Lecture 6
Concurrency in Go

CS3211 Parallel and Concurrent Programming

Outline

- Revisiting concurrency vs. parallelism
  - Types of parallelism
  - Amdahl's Law
- Concurrency and communication with Go
- Goroutines, channels in Go
- The sync package
- The Go memory model

Why study concurrency?

- Not a new concept!
  - Traditionally concurrency was achieved through task switching

- Increased prevalence of computers that can genuinely run multiple tasks in parallel rather than just giving the illusion of doing so
  - *Illusion* of concurrency vs. *true* concurrency

Concurrency vs. Parallelism

Concurrency

- Two or more tasks can start, run, and complete in overlapping time periods
- They might not be running (executing on CPU) at the same instant
- Two or more execution flows make progress at the same time by interleaving their executions or by executing instructions (on CPU) at exactly the same time

Structure

Parallelism

- Two or more tasks can run (execute) simultaneously, at the exact same time
- Tasks do not only make progress, but they also actually execute simultaneously

Execution

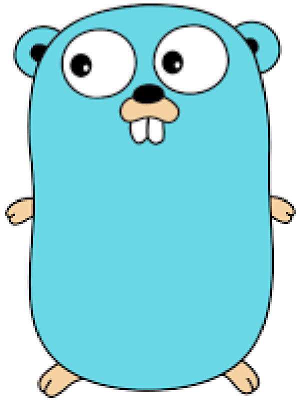
Concurrency in Programming Languages

- We *write* concurrent code
  - We *hope* that our code will run in parallel
- Ways of achieving concurrency:

Until now in CS3211 (C++):	**Next in CS3211 (Go):**
Model your program in terms of *threads*	Model your program in terms of tasks
Synchronize the access to the *memory* between them	Synchronize the tasks by making them communicate
Use *thread pools* to limit the number of threads that must be handled by the machine	

Programming languages with mascots!

- Go!



Go

- Programming language announced at Google in 2009
- Compiled programming language
- Statically typed
- (Partially) syntactically similar to C, but with
    - Memory safety
    - Garbage collection
    - CSP-style concurrency
- Compilers & tools: `gc`, `gccgo`, `gollvm`

Concurrent designs

- Types of parallelism
- Limiting factors for parallelism

Solutions to a problem

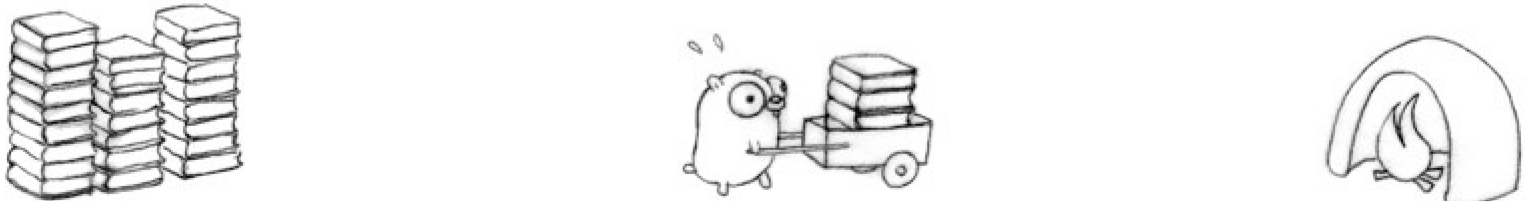
1. With only one gopher this will take too long

2. More gophers are not enough
   - need more carts

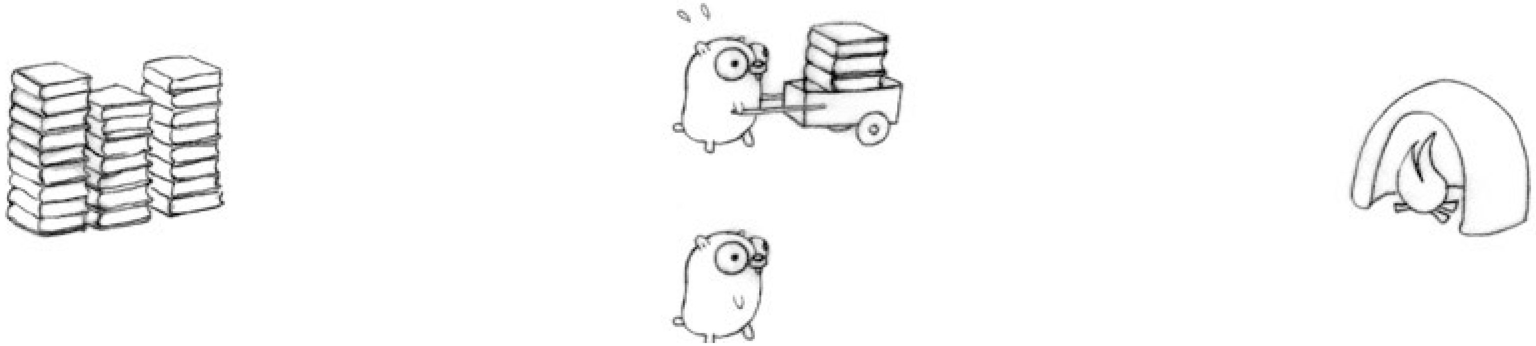
3. More gophers and carts
   - bottlenecks at the pile and incinerator
   - need to synchronize the gophers.

