Golang

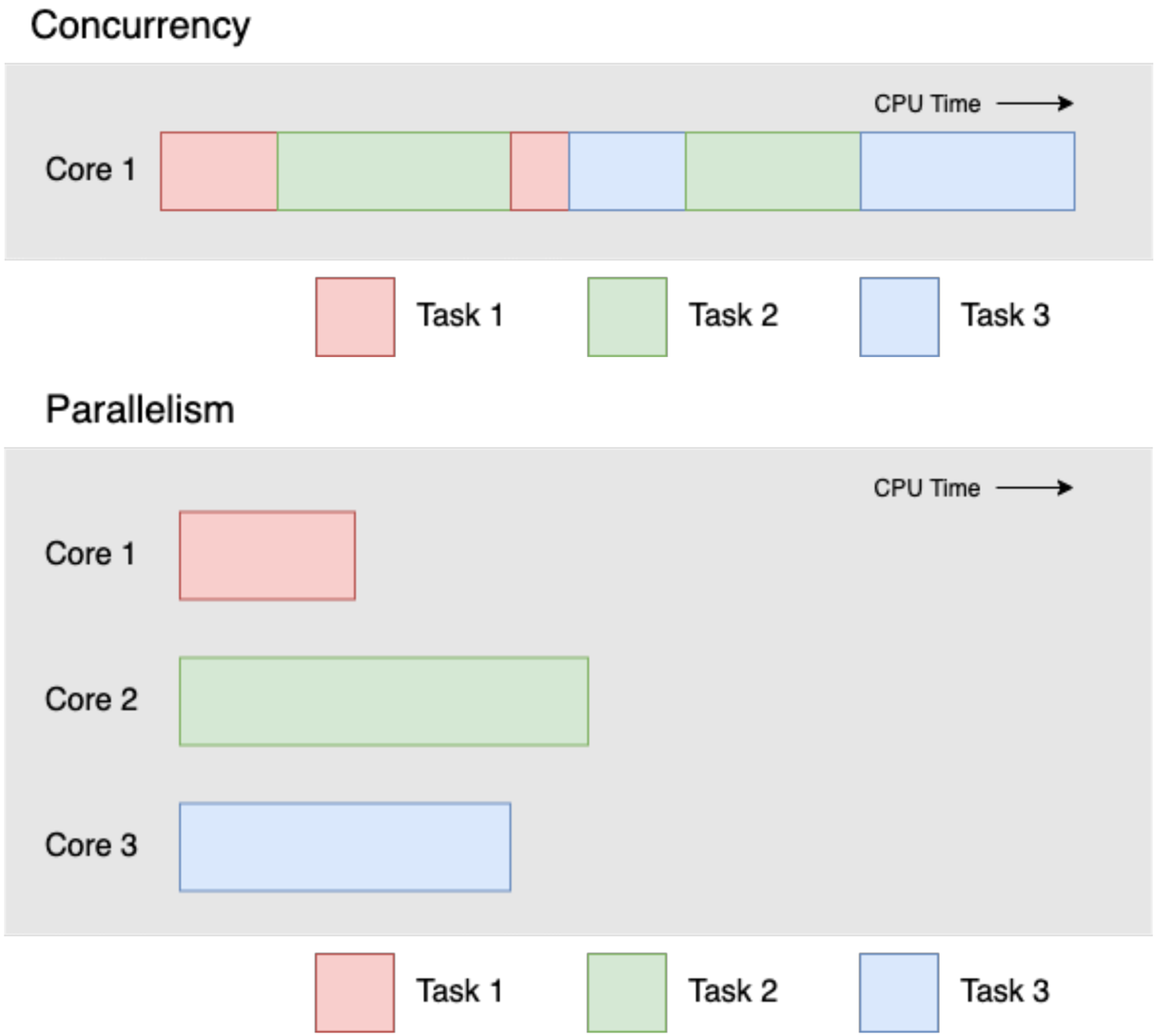
Асинхронная модель

План занятия

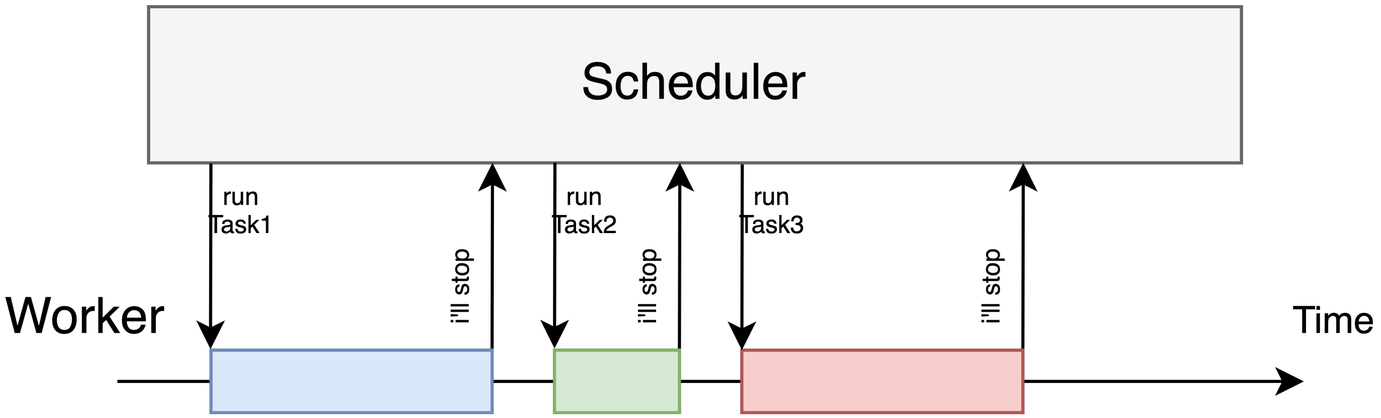
1. Golang Scheduler
2. Goroutines and synchronization primitives
3. Channels
4. Context
5. Pipelines

