ЛЕКЦИЯ 3.

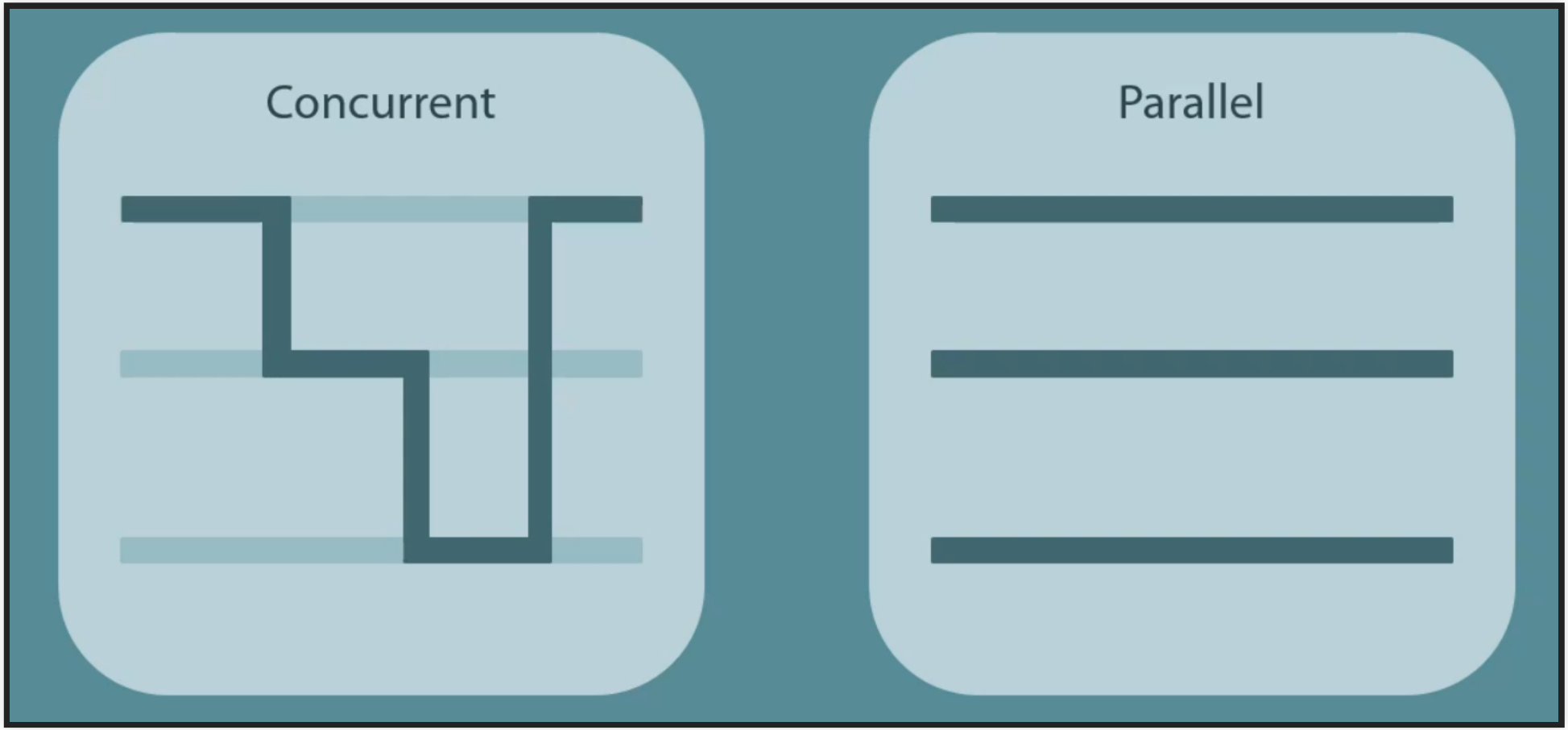
ВВЕДЕНИЕ В КОНКУРЕНТНОЕ ПРОГРАММИРОВАНИЕ

OZON

МОСКВА, 2021

ПАРАЛЛЕЛЬНОСТЬ И АСИНХРОННОСТЬ/ КОНКУРЕНТНОСТЬ

- Параллельность - возможность выполнять несколько потоков одновременно
- Конкурентность - возможность передавать управление другому потоку в процессе выполнения


Concurrent
Parallel

Проблема такого подхода - непредсказуемость этапов выполнения кода и синхронизация доступа к данным. Так же, не все алгоритмы можно распараллелить очевидным образом.

