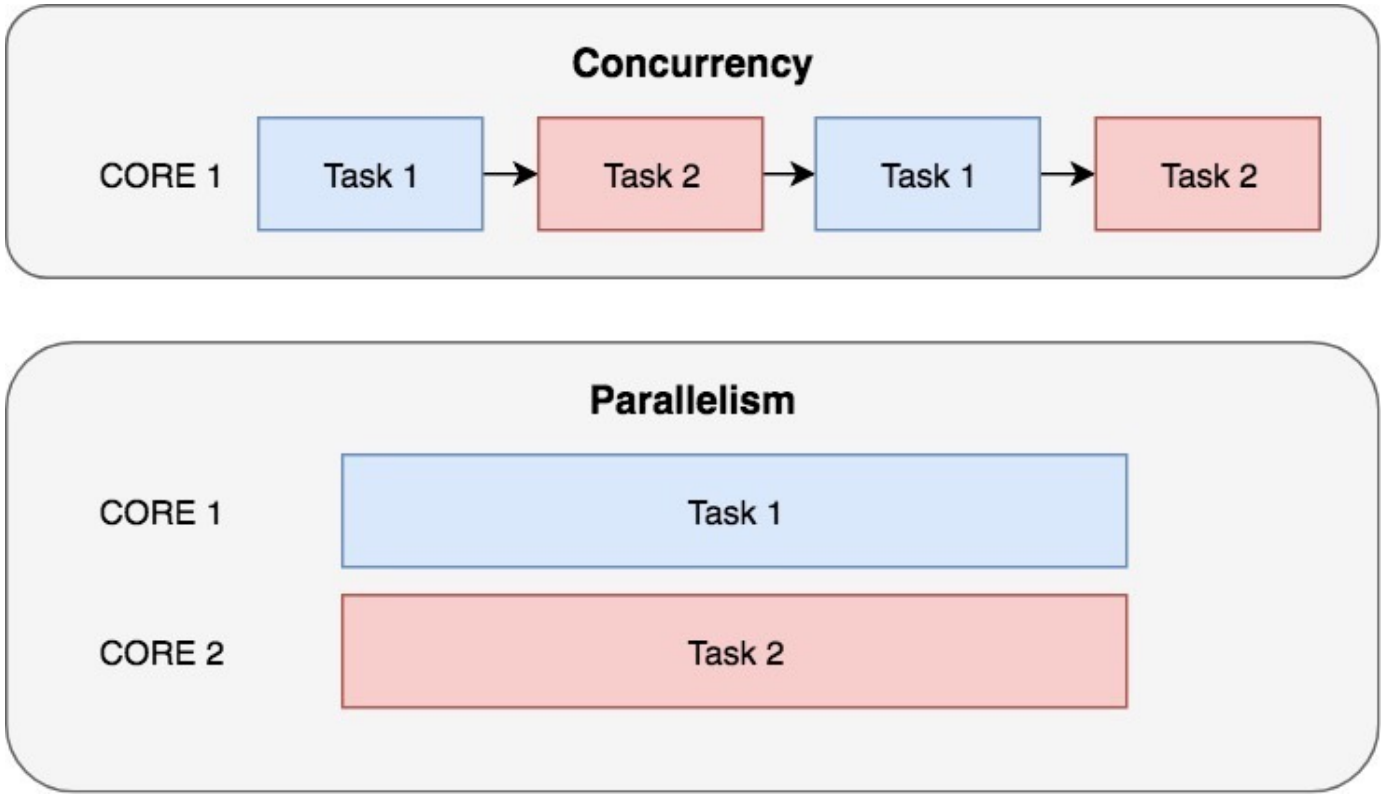
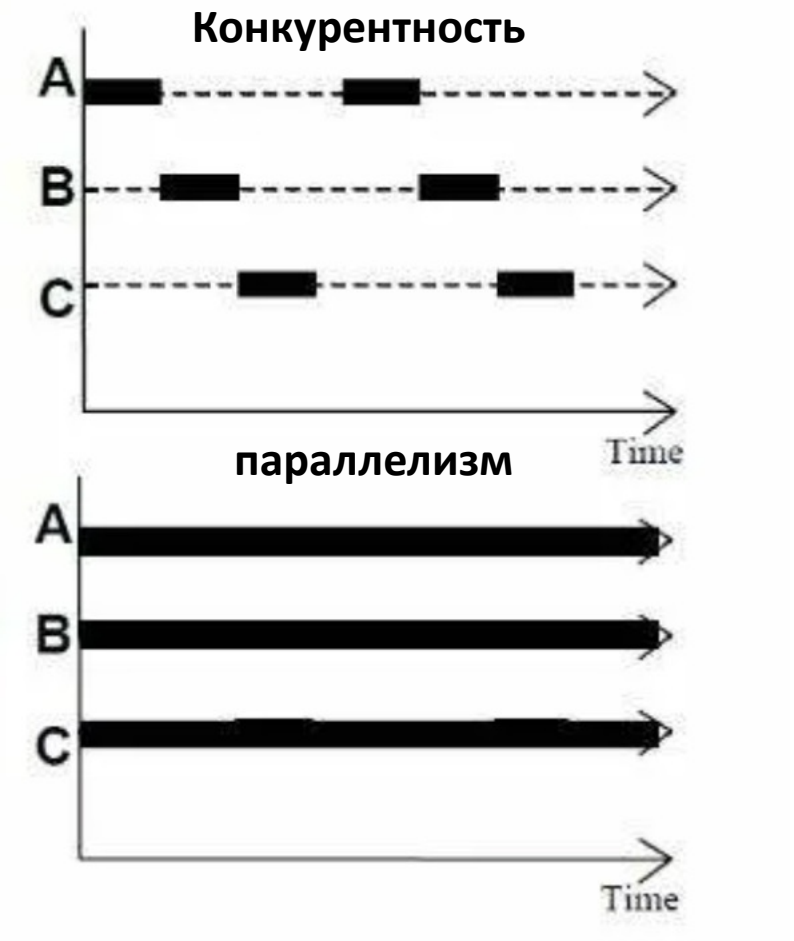
Лекция 3
Асинхрная модель в Go