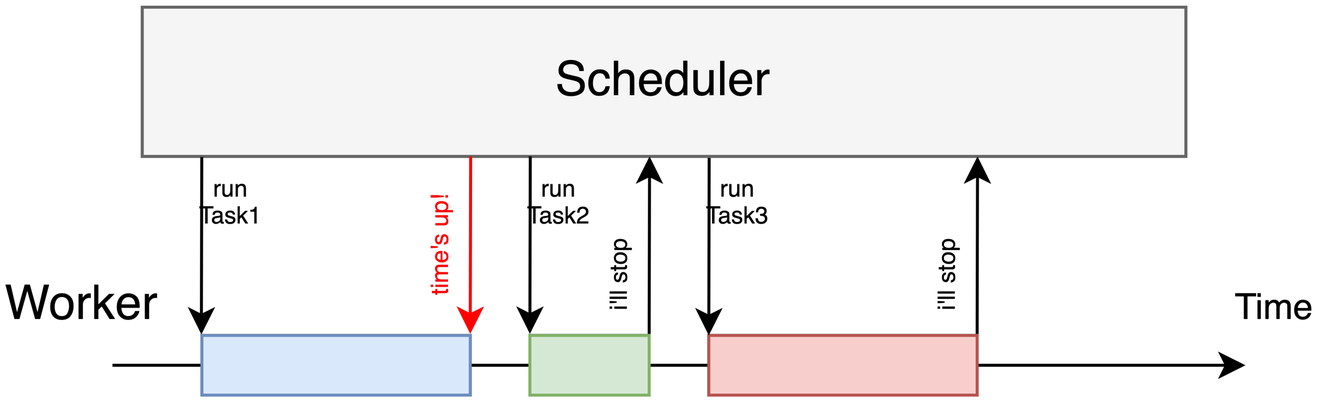
Go Scheduler

Конкурентность и параллелизм

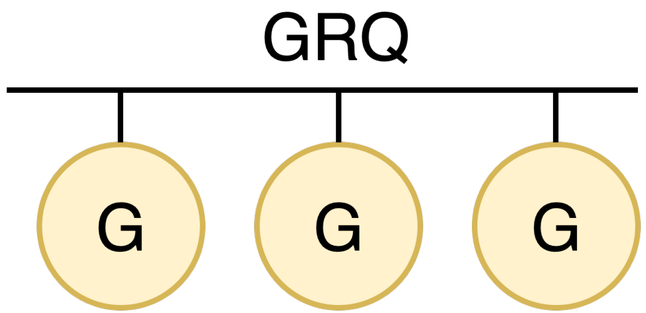


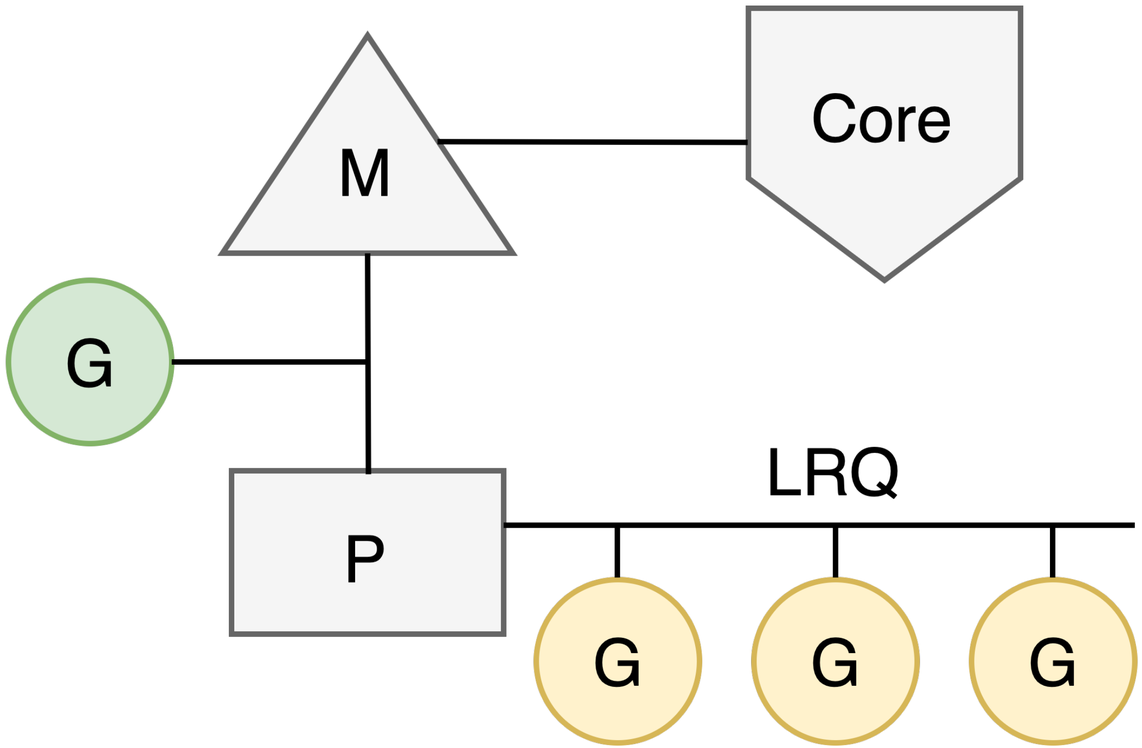
Вытесняющая и кооперативная многозадачность



