Golang #2

Concurrency

Oliver N.

Software Engineer

Why Go?

1. Single binary deployment
2. Minimal language
3. Easy concurrency
4. Full development environment
5. Multi-arch build
6. Low-level interface
7. Getting started quickly

Why Go?

1. Single binary deployment
2. Minimal language
3. Easy concurrency
4. Full development environment
5. Multi-arch build
6. Low-level interface
7. Getting started quickly

JavaScript

```
function *fibo() {
  let a = 0, b = 1;
  while (true) {
    yield a;
    [a, b] = [b, a+b];
  }
}
```

```
for (let value of fibo()) {
  console.log(value);
};
```

JavaScript

```
function *fibo() {
  let a = 0, b = 1;
  while (true) {
    yield a;
    [a, b] = [b, a+b];
  }
}
```

```
for (let value of fibo()) {
  console.log(value);
};
```

Single thread !

Golang

Run

```
func fibo(ch chan<- int) {
  a, b := 0, 1
  for {
    ch <- a
    a, b = b, a + b
  }
}
```

```
func main() {
  ch := make(chan int)
  go fibo(ch)
  for i:= 0; i < 10; i++ {
    fi := <- ch
    println(fi)
  }
}
```

Golang

Run

```
func fibo(ch chan<- int) {
  a, b := 0, 1
  for {
    ch <- a
    a, b = b, a + b
  }
}
```

```
func main() {
  ch := make(chan int)
  go fibo(ch)
  for i:= 0; i < 10; i++ {
    fi := <- ch
    println(fi)
```

```
$ export GOMAXPROCS=4
```

Golang #2

Concurrency

Concurrency in Go

1. What is concurrency?
2. Communicating with channel & select
3. Atomic accessing with mutex.Lock()
4. Detecting race condition with go race
5. Examples

What is concurrency?

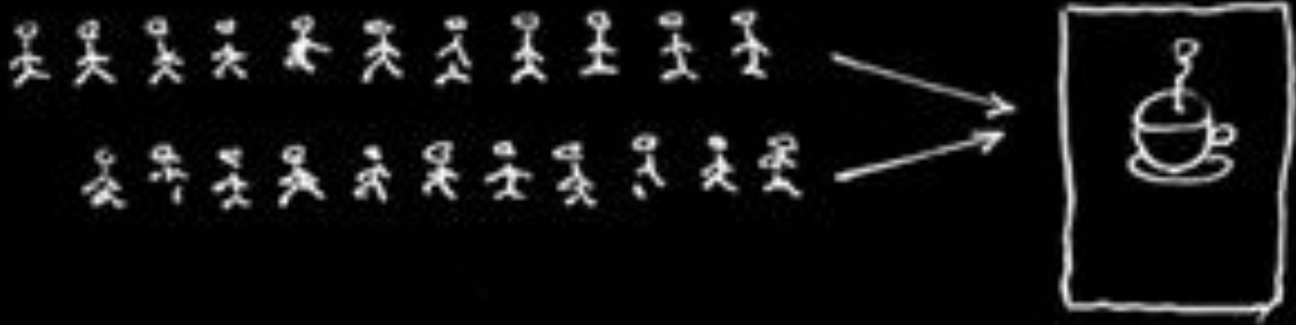
What is concurrency?

Concurrency is the composition of independently executing computations.