План занятия

- Планировщик Го и Го-рутины
- Примитивы синхронизации
- Подходы(паттерны) в многопоточным программировании

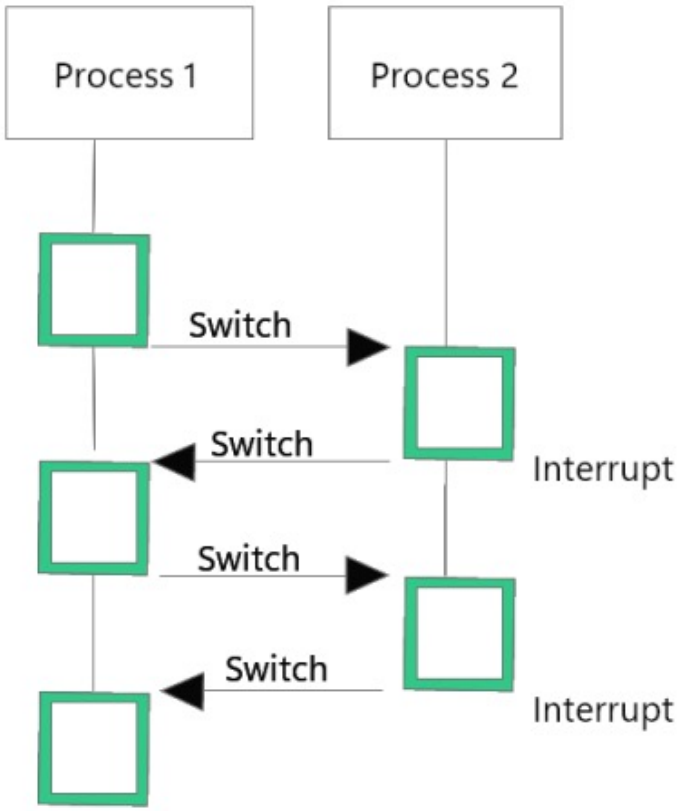
