Goroutines and Channels in Go

Session 15

Golang course by Exadel

31 Nov 2022

Sergio Kovtunenko
Lead backend developer, Exadel

Agenda

- ▶️ Goroutines
- ▶️ Channels: bi-directional and uni-directional, buffered and unbuffered
- ▶️ Channels Axioms
- ▶️ Select statement
- ▶️ Blocking vs. non-blocking flows
- ▶️ Rules for using channels
- ▶️ Next time...

Before everything else...

▶️ You can write production-grade applications in Go without knowing much about:

- goroutines
- channels
- mutexes, etc.

▶️ We need goroutines to **better utilize** our hardware!

Goroutines

Goroutines 💣

▶️ The go statement launches a function call as a goroutine

```
go f()
go f(x, y, ...)
```

▶️ A goroutine runs **concurrently** (but not necessarily in **parallel**)

▶️ A goroutine is a **thread of control within the program**, with its **own local variables and stack**.

▶️ We have no goroutine **ID** ! No "handler" associated with goroutine!

▶️ *Much cheaper to create and schedule* than operating system threads.

▶️ **Main** goroutine vs others.

▶️ Example: `code/01_goroutines_test.go`

Source: "Introduction to Go" by Dave Chaney

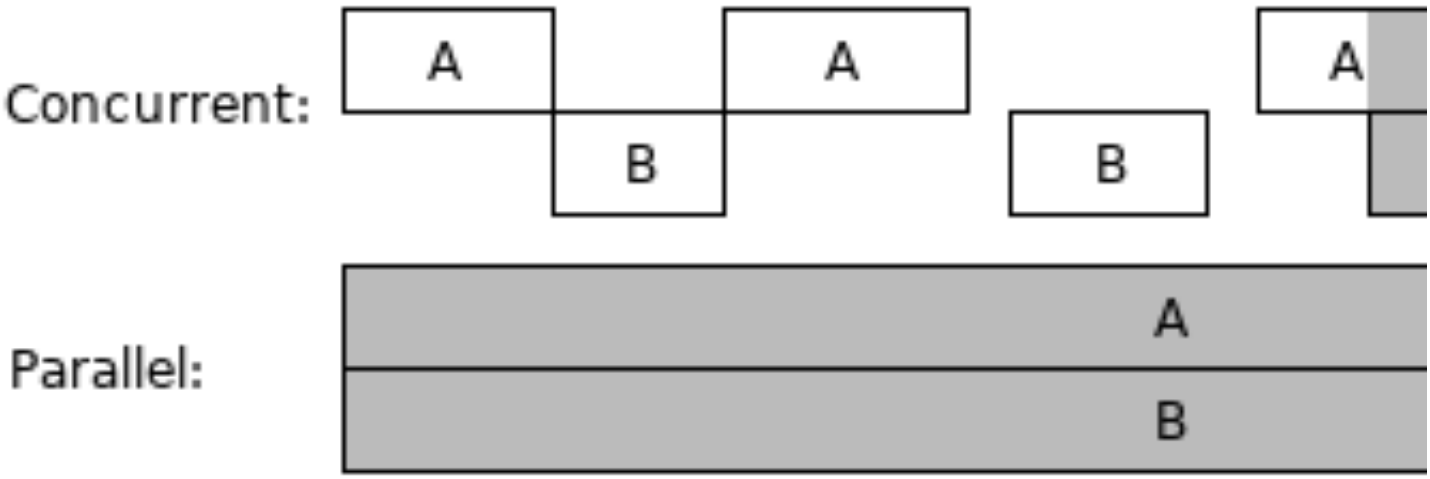
(https://github.com/davecheney/introduction-to-go)

Understanding Concurrent vs. Parallel Execution

▶️ Concurrency is about dealing with lots of things at once

▶️ Parallelism is about doing lots of things at once

▶️ Concurrency is about structure, parallelism is about execution



Source: https://talks.golang.org/2012/waza.slide#8

(https://talks.golang.org/2012/waza.slide#8)

Goroutines recap

▶️ Goroutines are user-space **lightweight** threads.