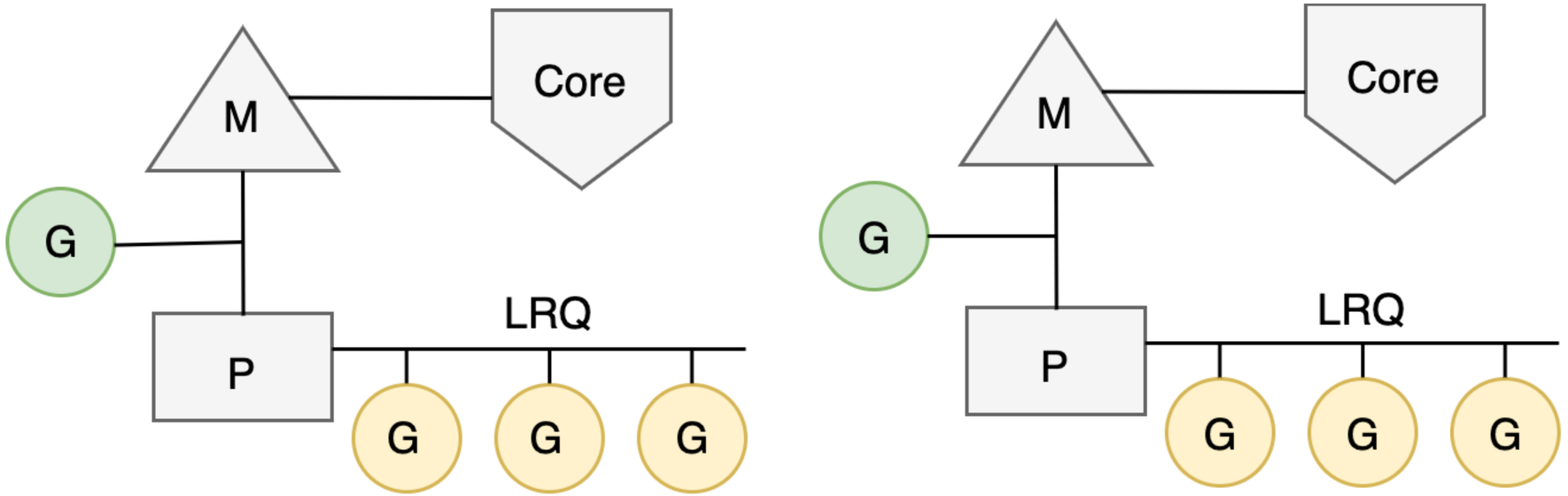


Планировщик

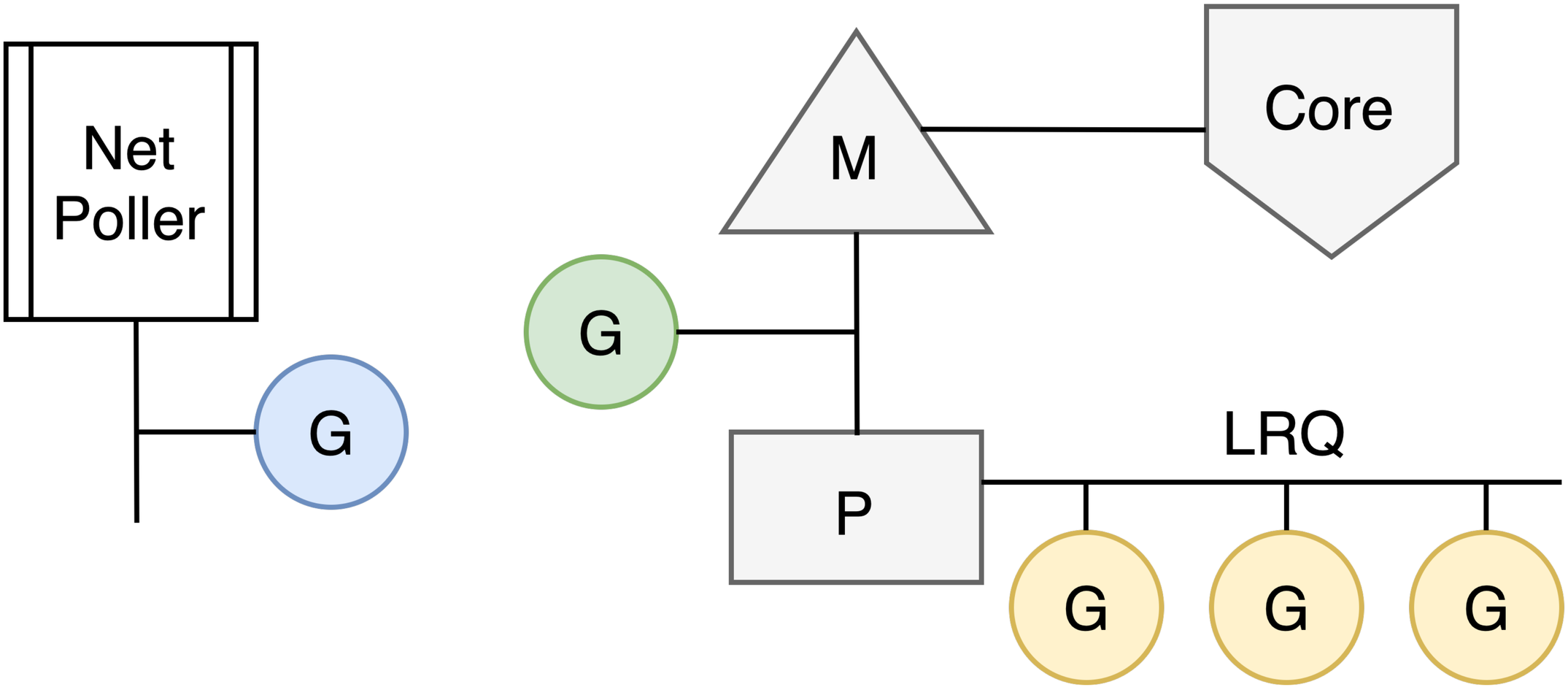




