Easing concurrency

with the sync package

$ go env

- 15+ years
- Staff Engineer @ PicPay
- Writer @ mfbmina.dev

go concurrency()

- concurrent x parallel
- goroutines
- maxprocs

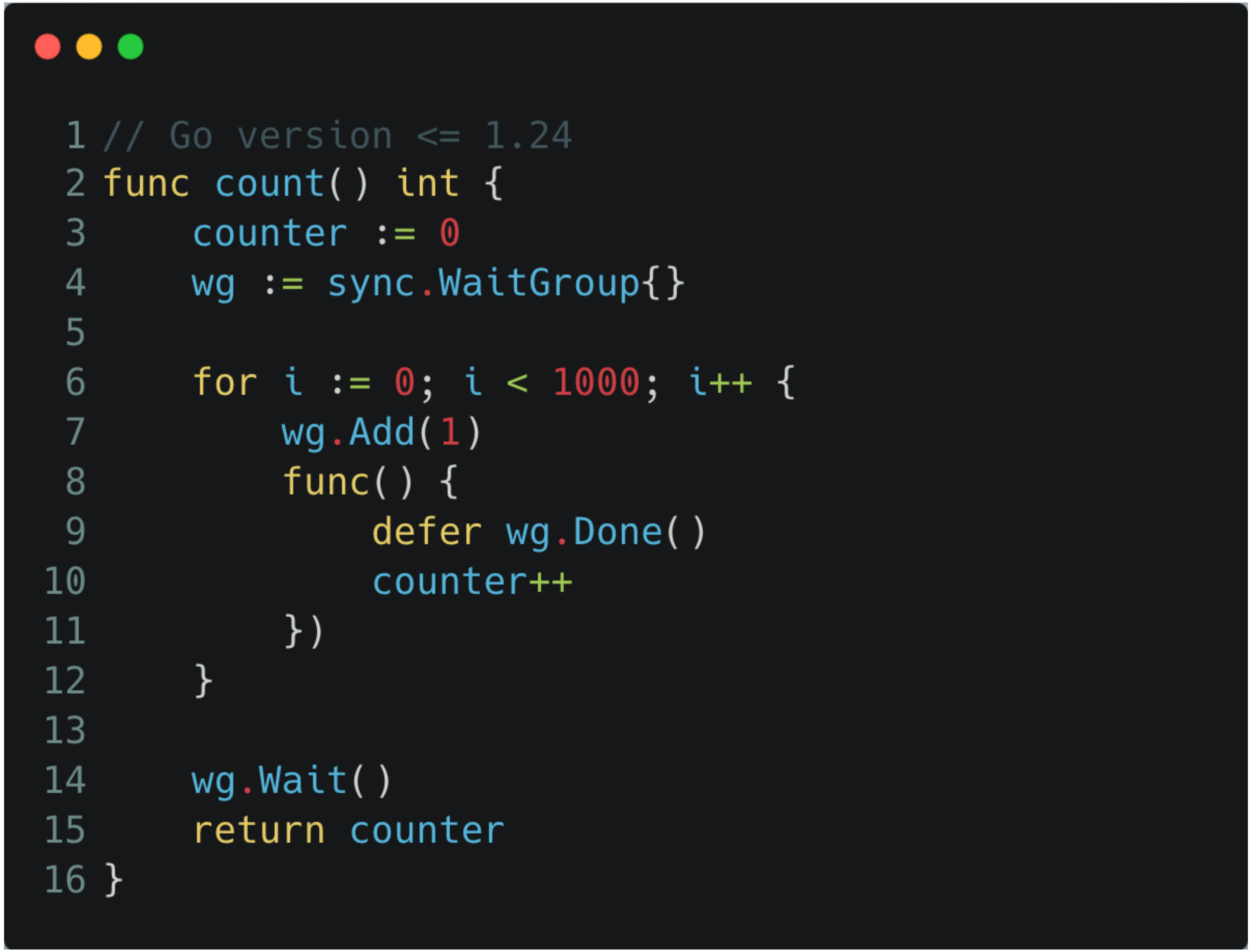
machine
processors
goroutines
M
M
P
P
P
G
G
G
G
G