Конкурентность и параллелизм



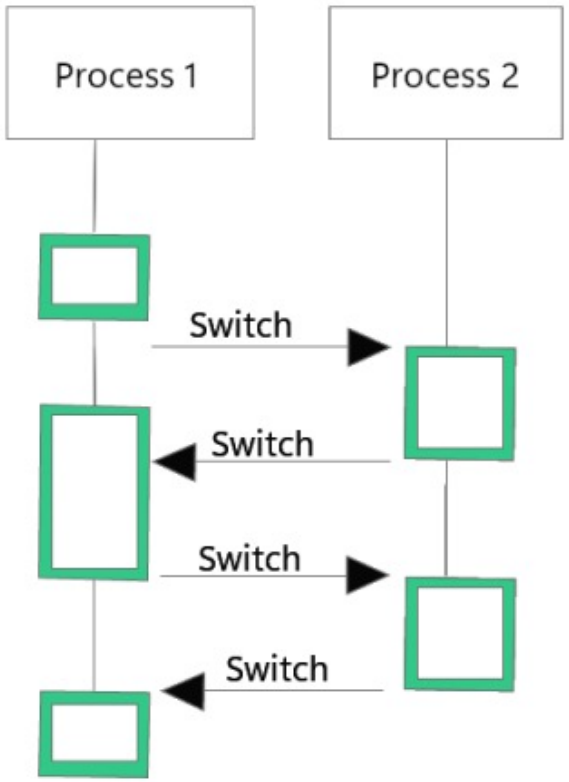


Кооперативная и вытесняющая многозадачность

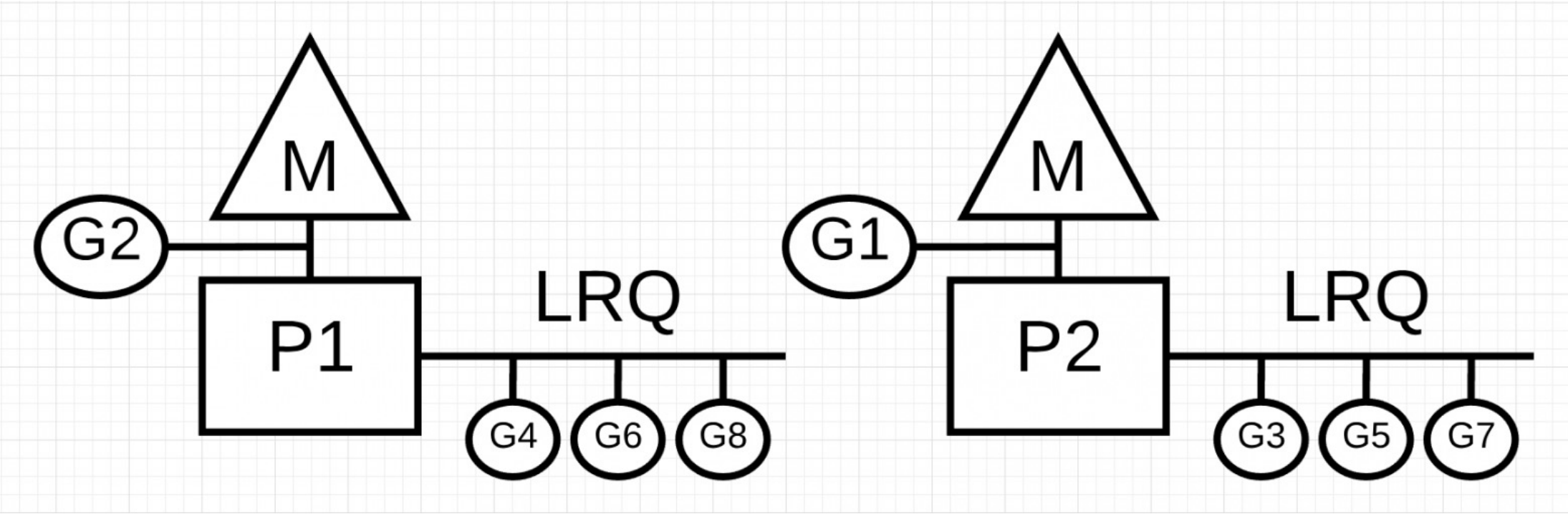
Вытесняющая



Кооперативная