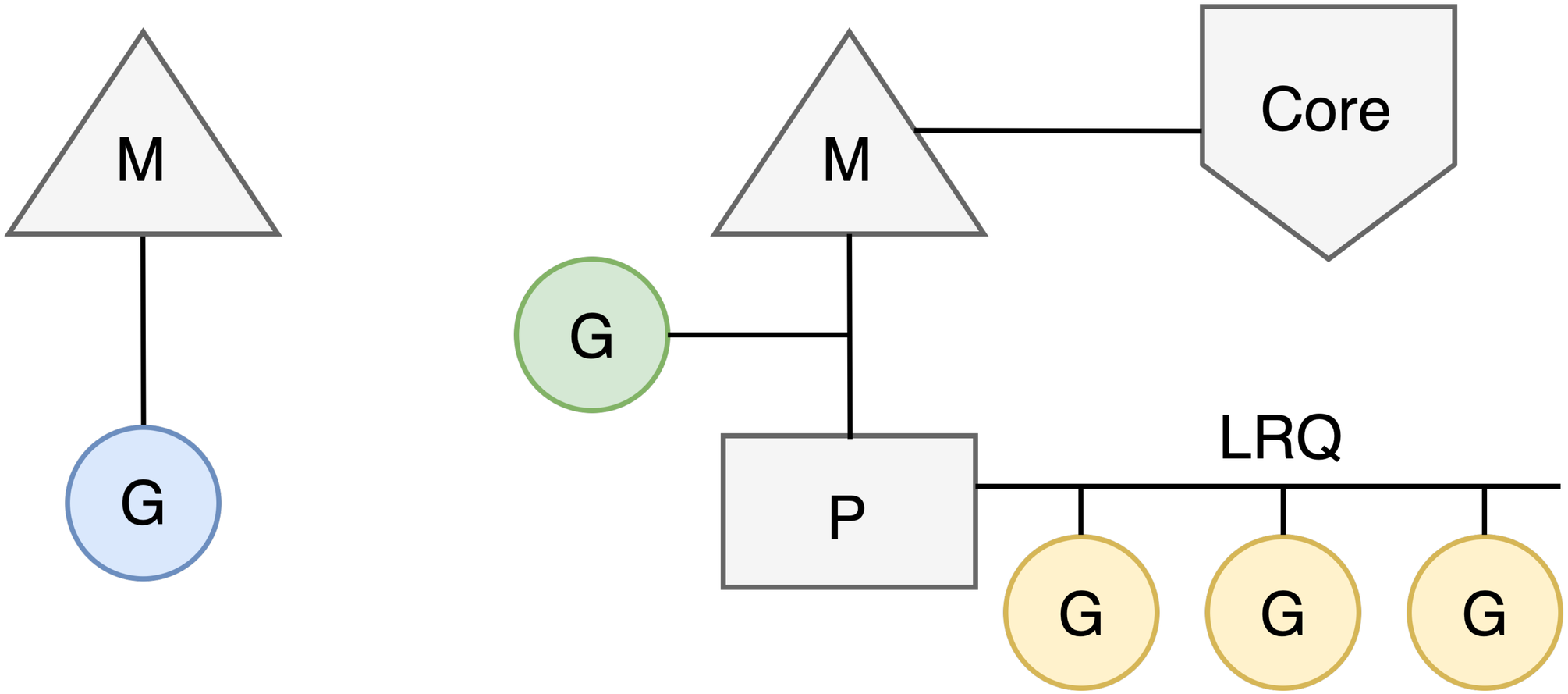
Планировщик



Асинхронный вызов



Синхронный (блокирующий) вызов (syscall)