*https://talks.golang.org/2012/waza.slide

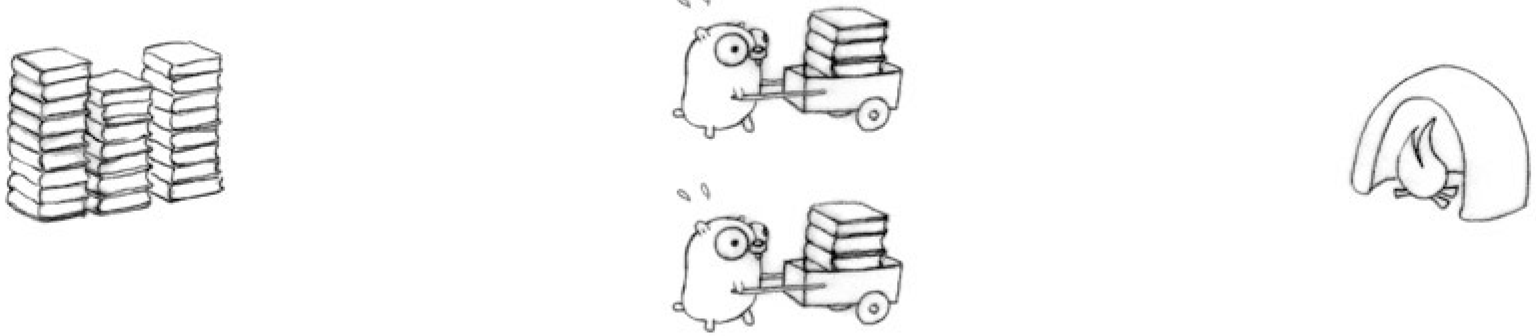
Sequential



More processes



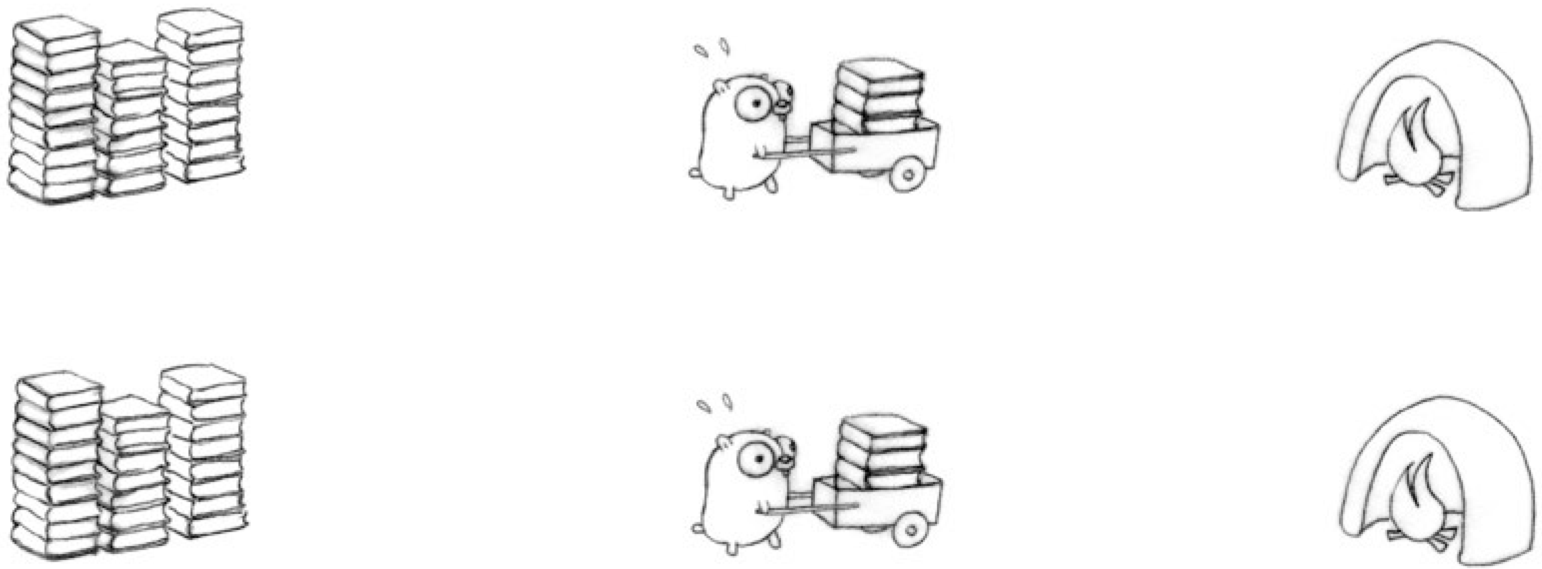
Processes can run same task: embarrassingly parallel, data parallelism



Concurrent composition

- Not automatically parallel!
  - However, it's automatically parallelizable!
- This can be twice as fast when running in parallel

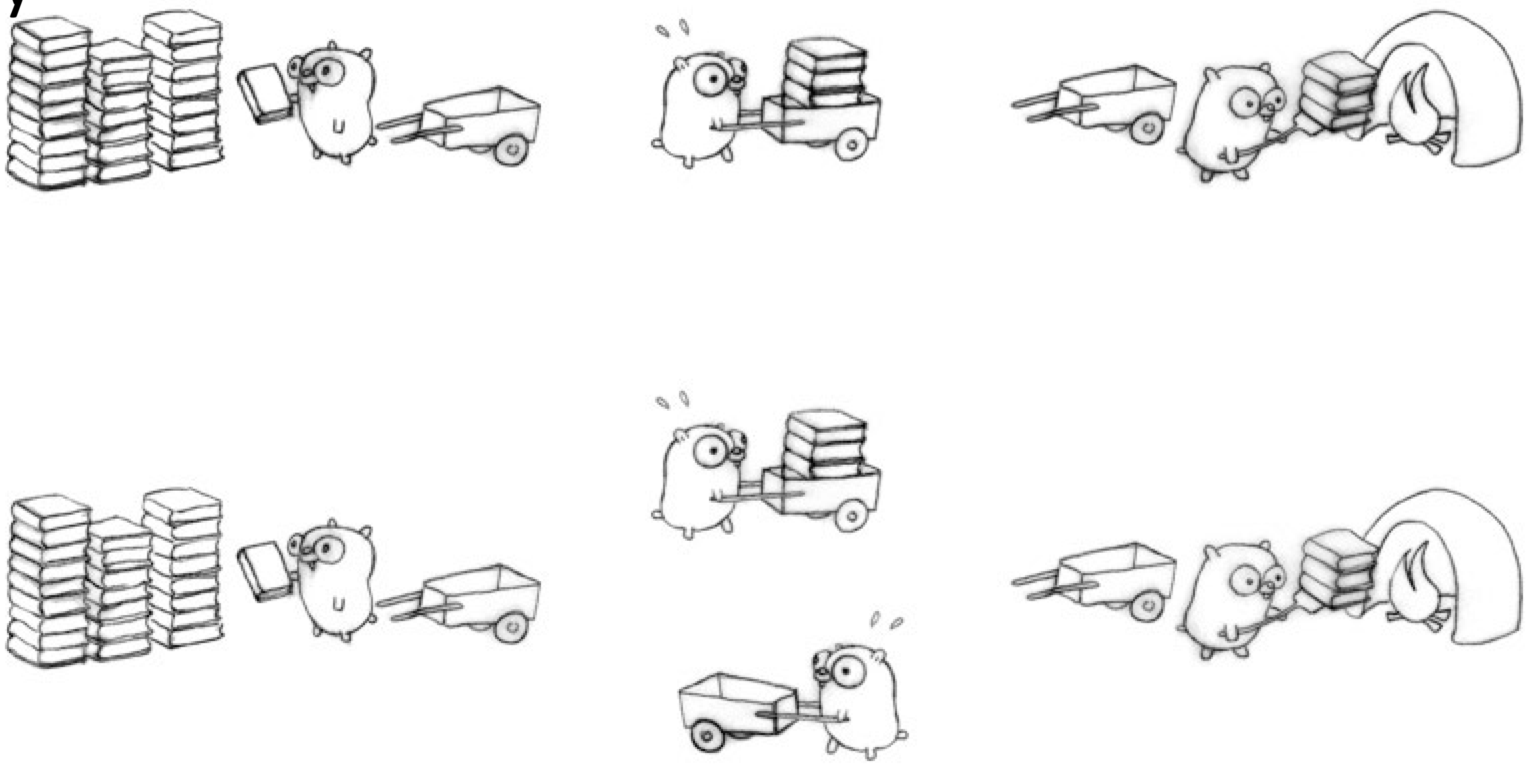
Same task: embarrassingly parallel, data parallelism