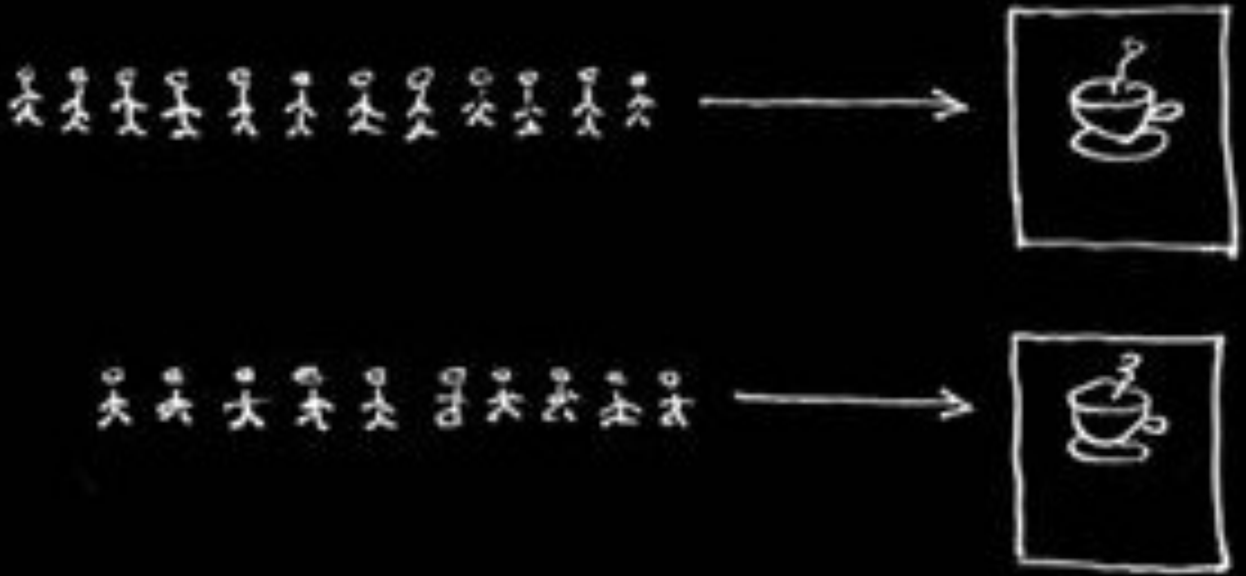
Concurrency is a way to structure software, particularly as a way to write clean code that interacts well with the real world.

It is not parallelism.