Горутины
и примитивы синхронизации

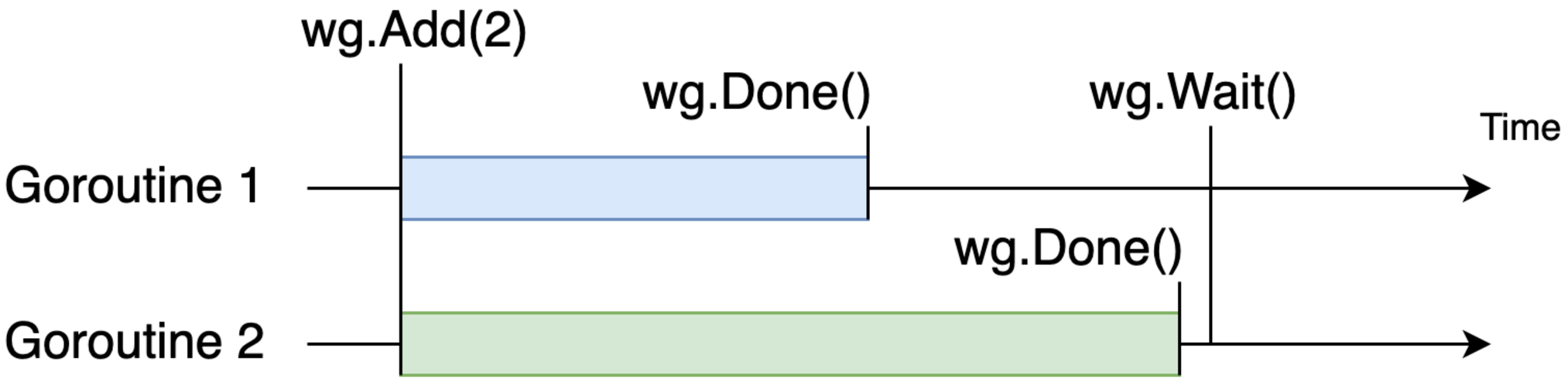
go func()

```
1  func main() {
2    go spinner(100 * time.Millisecond)
3    fibN := fib(n)
4    fmt.Printf("\rFibonacci(%d) = %d\n", n, fibN)
5  }
6  func spinner(delay time.Duration) {
7    for {
8      for _, r := range `-\|/` {
9        fmt.Printf("\r%c", r)
10       time.Sleep(delay)
11     }
12   }
13 }
14 func fib(x int) int {
15   if x < 2 {
16     return x
17   }
18   return fib(x-1) + fib(x-2)
19 }
```

WaitGroup

```
1  func main() {
2    runtime.GOMAXPROCS(1)
3    var wg sync.WaitGroup
4    wg.Add(2)
5    fmt.Println("Starting...")
6    go func() {
7      defer wg.Done()
8      for char := 'a'; char < 'a'+26; char++ {
9        //runtime.Gosched()
10       fmt.Printf("%c ", char)
11       time.Sleep(150 * time.Nanosecond)
12     }
13   }()
14   go func() {
15     defer wg.Done()
16     for char := 'A'; char < 'A'+26; char++ {
17       //runtime.Gosched()
18       fmt.Printf("%c ", char)
19       time.Sleep(150 * time.Nanosecond)
20     }
21   }()
22   wg.Wait()
23   fmt.Println("\nFinished")
24 }
```

WaitGroup



WaitGroup

```
type WaitGroup struct {
  noCopy noCopy

  state1 [3]uint32
}
```

Race

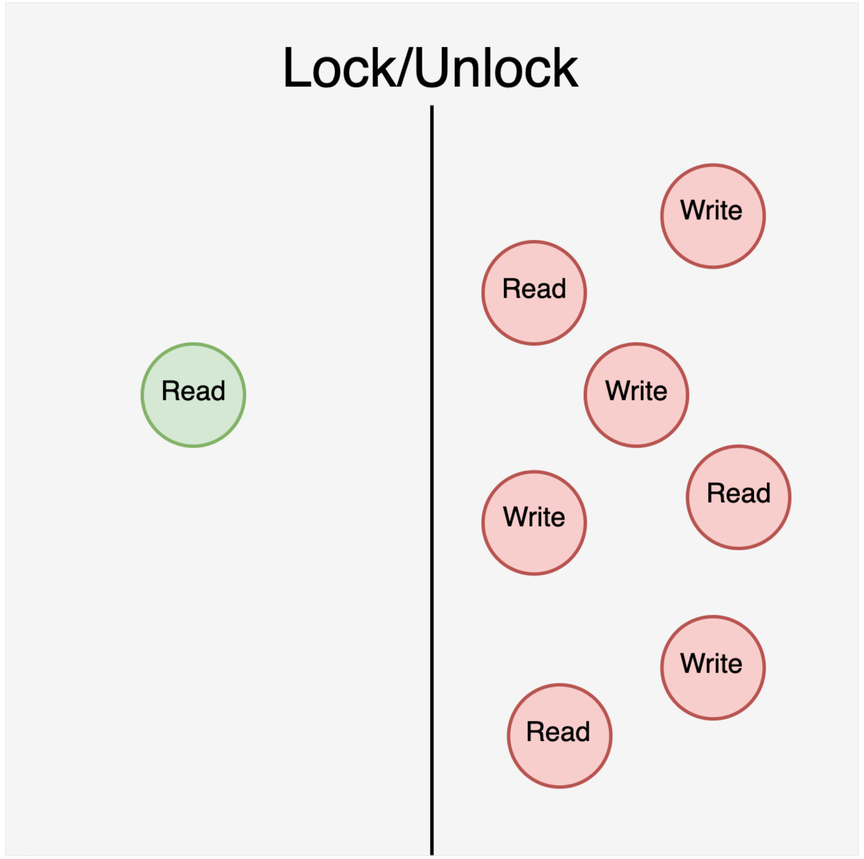
```go
var counter int
func main() {
  runtime.GOMAXPROCS(1)
  var wg sync.WaitGroup
  wg.Add(2)
  go incCounter(&wg) //routine #1
  go incCounter(&wg) //routine #2
  wg.Wait()
  fmt.Println("Final counter: ", counter)
}

func incCounter(wg *sync.WaitGroup) {
  defer wg.Done()
  for i := 0; i < 2; i++ {
    value := counter
    runtime.Gosched()
    value++
    counter = value
  }
}
```

