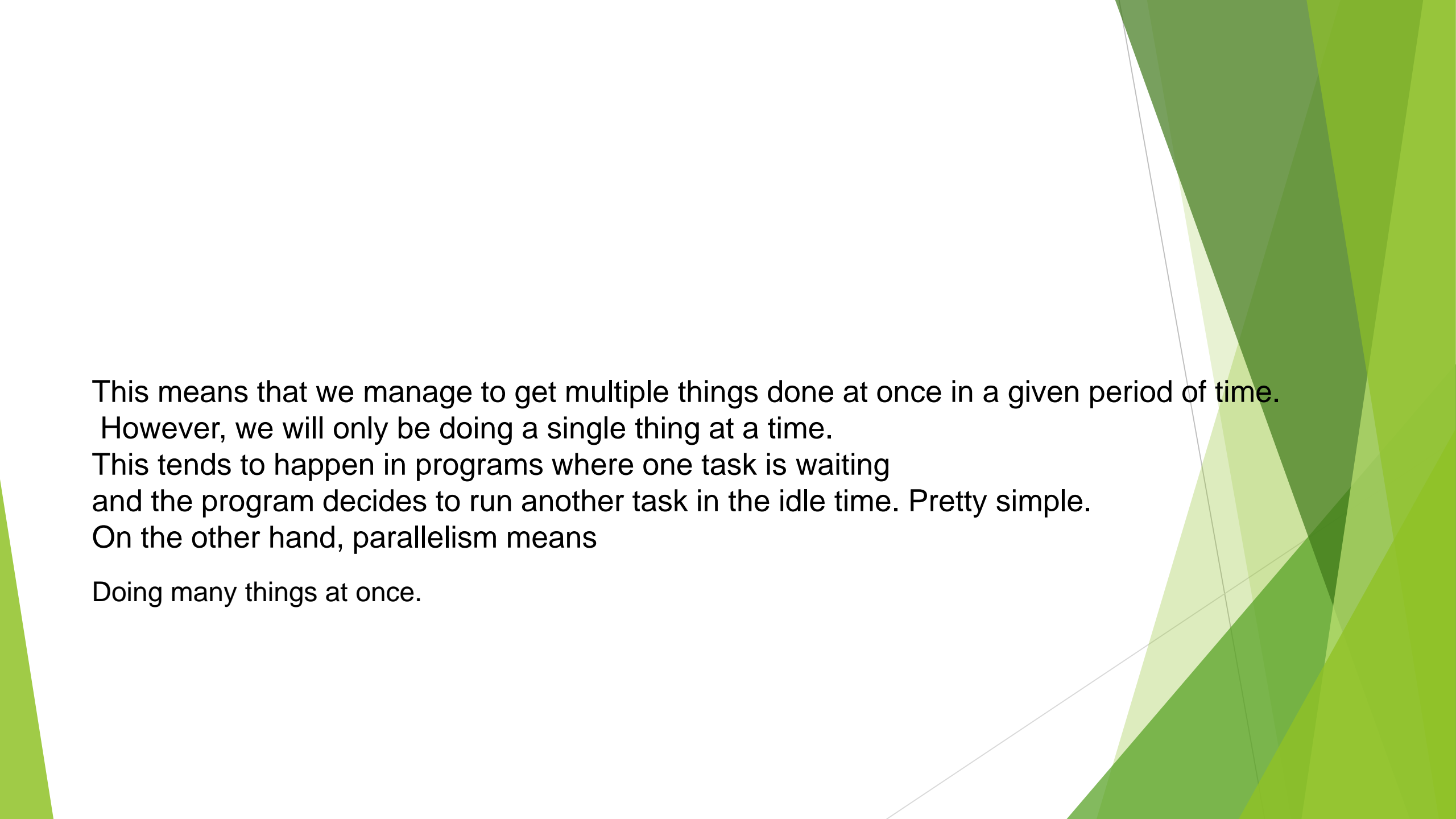


Go routines and Asynchronous Functions in Go

Miti Bhat

Concurrency & Parallelism

Concurrency and parallelism are two terms that are bound to come across often when looking into multitasking and are often used interchangeably. However, they mean two distinctly different things. Concurrency is all about... Dealing with many things at once.

The background of the slide features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, ranging from light lime to dark forest green. These shapes are positioned on the right side of the slide, creating a modern, layered effect.

This means that we manage to get multiple things done at once in a given period of time.

However, we will only be doing a single thing at a time.

This tends to happen in programs where one task is waiting and the program decides to run another task in the idle time. Pretty simple.

On the other hand, parallelism means

Doing many things at once.

- ▶ This means that even if we have two tasks, they are *continuously working without any breaks in between them*. The two tasks run independently and are not influenced by each other in any manner.
- ▶ We can also say that concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations¹. However, the components running in parallel, even inside a single application, have might have to communicate with each other. These communications happen between the components of even the simplest applications, and the overhead is generally low in concurrent systems. In the case when components run in parallel in multiple cores, this communication overhead could be (and generally is) higher. Hence parallel programs do not always result in faster execution times.