import sync

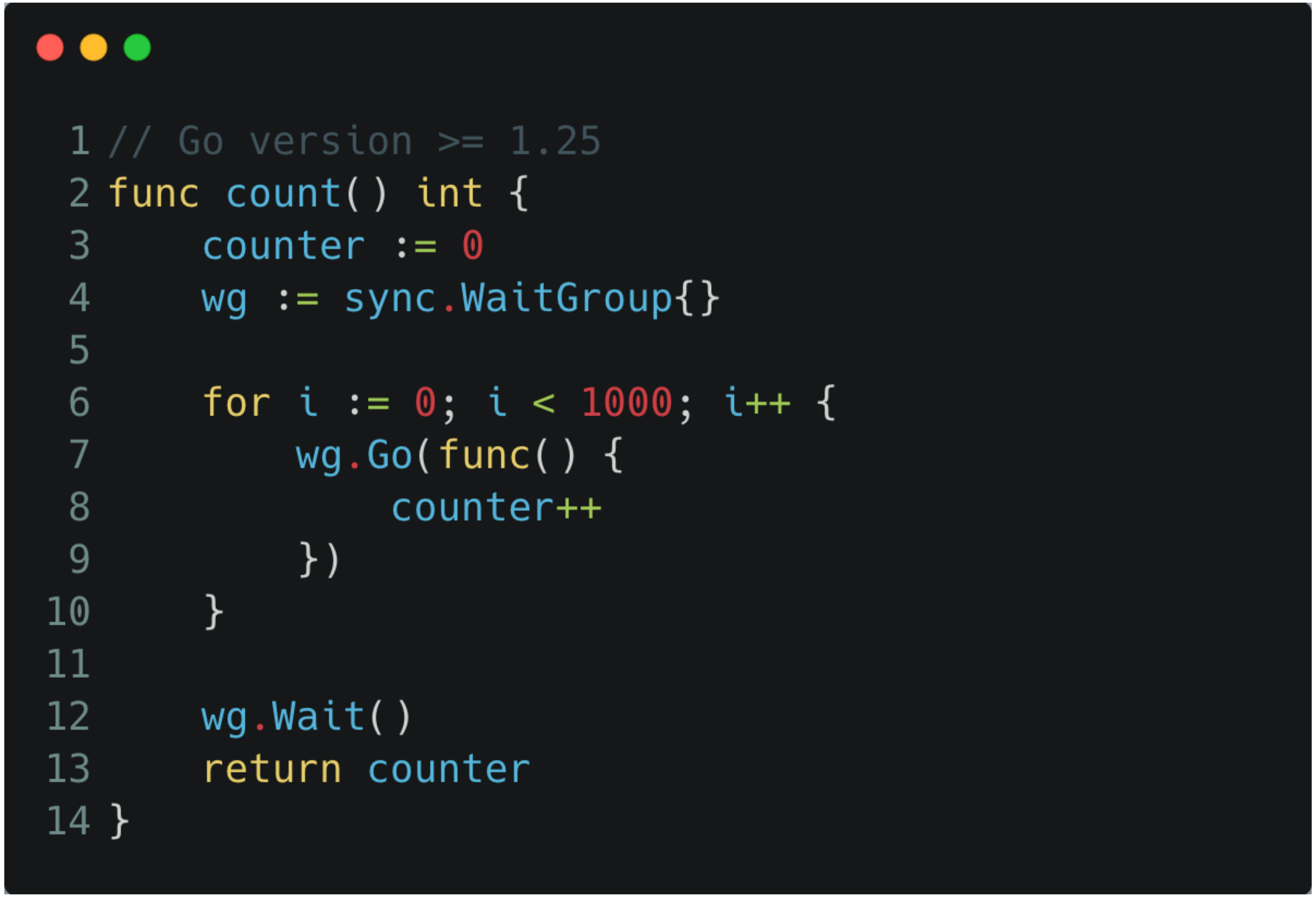
- waitgroups
- mutex & rw mutex
- atomic
- map
- once
- cond
- pool

wg := sync.Waitgroup{}

- Counter
- 1.24 <= Add( ), Done( ), Wait( )
- 1.25 >= Go( ), Wait( )



