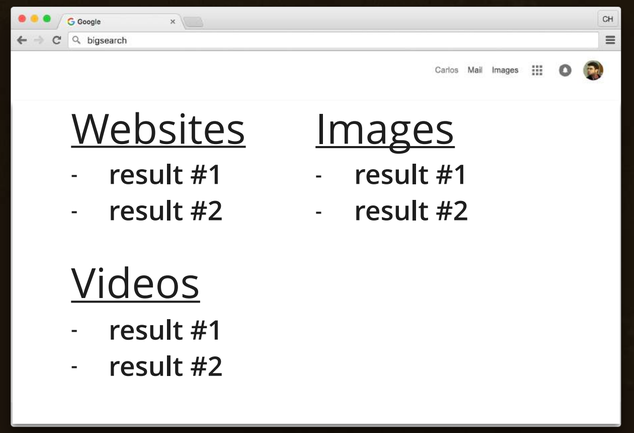
Takes as long as the sum of all of tasks

Concurrent Programs

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



Searches websites, images, and videos "at the same time"



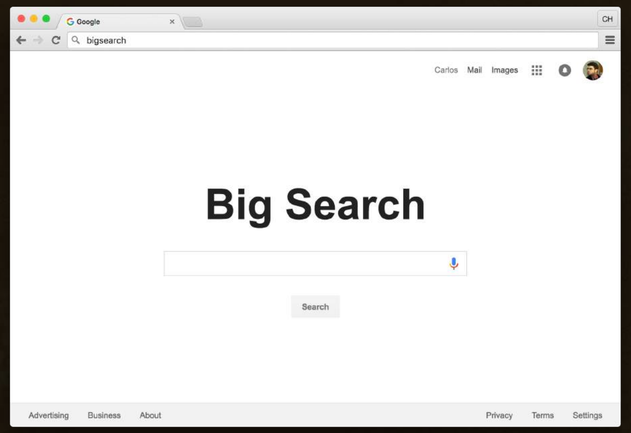
(Total: 9 seconds)

Can be executed in any order

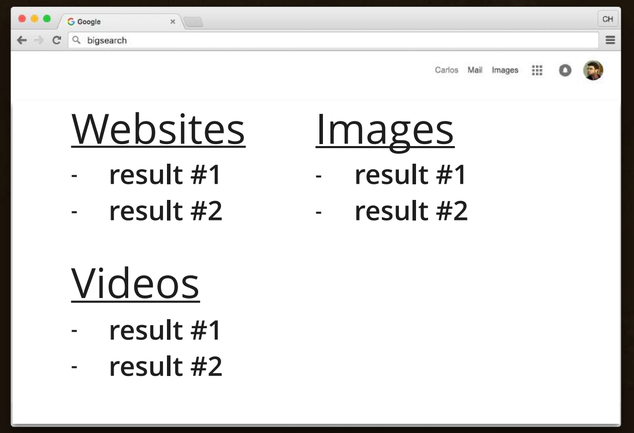
Also takes as long as the sum of all tasks, but tasks alternate time slices

Parallel Programs

In parallel programs, multiple tasks can be executed **simultaneously** (requires **multi-core** machines).



Searches websites, images, and videos **at the same time** *(really!)*



(Total: 4 seconds)