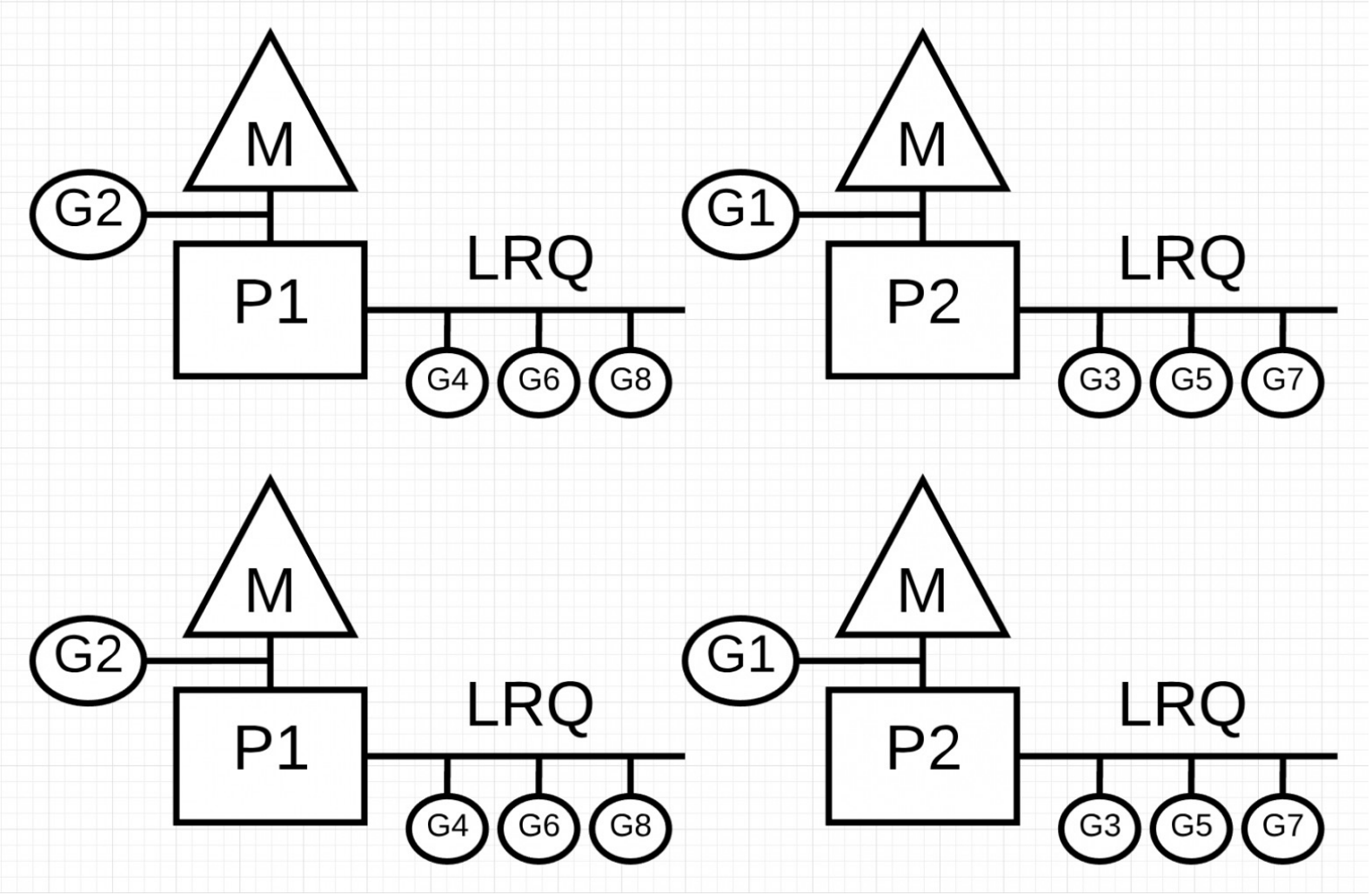


Планировщик в Го



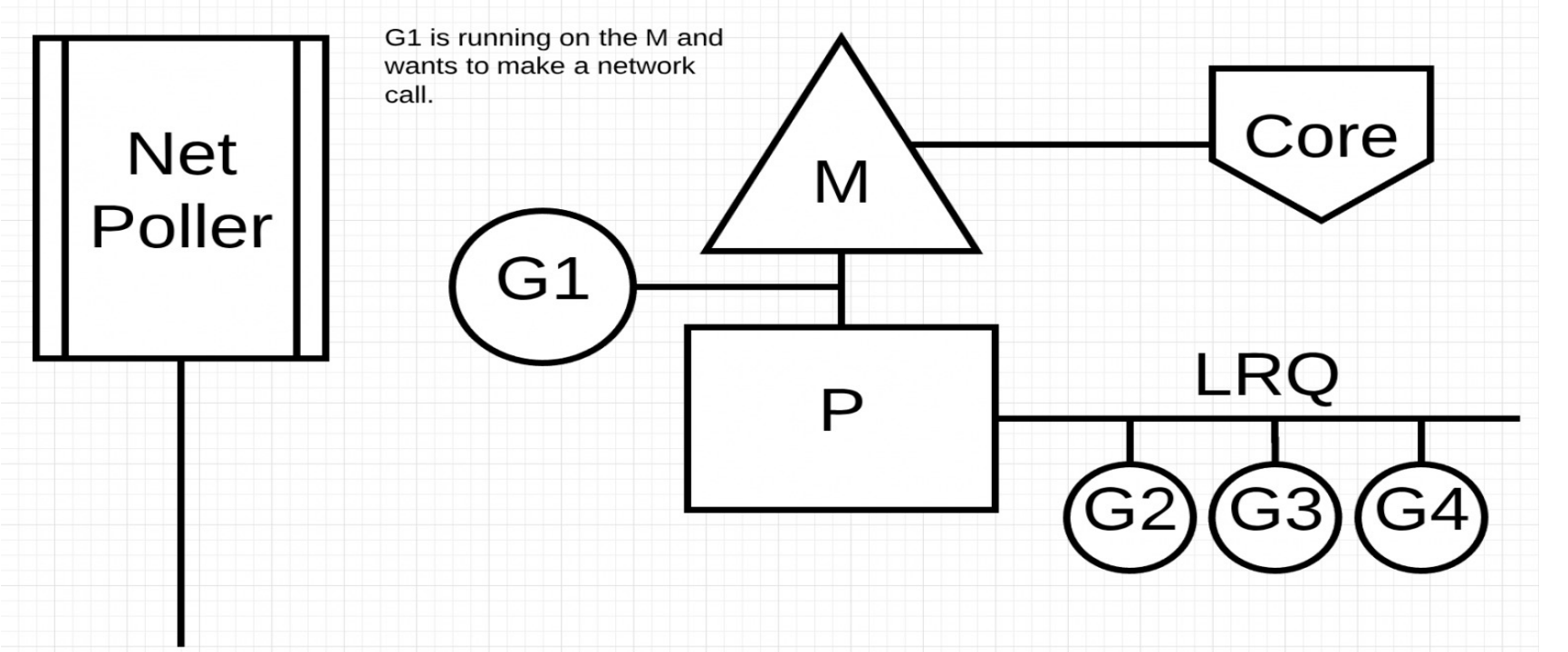
M(machine) – поток ОС / OS threads,
G — goroutines,
P — контекст планирования

