

Backend Golang

Урок 6

Многопоточность: потоки и горутины. Каналы. Шаблоны синхронизации (waitgroup, select, mutex)

Цели занятия

- Познакомиться с горутинами
- Понять разницу между потоками и горутинами
- Научиться базовым шаблонам конкурентного выполнения задач
- Научиться работать с каналами

Простой вызов функции

```
package main

import "fmt"

func main() {
    printHello()
}

func printHello() {
    fmt.Println("Hello!")
}
```



A terminal window titled "Run: go build main.go" showing the execution of Go code. The terminal displays environment setup for GOROOT and GOPATH, followed by the build command and its output, which is "Hello!".

```
Run: go build main.go x
▶ GOROOT=/usr/local/go #gosetup
🔧 GOPATH=/home/alikhan/go #gosetup
■ /usr/local/go/bin/go build -o /tmp/___go_bui
⚙ /tmp/___go_build_main_go
🗑 Hello!
```

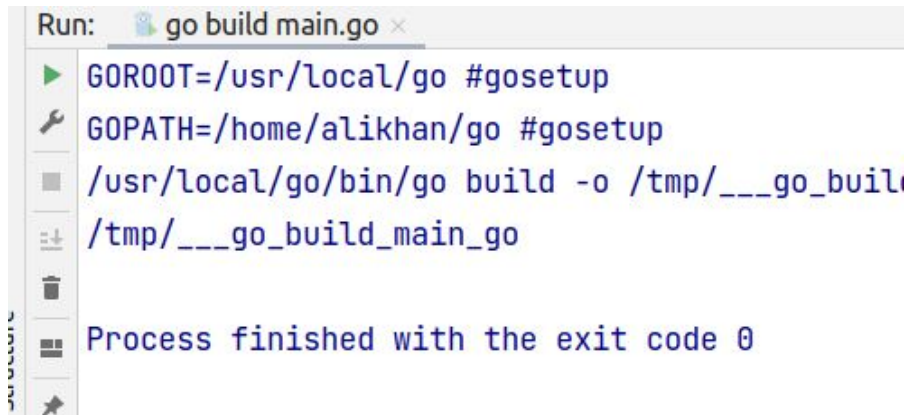
Запуск горутины

```
package main

import "fmt"

func main() {
    go printHello()
}

func printHello() {
    fmt.Println("Hello!")
}
```

A terminal window titled 'Run: go build main.go' with a close button. It shows the execution of the Go build command. The output includes environment variables for GOROOT and GOPATH, the build command itself, and a confirmation that the process finished with exit code 0. On the left side of the terminal, there is a vertical toolbar with icons for running, opening, saving, deleting, and other actions.

```
Run: go build main.go x
▶ GOROOT=/usr/local/go #gosetup
🔧 GOPATH=/home/alikhan/go #gosetup
▣ /usr/local/go/bin/go build -o /tmp/___go_build
⬇ /tmp/___go_build_main_go
🗑 
📁 
✦ Process finished with the exit code 0
```