2 seconds

Websites

3 seconds

Images

4 seconds

Videos

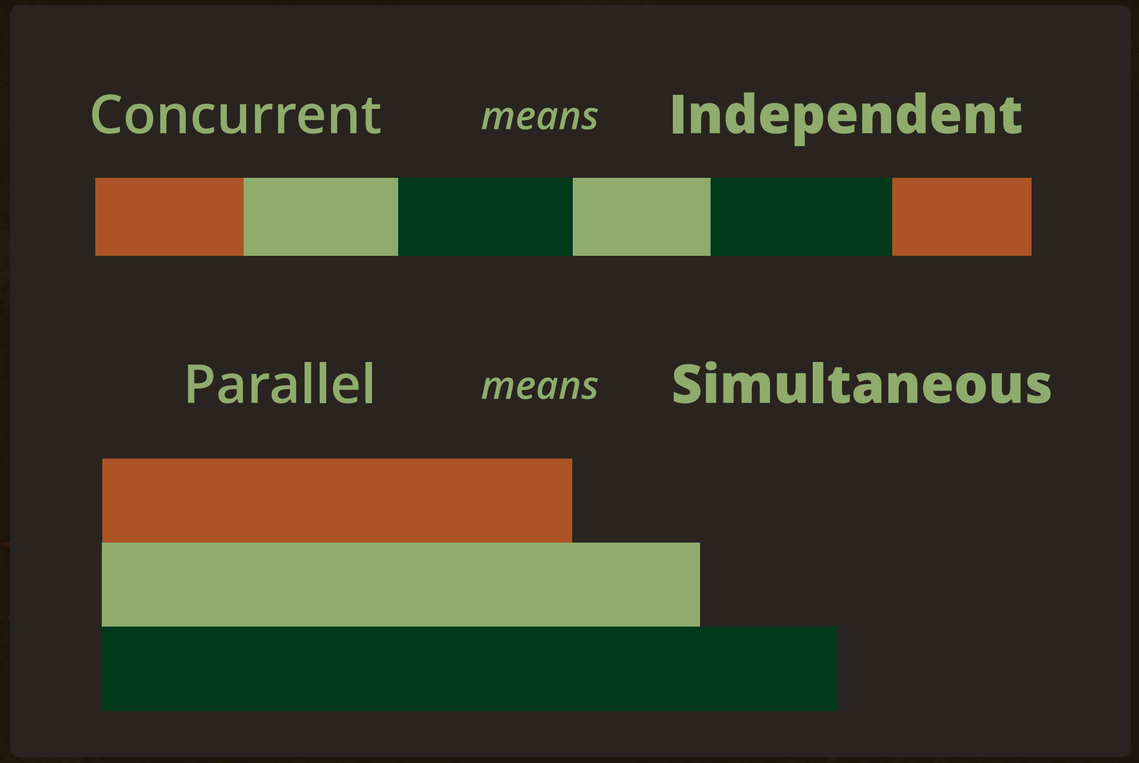
All executed at the same time

Takes as long as the slowest task

Concurrency Allows Parallelism

Concurrency and parallelism are **NOT** the same thing. The former means **independent,** which is a necessary step toward the latter, which means **simultaneous.**