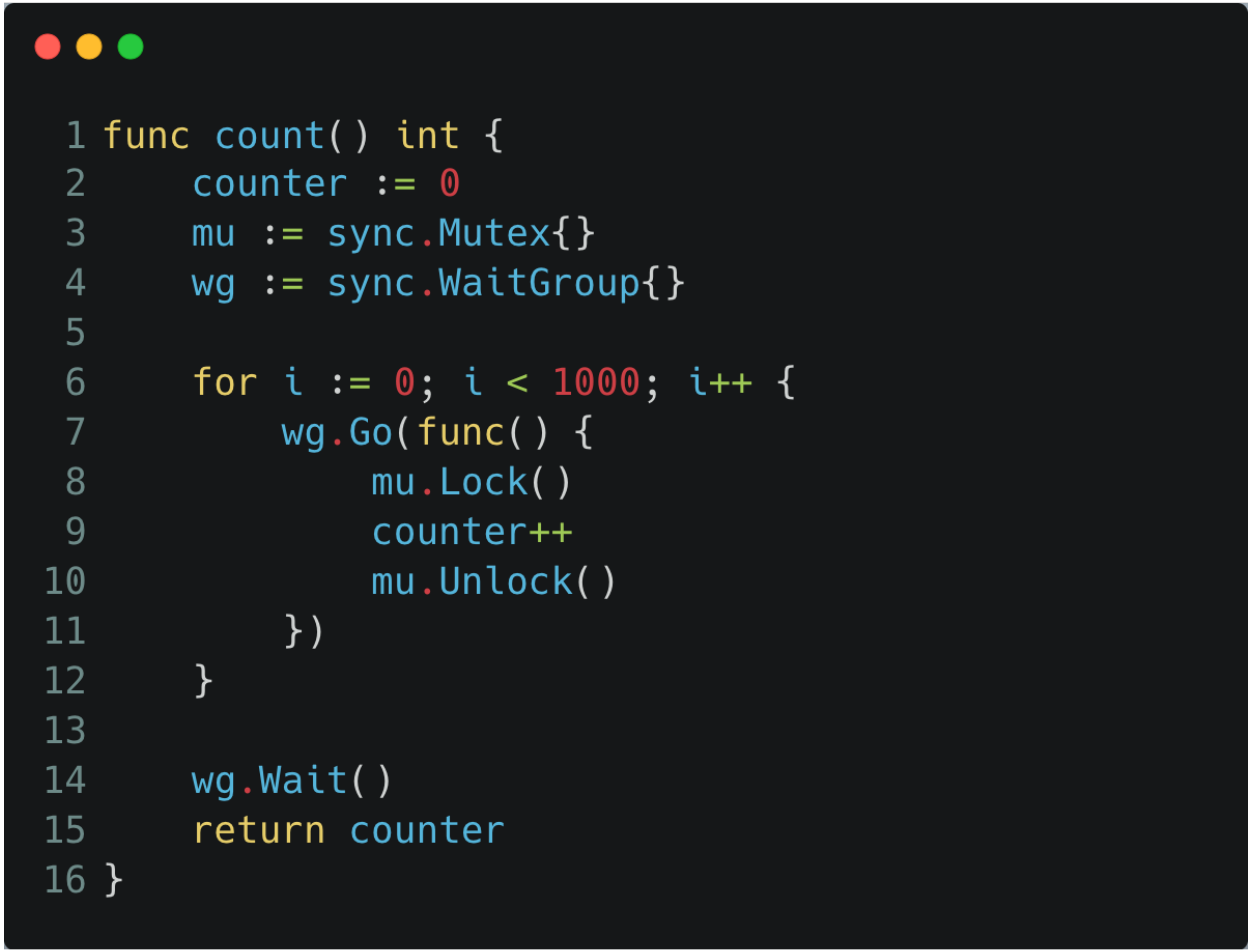
```go
// Go version <= 1.24
func count() int {
	counter := 0
	wg := sync.WaitGroup{}

	for i := 0; i < 1000; i++ {
		wg.Add(1)
		func() {
			defer wg.Done()
			counter++
		})
	}

	wg.Wait()
	return counter
}
```



```go
// Go version >= 1.25
func count() int {
    counter := 0
    wg := sync.WaitGroup{}

    for i := 0; i < 1000; i++ {
        wg.Go(func() {
            counter++
        })
    }

    wg.Wait()
    return counter
}
```