- Go runtime **scheduler** will manage them, not the OS. The scheduled is of **work-stealing** type.

▶️ Very **lightweight**!

- Owns a **small stack** for local variables: around 2k per 1 goroutine + **stack can grow** as needed
- Faster to create and do context switch
- You can have millions of them in your application

▶️ Goroutines are multiplexed onto OS threads

- We can limit the number of OS threads for program written in Go using `GOMAXPROCS` env var
- `runtime.GOMAXPROCS(32)`

`go` statement (1/2)

- ✓ A "**go**" statement starts the execution of a function call as an control, or **goroutine**, within the same address space.
- ✓ The expression must be a function or method call; it cannot be
  - Calls of built-in functions are restricted as for expressi
- ✓ The function *value and parameters* are evaluated as usual in the with a regular call, program execution does not wait for the in
  - Instead, the function begins executing independently in a
  - When the function *terminates*, its goroutine also *terminate*
  - If the function has any *return values*, they are discarded
- ✓ Example:

```
go Server()
go func(ch chan<- bool) { for { sleep(10); ch <- true }} (c)
```

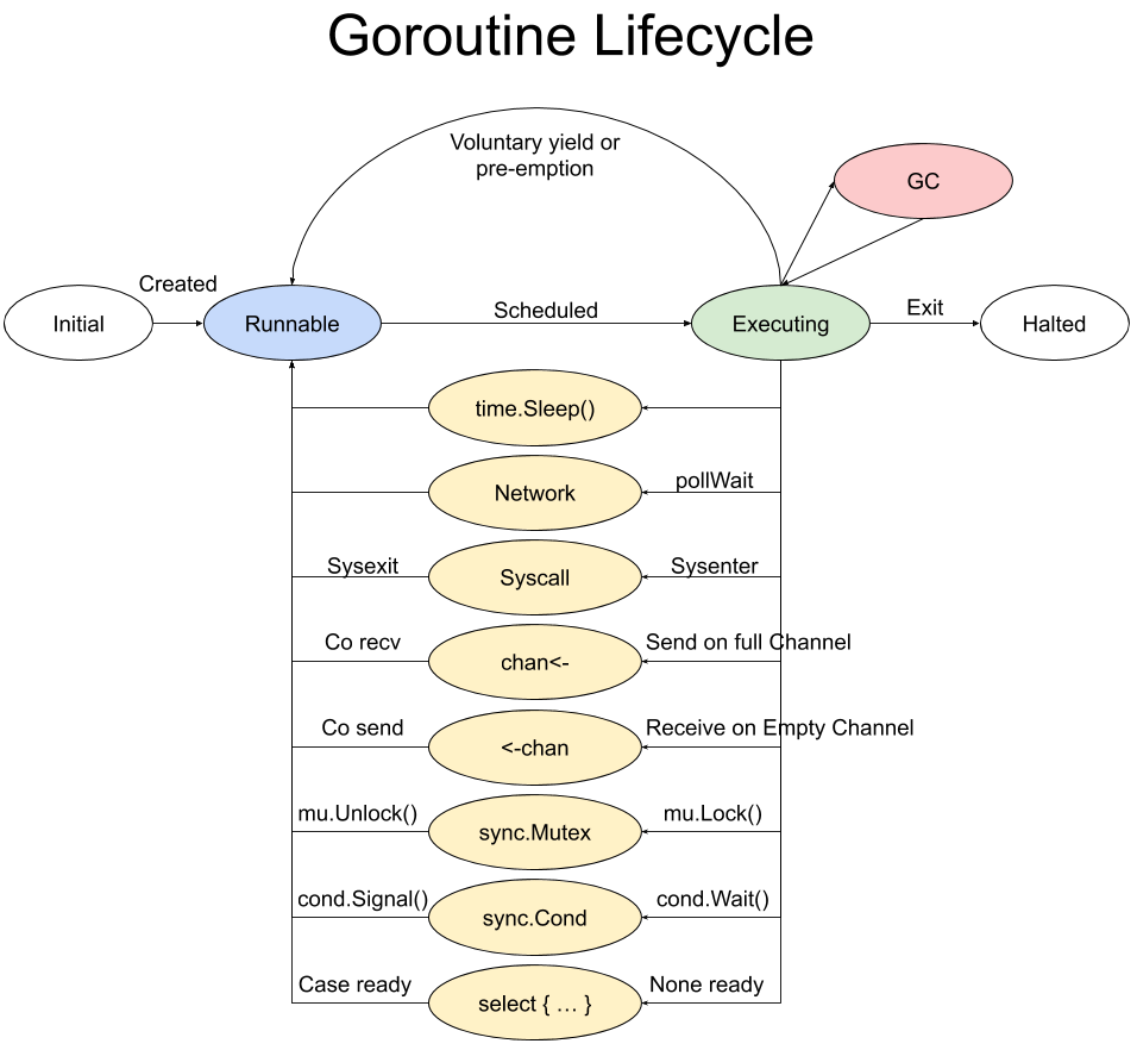
`go` statement (2/2)