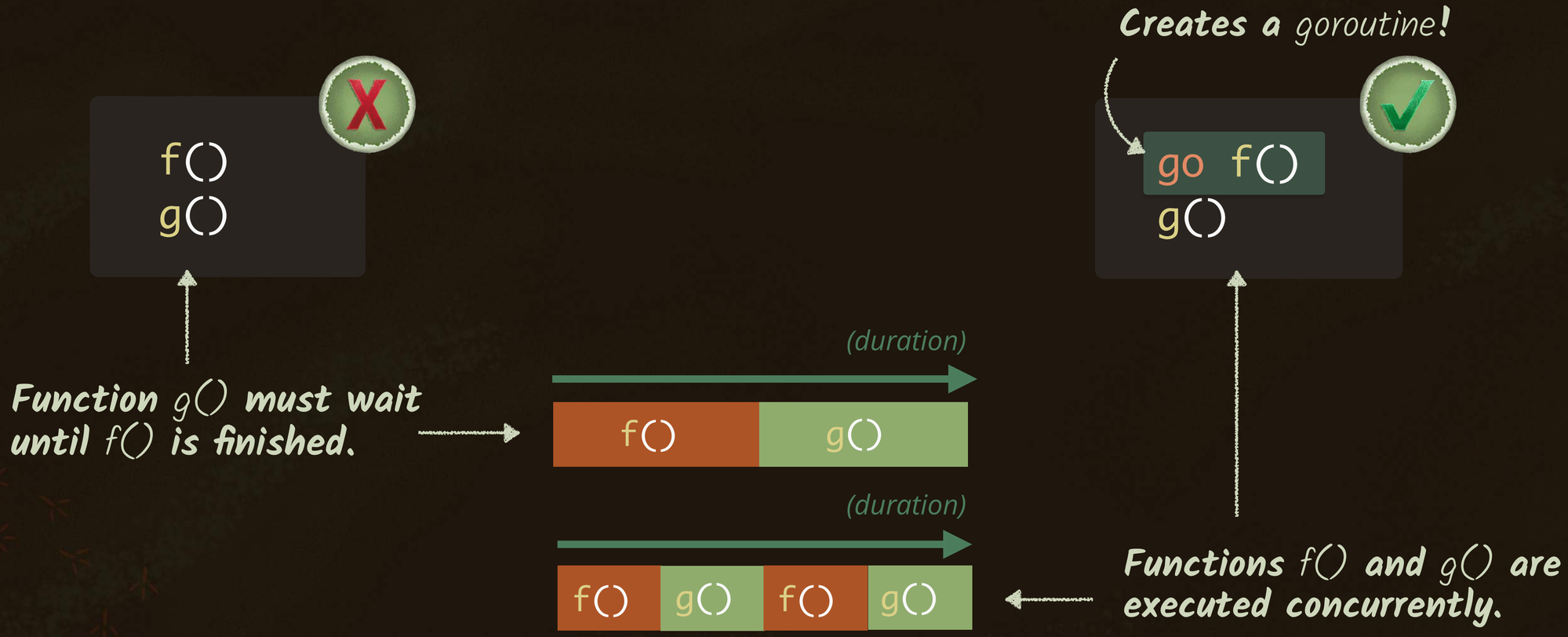
In short...



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

Concurrency With goroutines

A goroutine is a special function that executes **concurrently** with other functions. We create them with the go keyword (yes, go is also a keyword!).



On a single-core machine, concurrent code is unlikely to perform better than sequential code.

Looping and Printing Names

Let's write a new program that iterates through a slice and invokes a function printName() for each item.

```
package main

import "fmt"

func main() {
  names := []string{"Phil", "Noodles", "Barbaro"}
  for _, name := range names {
    printName(name)
  }
}

func printName(n string) {

}
```

Tracking Duration With the time Tool

The printName() function takes one argument and simply prints it to the console. We'll run our program with go run and use the Unix time command to track the duration of the execution.

```
...

func main() {
  names := []string{"Phil", "Noodles", "Barbaro"}
  for _, name := range names {
    printName(name)
  }
}

func printName(n string) {
  fmt.Println("Name: ", n)
}
```

Prints argument to the console

Determines duration of command passed to it.

```
$ time go run main.go
```

```
Name: Phil
Name: Noodles
Name: Barbaro

real 0m0.321s
```

Less than half a second

Heavy Processing Comes Into Play

Let's simulate a **time-consuming task** on printName(). We'll do this by adding a very costly mathematical operation using the math package from Go's standard library.

```
import (
  "fmt"
  "math"
)

func main() { ... }

func printName(n string) {
  result := 0.0
  for i := 0; i < 100000000; i++ {
    result += math.Pi * math.Sin(float64(len(n)))
  }
  fmt.Println("Name: ", n)
}
```

Import new package.

Time-consuming computation keeps the processor busy!

Sequential Tasks Are Blocking

When we add **heavy processing** to printName(), we can see a big **time increase on execution time.**

```
...
func main() {
  names := []string{"Phil", "Noodles", "Barbaro"}
  for _, name := range names {
    printName(name)
  }
}

func printName(n string) {
  result := 0.0
  for i := 0; i < 100000000; i++ {
    result += math.Pi * math.Sin(float64(len(n)))
  }
  fmt.Println("Name: ", n)
}
```

Each call to this function blocks the processor for almost 5 seconds!

```
$ time go run main.go
```

```
Name: Phil
Name: Noodles
Name: Barbaro

real 0m11.603s
```

Went from 0.3 to 11.6 seconds!