m := sync.Mutex{}

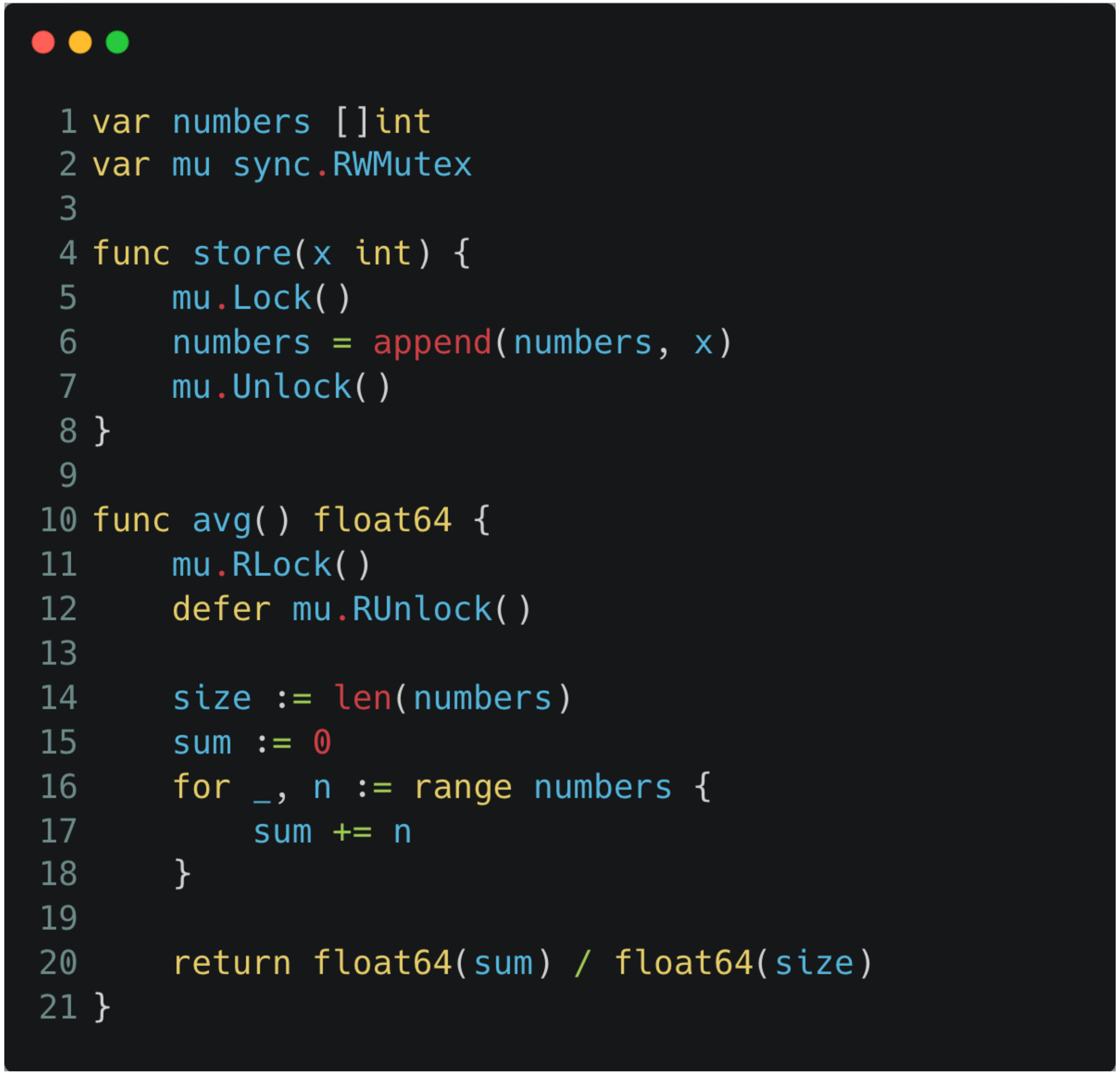
- Lock( )
- Unlock( )
- ⚠️ Deadlock ⚠️



```go
func count() int {
	counter := 0
	mu := sync.Mutex{}
	wg := sync.WaitGroup{}

	for i := 0; i < 1000; i++ {
		wg.Go(func() {
			mu.Lock()
			counter++
			mu.Unlock()
		})
	}

	wg.Wait()
	return counter
}
```

m := sync.RWMutex{}

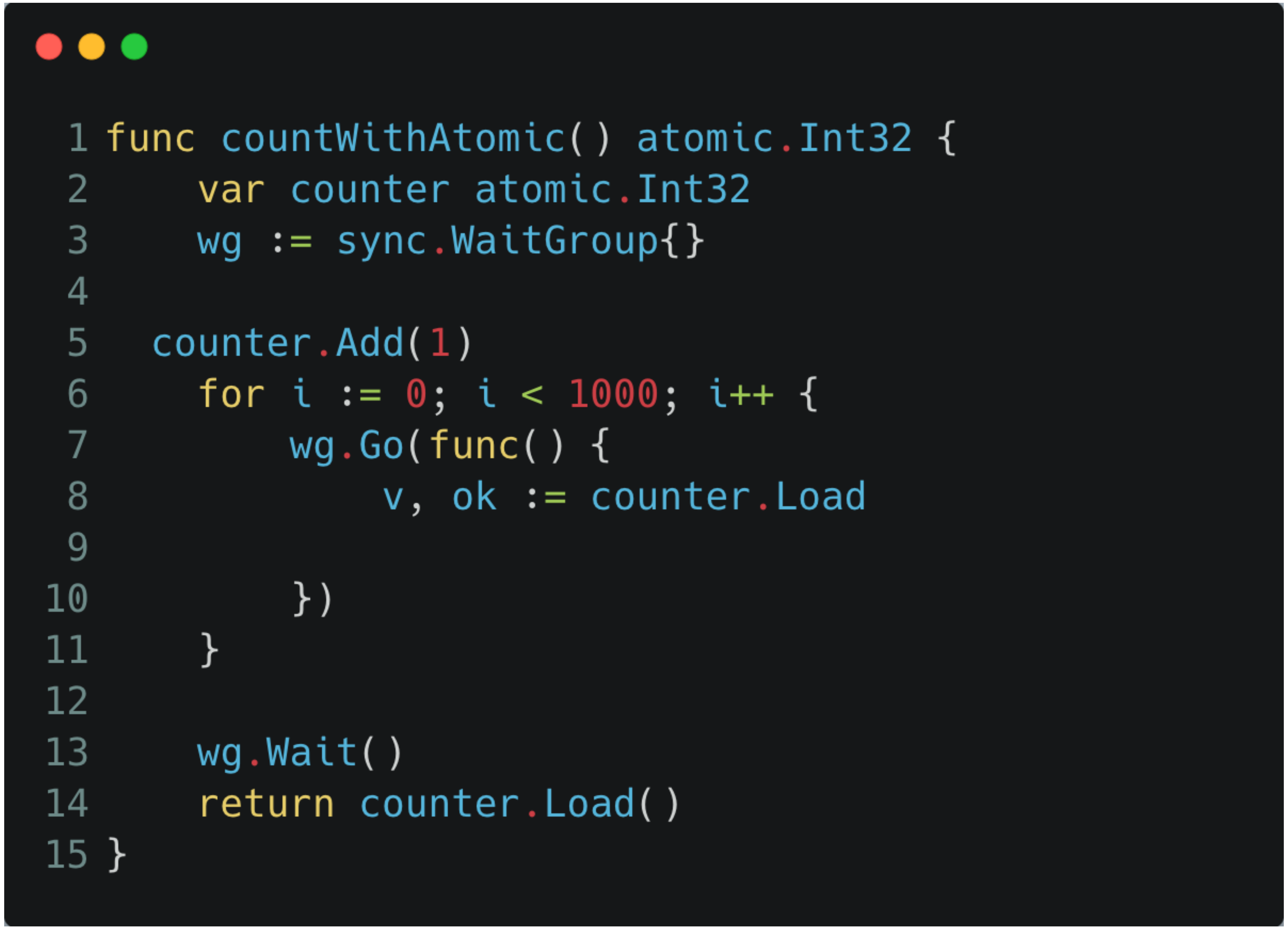
- Lock( ) & RLock( )
- Unlock( ) & RUnlock( )
- ⚠️ Deadlock & Starvation ⚠️



```go
var numbers []int
var mu sync.RWMutex

func store(x int) {
	mu.Lock()
	numbers = append(numbers, x)
	mu.Unlock()
}

func avg() float64 {
	mu.RLock()
	defer mu.RUnlock()

	size := len(numbers)
	sum := 0
	for _, n := range numbers {
		sum += n
	}

	return float64(sum) / float64(size)
}
```