Concurrent designs

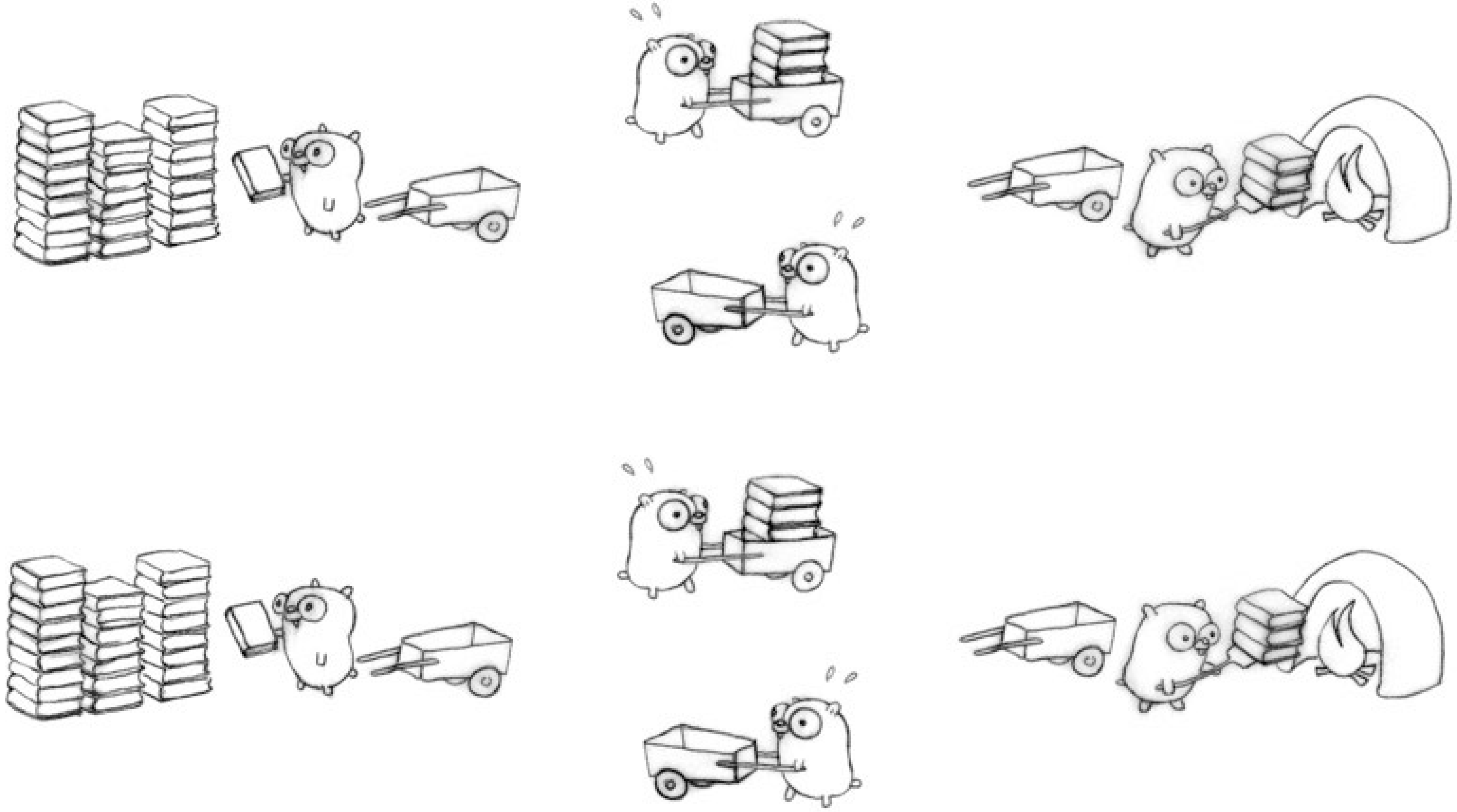
- Three gophers in action, but with (likely) delays
- Finer-grained concurrency
- Four distinct gopher procedures:
  - load books onto cart
  - move cart to incinerator
  - unload cart into incinerator
  - return empty cart
- Enables different ways to parallelize

Break the work into tasks: aka pipeline parallelism



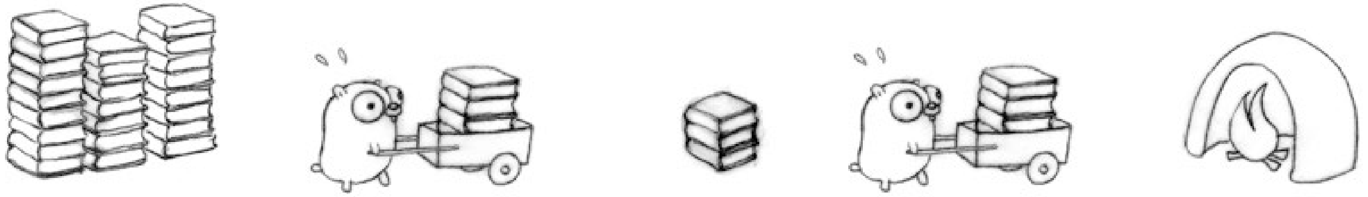
Concurrency enabled more parallelization