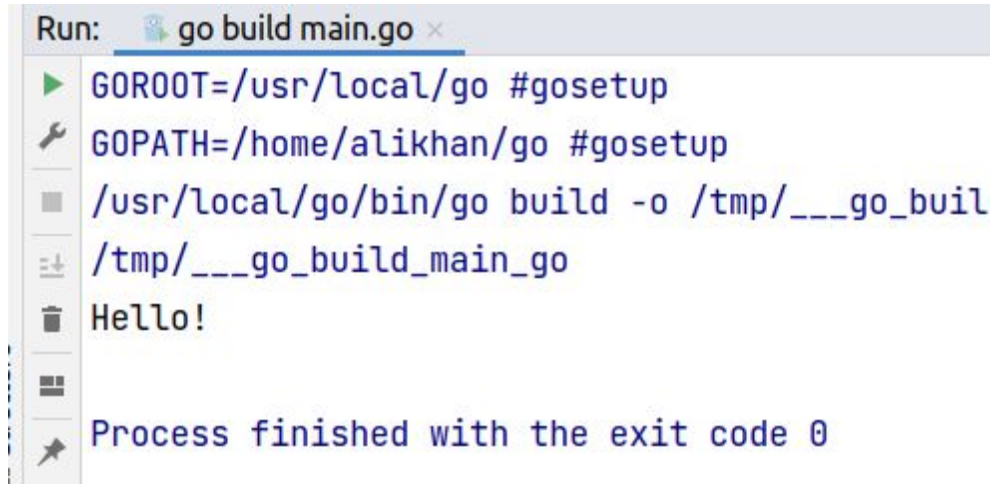
Подождем немного...

```
package main

import (
    "fmt"
    "time"
)

func main() {
    go printHello()
    time.Sleep(time.Millisecond)
}

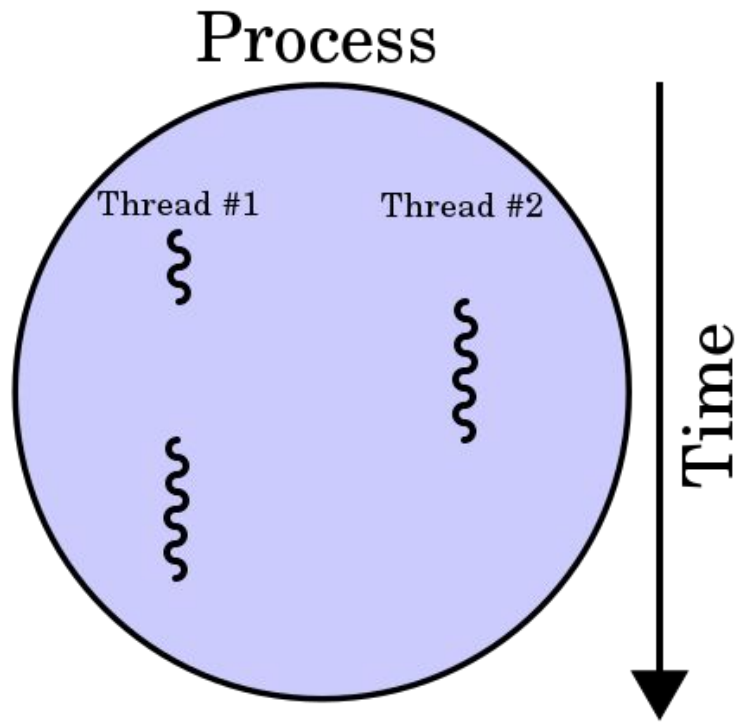
func printHello() {
    fmt.Println("Hello!")
}
```

A terminal window titled "Run: go build main.go" showing the execution of a Go program. The output includes environment variables for GOROOT and GOPATH, the build command, the output "Hello!", and a message stating "Process finished with the exit code 0".

```
Run: go build main.go x
▶ GOROOT=/usr/local/go #gosetup
🔧 GOPATH=/home/alikhan/go #gosetup
■ /usr/local/go/bin/go build -o /tmp/___go_buil
⚙ /tmp/___go_build_main_go
🗑 Hello!
📦
🚀 Process finished with the exit code 0
```

Потоки ОС (threads)

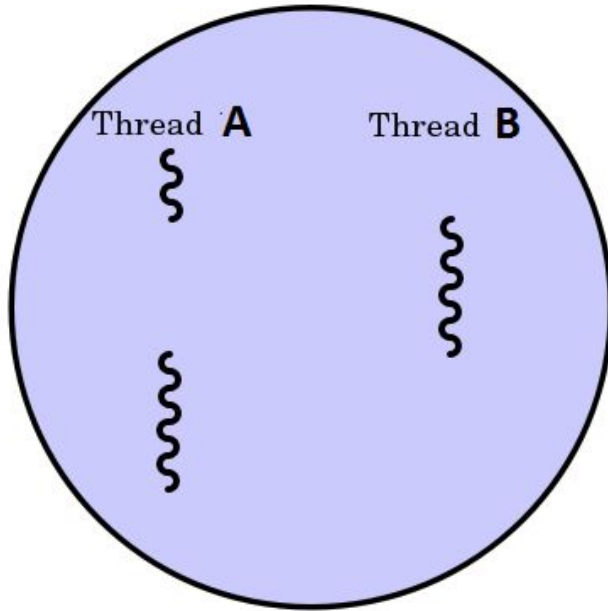
Поток выполнения — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Поток выполнения находится внутри процесса. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. Потоки выполнения разделяют инструкции процесса и его контекст (значения переменных, которые они имеют в любой момент времени).