import sync/atomic

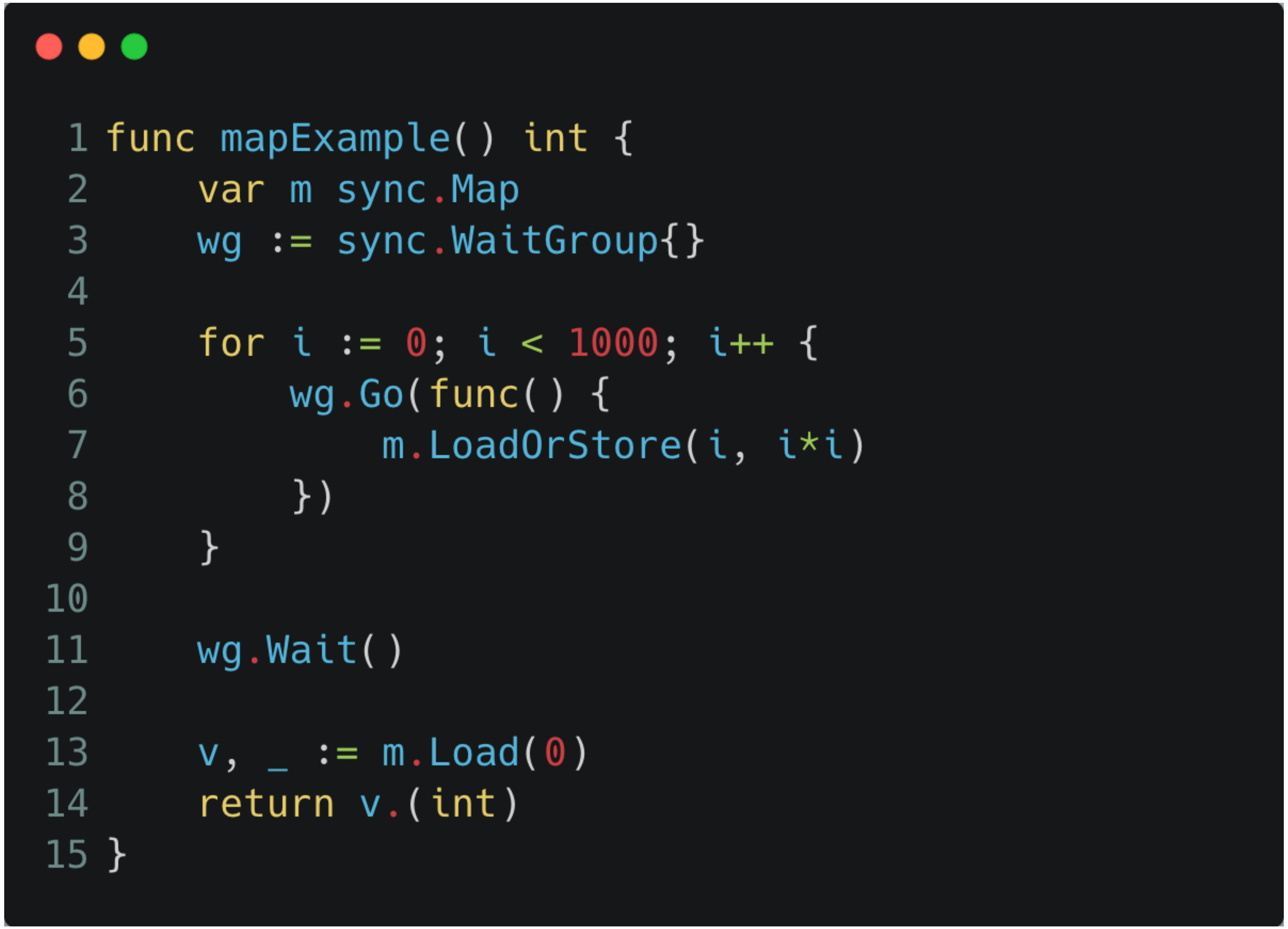
- Concurrent-safe types
- bool, int32, int64, pointer, uint32, uint64, uintpointer & value



```go
func countWithAtomic() atomic.Int32 {
    var counter atomic.Int32
    wg := sync.WaitGroup{}

  counter.Add(1)
    for i := 0; i < 1000; i++ {
        wg.Go(func() {
            v, ok := counter.Load

        })
    }

    wg.Wait()
    return counter.Load()
}
```