Running sequentially

Going Concurrent

Go programs **are NOT automatically aware** of newly created goroutines, **so the main function exits before the** goroutines **are finished.**

```
...
func main() {
  names := []string{"Phil", "Noodles", "Barbaro"}
  for _, name := range names {
    go printName(name)
  }
}

func printName(n string) {
  ...
}
```

Executes each function call on a new goroutine.

```
$ time go run main.go

real 0m0.314s
```

Back to being fast, but no names listed

Worry not, my friend. There's a built-in solution for this...

Adding Synchronization With WaitGroup

On the sync package from Go's standard library, there's a WaitGroup data type. We can use this type to make our program wait for goroutines to finish.

```
...
import (
  "fmt"
  "sync"
  "math"
)

func main() {
  names := []string{"Phil", "Noodles", "Barbaro"}
  var wg sync.WaitGroup
  ...
}
...
```

Import new package.

Declare a new variable of the sync.WaitGroup data type.

Waiting on goroutines

The Add **method sets the number of** goroutines **to wait for, and the** Wait **method prevents the program from exiting before all** goroutines **being tracked by our** WaitGroup **are finished executing.**

```
...

func main() {
  names := []string{"Phil", "Noodles", "Barbaro"}
  var wg sync.WaitGroup
  wg.Add(len(names))
  for _, name := range names {
    go printName(name)
  }
  wg.Wait()
}
...
```

The call to len() *returns the total number of names...*

...which is equal to the number of goroutines *we create inside the loop.*

Prevents program from exiting.

Updating WaitGroup

The Done **method must be called from each function that runs on a** goroutine **once it's finished. This gives the** WaitGroup **an update — like saying,** ***"Hey, there's one less*** goroutine ***you need to wait for."***

```
...
func main() {
  var wg sync.WaitGroup
  wg.Add(len(names))
  ...
  for _, name := range names {
    go printName(name, &wg)
  }
  wg.Wait()
}

func printName(n string, wg *sync.WaitGroup) {
  ...
  wg.Done()
}
```

Must pass a reference to *WaitGroup* ***so that we call*** *Done* ***on the original value and NOT on a copy.***

Inform the *WaitGroup* ***that the*** *goroutine* ***running this function is now finished!***

Single CPU — Concurrent and Synchronized

If we run our final code **specifying a single processor**, there's **no noticeable performance improvement.**

```
...
func main() {
  ...
  var wg sync.WaitGroup
  wg.Add(len(names))
  for _, name := range names {
    go printName(name, &wg)
  }
  wg.Wait()
}

func printName(n string) {
  ...
  wg.Done()
}
```

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

Run program on a single processor.

```
Name: Phil
Name: Noodles
Name: Barbaro

real 0m11.675s
```

Still slow

Running concurrent

Multiple CPUs — Parallel and Synchronized

The Go runtime **defaults to using all processors available**. Most machines today have more than one processor and our concurrent Go code can run **in parallel** with no changes!

```
...
func main() {
  ...
  var wg sync.WaitGroup
  wg.Add(len(names))
  for _, name := range names {
    go printName(name, &wg)
  }
  wg.Wait()
}

func printName(n string) {
  ...
  wg.Done()
}
```

Absence of GOMAXPROCS means all processors available will be used!

```
$ time go run main.go
```

```
Name: Phil
Name: Noodles
Name: Barbaro

real 0m4.172s
```

From 11 to 4.1 seconds!

Running in parallel!

```
printName("Phil", &wg)
printName("Noodles", &wg)
printName("Barbaro", &wg)
```