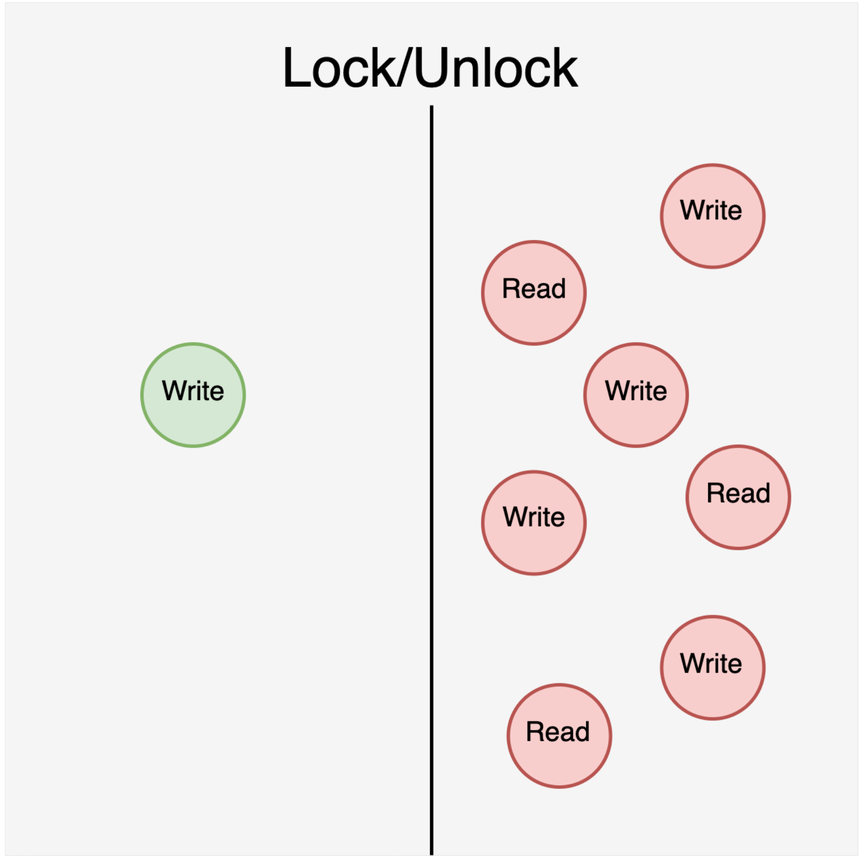
```
WARNING: DATA RACE
Read at 0x00000122fc90 by goroutine 8:
  main.incCounter()
      /Users/a.a.kozyrev/tcs/tfs-async/3_race/main.go:23 +0x77

Previous write at 0x00000122fc90 by goroutine 7:
  main.incCounter()
      /Users/a.a.kozyrev/tcs/tfs-async/3_race/main.go:26 +0x93

Goroutine 8 (running) created at:
  main.main()
      /Users/a.a.kozyrev/tcs/tfs-async/3_race/main.go:15 +0xd1

Goroutine 7 (finished) created at:
  main.main()
      /Users/a.a.kozyrev/tcs/tfs-async/3_race/main.go:14 +0xaf
==================
Final counter:  4
Found 1 data race(s)
```

Mutex

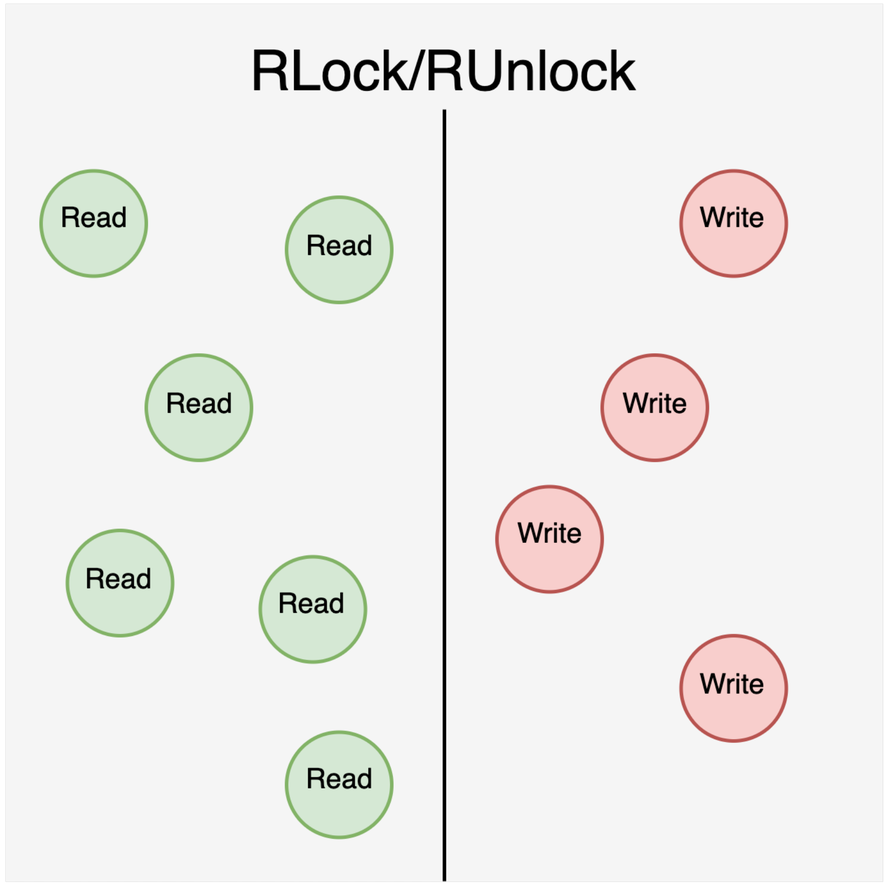
```
1  var counter int
2  var mu sync.Mutex
3  func main() {
4   runtime.GOMAXPROCS(1)
5   var wg sync.WaitGroup
6   wg.Add(2)
7   go incCounter(&wg) //routine #1
8   go incCounter(&wg) //routine #2
9   wg.Wait()
10
11  mu.Lock()
12  fmt.Println("Final counter: ", counter)
13  mu.Unlock()
14  }
15
16 func incCounter(wg *sync.WaitGroup) {
17  defer wg.Done()
18  for i := 0; i < 2; i++ {
19    mu.Lock()
20    value := counter
21    runtime.Gosched()
22    value++
23    counter = value
24    mu.Unlock()
25  }
26 }
```