Concurrent = Two Queues One Coffee Machine

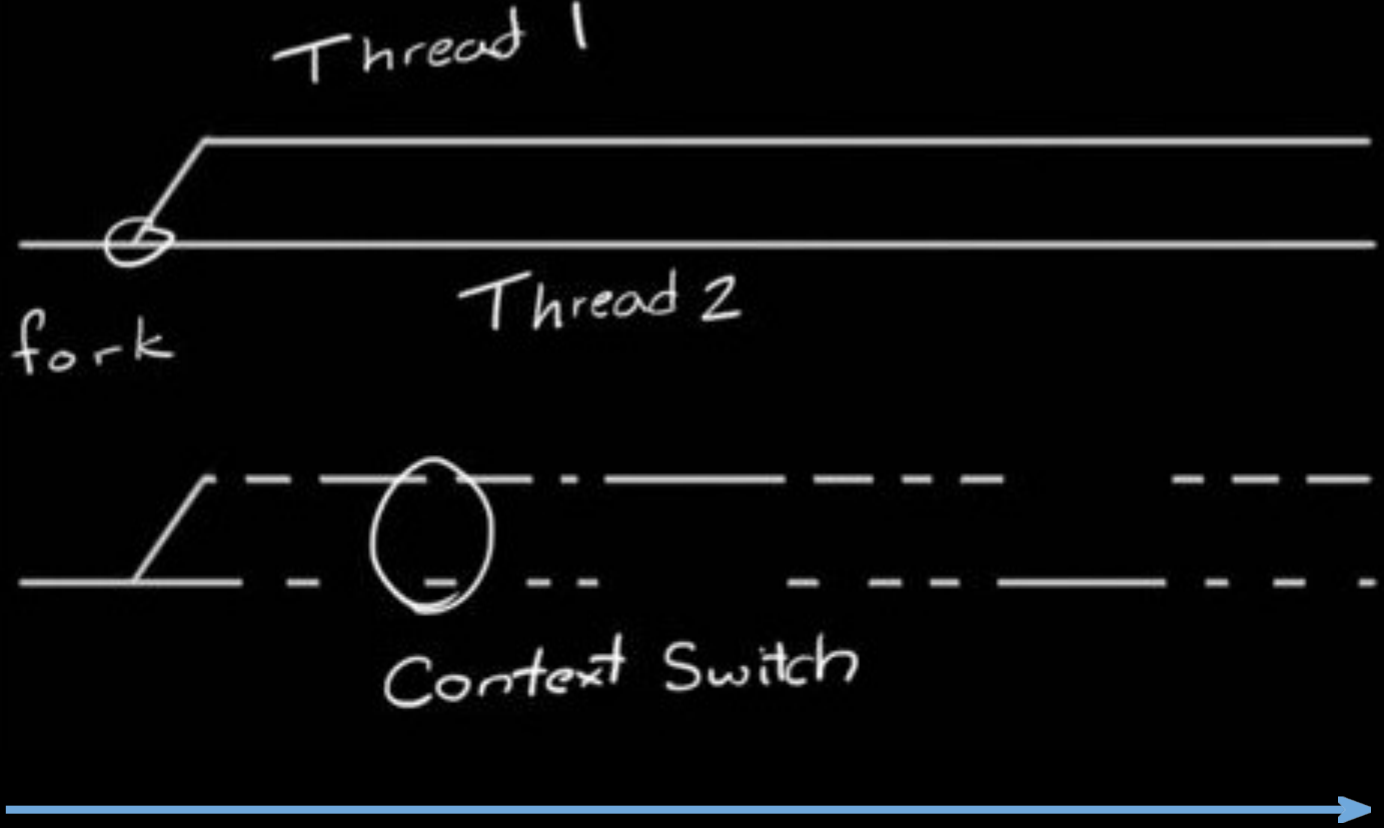


Parallel = Two Queues Two Coffee Machines

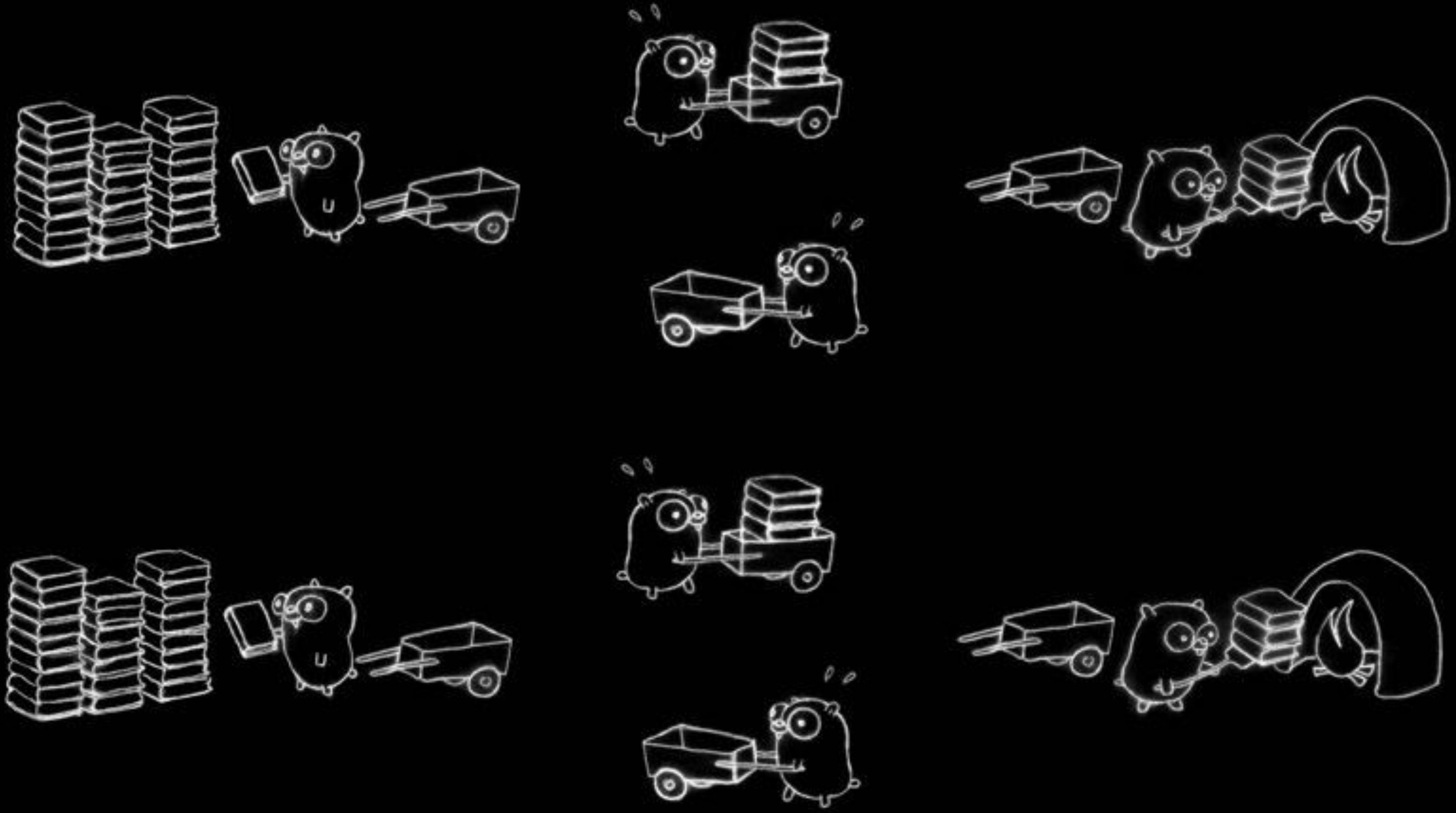


Parallelism

Concurrency
(without parallelism)



Time



Goroutine

Goroutine

It's an independently executing function, launched by a go statement.

It has its own call stack, which grows and shrinks as required.

It's very cheap. It's practical to have thousands, even hundreds of thousands of goroutines.

It's not a thread.

Example: Hello

Run

```
func hello(name string) {
  for {
    fmt.Println(“Hello from”, name)
    time.Sleep(200 * time.Millisecond)
  }
}

func main() {
  go hello(“Alice”)
  go hello(“Bob”)
  time.Sleep(1000 * time.Millisecond)
}
```

Communication

channel & select

Channel

`c := make(chan int)`	– Makes an unbuffered channel of ints
`c <- x`	– Sends a value on the channel
`<- c`	– Waits to receive a value on the channel
`x = <- c`	– Waits to receive a value and stores it in x
