

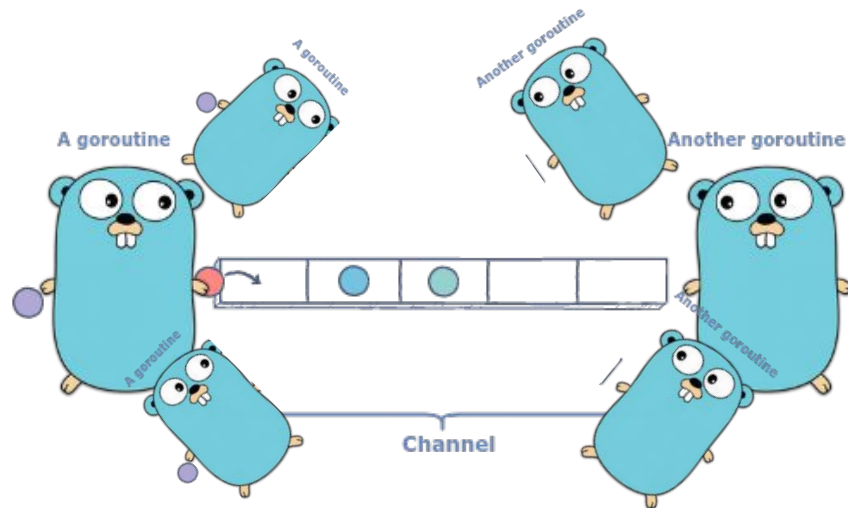
# CS3211 Tutorial 6

Advanced Go concurrency patterns  
Simon

*Adapted From Sriram's Slides*

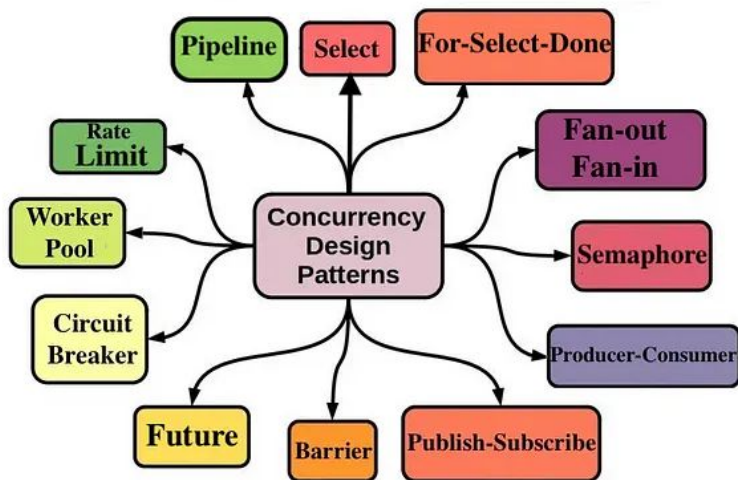
# What we've covered

- **Motivation:** concurrency **without shared memory**
- Channels (Tut 5)
  - Unbuffered vs buffered
  - Pushing into a single channel vs using per-goroutine channels
  - etc
- Waitgroups (Tut 5)
  - Handling synchronized exits from multiple goroutines
  - etc



# Today's Tutorial

- Goal: to understand common Go concurrency patterns
- Question: Why do we need to know *Concurrency Patterns*?



By the end of today's tutorial, you should  
be able to answer this question

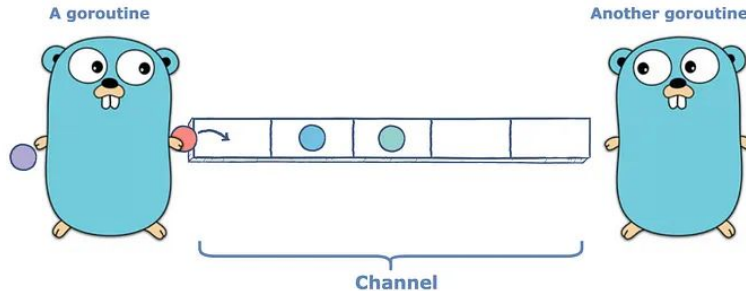
*How can we solve interesting, realistic, and  
complex problems with Go?*

*(maybe some useful techniques for Assignment 2?)*

# **1. Exit Conditions and Context**

# 1. Exit Conditions

- We have **many goroutines** that can run at the same time
- How do we manage their **various exit conditions**?
  - **Note:** you **cannot forcibly (non-cooperatively) kill a goroutine from another goroutine!**
  - <https://stackoverflow.com/questions/6807590/how-to-stop-a-goroutine>
  - Recall Tut 5 methods :) What's that?