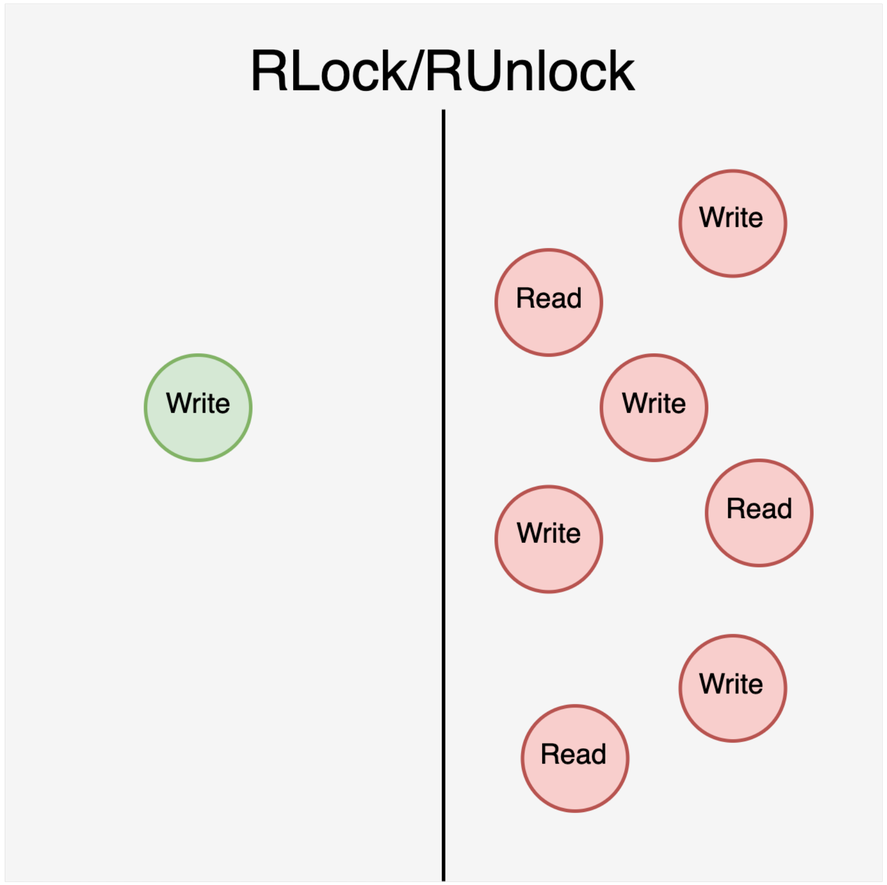
Mutex


Lock/Unlock
Read
Write
Read
Write
Write
Read
Write
Read


Lock/Unlock
Write
Write
Read
Write
Read
Write
Write
Read

RWMutex


RLock/RUnlock
Read
Read
Read
Read
Read
Read
Write
Write
Write
Write


RLock/RUnlock
Write
Write
Read
Write
Write
Read
Write
Write
Read

Atomic

```
1  var counter int64
2  func main() {
3    runtime.GOMAXPROCS(1)
4    var wg sync.WaitGroup
5    wg.Add(2)
6    go incCounter(&wg) //routine #1
7    go incCounter(&wg) //routine #2
8    wg.Wait()
9
10   fmt.Println("Final counter: ", counter)
11 }
12
13 func incCounter(wg *sync.WaitGroup) {
14   defer wg.Done()
15   for i := 0; i < 2; i++ {
16     atomic.AddInt64(&counter, 1)
17     runtime.Gosched()
18   }
19 }
```

x/sync

errgroup	Package errgroup provides synchronization, error propagation, and Context cancelation for groups of goroutines working on subtasks of a common task.
semaphore	Package semaphore provides a weighted semaphore implementation.
singleflight	Package singleflight provides a duplicate function call suppression mechanism.
syncmap	Package syncmap provides a concurrent map implementation.