`x, ok = <- c`	– ok will be false if channel is closed
`close(c)`	– Mark a channel not available to use

Example: Hello channel

Run

```
func hello(name string) chan string {
  c := make(chan string)
  go func() {
    for {
      c <- “Hello from “ + name
      time.Sleep(100 * time.Millisecond)
    }
  }()
  return c
}
```

```
func main() {
  a := hello(“Alice”)
  b := hello(“Bob”)
  for i := 0; i < 5; i++ {
     fmt.Println(<-a)
     fmt.Println(<-b)
  }
}
```

Example: Multi-flexing

Run

```
func fanIn(c1, c2 <-chan string) <-chan string
  { c := make(chan string)
  go func(){ for { c <- <-c1 } }()
  go func(){ for { c <- <-c2 } }()
  return c
}

func main() {
  c := fanIn(hello(“Alice”),
  hello(“Bob”)) for i:= 0; i < 10; i++ {
    fmt.Println(<-c)
  }
}
```

Example: Fibonacci

Run

```
func fibo(ch chan<- int) {
  a, b := 0, 1
  for true {
    ch <- a
    a, b = b, a + b
  }
}
```

```
func main() {
  ch := make(chan int)
  go fibo(ch)
  for i:= 0; i < 10; i++ {
    fi := <- ch
    println(fi)
  }
}
```