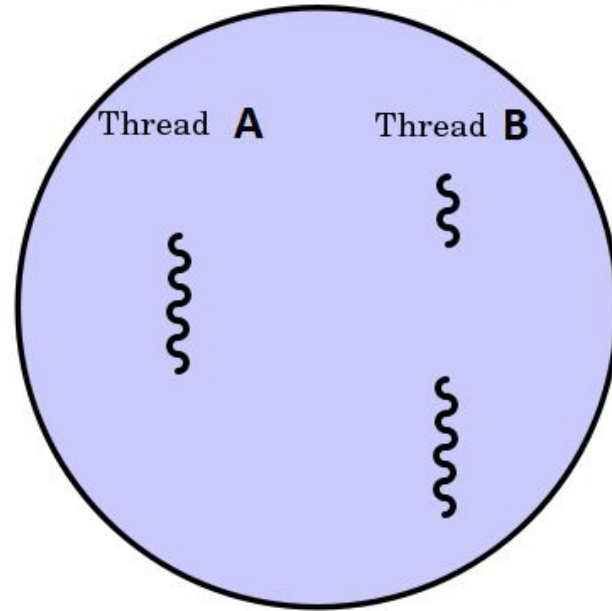
Процесс с двумя потоками выполнения на одном процессоре

Процессов может быть несколько

Process 1



Process 2



Горутины

Горутина (goroutine) — эта функция, выполняемая конкурентно с другими горутинами.

Горутины очень легковесны. Все расходы — это создание стека, который очень невелик, но при необходимости может расти.

Начальный размер стека горутины — 2 КБ.

Горутины и Машины

В исходном коде (`src/pkg/runtime/proc.c`) приняты следующие термины:

- **G (Goroutine)** — Горутина
- **M (Machine)** — Машина (поток ОС)

Планировщики

Каждая Машина работает в отдельном потоке и способна выполнять только одну Горутину в момент времени.

Планировщик **операционной системы**, в которой работает программа, переключает **Машины**.

Планировщик **Go runtime** переключает **Горутины**.

Число работающих машин ограничено переменной среды окружения **GOMAXPROCS** или функцией **runtime.GOMAXPROCS(n int)**.

По умолчанию, число Машин равно числу логических ядер процессора **runtime.NumCPU() int**.

Планировщик Go runtime

Цель планировщика (scheduler) заключается в распределении готовых к выполнению горутин (G) по свободным машинам (M).