- ✓ You don't need *"goroutine ID"* at all.
  - In Go there is NO goroutine exposed.
  - Potential problems include:
    - Because of the high potential for abuse, there is no identifier for the current goroutine in Go.
    - This may seem draconian, but this actually preserves package ecosystem: *it does not matter if you start a*

```
// That is, for any method of function F:
F()
// is almost exactly equivalent to:
done := make(chan struct{})
go func() {
     defer close(done)
     F()
}
<-done
```

- ✓ When you spawn goroutines, make it clear when - or whether - th
  - Try to keep concurrent code simple enough that goroutine
  - If that just isn't feasible, document when and why the go

Goroutine lifecycle



Source: "cmd/trace: problems and proposed improvements #33322" (https://github.com/golang/go/issues/33322)

Goroutine context switching

▶️ To understand context switching there is terminology: **M/G/P** :

▶️ **G** - represents a user space **Goroutine**

- it has its own stack and blocking info

▶️ **P** - represents a logical entity for available **Processors** or a scheduling context.

- The number of P is pre-decided (`GOMAXPROCS`) and *fixed during the run*.

▶️ **M** - stands for **Machines**, they are a representation for OS threads (like POSIX thread)

- **M** will run a set of **G**'s
- Every **Machine** (M) needs a **P** to run **G**.

▶️ The same names `M/G/P` is in use in internal Go's source code.

Understand Golang scheduler

▶️ Awesome 30-mins video: GopherCon 2018: Kavya Joshi - The Scheduler Saga (https://www.youtube.com/watch?v=YHRO5WQGh0k)

▶️ Another awesome video: Dmitry Vyukov — Go scheduler: Implementing language with lightweight concurrency

(https://www.youtube.com/watch?v=-K11rY57K7k)

▶️ Watch them! :)

▶️ Also useful article: Go Runtime Scheduler Design Internals

(https://freethreads.wordpress.com/2019/01/23/go-runtime-scheduler-design-internals/)

Channels

Channels: basic info

✓ A *channel* provides a mechanism for concurrently executing functi
and receiving values of a specified element type.
- The value of an uninitialized channel is `nil`.
  - A `nil` channel is *never ready* for communication.

✓ A single channel may be used in:
- *send* statements **OR**
- *receive* operations **OR**
- calls to the built-in functions `cap()` and `len()` by any numb
further synchronization.
- Channels act as first-in-first-out queues.
  - For example, if one goroutine *sends* values on a chann
receives them, the values are received *in the order s*

Channels: channel direction

- ✓ The optional `<-` operator specifies the **channel direction**, *send*
  - If *no direction* is given, the channel is **bidirectional.**
  - A channel may be *constrained* only to send or only to rece

```
chan T          // can be used to send and receive values of type T
chan<- float64  // can only be used to send float64s
<-chan int      // can only be used to receive ints
```

- ✓ The `<-` operator associates with the leftmost `chan` possible:

```
chan<- chan int    // same as chan<- (chan int)
chan<- <-chan int  // same as chan<- (<-chan int)
<-chan <-chan int  // same as <-chan (<-chan int)
chan (<-chan int)
```