ПРОЦЕССЫ, ПОТОКИ, ГОРУТИНЫ

ПРОЦЕССЫ

- Раздельные ресурсы
- Раздельная память
- Раздельные регистры

потоки

- Общие ресурсы
- Общая память
- Раздельные стек и регистры

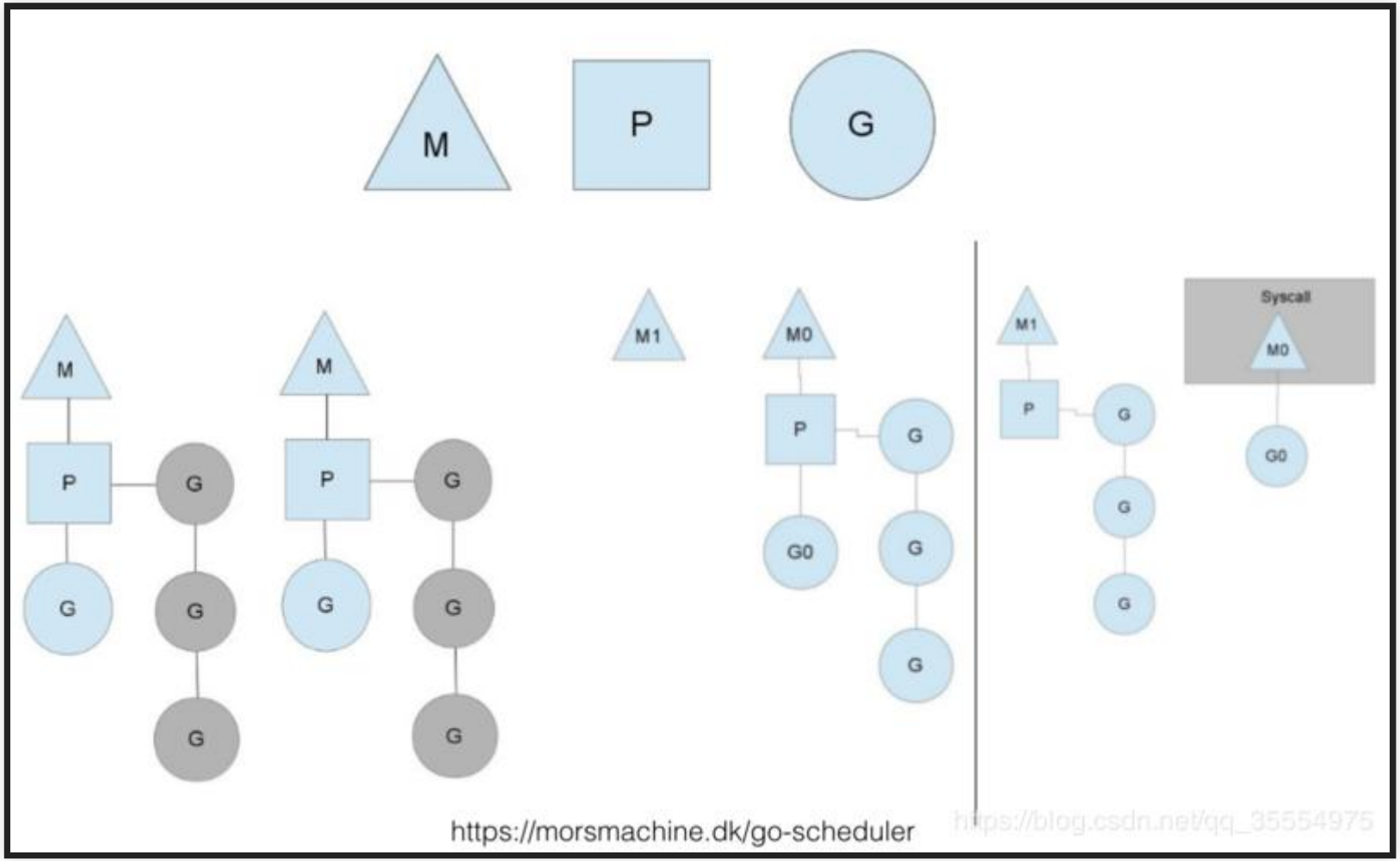
ГОРУТИНЫ

- Общие ресурсы
- Общая память
- Общий системный стек
- Общие регистры

Переключение потоков занимает ~12к операций процессора, переключение горутин - 2,4к оперций