m := sync.Map{}

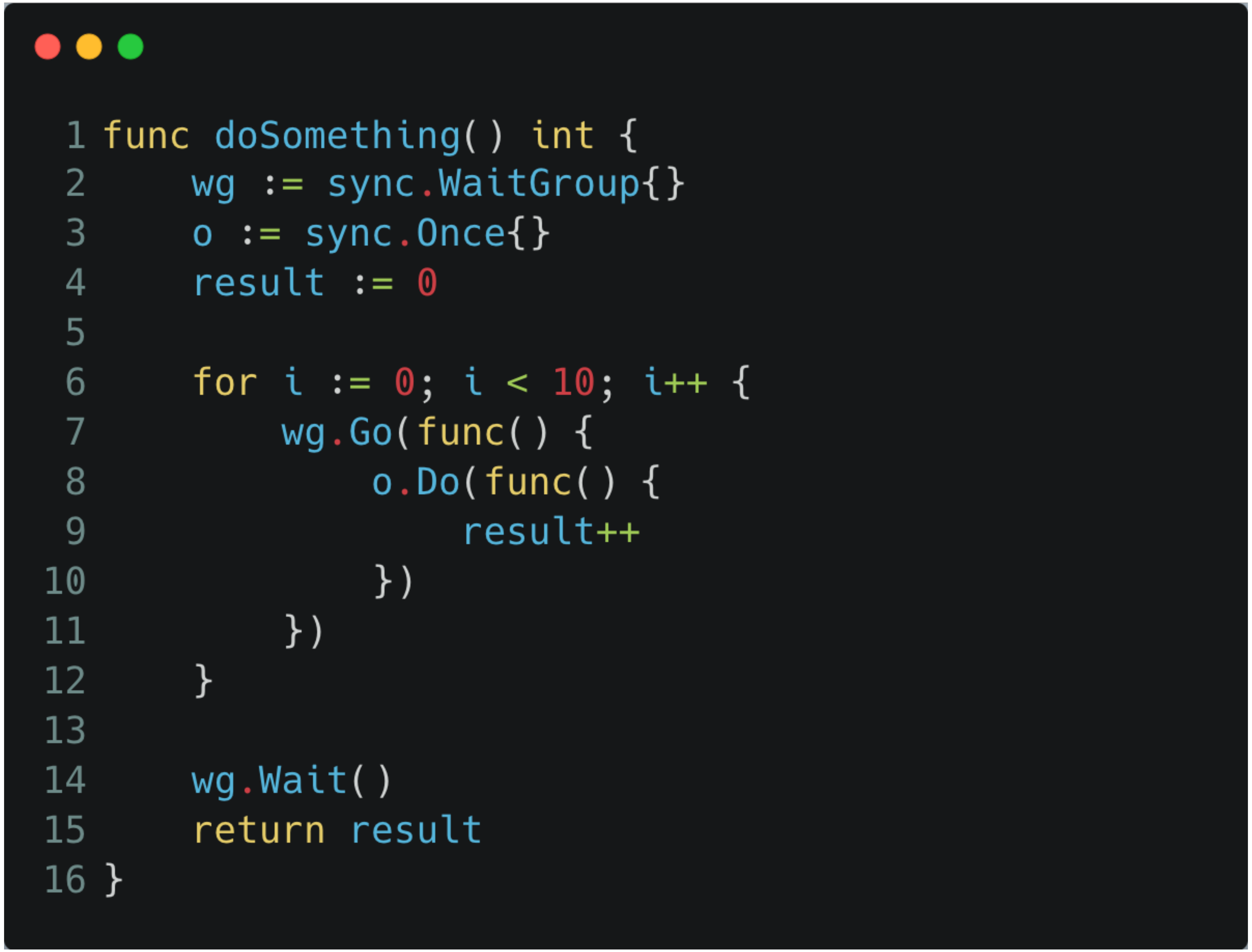
- Concurrent-safe map
- 1 write : N reads
- Goroutines read and write in distinct keys



```go
func mapExample() int {
    var m sync.Map
    wg := sync.WaitGroup{}

    for i := 0; i < 1000; i++ {
        wg.Go(func() {
            m.LoadOrStore(i, i*i)
        })
    }

    wg.Wait()

    v, _ := m.Load(0)
    return v.(int)
}
```