- Now parallelize on the other axis
  - 8 gophers

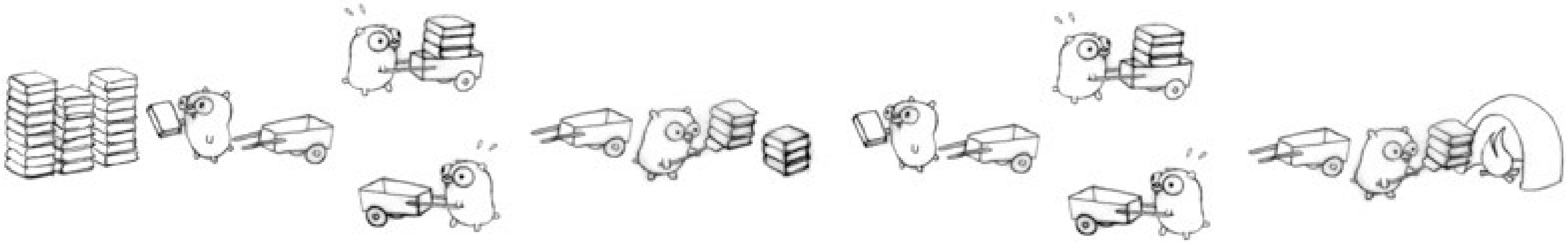


Other concurrent designs

- Design 1

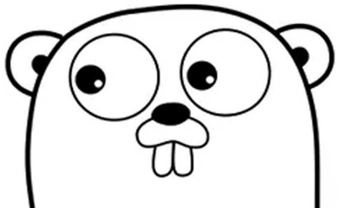


- Design 2



Back to computing

- In our book transport problem, substitute:
  - book pile => web content
  - gopher => CPU
  - cart => rendering, or networking
  - incinerator => proxy, browser, or other consumer
- It becomes a concurrent design for a scalable web service
  - Gophers are serving web content



Take-away points

- There are many concurrent designs
  - Many ways to break the processing down
- Finer level of granularity enables our program to scale *dynamically* when it runs to the amount of parallelism possible on the program's host
  - Amdahl's law in action!

Types of parallelism

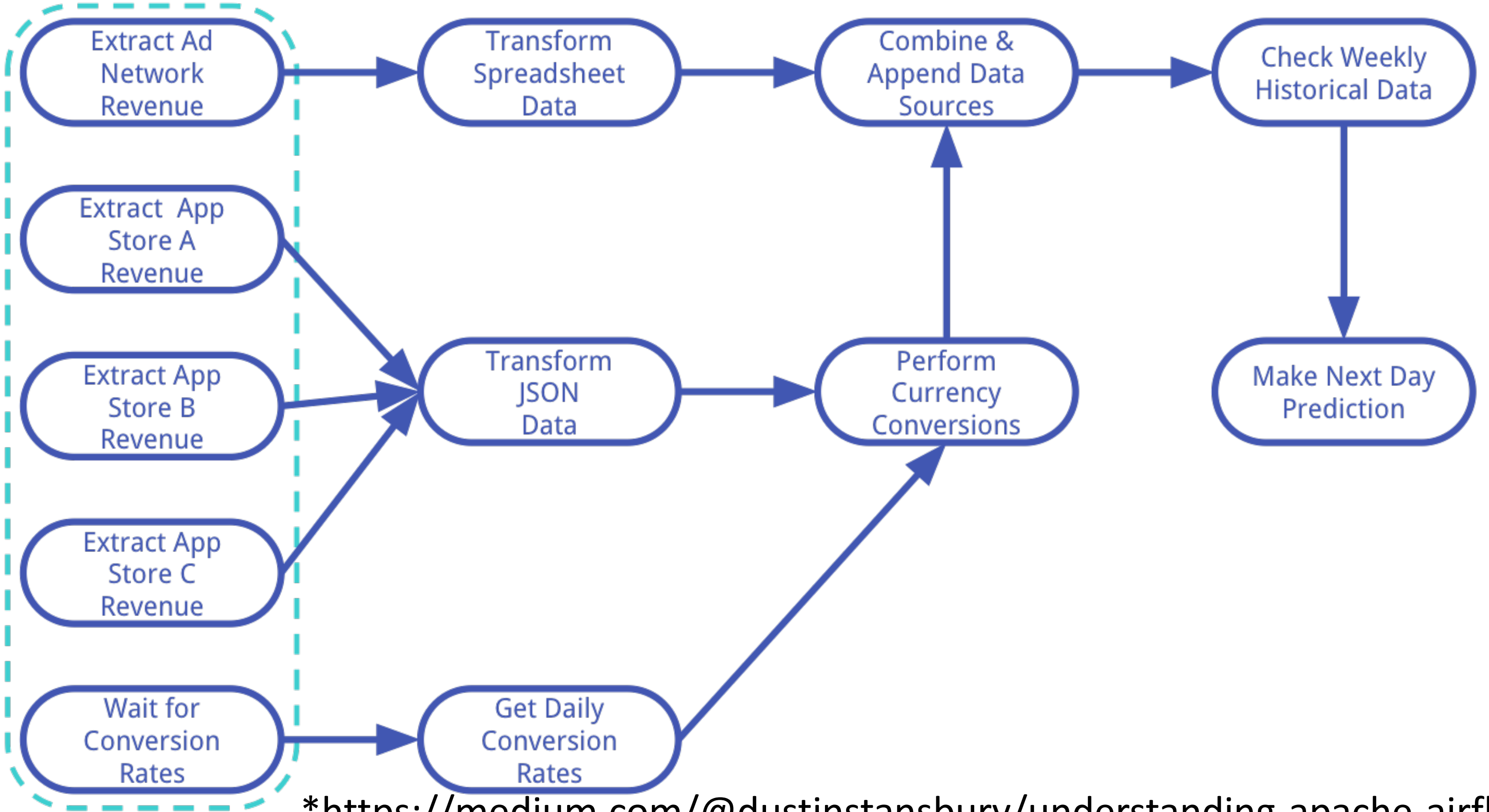
- Task parallelism
  - Do the same work faster
- Data parallelism
  - Embarrassingly parallel algorithms
  - Do more work in the same amount of time

Task Dependency Graph

- Can be used to visualize and evaluate the task decomposition strategy
- A **directed acyclic graph**:
  - Node: Represent each task, node value is the expected execution time
  - Edge: Represent **control dependency** between task

- Properties:
  - Critical path length: maximum (slowest) completion time
  - Degree of concurrency = Total Work / Critical Path Length
    - An indication of amount of work that can be done concurrently

An example



*https://medium.com/@dustinstansbury/understanding-apache-airflows-key-concepts-a96efed52b1a

Concurrent Programming Challenges

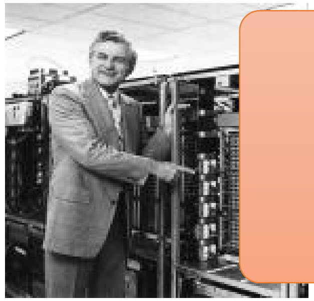
- Finding enough concurrency
- Granularity of tasks
- Coordination and synchronization

Parallel Program: **Speedup**

- Measure the benefit of parallelism
  - A comparison between sequential and parallel execution time

$$\boldsymbol{S_p(n) = \frac{T_{best\_seq}(n)}{T_p(n)}}$$

Amdahl's Law (1967)



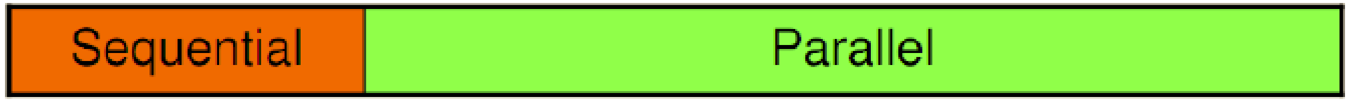
Speedup of parallel execution is limited by the fraction of the algorithm that cannot be parallelized (*f*).