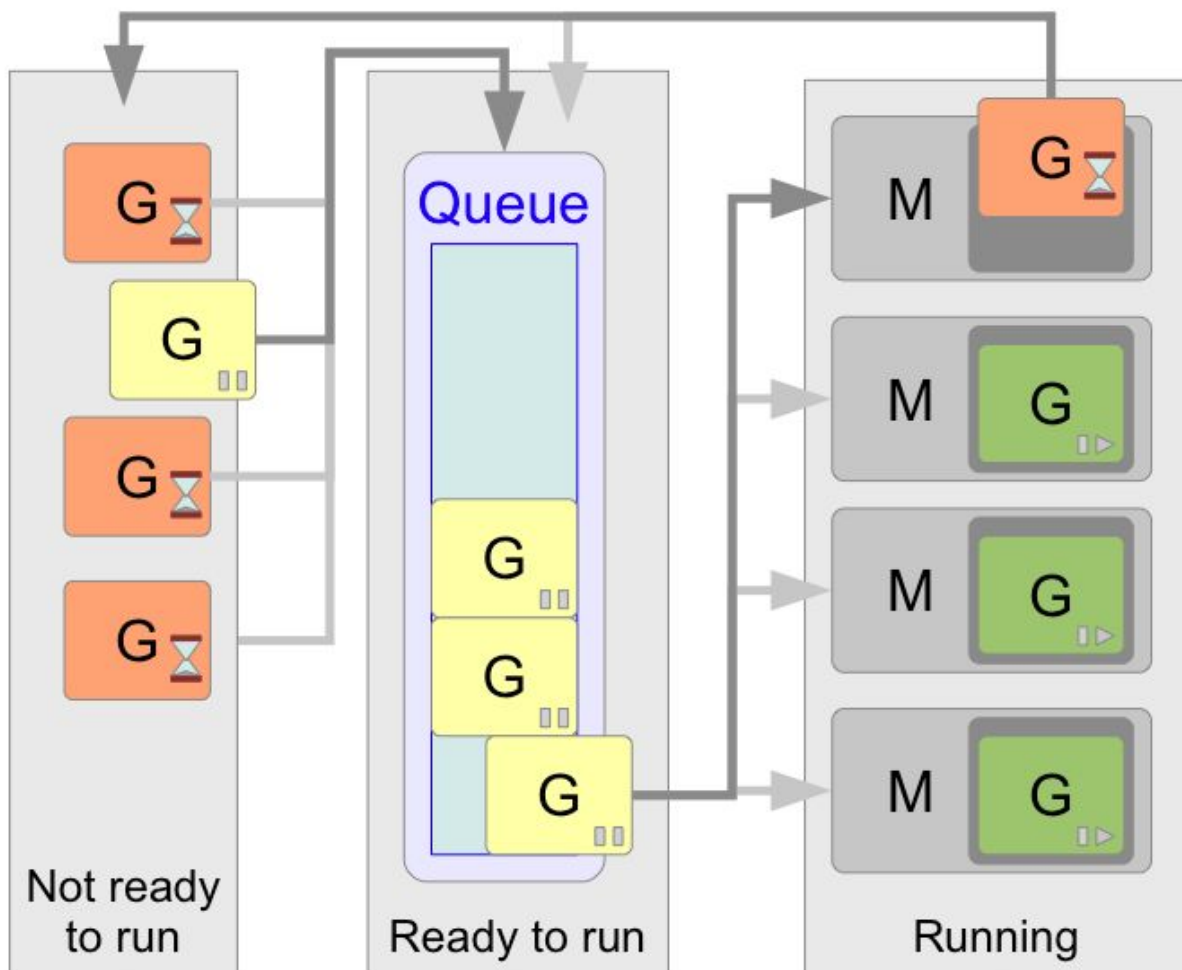
Готовые к исполнению горутин выполняются в порядке очереди FIFO. Исполнение горутин прерывается только тогда, когда она уже не может выполняться:

- из-за системного вызова
- `time.Sleep()`
- операции с мьютексами, каналами

Время выполнения горутины

Нет никаких гарантий по времени выполнения горутин и переключения между ними.

Неизвестно, в какой момент времени горутина будет запущена.



Задание 1

Реализовать функцию для подсчета длины html-страницы

<https://play.golang.org/p/gnaN0OabR8a>

```
// pageSize - TODO: implement
func pageSize(url string) (int, error) {
    return 0, fmt.Errorf("not implemented yet")
}
```

Запуск горутины

Для запуска горутины перед выходом функции необходимо прописать go:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    go func() {
        fmt.Println("Анонимная горутина")
    }()
    time.Sleep(time.Millisecond)
}
```


Запуск множества горутин

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0; i < 10; i++ {
        go printNumber(i)
    }
    time.Sleep(time.Millisecond)
}

func printNumber(n int) {
    fmt.Println(n)
}
```

Run: go build main.go

```
▶ GOROOT=/usr/local/go #gosetup
🔧 GOPATH=/home/alikhan/go #gosetup
■ /usr/local/go/bin/go build -o /tmp/___go_b
⇅ /tmp/___go_build_main_go
🗑 9
📊 4
➦ 5
6
7
8
1
2
3
0

Process finished with the exit code 0
```

Что будет, если убрать time.Sleep()

```
package main

import (
    "fmt"
)

func main() {
    for i := 0; i < 10; i++ {
        go printNumber(i)
    }
    //time.Sleep(time.Millisecond)
}

func printNumber(n int) {
    fmt.Println(n)
}
```

A terminal window titled 'Run: go build main.go x' showing the output of the Go build command. The output includes environment variables for GOROOT and GOPATH, the build command itself, and a confirmation that the process finished with exit code 0.

```
Run: go build main.go x
GOROOT=/usr/local/go #gosetup
GOPATH=/home/alikhan/go #gosetup
/usr/local/go/bin/go build -o /tmp/___go_b
/tmp/___go_build_main_go
Process finished with the exit code 0
```