o := sync.Once{}

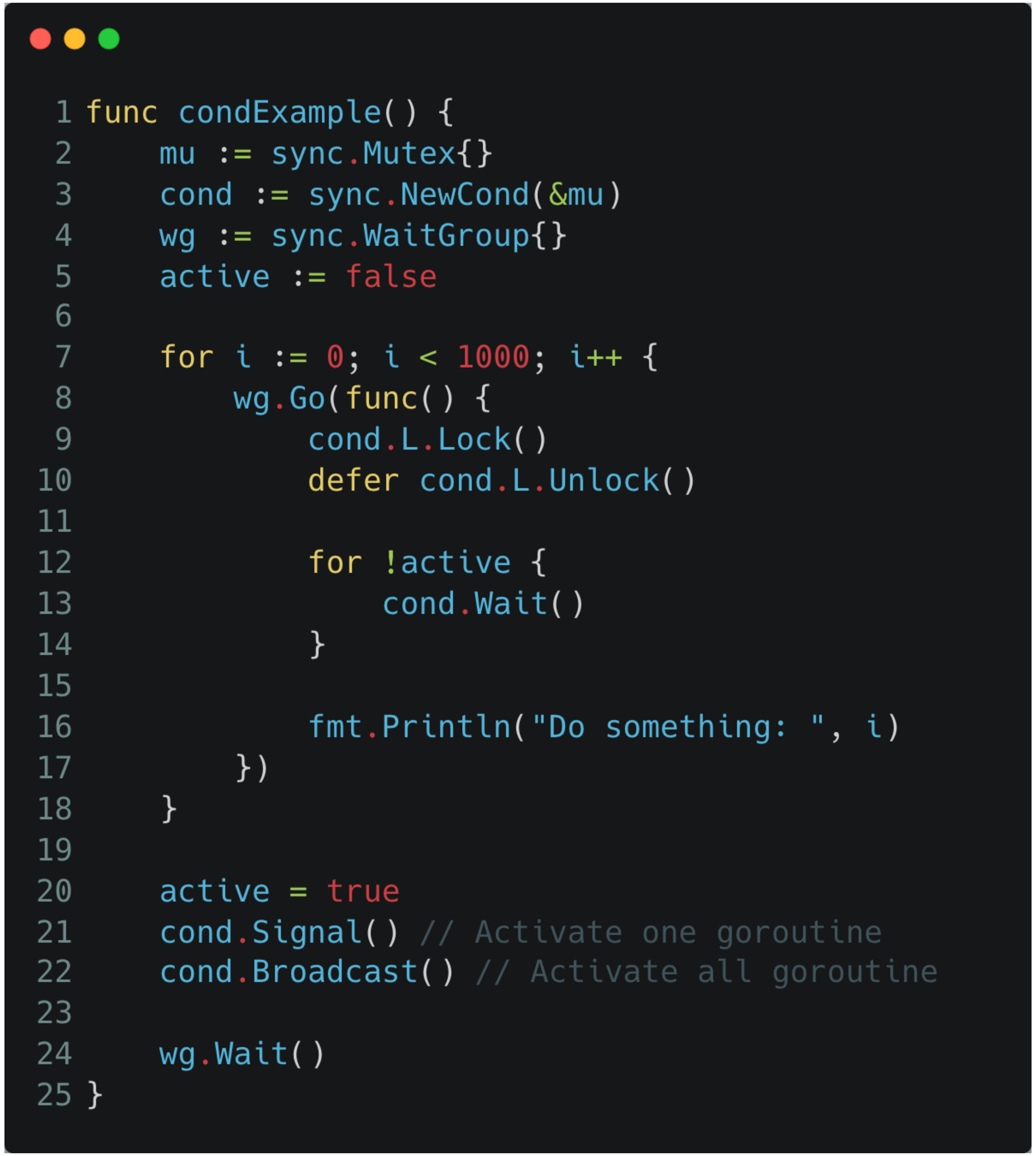
- Avoid doing something multiple times
- If it panics, it will not be retried



```go
func doSomething() int {
    wg := sync.WaitGroup{}
    o := sync.Once{}
    result := 0

    for i := 0; i < 10; i++ {
        wg.Go(func() {
            o.Do(func() {
                result++
            })
        })
    }

    wg.Wait()
    return result
}
```