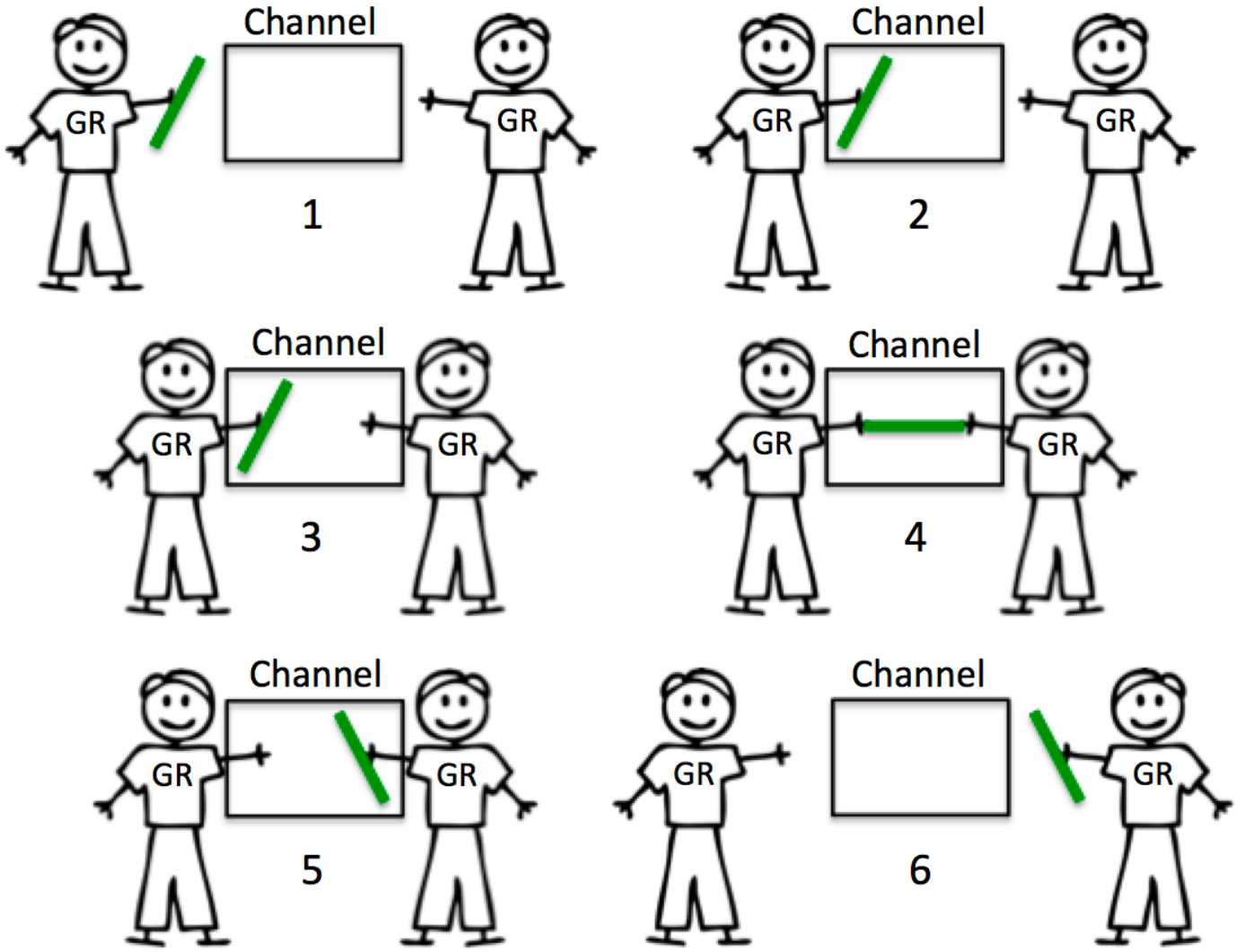
- ✓ A new, *initialized channel* value can be made using the built-ir the channel type and an **optional** capacity as arguments:

```
unbufferedChannel := make(chan int)
bufferedChannel := make(chan int, 100)
```

Channels: buffered or non-buffered

- ✓ Channels can be **buffered** or **non-buffered**.
- ✓ Unbuffered channels.
  - If *channel capacity* is zero or absent, the channel is unbuf
  - The communication succeeds only when both a *sender* and *rece*
- ✓ Buffered channels.
  - Channels can be buffered. Provide the buffer length as the initialize a buffered channel.
  - *Sends* to a buffered channel *block* only when the buffer is f
  - *Receives* *block* when the buffer is empty.
- ✓ A `channel` may be *closed* with the built-in function `close()`.
  - The multi-valued assignment form of the receive operator re was sent before the channel was closed.