Example: Unique ID service

Run

```
func startIdService() chan int {
  c := make(chan int)
  counter := 0
  go func() {
    for {
      counter++
      c <- counter
    }
  }()
  return c
}
```

```
func main() {
  c := startIdService()
  id1 := <- c
  id2 := <- c
  fmt.Println(id1, id2)
}
```

Select

```
select {    // Try executing each statement until one is available
  case <- c1:        // Try reading from c1
  case x := <- c2    // Try reading from c2 to x
  case c3 <- value      // Try sending to c3
  default:           // Run if no other statement available
}
```

Example: Hello channel & select

Run

```
func hello(name string) chan string {
  c := make(chan string)
  go func() {
    for i := 0; i < 5; i++ {
      c <- “Hello from “ + name
      time.Sleep(100 * time.Millisecond)
    }
  }()
  return c
}
```

```
func main() {
  a := hello(“Alice”)
  b := hello(“Bob”)
  for {
    select {
    case v, ok := <-a:
      if !ok {
        return
      }
      fmt.Println(v)
    }
  }
```

Example: First response

```
func get(c chan string, url string) {
  if res, err := http.Get(url); err == nil
    { data, _ := ioutil.ReadAll(res.Body))
    c <- string(data)
  }
}
func main() {
  first := make(chan string)
  for _, url := range []string{ “http://example.com”, “http://google.com” }
    { go get(first, url)
  }
  body := <- first
}
```

Example: Timeout

Run

```
func timeout(t time.Duration) <-chan int
  { c := make(chan int)
  go func() {
    time.Sleep(t)
    close(c)
  }()
  return c
}
```

```
func main() {
  chTime := timeout(time.Second)
  chBody := make(chan string)
  go get(“http://example.com”)
  select {
    case body := <-chBody
      fmt.Println(body)
    case <-chTime:
      fmt.Println(“Timeout!”)
  }
}
```

Atomic access

mutex.Lock()

mutex.Lock()

```
var m sync.Mutex   // Make a new mutex

m.Lock()

m.Unlock()
```

// Code between Lock() and Unlock() can only be executed in one goroutine at the same time.

Example: Lock() - 1

Run

```
var a = make([]int, 0)

func add(i int) {
    time.Sleep(time.Duration(rand.Intn(100)) *
    time.Millisecond) a = append(a, i)
}
func main() {
    for i := 0; i < 10; i++ {
        go add(i)
    }
    time.Sleep(time.Second)
    fmt.Println(a)
}
```

Example: Lock() - 2

Run

```
var a = make([]int, 0)
var m sync.Mutex

func add(i int) {
    m.Lock()
    defer m.Unlock()

    time.Sleep(time.Duration(rand.Intn(100)) *
    time.Millisecond) a = append(a, i)
}
```

Example: Wait Group

```
var wg sync.WaitGroup
var urls = []string{ “http://www.golang.org/”, “http://www.google.com/” }
for _, url := range urls {
  wg.Add(1)
  go func(url string) {
    defer wg.Done()
    http.Get(url)
  }(url)
}

wg.Wait()
```

Detecting race condition

go race

Detecting race condition

```
go run -race sample.go
go test -race sample
go build -race sample
go get -race sample
```

Sample output

```
$ go run -race examples/race.go

==================
WARNING: DATA RACE
Read by goroutine 10:
  main.add()
      /home/i/x/src/go2/examples/race.go:18 +0x5a

Previous write by goroutine
  14: main.add()
      /home/i/x/src/go2/examples/race.go:18 +0x12a
```

Rules

Rules

- Use channel to synchronize between goroutine
- Only one goroutine can read and write a variable
  - Or use mutex.Lock()
- close(c): Use like sending an EOF value. Only sending goroutine should call close()

Golang #2: Concurrency

Thanks for your listening

Oliver N.

Software Engineer

Read more

- https://blog.golang.org/share-memory-by-communicating
- https://blog.golang.org/concurrency-is-not-parallelism
- http://talks.golang.org/2012/concurrency.slide
- http://talks.golang.org/2013/advconc.slide
- http://gary.burd.info/go-websocket-chat