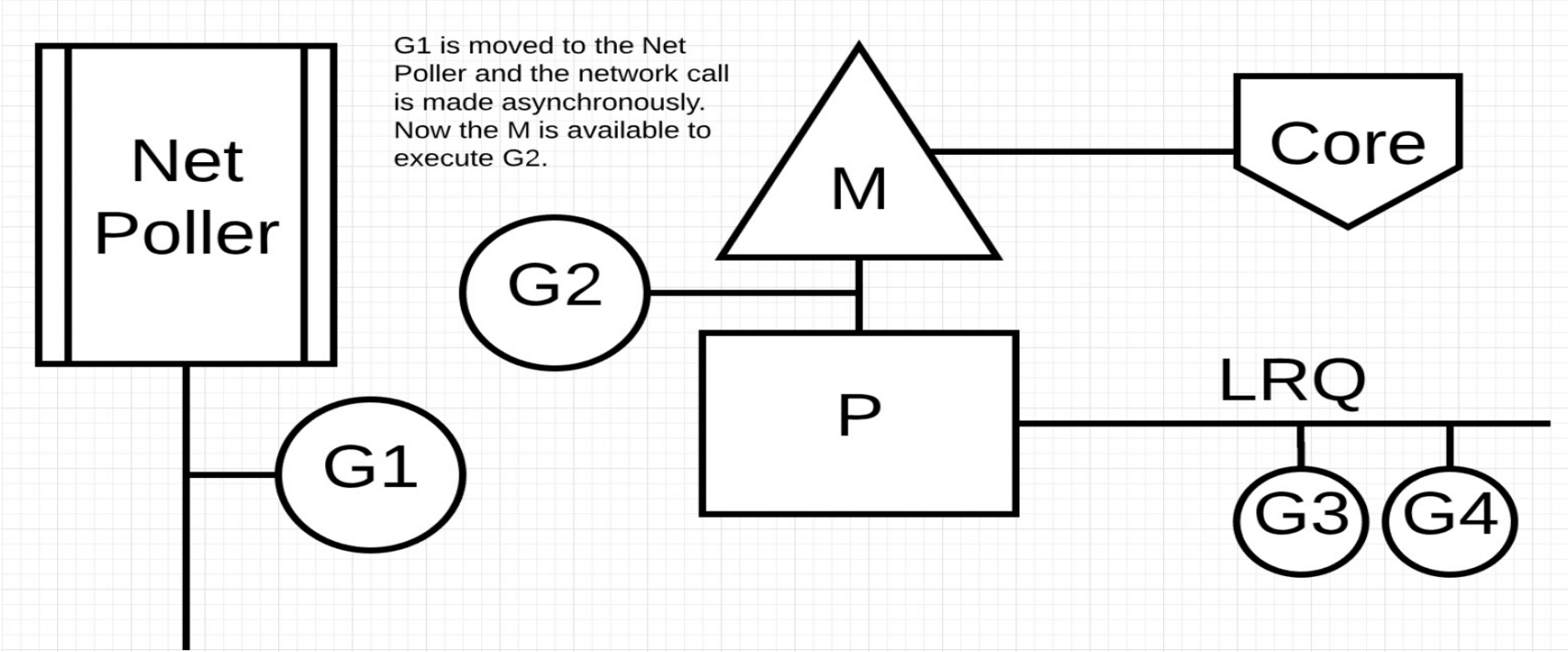
Планировщик в Го



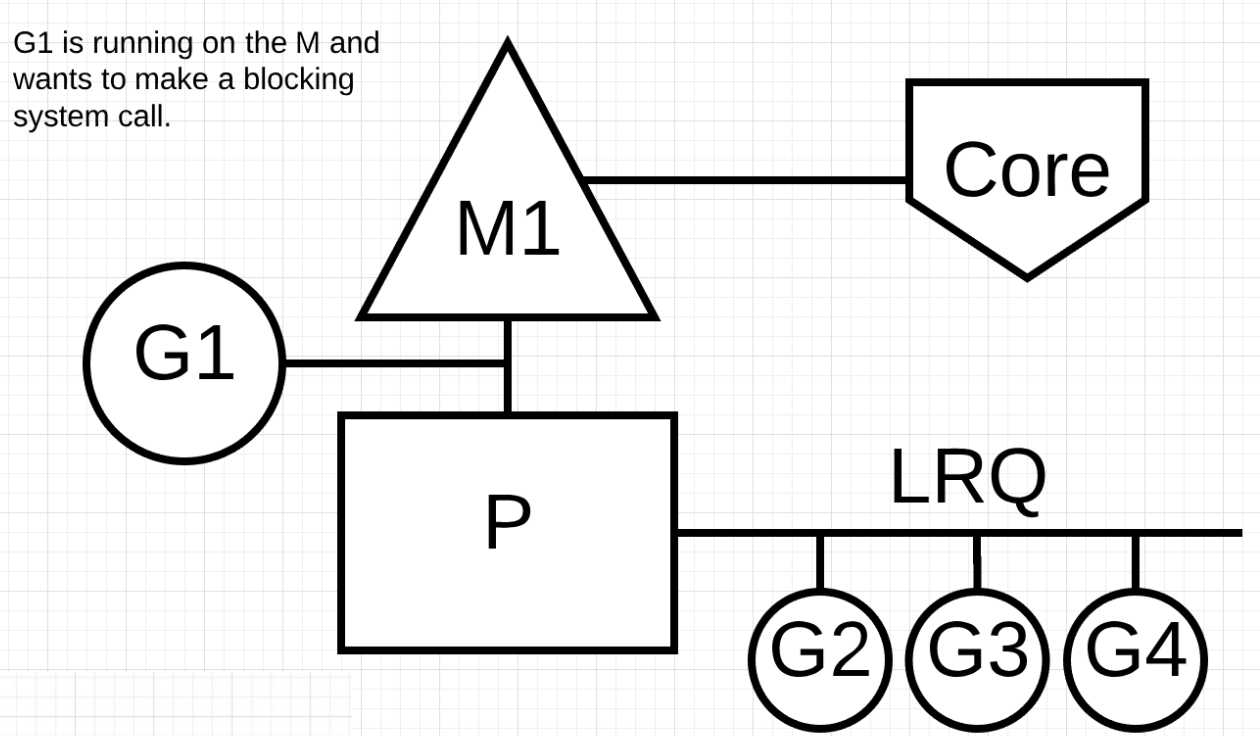
Неблокирующие системные вызовы

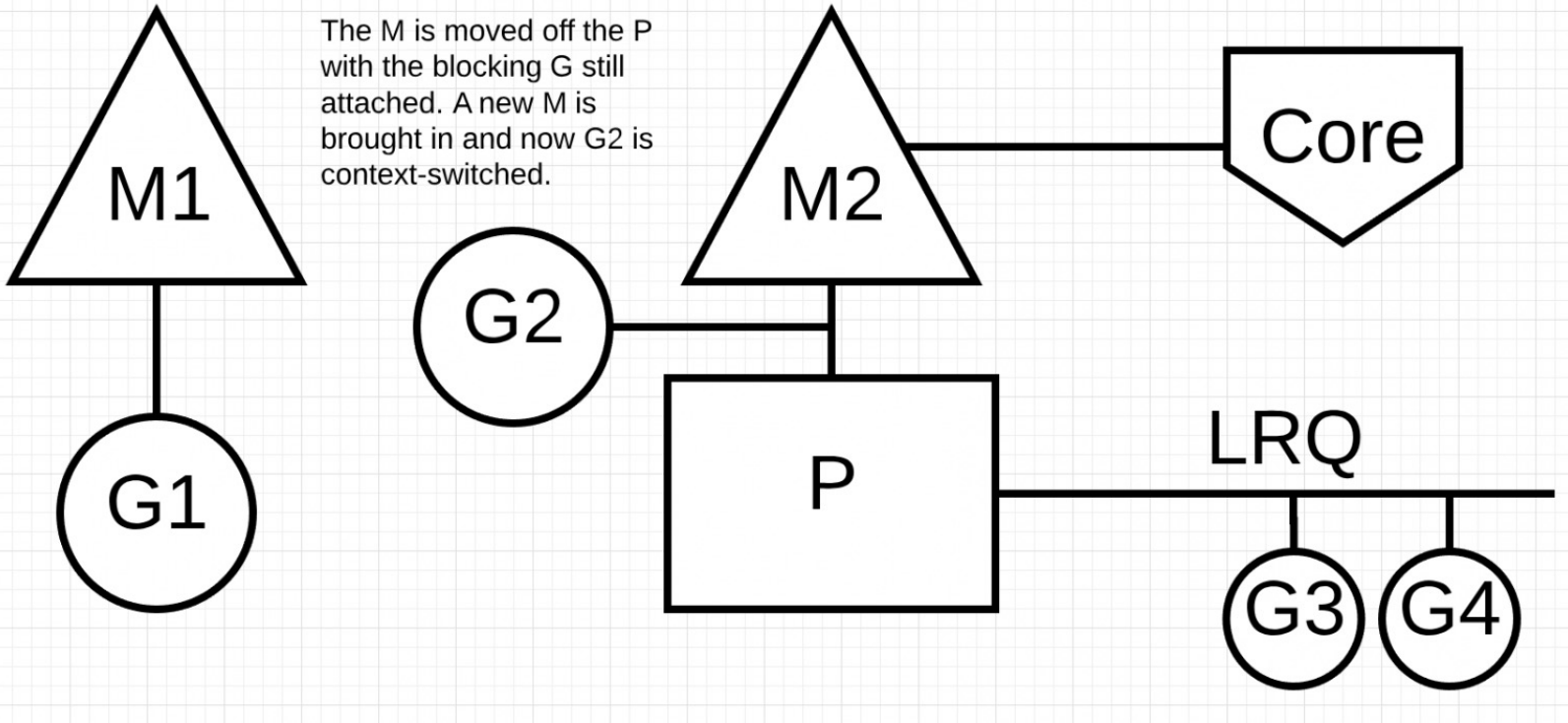
механизм
Net Poller





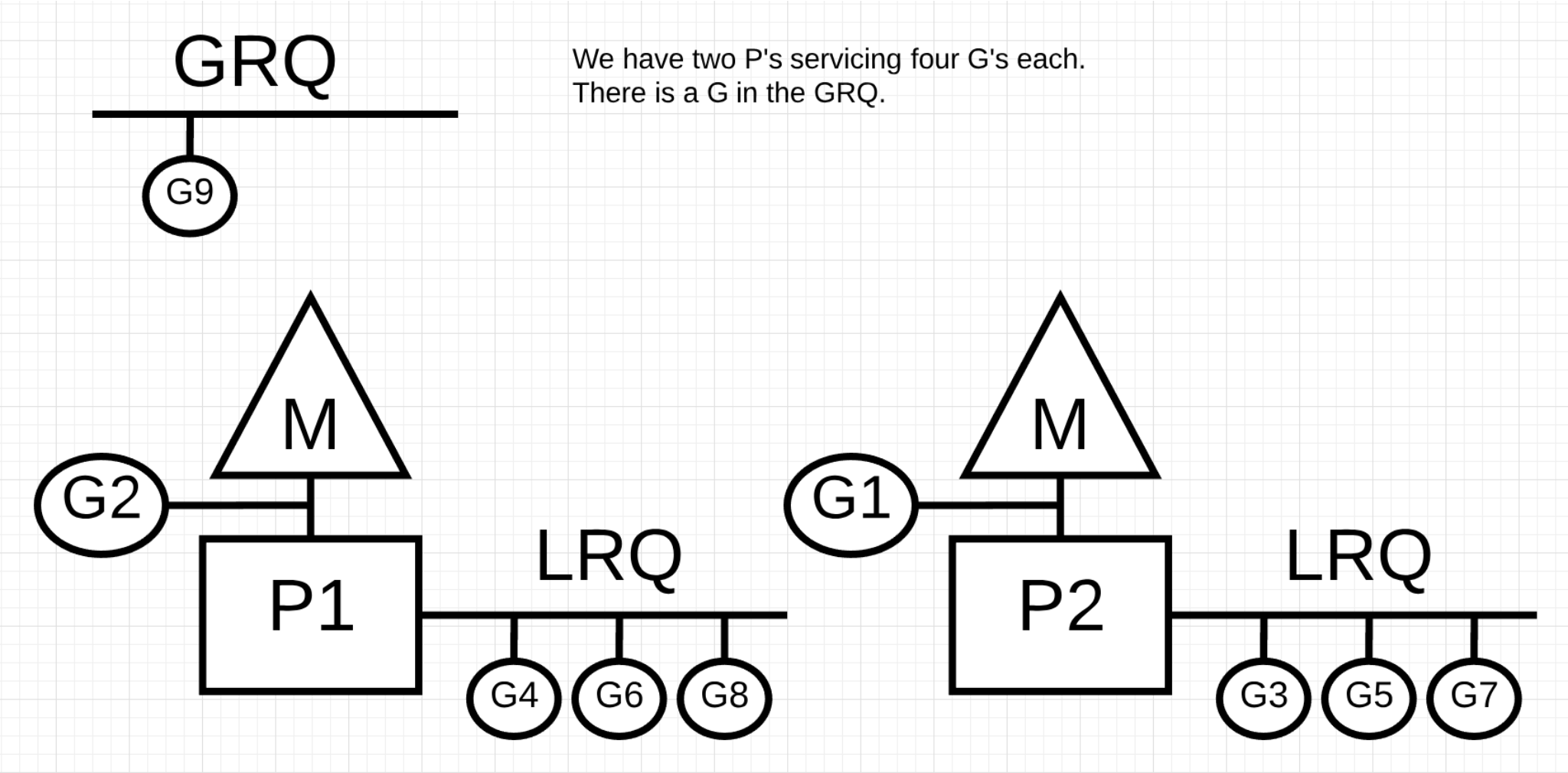
Блокирующие системные вызовы





Глобальная очередь горутин - GRQ

Work stealing



go func()

Горутины в Go запускаются с использованием ключевого слова

go

```go
func someWork(id string) {  6 usages  new *
    fmt.Println(id, ": начало работы", time.Now())

    // Имитация работы
    time.Sleep(time.Second)

    fmt.Println(id, ": завершение работы", time.Now())
}

func main() {  new *