- f ($0 \leq f \leq 1$) is called the sequential fraction
- Also known as fixed-workload performance

- The most well-known law for discussing speedup performance
  - Applicable at all levels of parallelism

Amdahl's Law: Implication

- Sequential execution time:



$f \times T_*(n)$ $\quad$ $(1 - f) \times T_*(n)$

- Parallel execution time:

$f \times T_*(n)$ $\quad$ $\dfrac{(1 - f) \times T_*(n)}{p}$

$$S_p(n) = \frac{T_*(n)}{f \times T_*(n) + \frac{1-f}{p} T_*(n)} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

Outline

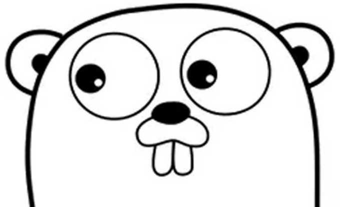
- Revisiting concurrency vs. parallelism
  - Types of parallelism
  - Amdahl’s Law
- Concurrency and communication with Go
- Goroutines, channels in Go
- The sync package
- The Go memory model

Concurrency + Communication

- Go model – based on Communicating Sequential Processes (CSP)*
  - Concurrency: structure a program by breaking it into pieces that can be executed independently
  - Communication: coordinate the independent executions

*C. A. R. Hoare: Communicating Sequential Processes (CACM 1978)

- Ideas of CSP
  - Refined to process calculus
  - Can be used to reason about program correctness

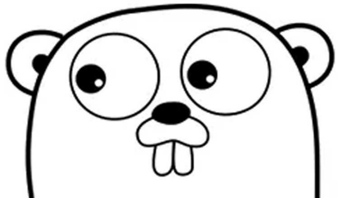


Abstractions in Go

Concurrency

- Goroutines
  - A function running independently
  - Spin up (start) a goroutine using

    `go function_name`
  - Run on OS threads

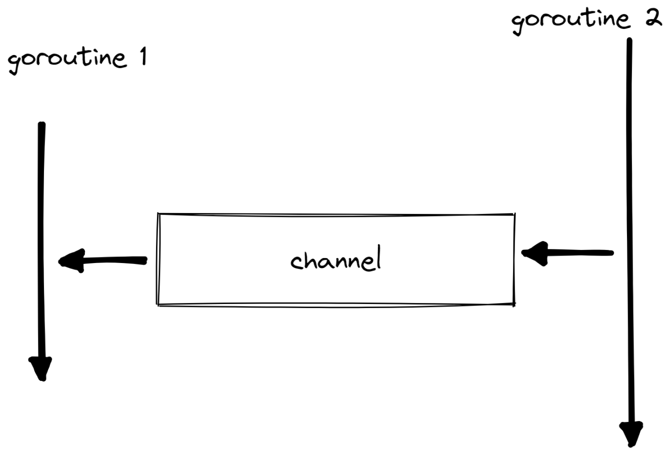


Communication

- Channels
  - Goroutines can write to and read from a channel

    `channel<-`

    `<-channel`
- `select` statements



Abstractions in Go

Concurrency

- Goroutines
  - A function running independently
  - Spin up (start) a goroutine using

  `go function_name`

  - Run on OS threads

- Tasks

Communication

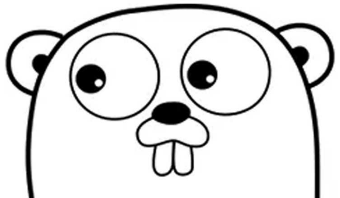
- Channels
  - Goroutines can write to and read from a channel

  `channel<-`

  `<-channel`

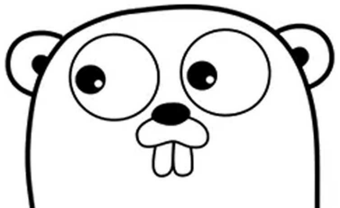
- `select` statements

- Dependencies



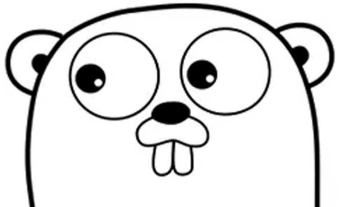
Goroutines

- Function running **independently**
  - In the **same address space** as other goroutines
  - Like & in shell
- Cheaper than threads
- Goroutines follow the fork-join model



Running Goroutines

- Runtime *multiplexes* goroutines onto OS threads
  - Automatic scheduling – mapping M:N
  - **Decouples concurrency from parallelism**
- Goroutine is a special class of coroutine (concurrent subroutine)
  - When a *goroutine blocks*, that *thread blocks*
    - but no other goroutine blocks
  - Preemptable: Go's runtime can suspend them

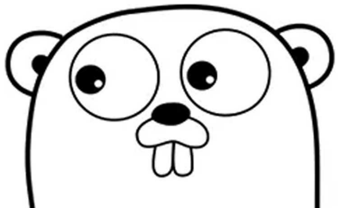


Goroutines example

- Line 3: the main goroutine is automatically created and started when the process begins
- Line 4: start a goroutine using keyword go
- Line 8: the print might never happen because the main goroutine finishes execution before `sayHello` completes
- Line 11: anonymous function

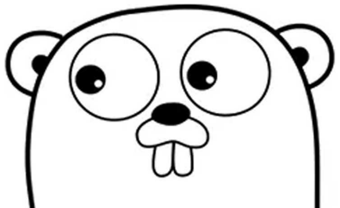
```
3  func main() {
4      go sayHello()
5  // continue doing other things
6  }
7  func sayHello() {
8      fmt.Println("hello")
9  }
```

```
11 go func() {
12     fmt.Println("hello")
13 }()
14 // continue doing other things
```