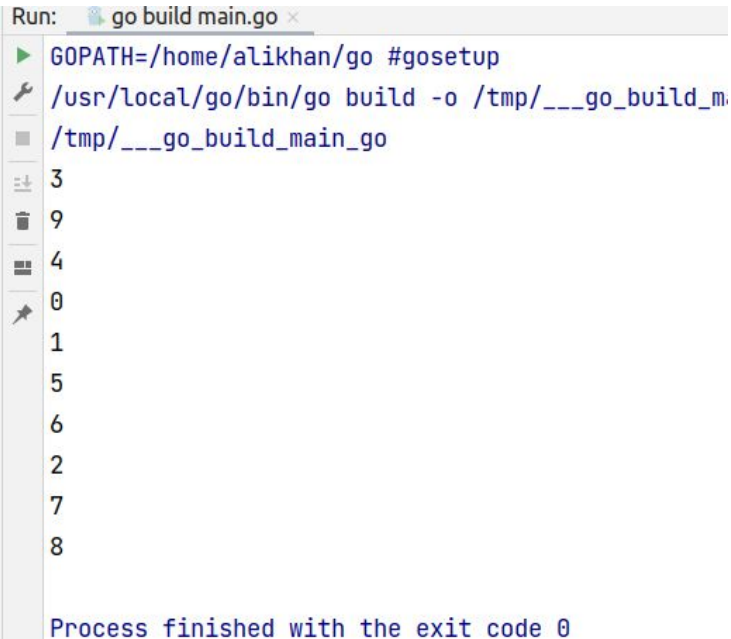
sync.WaitGroup

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go printNumber(i, &wg)
    }
    wg.Wait()
}

func printNumber(n int, wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Println(n)
}
```



The terminal window shows the command `go build main.go` being executed. The output displays the GOPATH, the command used to build the binary, and the path to the resulting executable. A list of numbers from 0 to 9 is printed, indicating that the program successfully executed 10 parallel goroutines and waited for them to complete. The final message states that the process finished with the exit code 0.

```
Run: go build main.go x
▶ GOPATH=/home/alikhan/go #gosetup
🔧 /usr/local/go/bin/go build -o /tmp/___go_build_m
■ /tmp/___go_build_main_go
⏏ 3
🗑 9
📊 4
📁 0
➦ 1
5
6
2
7
8

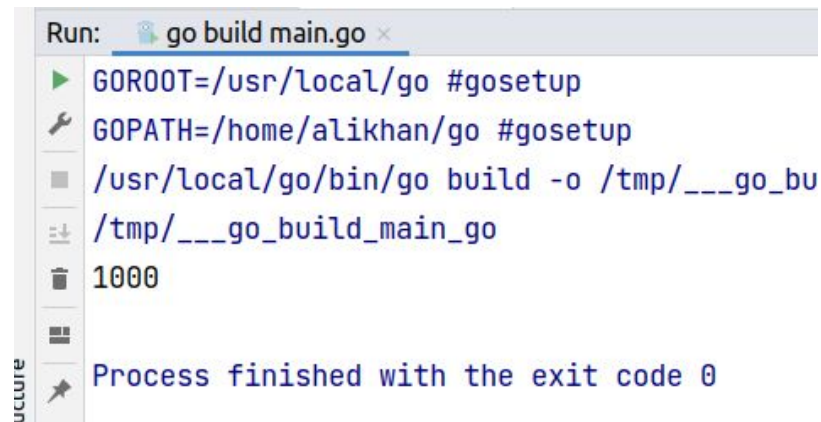
Process finished with the exit code 0
```

Последовательный счетчик

```
package main

import "fmt"

func main() {
    var counter int
    for i := 0; i < 1000; i++ {
        counter++
    }
    fmt.Println(counter)
}
```




```
Run: go build main.go x
GOROOT=/usr/local/go #gosetup
GOPATH=/home/alikhan/go #gosetup
/usr/local/go/bin/go build -o /tmp/___go_bu
/tmp/___go_build_main_go
1000
Process finished with the exit code 0
```

Конкурентный счетчик

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    var counter int
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            counter++
        }()
    }
    wg.Wait()
    fmt.Println(counter)
}
```



The terminal window shows the command 'go build main.go' being executed. The output displays the Go environment setup, including the GOPATH and the build command. The build process completes successfully, and the process exits with code 0.