c := sync.Cond{}

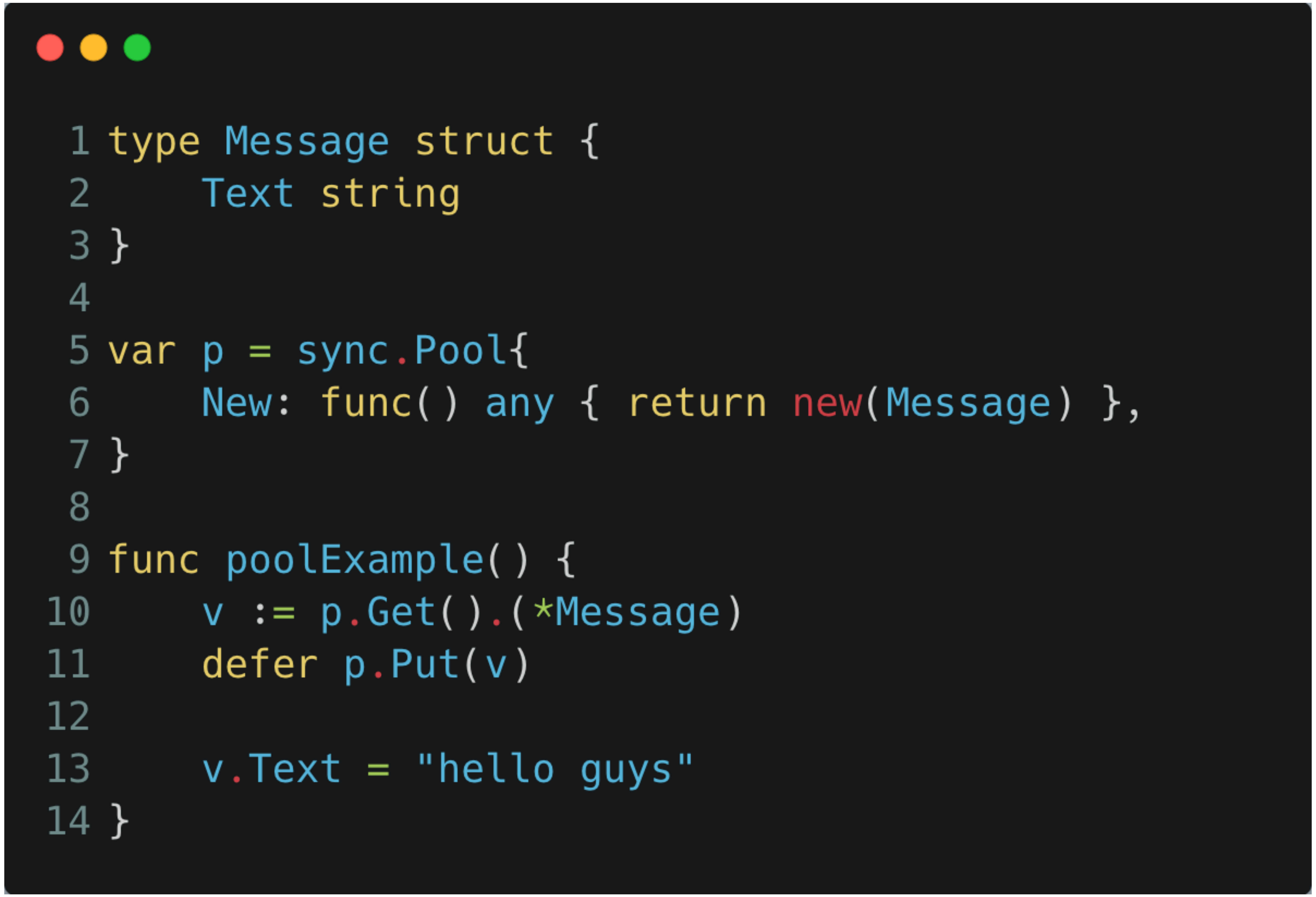
- If something happens, allow goroutine work
- Signal( )
- Broadcast( )
- Wait( )



```go
func condExample() {
    mu := sync.Mutex{}
    cond := sync.NewCond(&mu)
    wg := sync.WaitGroup{}
    active := false

    for i := 0; i < 1000; i++ {
        wg.Go(func() {
            cond.L.Lock()
            defer cond.L.Unlock()

            for !active {
                cond.Wait()
            }

            fmt.Println("Do something: ", i)
        })
    }

    active = true
    cond.Signal() // Activate one goroutine
    cond.Broadcast() // Activate all goroutine

    wg.Wait()
}
```

p := sync.Pool{}

- Short lived objects on memory
- Relieves pressure on GC



```go
type Message struct {
    Text string
}

var p = sync.Pool{
    New: func() any { return new(Message) },
}

func poolExample() {
    v := p.Get().(*Message)
    defer p.Put(v)

    v.Text = "hello guys"
}
```

wg.Done( )