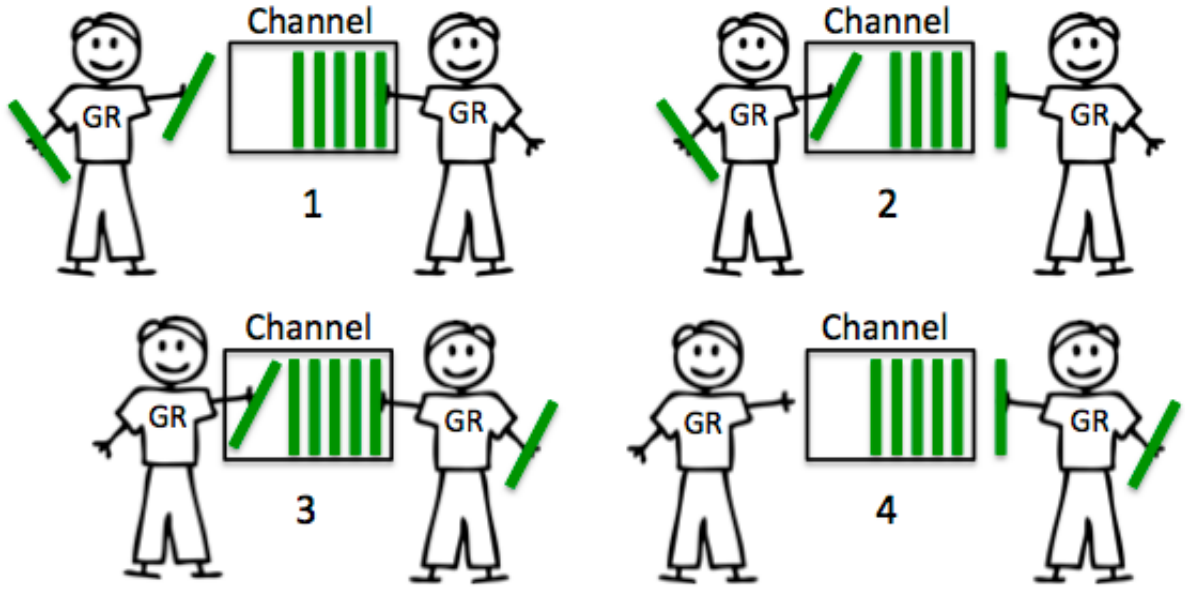
Visualization of non-buffered channels



Source: "stackoverflow: What are channels used for?"

(https://stackoverflow.com/a/39831976)

Visualization of buffered channels



Source: "stackoverflow: What are channels used for?"

(https://stackoverflow.com/a/39831976)

Closing a channel

- ✓ For a channel `c`, the built-in function `close(c)` records that no r channel.
  - It is an error if `c` is a *receive-only channel*.
- ✓ **Sending** to or **closing** a *closed* channel causes a **run-time panic.**
- ✓ Closing the `nil` channel also causes a **run-time panic.**
- ✓ It is completely normal to not to close channels manually.
  - They will be automatically garbage collected once related (finishes.

Best Practice: Formal arguments for goroutine functions

✓ Prefer using formal arguments for the channels you pass to go-r channels in global scope.

