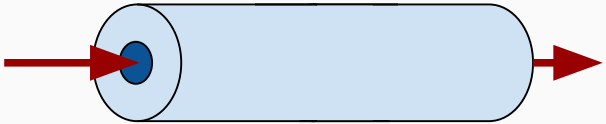
Programming in Go

Matt Holiday
Christmas 2020

Share memory by communicating

Channels

A channel is a one-way communications pipe

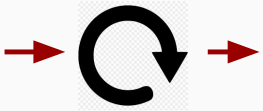


- things go in one end, come out the other
- in the same order they went in
- until the channel is closed
- **multiple readers & writers can share it safely**

Sequential process

Looking at a single independent part of the program,
it appears to be sequential

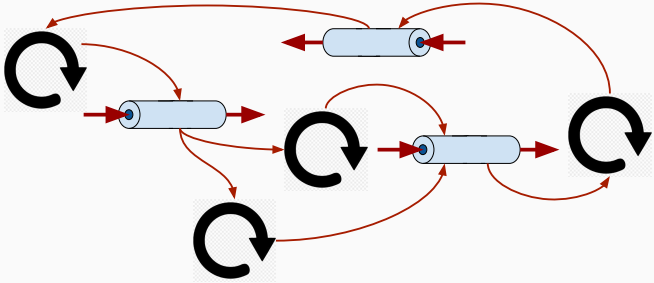
```
for {
    read()
    process()
    write()
}
```



This is perfectly natural if we think of reading & writing files or network sockets

Communicating sequential processes

Now put the parts together with channels to communicate



- each part is independent
- all they share are the channels between them
- **the parts can run in parallel as the hardware allows**

Communicating sequential processes

Concurrency is **always** hard
(the human brain didn't evolve for this, sorry :-)

CSP provides a model for thinking about it that makes it **less hard**
(take the program apart and make the pieces talk to each other)

"Go doesn't force developers to embrace the asynchronous ways of event-driven programming. ... That lets you **write asynchronous code in a synchronous style**. As people, we're much better suited to writing about things in a synchronous style."

— Andrew Gerrand

Goroutines

A goroutine is a unit of **independent execution** (coroutine)

It's easy to start a goroutine: put `go` in front of a *function call*

The trick is knowing how the goroutine will stop:

- you have a well-defined loop terminating condition, or
- you signal completion through a channel or context, or
- you let it run until the program stops

But you need to make sure it doesn't get blocked by mistake

Channels

A channel is like a one-way socket or a Unix pipe
(except it allows multiple readers & writers)

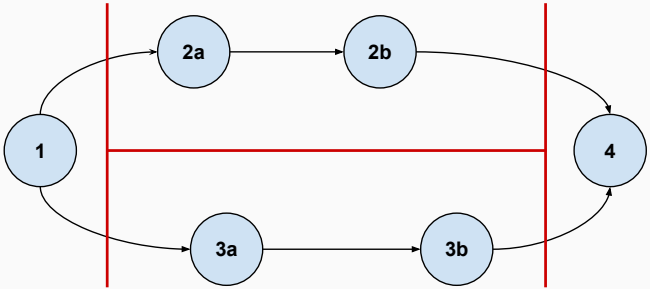
It's a method of synchronization as well as communication

We know that a send (write) always **happens before** a receive (read)

It's also a vehicle for *transferring ownership* of data, so that only one goroutine at a time is writing the data (avoid race conditions)

"Don't communicate by sharing memory; instead, **share memory by communicating**." — Rob Pike

Partial order



- part 1 **happens before** parts of 2 or 3
- both 2 and 3 **happen before** part 4
- the parts of 2 and 3 are ordered among themselves

Concurrency Example 1: Parallel Get

```
func main() {
    results := make(chan result)   // channel for results
    list := []string{"https://amazon.com", "https://google.com",
                     "https://nytimes.com", "https://wsj.com",
    }
    for _, url := range list {
        go get(url, results)       // start a CSP process
    }
    for range list {               // read from the channel
        r := <-results
        if r.err != nil {
            log.Printf("%-20s %s\n", r.url, r.latency)
        } else {
            log.Printf("%-20s %s\n", r.url, r.err)
        }
    }
}
```