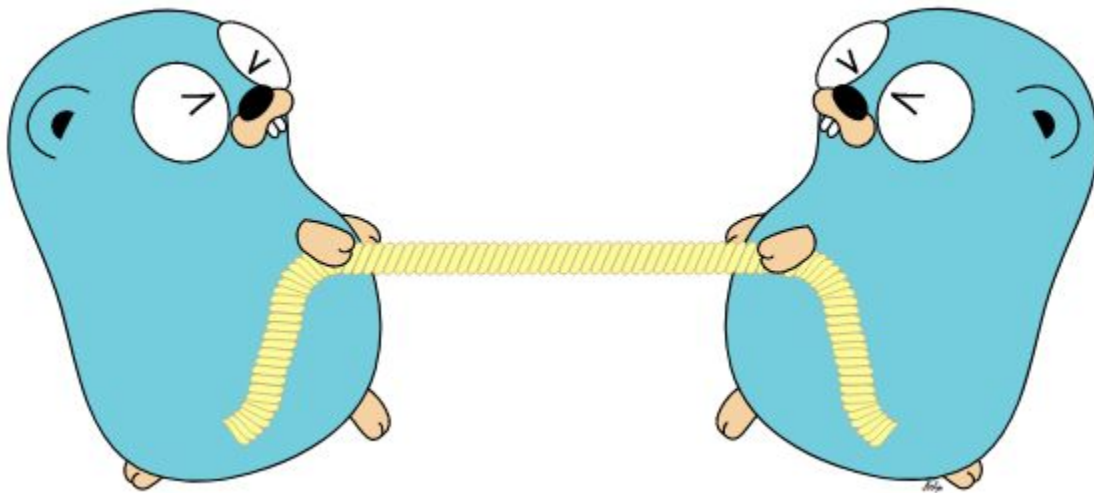
```
Run: go build main.go x
GOROOT=/usr/local/go #gosetup
GOPATH=/home/alikhan/go #gosetup
/usr/local/go/bin/go build -o /tmp/___go_build_
/tmp/___go_build_main_go
921
Process finished with the exit code 0
```

Data Race Condition

При конкурентной записи в разделяемую память возникает состояние гонки

```
counter++
```

```
temp := counter + 1  
counter = temp
```



Race Detector

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

```
Terminal: Local x +
alikhhan@cerebro:~/go/src/github.com/alikhhanmurzayev/test_project$ go run --race main.go
=====
WARNING: DATA RACE
Read at 0x00c0000160d8 by goroutine 8:
    main.main.func1()
        /home/alikhhan/go/src/github.com/alikhhanmurzayev/test_project/main.go:12 +0x38

Previous write at 0x00c0000160d8 by goroutine 7:
    main.main.func1()
        /home/alikhhan/go/src/github.com/alikhhanmurzayev/test_project/main.go:12 +0x4e

Goroutine 8 (running) created at:
    main.main()
        /home/alikhhan/go/src/github.com/alikhhanmurzayev/test_project/main.go:11 +0x84

Goroutine 7 (finished) created at:
    main.main()
        /home/alikhhan/go/src/github.com/alikhhanmurzayev/test_project/main.go:11 +0x84
=====
9960
Found 1 data race(s)
exit status 66
alikhhan@cerebro:~/go/src/github.com/alikhhanmurzayev/test_project$
```

sync.Mutex

Блокировка разделяемой памяти
мьютексом помогает избавиться от
гонки.

`mu.Lock()` — заблокировать ресурс

`mu.Unlock()` — разблокировать

```
Run: go build main.go x
▶ GOROOT=/usr/local/go #gosetup
✎ GOPATH=/home/alikhan/go #gosetup
■ /usr/local/go/bin/go build -o /tmp/___go_build
  /tmp/___go_build_main_go
  1000
■
✎ Process finished with the exit code 0
```

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    var mu sync.Mutex
    var counter int
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            mu.Lock()
            defer mu.Unlock()
            counter++
        }()
    }
    wg.Wait()
    fmt.Println(counter)
}
```


Задание 2

Модифицировать программу из задания 2 следующим образом:

- подсчет длин страниц выполнять конкурентно;
- результаты записывать в `map[string]int`, где ключ — url, значение — длина;
- исключить Data Race Condition.

Каналы (channels)

Каналы — средство передачи данных между горутинами (и не только).

В каналы можно отправлять объекты, получать из них объекты.

- Работают по принципу очереди (FIFO);
- Типизированы (работают со значениями определенного типа)

Создание канала

Значение по умолчанию — nil