РАНТАЙМ GO

- Виртуальный процессор **P** - представляет каждое виртуальное ядро или поток Hyper-Treading
- Реальные поток ОС для рантайма Go **M** - создается по числу виртуальных процессоров **P**
- Горутина **G** - создаются по необходимости, как минимум одна для main


M
P
G
M
P
G
G
G
G
M
P
G
G
G
M1
M0
P
G
G0
G
G
M1
P
G
G
G
Syscall
M0
G0
https://morsmachine.dk/go-scheduler
https://blog.csdn.net/qq_35554975

Управление G осуществляется шедулером Go, а не ОС. Переключение может быть выполнено в следующих случаях:

1. Запуск новой горутины
2. Сборка мусора
3. Операция синхронизации
4. Системный вызов

Может быть выполнено не значит точно будет выполнено

ГОРУТИНЫ

Горутина - это структура, которая выполняет переданную функцию.

Самая тяжелый по памяти элемент структуры - stack. По умолчанию выделяется 2Кб

В процессе выполнения стек может увеличиваться, если потребуется.

У стека есть максимальный размер. 1Гб для 64бит, 250Кб для 32бит

https://github.com/golang/go/blob/f296b7a6f045325a2

СОЗДАНИЕ УДАЛЕНИЕ ГОРУТИН

```
go func(trolley Trolley) {
    burnBooks(trolley)
}(trolley)
```

```
go func() {
    burnBooks(trolley)
}()
```

```
go burnBooks(trolley)
```

```
go trolley.Load(pile)
```

СКОЛЬКО ТУТ ГОРУТИН?

```
func main() {
    fmt.Printf(
        "Goroutines: %d",
        runtime.NumGoroutine(),
    )
}
```

ЗАМЫКАНИЯ

```
func main() {
    for i := 0; i < 5; i++ {
        go func() {
            time.Sleep(time.Second)
            fmt.Print(i)
        }()
    }
    time.Sleep(2 * time.Second)
}
```

Как исправить?

ЗАМЫКАНИЯ

```
func main() {
    for i := 0; i < 5; i++ {
        go func(counter int) {
            time.Sleep(time.Second)
            fmt.Print(counter)
        }(i)
    }
    time.Sleep(2 * time.Second)
}
```