- You can get *more compiler checking* this way, and *better m*
- Example:

```
func main() {
    c := make(chan string)

    for i := 1; i <= 2; i++ {
        go func(i int, co chan<- string) {
            for j := 1; j <= 2; j++ {
                co <- fmt.Sprintf("hi from %d.%d", i, j)
            }
        }(i, c)
    }

    for i := 1; i <= 4; i++ {
        fmt.Println(<-c)
    }
}
// OUTPUT:
// Hi from 2.1
// Hi from 2.2
// Hi from 1.1
// Hi from 1.2
```

Source: "Multiple goroutines listening on one channel" StackOverflow (http://stackoverflow.com/questions/15715605/multiple-goroutines-listening-on-one-channel/15721380#15721380)

Best Practice: Avoid both reading and writing on the same channel

- ✓ **Avoid both reading and writing on the same channel** in a particula 'main' one).
  - Otherwise, deadlock is a much greater risk.
- ✓ It is generally a good principle to *view buffering as a performar*
  - If your program does not deadlock without buffers, it won't (**but the converse is not always true**).
  - So, as another rule of thumb, start without buffering then

Channel Axioms: send to a nil channel blocks forever

✓ A send to a **nil** channel blocks forever:

```
package main

func main() {
    var c chan string
    c <- "let's get started" // deadlock
}
```

Source: "Channel Axioms" by Dave Cheney

(https://dave.cheney.net/2014/03/19/channel-axioms)

Channel Axioms: receive from a nil channel blocks forever

✓ A receive from a **nil** channel blocks forever:

```
package main
import "fmt"

func main() {
    var c chan string
    fmt.Println(<-c) // deadlock
}
```

- *Reason why that happened*:
  - If the channel is not initalised then its *buffer siz*
  - If the size of the channel's buffer is zero, then th
  - If the channel is unbuffered, then a send will block ready to receive.
  - If the channel is **nil** then the sender and receiver h they are both blocked waiting on independent channel

Source: "Channel Axioms" by Dave Cheney

(https://dave.cheney.net/2014/03/19/channel-axioms)

Channel Axioms: send to a closed channel panics

✓ A send to a **closed channel** *panics*:

```
package main
import "fmt"

func main() {
    var c = make(chan int, 100)
    for i := 0; i < 10; i++ {
        go func() {
            for j := 0; j < 10; j++ {
                c <- j
            }
            close(c)
        }()
    }
    for i := range c {
        fmt.Println(i)
    }
}
```

Source: "Channel Axioms" by Dave Cheney

(https://dave.cheney.net/2014/03/19/channel-axioms)

Channel Axioms: receive from a closed channel returns the zero value immediately

✓ A receive from a **closed channel** returns the *zero value immediat*

```go
package main
import "fmt"

func main() {
    c := make(chan int, 3)
    c <- 1
    c <- 2
    c <- 3
    close(c)
    for i := 0; i < 4; i++ {
        fmt.Printf("%d ", <-c) // prints 1 2 3 0
    }
}
```

Source: "Channel Axioms" by Dave Cheney

(https://dave.cheney.net/2014/03/19/channel-axioms)

Next time...

We will continue next time

▶️ We didn't discuss today:

- `select` statement
- goroutine gotcha's

Homework:

- "3 Golang Channel Red Flags" (https://hackmongo.com/post/three-golang-channel-red-flags/)
- Must read: Channel Axioms (https://dave.cheney.net/2014/03/19/channel-axioms)
- Channels in Go (https://go101.org/article/channel.html)
- Principles of designing Go APIs with channels (https://inconshreveable.com/07-08-2014/principles-of-designing-go-apis-with-channels/)
- Understanding real-world concurrency bugs in Go (https://blog.acolyer.org/2019/05/17/understanding-real-world-concurrency-bugs-in-go/)
- Just for information, not to be used: A pattern for overcoming non-determinism of Golang select statement (https://medium.com/@pedram.esmaeeli/a-pattern-for-overcoming-non-determinism-of-golang-select-statement-139dbe93db98)

Next session...

▶️ Session16:

Most useful packages: context, sync

- Context package
- Sync package

Thank you

Golang course by Exadel

31 Nov 2022

Sergio Kovtunenko
Lead backend developer, Exadel
skovtunenko@exadel.com (mailto:skovtunenko@exadel.com)
https://github.com/skovtunenko (https://github.com/skovtunenko)
@realSKovtunenko (http://twitter.com/realSKovtunenko)