    go someWork( id: "Work 1")
    go someWork( id: "Work 2")
    go someWork( id: "Work 3")
}
```

WaitGroup

```go
func main() {  new *
    wg := sync.WaitGroup{}

    someWork := func(id string) {
        fmt.Println(id, ": начало работы", time.Now())

        // Имитация работы
        time.Sleep(time.Second)

        fmt.Println(id, ": завершение работы", time.Now())
        wg.Done()
    }

    wg.Add( delta: 1)
    go someWork( id: "Work 1")
    wg.Add( delta: 1)
    go someWork( id: "Work 2")

    wg.Wait()
}
```

WaitGroup

- передача вейтгруппы в функцию должна осуществляться по указателю

- инкремент вейтгруппы должен быть вызван до вызова горутины

Проблемы многопоточного кода

- race condition (условие гонки)

ситуация, когда несколько горутин (или потоков) одновременно обращаются к одному и тому же ресурсу (например, переменной) и хотя бы одна из них изменяет это состояние. В результате, конечный результат зависит от порядка, в котором выполняются горутины, что приводит к непредсказуемому поведению и ошибкам.

- deadlock

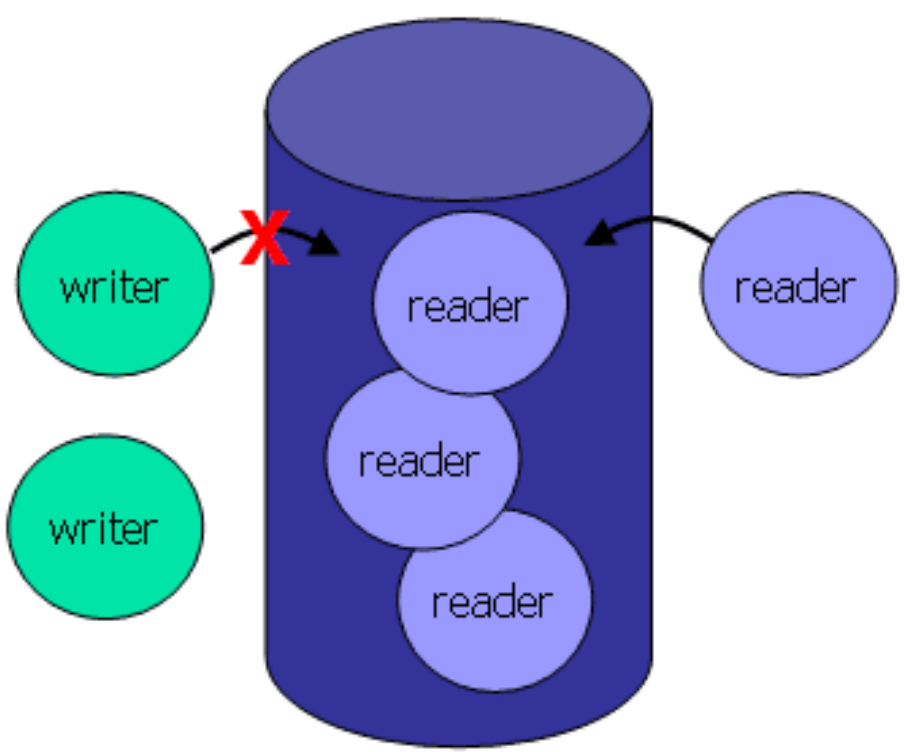
ситуация, когда две или более горутины (или потока) блокируют друг друга, ожидая освобождения ресурсов, и в результате ни одна из них не может продолжить выполнение.