КАНАЛЫ

Горутина сама по себе мало кому интересна

- Возврат результата
- Получение новых данных

Канал - очередь сообщений, которая умеет работать в многопоточной среде

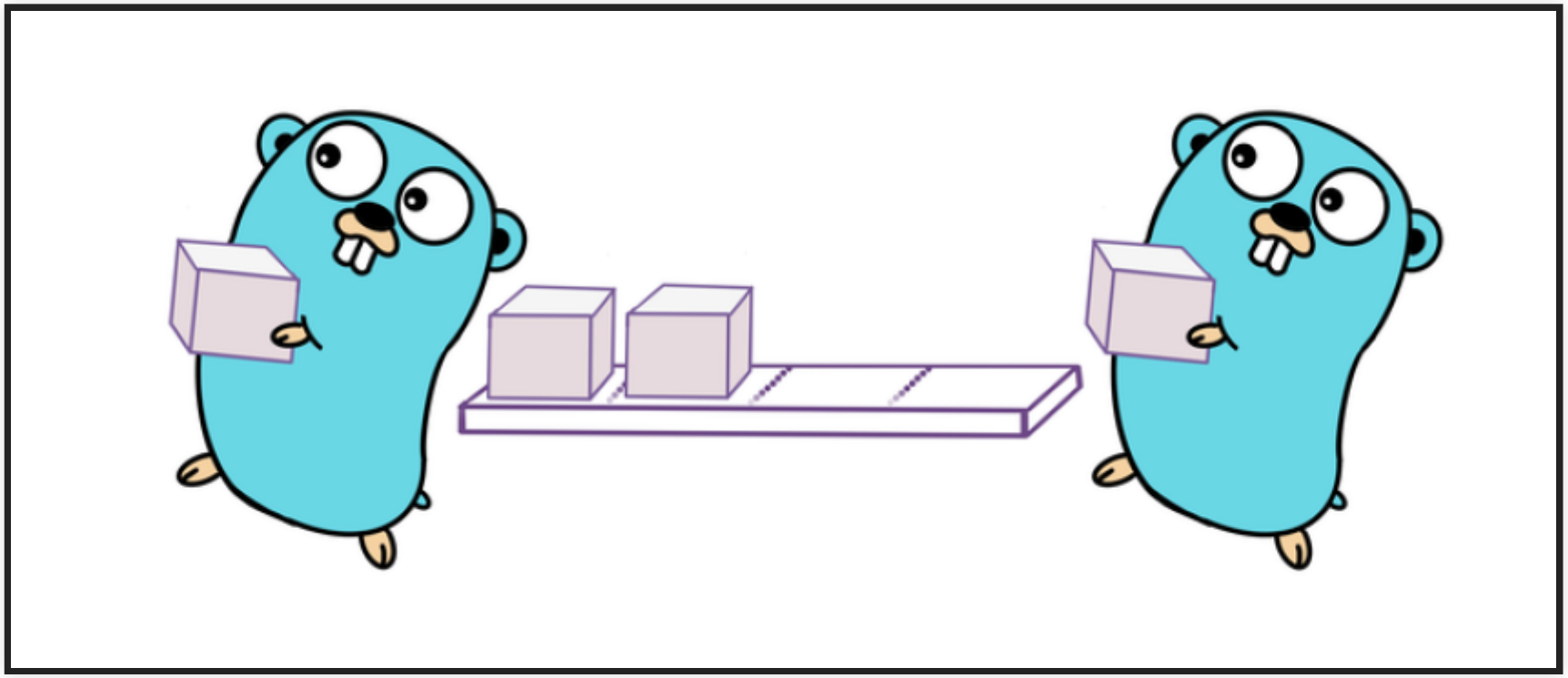
```
intCh := make(chan int)
chanCh := make(chan chan int)
sliceCh := make(chan []string)
interfaceCh := make(chan io.Reader)
```

Буферизованный канал - канал в котором есть место для нескольких сообщений

```
bufCh := make(chan bool, 10)
```

Закрытия канала

```
close(ch)
```

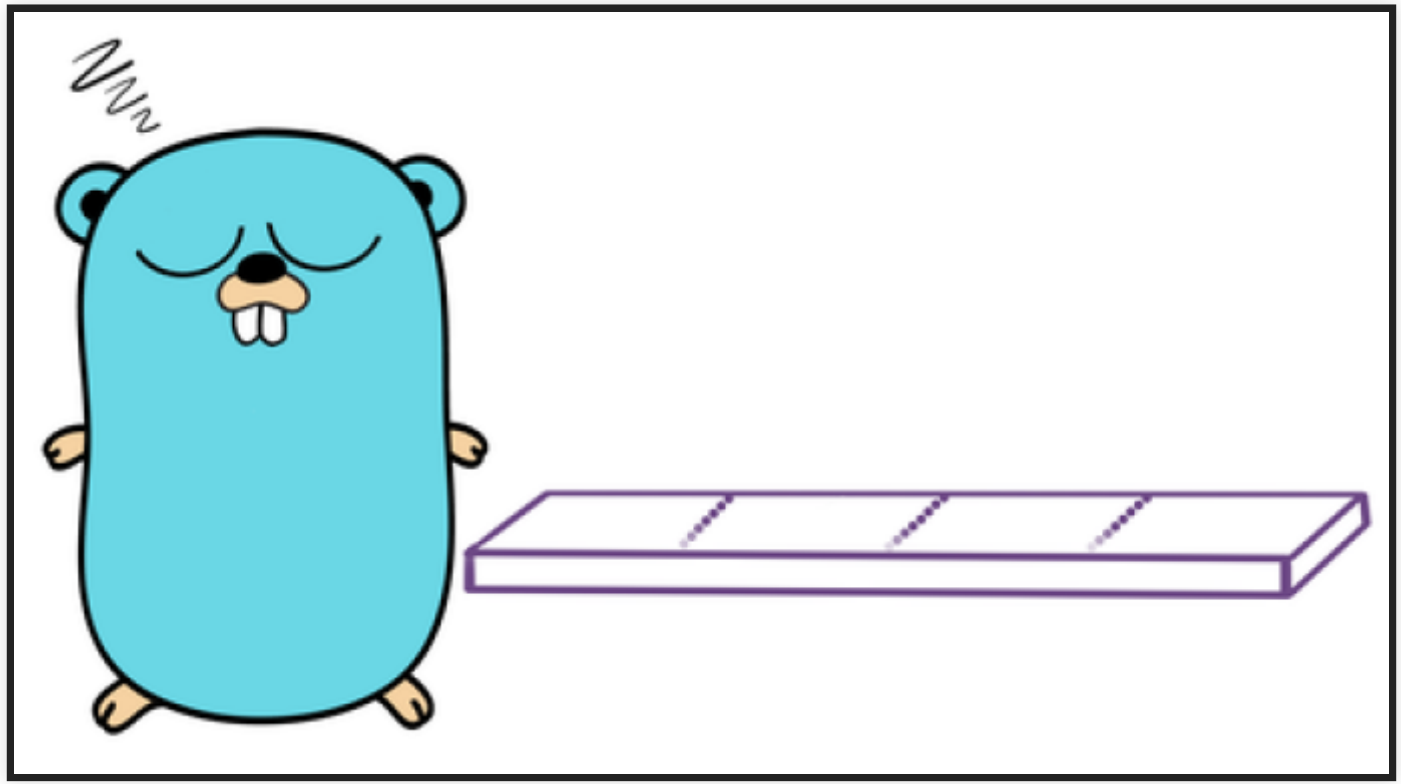


```
var ch chan int

// cap(ch) - ?, len(ch) - ?, ch -?

ch = make(chan int, 5)
ch <- someVariable

// cap(ch) - ?, len(ch) - ?

a := <- ch
```