```
var ch chan int  
ch = make(chan int)
```

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

Использование канала

<https://play.golang.org/p/5MBNI7AKKMV>

Типы каналов

- Однонаправленные (только чтение или запись), двунаправленные;
- Буферизованные, небуферизованные;
- Открытые, закрытые

Однонаправленные каналы

```
var writeOnly chan<- string  
var readOnly <-chan string
```

<https://play.golang.org/p/nVU6l1VtdcX>

Буферизованные каналы

<https://play.golang.org/p/kqGGoE4jF4Y>

Открытые и закрытые каналы

```
package main

import "fmt"

func main() {
    ch := make(chan string, 1)
    ch <- "hello"
    close(ch)

    s, ok := <-ch
    fmt.Println(s, ok) // "hello" true

    s, ok = <-ch
    fmt.Println(s, ok) // "" false

    fmt.Println(<-ch) // ""

    ch <- "world" // panic: send on closed channel
}
```


Читаем из канала, пока он не закрыт

```
for {  
    x, ok := <-ch  
    if !ok {  
        break  
    }  
    fmt.Println(x)  
}
```

```
for x := range ch {  
    fmt.Println(x)  
}
```

Мультиплексирование (select)

- select ждет, когда один из каналов будет готов к чтению или записи
- Если никто не готов, срабатывает default (если он объявлен)

```
select {  
  case <-ch1:  
    // read from channel  
  case ch2 <- 23:  
    // wrote to channel  
  case , ok := <-ch3:  
    if !ok {  
      // ch3 is closed  
      return  
    }  
  default:  
    // channels are not ready  
}
```

Замыкание

https://play.golang.org/p/yH7n_kBNKY2

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0; i < 5; i++ {
        go func() {
            fmt.Println(i)
        }()
    }
    time.Sleep(time.Millisecond)
}
```

Что происходит при запуске горутины?

1. Вычисляются аргументы функции
2. Горутина кладется в очередь ожидания

```
package main

import (
    "fmt"
    "time"
)

func main() {
    go printString(getArgument())
    // in getArgument
    // some argument
    time.Sleep(time.Millisecond)
}

func printString(s string) {
    fmt.Println(s)
}

func getArgument() string {
    fmt.Println("in getArgument")
    return "some argument"
}
```

Замыкание

<https://play.golang.org/p/Jw5xk1zCnU8>

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0; i < 5; i++ {
        num := i
        go func() {
            fmt.Println(num)
        }()
    }
    time.Sleep(time.Millisecond)
}
```

ДЗ

- Пройти go tour по concurrency: <https://tour.golang.org/concurrency>
- Легендарное видео Роба Пайка: <https://youtu.be/oV9rvDIIKEg>
- Многопроцессность, многопоточность, асинхронность в Python и не только: <https://youtu.be/Jlp14T9bvvc>
- Асинхронный Python-код медленнее обычного кода: <https://youtu.be/z7WIm0iZcOU>
- Понять, что такое hyperthreading: <https://youtu.be/wnS50IJicXc>, <https://youtu.be/lrT9BI0MCXQ>
- Почитать про планировщик: <https://rakyll.org/scheduler/>, https://medium.com/@ankur_anand/illustrated-theses-of-go-runtime-scheduler-74809ef6d19b
- Почитать <https://habr.com/ru/company/otus/blog/527748/>

Д3. Конкурентное выполнение задач

Написать функцию, которая на вход принимает слайс задач `tasks` и `E` — максимально допустимое количество ошибок в задачах. Если количество задач с ошибками превысило `E`, функция `Execute` должна вернуть соответствующую ошибку.

Написать тесты к функции.

```
// Execute - TODO: implement
func Execute(tasks []func() error, E int) error {
    return nil
}
```

Д3. Конкурентное выполнение задач — продолжение

Написать функцию, которая на вход принимает слайс задач `tasks` и `E` — максимально допустимое количество ошибок в задачах. Если количество задач с ошибками превысило `E`, стоит прекратить выполнение всех запущенных задач (если это возможно), а функция `Execute` должна вернуть соответствующую ошибку.

Для наглядности завершения задач можно в них запускать циклы и при каждой итерации проверять контекст.

Написать тесты к функции.

```
// Execute - TODO: implement
func Execute(tasks []func(ctx context.Context) error, E int) error {
    return nil
}
```