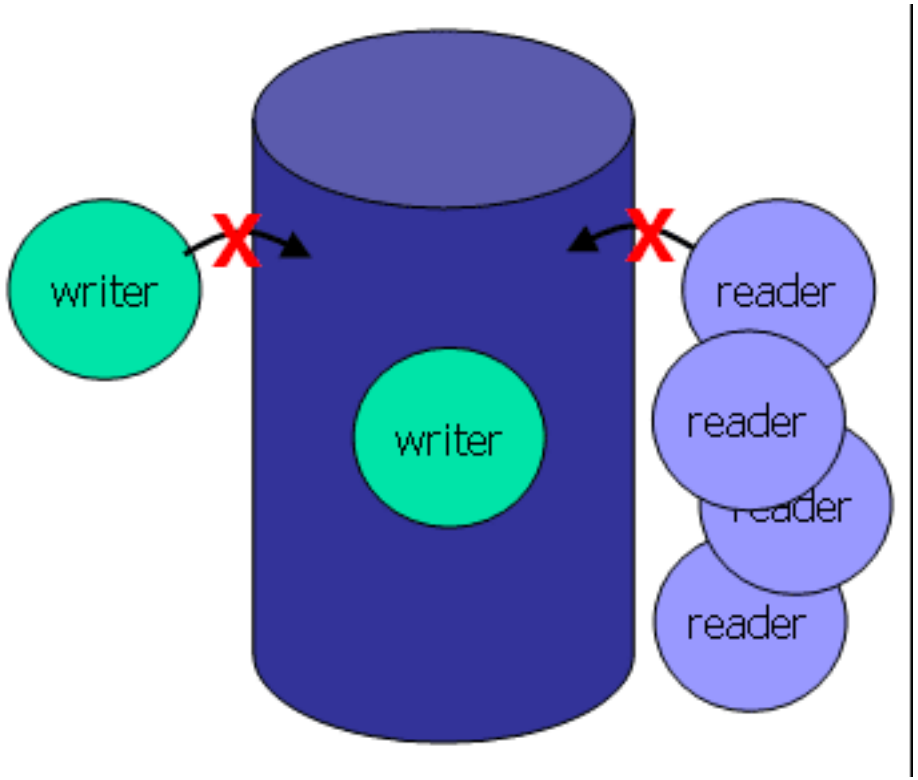
Mutex

```go
var counter int  1 usage  new *
var mutex sync.Mutex  2 usages  new *

func increment() {  2 usages  new *
    for i := 0; i < 5000; i++ {
        mutex.Lock()
        counter++
        mutex.Unlock()
    }
}

func main() {  new *
    go increment()
    go increment()
}
```

sync.RWMutex

sync.RWMutex предоставляет механизм блокировки, который позволяет нескольким горутинам одновременно читать данные, но изменять данные может только одна горутина.





Каналы (Channel)

Каналы (channels) в Go — это механизм для:

- обмена данными между горутинами.
- Синхронизации горутин

Каналы

- объявдение

ch := make(chan тип)

- закрытие канала

close(ch)

- запись в канал

ch <- val

- чтение (1 возвращаемый аргумента)

val <- ch

- чтение (2 возвращаемых аргумента)

val, ok <- ch // ok – закрыт ли канал

```go
func main() {  new *
    ch := make(chan string)

    go func() {
        for i := 0; i <= 9; i++ {
            ch <- fmt.Sprintf( format: "Hello #%d", i)
        }

        close(ch)
    }()

    for {
        v, ok := <-ch
        if !ok {
            fmt.Println( a...: "channel closed")
            break
        }

        fmt.Println(v)
    }
}
```

Каналы

```go
func main() {  new *
    ch := make(chan string)

    go func() {
        for i := 0; i <= 9; i++ {
            ch <- fmt.Sprintf( format: "Hello #%d", i)
        }

        close(ch)
    }()

    for {
        v, ok := <-ch
        if !ok {
            fmt.Println( a...: "channel closed")
            break
        }

        fmt.Println(v)
    }
}
```

Каналы

select – одновременное чтение нескольких каналов

```go
func main() {  new *
    ch1 := make(chan string)
    ch2 := make(chan string)

    go writer1(ch1)
    go writer2(ch2)

    for i := 0; i < 6; i++ {
        select {
        case msg := <-ch1:
            fmt.Println(msg)
        case msg := <-ch2:
            fmt.Println(msg)
        }
    }
}
```

Context

```go
func main() {  new *
    ch := make(chan int)
    ctx, cancel := context.WithCancel(context.Background())

    go writer(ch)

    go func() {
        for {
            select {
            case msg := <-ch:
                fmt.Println(msg)
            case _ = <-ctx.Done():
                fmt.Println( a...: "got signal from done chan. exiting")
                return
            }
        }
    }()

    time.Sleep(3 * time.Second)
    cancel()
    time.Sleep(100 * time.Millisecond)
}
```

Pipeline

```go
func gen(nums ...int) <-chan int {
  out := make(chan int)
  go func() {
    for _, n := range nums {
      out <- n
    }
    close(out)
  }()
  return out
}

func sq(in <-chan int) <-chan int {
  out := make(chan int)
  go func() {
    for n := range in {
      out <- n * n
    }
    close(out)
  }()
  return out
}
```

```go
func main() {
  c := gen(2, 3)
  out := sq(c)

  fmt.Println(<-out) // 4
  fmt.Println(<-out) // 9
}
```

https://go.dev/blog/pipelines

Worker Pool

```go
func worker(wg *sync.WaitGroup, jobs <-chan int, results chan<- int) {  1 usage  new *
    for j := range jobs {
        results <- j * 10
    }
    wg.Done()
}

func main() {  new *
    wg := &sync.WaitGroup{}
    inChan := getData( n: 100)

    resultsChan := make(chan int, numJobs)

    for w := 0; w <= numJobs; w++ {
        wg.Add( delta: 1)
        go worker(wg, inChan, resultsChan)
    }

    go func() {
        wg.Wait()
        close(resultsChan)
    }()

    for res := range resultsChan {
        fmt.Println(res)
    }
}
```

Вопросы