ПЕРЕДАЧА В ФУНКЦИИ

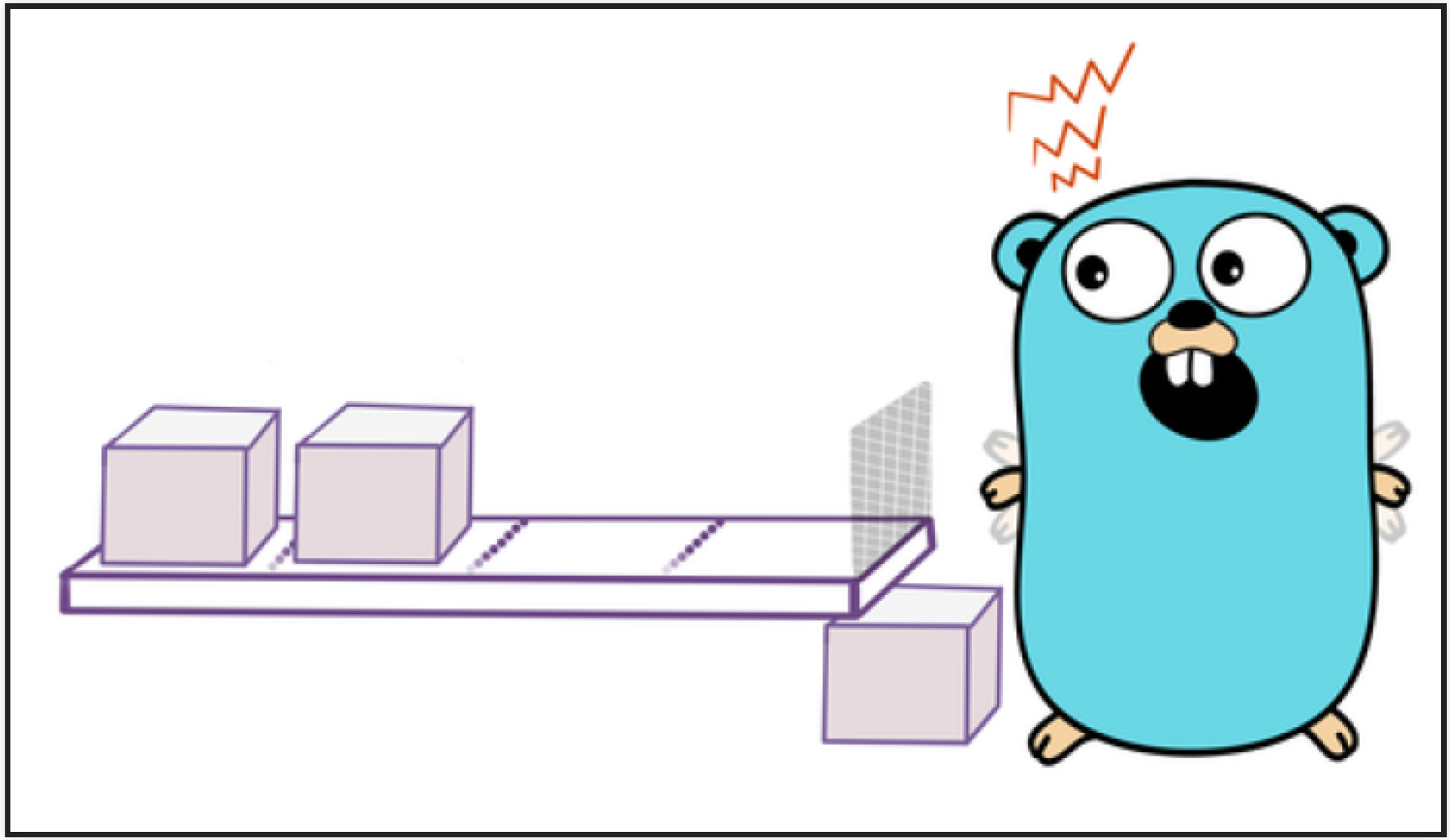
```
func f(ch <-chan int) {...}
fucn f1(ch chan<- int) {...}

ch := make(chan int)
bufCh := make(chan int, 2)
f(ch)
f1(bufCh)
```