Concurrency is an inherent part of the Go programming language, and it's handled using goroutines **and** channels

goroutines

The definition of a goroutine is as simple as the following:

A goroutine is a lightweight thread managed by the Go runtime.

The Golang official site states³ that they're called *goroutines* because the existing terms—threads, coroutines, processes, and so on—convey inaccurate connotations.

A goroutine has a simple model: it is a function executing concurrently with other goroutines in the same address space.

It is lightweight, costing little more than the allocation of stack space.


And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required.

- ▶ Moreover, the goroutines are multiplexed to a fewer number of OS threads, thus there might be only one thread in a program with thousands of goroutines. If one goroutine should block, such as while waiting for I/O, then another OS thread is created and the remaining goroutines are moved to the new OS thread and continue to run. Their design hides many of the complexities of thread creation and management.
- ▶ The cost of creating a Goroutine is tiny when compared to a thread. Hence it's common for Go applications to have thousands of goroutines running concurrently.

- ▶ Prefix a function or method call with the `go` keyword to run the call in a new goroutine. When the call completes, the goroutine exits, silently. (The effect is similar to the Unix shell's `&` notation for running a command in the background.). Let's make an example.
- ▶ Demo `goutine1.go`

```
package main
import ( "fmt" "time" )
func say(s string) { for i := 0; i < 5; i++){
time.Sleep(100 * time.Millisecond)
fmt.Println(s)
}
}
func main() {
go say("world")
say("hello")
}
```


- ▶ You can easily test the pseudo-random behavior of printed output: the first time `say` is called doesn't block the execution of the main function, thus the *hello* string appears interleaved by the world string (5 times each). Let's go ahead introducing the concept of channels.

- 
- ▶ Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine, thus make many goroutines communicate between each other - actually, orchestrate them - using channels (aka... memory). In fact, it's pretty known that Go's approach to concurrency differs from the traditional use of (not only) threads, but shared memory as well. Philosophically, the idea behind Go can be summarized by the following sentence:
 - ▶ Don't communicate by sharing memory; share memory by communicating.

- Channels can be thought of as a pipe using which goroutines communicate. They allow you to pass references to data structures between goroutines, so if you consider this as passing around ownership of the data (the ability to read and write it), they become a powerful and expressive synchronization mechanism. Moreover, channels by design prevent race conditions from happening when accessing shared memory using goroutines.

```
package main
```

```
import "fmt"
```

```
func sum(s []int, c chan int) {
```

```
    sum := 0
```

```
    for _, v := range s {
```

```
        sum += v
```

```
    }
```

```
    c <- sum // send sum to c
```

```
}
```

```
}
```

```
//Demo goroutine2.go
```

```
func main() {
```

```
    s := []int{7, 2, 8, -9, 4, 0}
```

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

```
    go sum(s[:len(s)/2], c)
```

```
    go sum(s[len(s)/2:], c)
```

```
    x, y := <-c, <-c // receive from c
```

```
    fmt.Println(x, y, x+y)
```

```
}
```

- ▶ Another way to think about the channels is as typed conduits through which you can send and receive values with the channel operator, `<-`. In the code above, `c <- sum` send `v` to the channel `sum` and `x, y := <-c, <-c` receive two times `v` from the channel `c` and assign the respective values to `x` and `y`.
- ▶ Done? Then we should be ready and pretty confident to map this concept to the well known `async/await` pattern. Let's go ahead!

```
//A javascript code sample...
const sleep = require('util').promisify(setTimeout)
  async function myAsyncFunction() {
    await sleep(2000) return 2
  };
(
  async function() {
    const result = await myAsyncFunction();
    // outputs `2` after two seconds
    console.log(result);
  }
)();
```

- What this code does should be simple to understand: it simply simulates a workload of 2 seconds and asynchronously waits for it to be completed. Also, since the run of a script from the shell is synchronous, you have to await for the execution of `myAsyncFunction` from inside an async context, otherwise the Node.js runtime will complain. You should be able to copy and paste the code inside a `test.js` file and run it from the console with `node test.js`.

```
package main
```

```
import (  
    "fmt"  
    "time"  
)
```

```
func myAsyncFunction() <-chan int32 {  
    r := make(chan int32)  
    go func() {  
        defer close(r)  
        // func() core (meaning, the operation to be completed)  
        time.Sleep(time.Second * 2)  
        r <- 2  
    }()  
    return r  
}  
  
func main() {  
    r := <-myAsyncFunction()  
    // outputs `2` after two seconds  
    fmt.Println(r)  
}
```


- ▶ As you can see, we used both a goroutine and a channel, introduced in the beginning. Let's see in detail the pattern used to implement the async mechanism. First of all, the async function explicitly returns a `<-chan [your_type]` where `your_type` could be whatever you want. In this case, it's a simple `int32` number. Within the function you want to run asynchronously, create a channel by using the `make(chan [your_type])` and return the created channel at the end of the function. Finally, start an anonymous goroutine by the `go myAsyncFunction() {...}` and implement the function's logic inside that anonymous function. Return the result by sending the value to the channel. At the beginning of the anonymous function, add `defer close(r)` to close the channel once done.
- ▶ To "await" behavior is implemented by simply read the value from channel, with `r := <-myAsyncFunction()`. And This Is It.