Goroutines are lightweight

- A newly minted goroutine is given a few kilobytes, which is almost always enough
  - When it isn’t, the runtime grows (and shrinks) the memory for storing the stack automatically
- The CPU overhead averages about three cheap instructions per function call
- It is practical to create hundreds of thousands of goroutines in the same address space
- Goroutines are not garbage collected!
  - Programmer should prevent goroutine leaks



Goroutines

- Line 12: Join point
- Line 7: The goroutine is running a closure that has closed over the iteration variable `salutation`

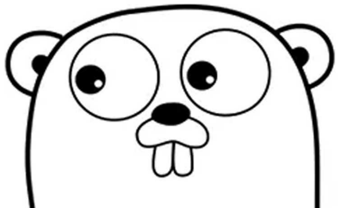
```
3  func main() {
4      var wg sync.WaitGroup
5      for _, salutation := range []string{"hello", "greetings", "good day"} {
6          wg.Add(1)
7          go func() {
8              defer wg.Done()
9              fmt.Println(salutation)
10         }()
11     }
12     wg.Wait()
13 }
```

Go runtime is observant enough to know that a reference to the `salutation` variable is still being held, and therefore will transfer the memory to the heap so that the goroutines can continue to access it.

Goroutines

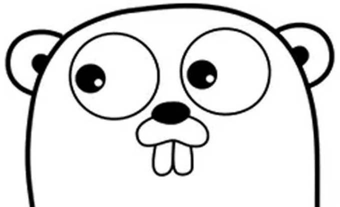
- Line 19: write this loop is to pass a copy of `salutation` into the closure

```
13      var wg sync.WaitGroup
14      for _, salutation := range []string{"hello", "greetings", "good day"} {
15          wg.Add(1)
16          go func(salutation string) {
17              defer wg.Done()
18              fmt.Println(salutation)
19          }(salutation)
20      }
21      wg.Wait()
22  }
```



Potential issues with shared memory

- Need to synchronize access to shared memory locations
  - Similar to what we did in C++
  - `sync` package
- Don’t rely on shared memory for memory locations that are modified
  - Never modify a shared memory location
- Use **channels** instead of modifying shared memory

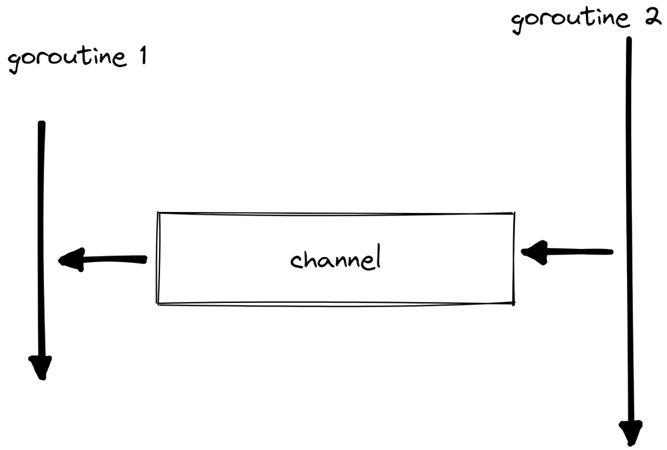


Channels

- Serves as a conduit for a stream of information
  - Like the pipe (|) in shell
- Values may be passed along the channel, and then read out downstream
  - Pass a value into a chan variable, and then somewhere else in your program read it off the channel
- No knowledge is required about the other parts of your program that work with the channel
- A channel is a reference to a place in memory where the channel resides
  - Channels (references of channels) can be passed around your program

Channels

- Typed
- Bi-/uni- directional
- Blocking
  - Write to a channel that is full waits until the channel has been emptied
  - Read from a channel that is empty waits until at least one item is placed on it
  - Can cause deadlocks!



Creating a channel

- Lines 4-5: declare and create a bidirectional channel using built-in make function

```
4  var dataStream chan interface{}
5  dataStream = make(chan interface{})
```

- Lines 8-9: declare a unidirectional channel

```
8  var receiveChan <-chan interface{}
9  var sendChan chan<- interface{}
10 dataStream := make(chan interface{})
11 // Valid statements:
12 receiveChan = dataStream
13 sendChan = dataStream
```

- Line 22: receive will block until `timerChan` delivers.
  - Value sent is other goroutine's completion *time*

```
16 timerChan := make(chan time.Time)
17 go func() {
18     time.Sleep(deltaT)
19     timerChan <- time.Now()
20 }()
21 // Do something else; when ready, receive.
22 completedAt := <-timerChan
```

Blocking operations

- Lines 23, 26: blocking read and write
- Line 25: ok Boolean indicates whether the read was
  - a value generated by a **write**, or
  - a default value generated from a **closed** channel

```go
21 stringStream := make(chan string)
22 go func() {
23     stringStream <- "Hello channels!"
24 }()
25 salutation, ok := <-stringStream
26 fmt.Printf("(%v): %v", ok, salutation)
```

- Line 33: reading from a closed channel
  - Allowed any number of times

```go
31 intStream := make(chan int)
32 close(intStream)
33 integer, ok := <- intStream
34 fmt.Printf("(%v): %v", ok, integer)
```