ЧТО ПРОИЗОЙДЕТ ПРИ ЧТЕНИИ ИЗ ПУСТОГО КАНАЛА?

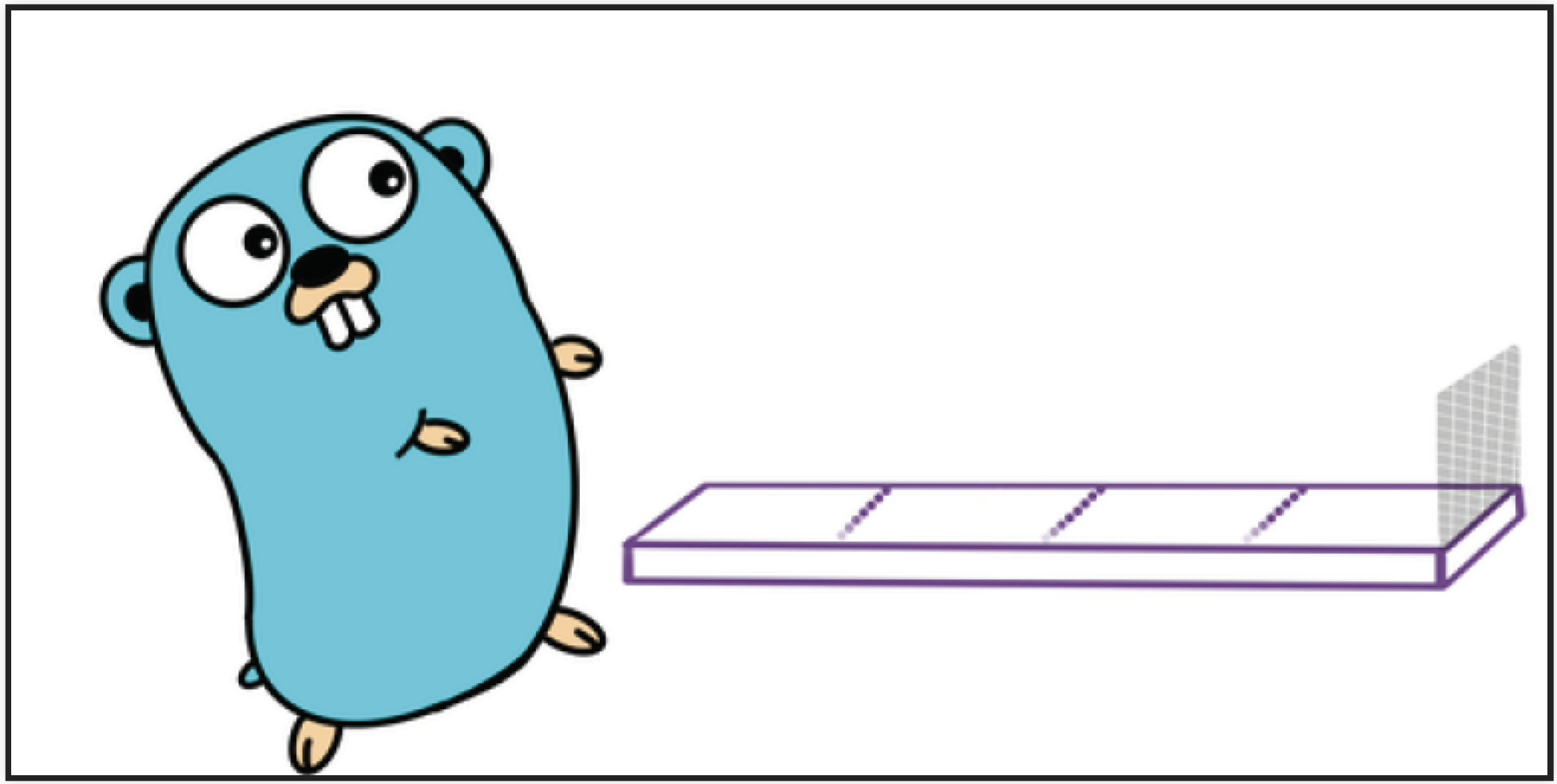