- ▶ `Promise.all()`
- ▶ Unfortunately, things get more complicated as soon as you realized what you can do with `async/await`: another common scenario is when you start multiple `async` tasks then wait for all of them to finish and gather their results. Doing that is quite simple in Javascript (it is? it depends I guess). A pretty-simple to describe a way to achieve it is by using the `Promise.all()` primitive:

```
const myAsyncFunction = (s) =>
{
  return new Promise((resolve)
=> {
    setTimeout(() =>
resolve(s), 2000);
  })
};
```

```
(async function() {
  const result = await Promise.all([
    myAsyncFunction(2),
    myAsyncFunction(3)
  ]);
  // outputs `[2, 3]` after three
seconds
  console.log(result);
})();
```

- ▶ The await this time is done across a list of Promises: pay attention, because of the `.all()` signature takes an array as input. The `.all()` resolve all promises passed as an iterable object, short-circuits when an input value is rejected, is resolved successfully when all the promises in the array are resolved and rejected at first rejected of them.
- ▶ We achieve the same behavior with a Golang script:

```
package main
import (
    "fmt"
    "time"
)
func myAsyncFunction(s int32)
<-chan int32 {
    r := make(chan int32)
    go func() {
        defer close(r)
        // func() core (meaning,
        the operation to be completed)
```

```
time.Sleep(time.Second * 2)
    r <- s
    }()
    return r
}

func main() {
    firstChannel, secondChannel :=
myAsyncFunction(2), myAsyncFunction(3)
    first, second := <-firstChannel, <-
secondChannel
    // outputs `2, 3` after three seconds
    fmt.Println(first, second)
}
```

- ▶ In both snippets of code we just packaged a function taking as parameter the number of seconds to simulate a workload. The `await` is implemented using the channels receive operation, nothing more than the `<-` operator.
- ▶ `Promise.race()`
- ▶ Sometimes, a piece of data can be received from several sources to avoid high latencies, or there're cases that multiple results are generated but they're equivalent and the only first response is consumed. This first-response-win pattern is quite popular.

```
const myAsyncFunction = (s) =>
{
  return new Promise((resolve)
=> {
    setTimeout(() =>
resolve(s), 2000);
  })
};
```

```
(async function() {
  const result = await Promise.race([
    myAsyncFunction(2),
    myAsyncFunction(3)
  ]);
  // outputs `2` after three seconds
  console.log(result);
})();
```

The expected behavior is that 2 is always returned before the second Promise returned by myAsyncFunction(3) got resolved. This is natural due to the nature of .race() that implements the first-win pattern mentioned above. In Golang, this can be obtained similarly by using the select statement: let's make an example.

```
package main

import (
    "fmt"
    "time"
)

func myAsyncFunction(s int32) <-chan
int32 {
    r := make(chan int32)
    go func() {
        defer close(r)
        // func() core (meaning, the
operation to be completed)
        time.Sleep(time.Second * 2)
        r <- s
    }()
    return r
}
```

```
func main() {
    var r int32
    select {
        case r = <-myAsyncFunction(2):
        case r = <-myAsyncFunction(3):
    }
    // outputs `2` after three seconds
    fmt.Println(r)
}
```


- ▶ The cool thing about channels is that you can use Go's select statement to implement concurrency patterns and wait on multiple channel operations. In the snippet above, we use select to await both of the values simultaneously, choosing, in this case, the first one that arrives: once again, 2 is always returned before a value appear is retrieved from the channel populated by the `myAsyncFunction(3)`.
- ▶ However, we've seen that basic sends and receives on channels are blocking. We can use select with a default clause to implement non-blocking sends, receives, and even non-blocking multi-way selects. Let's take the example below

```
package main
```

```
import "fmt"
```

```
func main() {
```

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

```
    signals := make(chan bool)
```

```
    select {
```

```
        case msg := <-messages:
```

```
            fmt.Println("received  
message", msg)
```

```
        default:
```

```
            fmt.Println("no message  
received")
```

```
    }
```

```
    msg := "hi"
```

```
    select {
```

```
        case messages <- msg:
```

```
            fmt.Println("sent message", msg)
```

```
        default:
```

```
            fmt.Println("no message sent")
```

```
    }
```

```
    select {
```

```
        case msg := <-messages:
```

```
            fmt.Println("received message", msg)
```

```
        case sig := <-signals:
```

```
            fmt.Println("received signal", sig)
```

```
        default:
```

```
            fmt.Println("no activity")
```

```
    }
```

```
}
```

- The code above implements a non-blocking receive. If a value is available on messages then select will take the <-messages case with that value. If not it will immediately take the default case. A non-blocking send works similarly. Here msg cannot be sent to the messages channel, because the channel has no buffer and there is no receiver. Therefore the default case is selected. We can use multiple cases above the default clause to implement a multi-way non-blocking select. Here we attempt non-blocking receives on both messages and signals.

Conclusion

As you can see, the await/async basic patterns are easily portable to a Golang code. But... this was just a tasting: you can get so much more using buffered channels, signals and context. I will talk about all of this next time! Stay tuned and thank you for reading.