Synchronizing using channels

- Line 48: *ranging* over a channel
  - The loop doesn't need an exit criteria

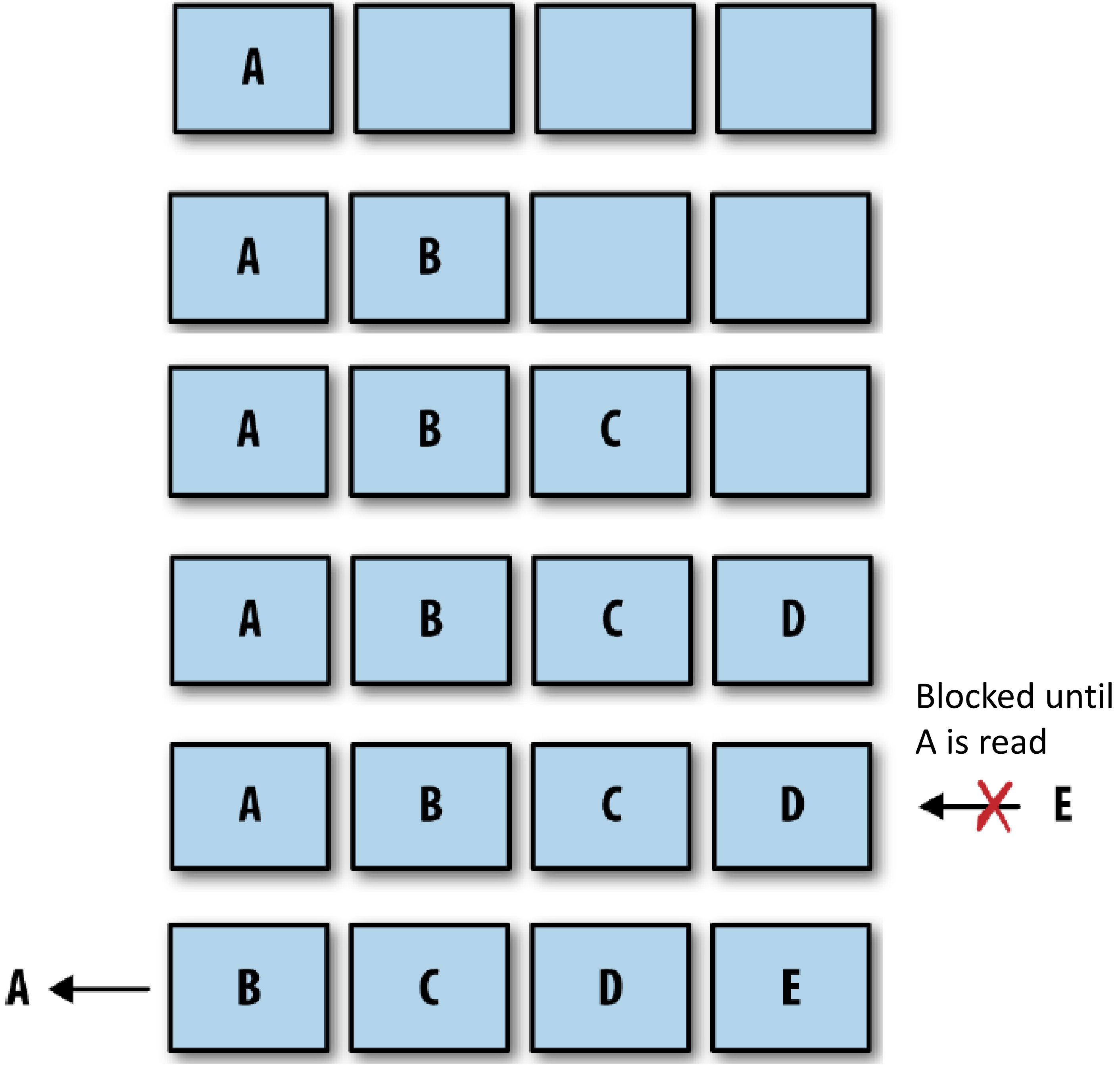
- Line 62: instead of writing n times to the channel to unblock each goroutine, you can simply close the channel

```go
41      intStream := make(chan int)
42      go func() {
43          defer close(intStream)
44          for i := 1; i <= 5; i++ {
45              intStream <- i
46          }
47      }()
48      for integer := range intStream {
49          fmt.Printf("%v ", integer)
50      }
```

```go
51  begin := make(chan interface{})
52  var wg sync.WaitGroup
53  for i := 0; i < 5; i++ {
54      wg.Add(1)
55      go func(i int) {
56          defer wg.Done()
57          <-begin
58          fmt.Printf("%v has begun\n", i)
59      }(i)
60  }
61  fmt.Println("Unblocking goroutines...")
62  close(begin)
63  wg.Wait()
```

Buffered channel

```
c := make(chan rune, 4)
```



Operation	Channel state	Result
Read	`nil`	Block
	Open and Not Empty	Value
	Open and Empty	Block
	Closed	<default value>, false
	Write Only	Compilation Error
Write	`nil`	Block
	Open and Full	Block
	Open and Not Full	Write Value
	Closed	**panic**
	Receive Only	Compilation Error
`close`	`nil`	**panic**
	Open and Not Empty	Closes Channel; reads succeed until channel is drained, then reads produce default value
	Open and Empty	Closes Channel; reads produces default value
	Closed	**panic**
	Receive Only	Compilation Error

Ownership of a channel

- Owner is the goroutine that instantiates, writes, and closes a channel
- Useful when reasoning about program correctness
- Unidirectional channels
  - Owners have a write-access view into the channel (chan or chan<-)
  - Utilizers only have a read-only view into the channel (<-chan)

Owner should	**Consumer should**
• Instantiate the channel	• Know when a channel is closed
• Perform writes, or pass ownership to another goroutine	• Responsibly handle blocking for any reason
• Close the channel	
• Encapsulate 1.-3. and expose them via a reader channel	

Ownership increases safety

- Because we’re the one initializing the channel, we remove the risk of deadlocking by writing to a nil channel
- Because we’re the one initializing the channel, we remove the risk of panicing by closing a nil channel
- Because we’re the one who decides when the channel gets closed, we remove the risk of panicing by writing to a closed channel
- Because we’re the one who decides when the channel gets closed, we remove the risk of panicing by closing a channel more than once
- We wield the type checker at compile time to prevent improper writes to our channel

`select` statement

- Compose channels together in a program to form larger abstractions
- Bind together channels
  - locally, within a single function or type,
  - globally, at the intersection of two or more components in a system
- Help safely bring channels together with concepts like cancellations, timeouts, waiting, and default values
- Similar in syntax with a `switch` block
  - BUT `case` statements aren't tested sequentially, and execution won't automatically fall through if none of the criteria are met

Behavior of `select`

- All channel reads and writes (case statements) are considered simultaneously to see if any of them are ready
  - populated or closed channels in the `case` of reads
  - channels that are not at capacity in the case of writes
- The entire select statement blocks if none of the channels are ready
- Handle the following:
  - Multiple channels have something to read
  - There are never any channels that become ready
  - We want to do something, but no channels are currently ready

Multiple channels have something to read

- Output:
  ```
  c1Count: 505
  c2Count: 496
  ```
- Go runtime will perform a pseudorandom uniform selection over the set of case statements
  - Each case has an equal chance of being selected

```go
c1 := make(chan interface{}); close(c1)
c2 := make(chan interface{}); close(c2)
var c1Count, c2Count int
for i := 1000; i >= 0; i-- {
    select {
    case <-c1:
        c1Count++
    case <-c2:
        c2Count++
    }
}
fmt.Printf("c1Count: %d\nc2Count: %d\n", c1Count, c2Count)
```

Channels are not ready

- Never ready: timeout
  - Line 44: `time.After` returns a channel that sends the current time after a `time.Duration`
- Do work while waiting: use `default`

```
41    var c <-chan int
42    select {
43    case <-c:
44    case <-time.After(1 * time.Second):
45        fmt.Println("Timed out.")
46    }
```