Channel

hchan

```
type hchan struct {
 qcount   uint           // total data in the queue
 dataqsiz uint           // size of the circular queue
 buf      unsafe.Pointer // points to an array of dataqsiz elements
 elemsize uint16
 closed   uint32
 elemtype *_type // element type
 sendx    uint   // send index
 recvx    uint   // receive index
 recvq    waitq  // list of recv waiters
 sendq    waitq  // list of send waiters

 lock mutex
}

type waitq struct {
 first *sudog
 last  *sudog
}

type sudog struct {
 g *g

 next *sudog
 prev *sudog
 elem unsafe.Pointer // data element (may point to stack)
 // ...
}
```

unbuffered

```
1  func main() {
2    unbuffered := make(chan string)
3    wg := sync.WaitGroup{}
4    wg.Add(1)
5    go func() {
6      defer wg.Done()
7      for {
8        v, ok := <- unbuffered
9        if !ok {
10         fmt.Println("stop reader")
11         return
12       }
13       fmt.Println(v)
14     }
15   }()
16   wg.Add(1)
17   go func() {
18     defer wg.Done()
19     for i := 0; i<= 9; i++ {
20       unbuffered <- fmt.Sprintf("Hello #%d", i)
21     }
22     close(unbuffered)
23   }()
24   wg.Wait()
25 }
```

buffered

```
1  func main() {
2   buffered := make(chan string, 10)
3   wg := sync.WaitGroup{}
4   wg.Add(1)
5   go func() {
6      defer wg.Done()
7      for i := 0; i<= 9; i++ {
8         fmt.Println("write to channel")
9         buffered <- fmt.Sprintf("Hello #%d", i)
10     }
11     close(buffered)
12     fmt.Println("close channel")
13  }()
14  time.Sleep(time.Second * 2)
15  wg.Add(1)
16  go func() {
17     defer wg.Done()
18     for {
19        v, ok := <- buffered
20        if !ok {
21           fmt.Println("stop reader")
22           return
23        }
24
25        fmt.Println(v)
26     }
27  }()
28  wg.Wait()
29 }
```

for range

```
1  func main() {
2    unbuffered := make(chan string)
3    wg := sync.WaitGroup{}
4    wg.Add(1)
5    go func() {
6      defer wg.Done()
7      for v := range unbuffered {
8        fmt.Println(v)
9      }
10     fmt.Println("stop reader")
11   }()
12   wg.Add(1)
13   go func() {
14     defer wg.Done()
15     for i := 0; i<= 9; i++ {
16       unbuffered <- fmt.Sprintf("Hello #%d", i)
17     }
18     close(unbuffered)
19   }()
20   wg.Wait()
21 }
```

channel direction

```
1  func main() {
2    wg := sync.WaitGroup{}
3    ch := func(wg *sync.WaitGroup) <-chan string {
4      out := make(chan string)
5      wg.Add(1)
6      go func() {
7        defer wg.Done()
8        for i := 0; i<= 9; i++ {
9          out <- fmt.Sprintf("Hello #%d", i)
10       }
11       close(out)
12     }()
13     return out
14   }(&wg)
15   wg.Add(1)
16   go func(in <- chan string) {
17     defer wg.Done()
18     for v := range in {
19       fmt.Println(v)
20     }
21     fmt.Println("stop reader")
22   }(ch)
23   wg.Wait()
24 }
```

context/select

```
1  func main() {
2    ctx, cancel := context.WithCancel(context.Background())
3
4    go func() {
5      ticker := time.NewTicker(time.Second)
6      for {
7        select {
8        case <-ctx.Done():
9          fmt.Println("ctx done")
10         return
11       case <-ticker.C:
12         fmt.Println(time.Now().Format(time.RFC1123))
13       }
14     }
15   }()
16
17   time.Sleep(time.Second * 10)
18   cancel()
19 }
```

context

```
type Context interface {
  // Deadline returns the time when work done on behalf of this context
  // should be canceled. Deadline returns ok==false when no deadline is
  // set. Successive calls to Deadline return the same results.
  Deadline() (deadline time.Time, ok bool)

  // Done returns a channel that's closed when work done on behalf of this
  // context should be canceled. Done may return nil if this context can
  // never be canceled. Successive calls to Done return the same value.
  // The close of the Done channel may happen asynchronously,
  // after the cancel function returns.
  Done() <-chan struct{}

  // If Done is not yet closed, Err returns nil.
  // If Done is closed, Err returns a non-nil error explaining why:
  // After Err returns a non-nil error, successive calls to Err return the same error.
  Err() error

  // Value returns the value associated with this context for key, or nil
  // if no value is associated with key. Successive calls to Value with
  // the same key returns the same result.
  Value(key interface{}) interface{}
}
```

Pipelines

pipelines

```
1  func gen(nums ...int) <-chan int {
2    out := make(chan int)
3    go func() {
4      for _, n := range nums {
5        out <- n
6      }
7      close(out)
8    }()
9    return out
10 }
11
12 func sq(in <-chan int) <-chan int {
13   out := make(chan int)
14   go func() {
15     for n := range in {
16       out <- n * n
17     }
18     close(out)
19   }()
20   return out
21 }
```

```
1 func main() {
2   c := gen(2, 3)
3   out := sq(c)
4
5   fmt.Println(<-out) // 4
6   fmt.Println(<-out) // 9
7 }
```