ЧТО ПРОИЗОЙДЕТ ПРИ ЗАПИСИ В ЗАКРЫТЫЙ КАНАЛ?

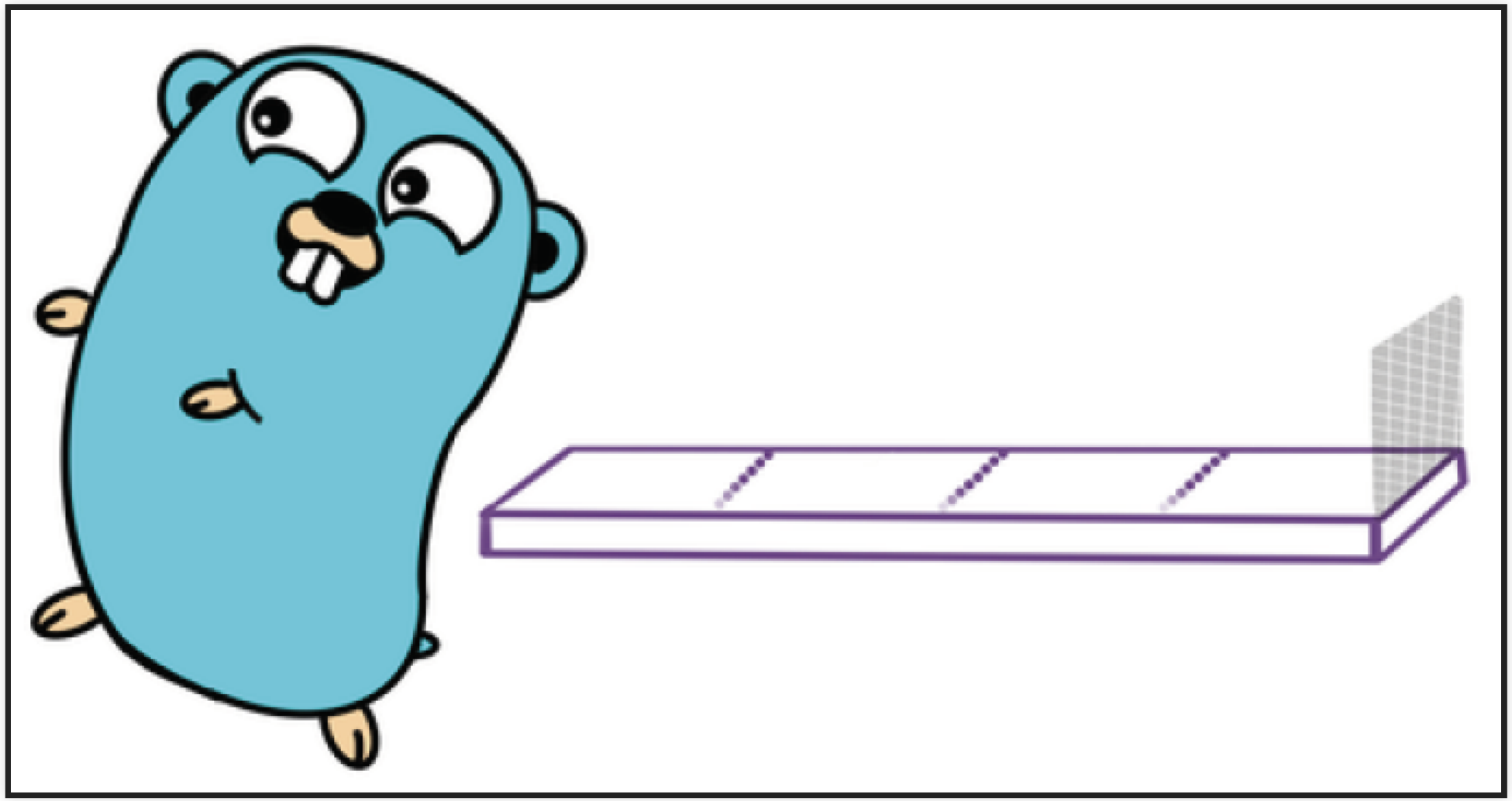
НИЧЕГО ХОРОШЕГО:



ЧТО ПРОИЗОЙДЕТ ПРИ ЧТЕНИИ ИЗ ЗАКРЫТОГО КАНАЛА?



НУЛЕВОЕ ЗНАЧЕНИЕ:



УСПЕШНОСТЬ ЧТЕНИЯ

Как отличить нулевое значение в закрытом канале от просто нулевого значения:

```
v, ok := <- ch
```

ВОПРОС СО ЗВЕЗДОЧКОЙ

```
var ch chan int

x, ok := <- ch

ch <- 1
```

Что будет?

ЧТЕНИЕ КАНАЛОВ В ЦИКЛЕ

```
for v := range ch{
    ...
}
```

СИНХРОНИЗАЦИЯ ПОТОКОВ

```
func longOperation(i int) {
    time.Sleep(time.Second)
    fmt.Printf("%v passed\n", i)
}

func main() {
    for i := 0; i < 5; i++ {
        go longOperation(i)
    }
    fmt.Println("That's all!")
}
```

```
func longOperation(i int, wg *sync.WaitGroup) {
    defer wg.Done()
    ...
}

wg := new(sync.WaitGroup)
wg.Add(5) // Вариант 1
for i := 0; i < 5; i++ {
    // wg.Add(1) - Вариант 2
    go longOperation(i, wg)
}
wg.Wait()
```

ПАТТЕРНЫ ИСПОЛЬЗОВАНИЯ

1. Сбор данных из нескольких источников

```
func producer(ch chan <- string)
ch := make(chan string)
for i := 0; i < 10; i ++ {
   go producer(ch)
}

for x := range ch {
    // обрабатываем данные
}
```

2. Раздача данных для обработки нескольким воркерам

```
func consumer(ch <-chan string)
ch := make(chan string)
for i := 0; i < 10; i ++ {
   go consumer(ch)
}

for {
   ch <- time.Now().Format("Mon Jan 2 15:04:05 -0700 MST 2006"
   ...
}
```

3. Синхронизация потоков выполнения

```
func worker(ch <-chan int) {
   for {
      v, ok := <- ch
      if !ok {
         return
      }
      fmt.Printf("%v\n")
      time.Sleep(time.Second)
   }
}

ch := make(chan int 5)
for i := 0; i < 100; i++ {
   ch <- i
}
```

SELECT - ПРИОРИТЕТНОЕ ЧТЕНИЕ

select - switch для каналов

```
select {
case x, ok := <- intChan:
    fmt.Printf("Read int %v, ok: %v\n", x, ok)

case x, ok := <- floatChan:
    fmt.Printf("Read float %v, ok: %v\n", x, ok)
case sendTo <- "Hello world!":
    fmt.Println("Sended!\n")
default:
    fmt.Println("Are you sure?!\n")
}
```

Завершение работы сервиса через сигналы ОС

```
c := make(chan os.Signal, 1)
signal.Notify(c, os.Interrupt)

for {
    select {
    case <- c:
        // Прибираем за собой
        fmt.Println("Получен сигнал завершения\n")
        return
    case data, ok <- dataChan:
        if !ok {
            fmt.Println("Нештатная ситуацияб завершаемся!")
            return
        }
        // Что-то делаем с данными
```

ВАЖНЫЕ ПРАВИЛА

1. Закрывает канал тот кто в него пишет
2. Если пишет несколько продюсеров, закрывает канал тот кто создал его и продюсеров
3. Не закрытый канал держит ресурсы, закрывайте их явно