Concurrency Example 1: Parallel Get

```
type result struct {
    url     string
    err     error
    latency time.Duration
}

func get(url string, ch chan<- result) {
    start := time.Now()

    if resp, err := http.Get(url); err != nil {
        ch <- result{url, err, 0}     // error response
    } else {
        t := time.Since(start).Round(time.Millisecond)
        ch <- result{url, nil, t}     // normal response
        resp.Body.Close()
    }
}
```

Concurrency Example 2: Stream of IDs

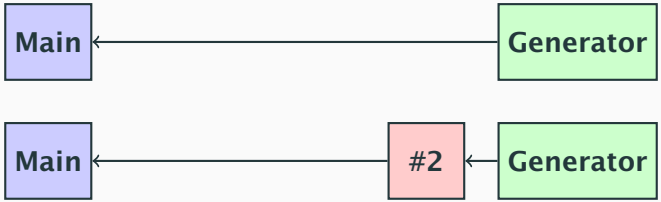
```
var nextID = 0

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "<h1>You got %v<h1>", nextID)

    // unsafe - data race

    nextID++
}

func main() {
    http.HandleFunc("/", handler)
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal(err)
    }
}

// simple HTTP server example from Francesc Campoy
```

Concurrency Example 2: Stream of IDs

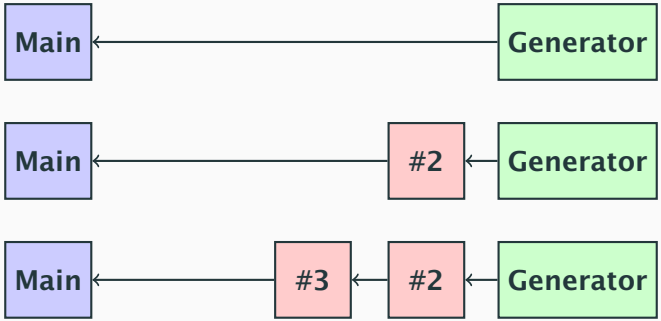
```
var nextID = make(chan int)

func handler(w http.ResponseWriter, q *http.Request) {
    fmt.Fprintf(w, "<h1>You got %v<h1>", <-nextID)
}

func counter() {
    for i := 0; ; i++ {
        nextID <- i
    }
}

func main() {
    go counter()
    http.HandleFunc("/", handler)
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal(err)
    }
}
```

Concurrency Example 3: Prime Sieve

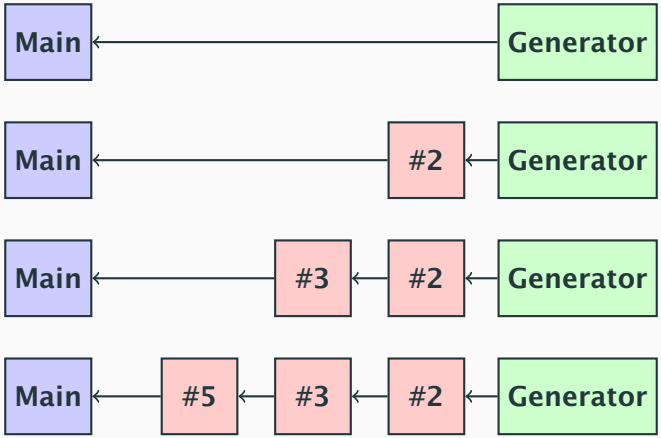
Concurrency Example 3: Prime Sieve



Concurrency Example 3: Prime Sieve



Concurrency Example 3: Prime Sieve



Concurrency Example 3: Prime Sieve

```
// Doug McIlroy (1968) via Tony Hoare (1978)
// code example from the Go language spec

func generate(limit int, ch chan<- int) {
    for i := 2; i < limit ; i++ {
        ch <- i
    }
    close(ch)
}

func filter(src <-chan int, dst chan<- int, prime int) {
    for i := range src {
        if i % prime != 0 {
            dst <- i
        }
    }
    close(dst)
}
```

Concurrency Example 3: Prime Sieve

```
func sieve(limit int) {
    ch := make(chan int)
    go generate(limit, ch)
    for {
        prime, ok := <-ch
        if !ok {
            break
        }
        ch1 := make(chan int)
        go filter(ch, ch1, prime)
        ch = ch1
        fmt.Print(prime, " ")
    }
}

func main() {
    sieve(100) // 2 3 5 7 11 13 17 19 23 29 31 37 41 43 ...
}
```