For-select loop

- Allows a goroutine to make progress on work while waiting for another goroutine to report a result

```go
done := make(chan interface{})
go func() {
    time.Sleep(5*time.Second)
    close(done)
}()
workCounter := 0
loop:
for {
    select {
    case <-done:
        break loop
    default:
    }
    // Simulate work
    workCounter++
    time.Sleep(1*time.Second)
}
fmt.Printf("%v cycles of work.\n", workCounter)
```

The sync package

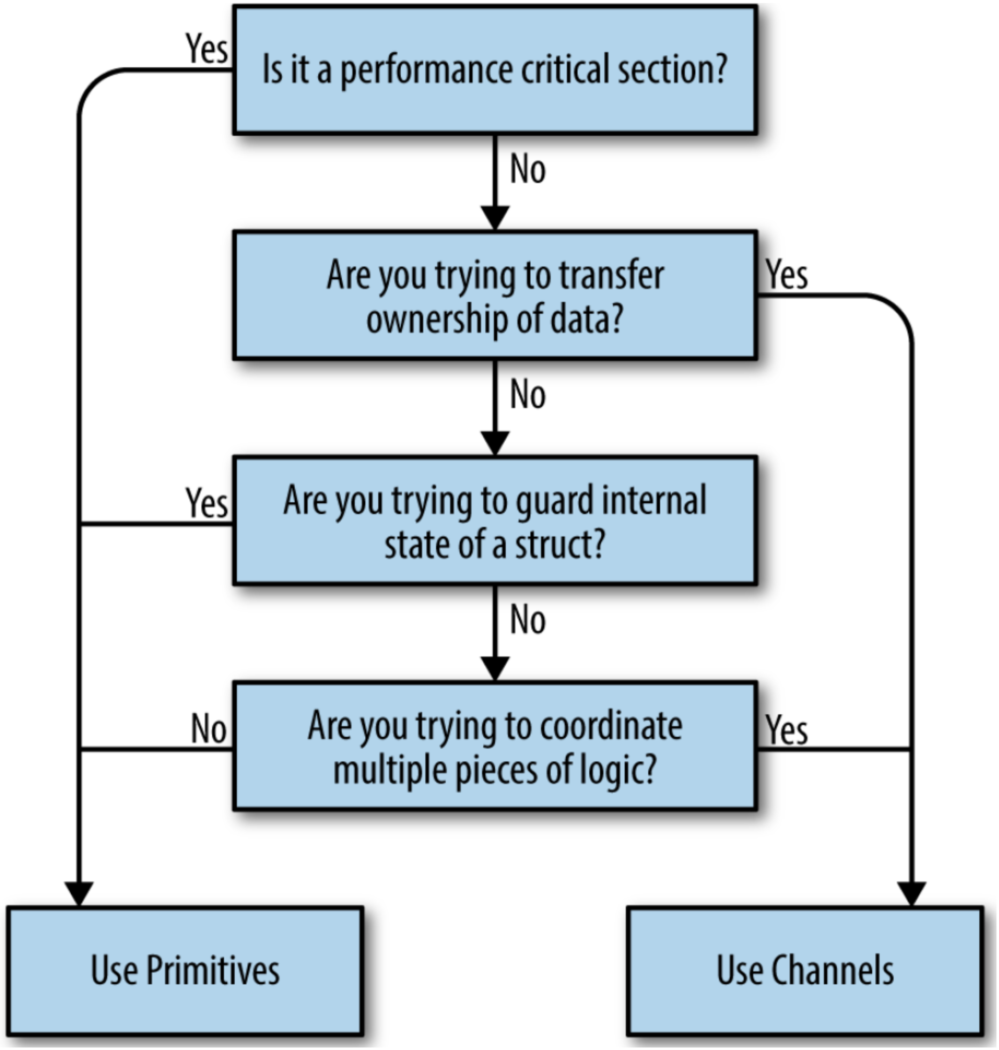
Regarding mutexes, the sync package implements them, but we hope Go programming style will encourage people to try higher-level techniques. In particular, consider structuring your program so that *only one goroutine at a time is ever responsible for a particular piece of data.*

Do not communicate by sharing memory.

Instead, share memory by communicating.

Sync package

- Used mostly in small scopes such as a `struct`
- Contains
  - `WaitGroup`: wait for a set of concurrent operations to complete
  - Synchronization primitives
    - `Mutex` and `RWMutex`
    - `Cond`
    - `Once`
  - Basic constructs
    - `Pool`



The Go memory model

- Specifies the conditions under which reads of a variable in one goroutine can be guaranteed to observe values produced by writes to the same variable in a different goroutine
  - Happens Before
  - Synchronization of goroutines and channels

Happens before

- Within a single goroutine, reads and writes must behave as if they executed in the order specified by the program (**sequenced before**)
- The execution order observed by one goroutine may differ from the order perceived by another
- To guarantee that a read r of a variable v observes a particular write w to v, ensure that w is the only write r is allowed to observe. That is, r is guaranteed to observe w if both of the following hold:
    - w happens before r.
    - Any other write to the shared variable v either happens before w or after r.
- The **happens before** relation is defined as the transitive closure of the union of the **sequenced before** and **synchronized before** relations.

Synchronized before

- The go statement that starts a new goroutine **synchronized before** the goroutine's execution begins
- The exit of a goroutine is not guaranteed to be **synchronized before** any event in the program
- A *send* on a channel is **synchronized before** the completion of the corresponding receive from that channel.
- The *closing* of a channel is **synchronized before** a receive that returns a zero value because the channel is closed
- A *receive* from an unbuffered channel is **synchronized before** the send on that channel completes
- The kth receive on a channel with capacity C is **synchronized before** the (k+C)th send from that channel completes.

Send-receive Synchronized Before

- A *send* on a channel is **synchronized before** the completion of the corresponding receive from that channel.

```
var c = make(chan int, 10)
var a string

func f() {
        a = "hello, world"
        c <- 0
}

func main() {
        go f()
        <-c
        print(a)
}
```

- A *receive* from an unbuffered channel is **synchronized before** the send on that channel completes

```
var c = make(chan int)
var a string

func f() {
        a = "hello, world"
        <-c
}

func main() {
        go f()
        c <- 0
        print(a)
}
```

Summary

- Go helps distinguish between concurrency and parallelism
- Using a different way to implement concurrency based on CSP
  - Goroutines and channels

References

- “Concurrency in Go” by Katherine Cox-Buday, 2017.
- “Concurrency is not Parallelism” by Rob Pike
- https://go.dev/ref/mem