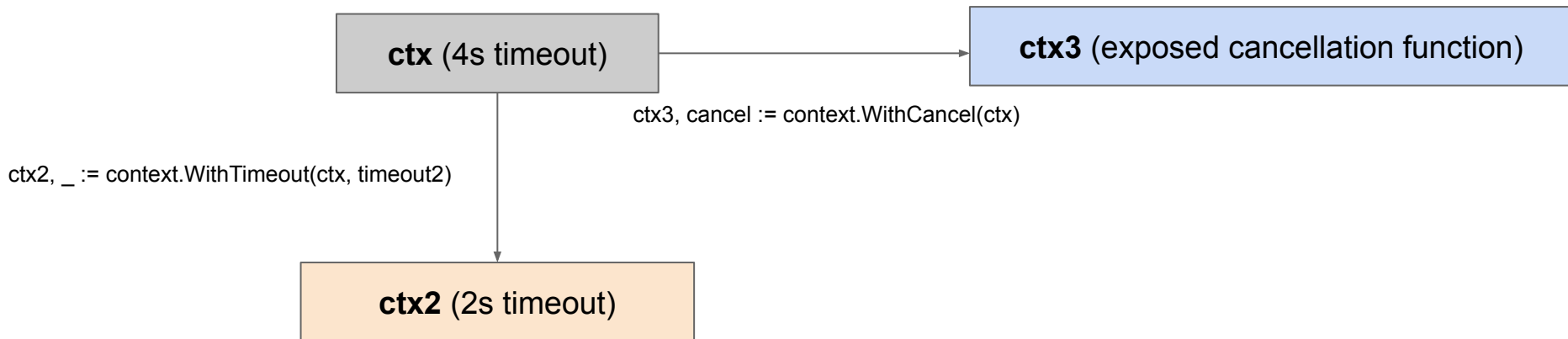
# 1. Exit Conditions

- But in Real Life, there are more complex exit conditions. Imagine if we have N goroutines
- **Goroutine 1 (main): must stop after 4 seconds**
  - E.g., network request that you know cannot take longer than 4 seconds (TTL)
- **Goroutine 2: must stop after 2 seconds or when goroutine 1 exits**
  - Shorter network request that is gets some related information for the first goroutine
- **Goroutine 3: must stop if the program receives a termination signal (SIGINT/TERM) or when goroutine 1 exits**
  - Maybe this goroutine is responsible for request cleanup on SIGINT but also is meaningless without the first goroutine succeeding

# 1. Exit Conditions

- Solution: we can build a **context tree**!  
(<https://pkg.go.dev/context>)

`ctx, _ := context.WithTimeout(context.Background(), timeout1)`





# 1. Exit Conditions

- What happens if we no longer make **ctx2** depend on **ctx1**? [p]

```
ctx, _ := context.WithTimeout(context.Background(), timeout1)

                                context.Background()
ctx2, _ := context.WithTimeout(ctx, timeout2)
go func() {
    <-ctx2.Done()
    fmt.Printf("ctx2 done at %v\n", time.Now().Sub(startTime))
}()

ctx3, cancel := context.WithCancel(ctx)
go func() {
    <-ctx3.Done()
    fmt.Printf("ctx3 done at %v\n", time.Now().Sub(startTime))
}()

go func() {
    <-handleSigs()
    cancel()
    fmt.Printf("signal in at %v\n", time.Now().Sub(startTime))
}()

<-ctx.Done()
```

# 1. Exit Conditions

- What happens if we no longer make **ctx2** depend on **ctx1**? [p]
- Nothing happens! ctx2 timeout is 2s, ctx is 4 seconds, the inheritance is pretty unnecessary in this specific scenario.

```
ctx, _ := context.WithTimeout(context.Background(), timeout1)
                                context.Background()
ctx2, _ := context.WithTimeout(ctx, timeout2)
go func() {
    <-ctx2.Done()
    fmt.Printf("ctx2 done at %v\n", time.Now().Sub(startTime))
}()

ctx3, cancel := context.WithCancel(ctx)
go func() {
    <-ctx3.Done()
    fmt.Printf("ctx3 done at %v\n", time.Now().Sub(startTime))
}()

go func() {
    <-handleSigs()
    cancel()
    fmt.Printf("signal in at %v\n", time.Now().Sub(startTime))
}()

<-ctx.Done()
```

# 1. Exit Conditions

- Apparently the “ctx3 done at..” message can not be printed. Why?

```
ctx, _ := context.WithTimeout(context.Background(), timeout1)

ctx2, _ := context.WithTimeout(ctx, timeout2)
go func() {
    <-ctx2.Done()
    fmt.Printf("ctx2 done at %v\n", time.Now().Sub(startTime))
}()

ctx3, cancel := context.WithCancel(ctx)
go func() {
    <-ctx3.Done()
    fmt.Printf("ctx3 done at %v\n", time.Now().Sub(startTime))
}()

go func() {
    <-handleSigs()
    cancel()
    fmt.Printf("signal in at %v\n", time.Now().Sub(startTime))
}()

<-ctx.Done()
```

# 1. Exit Conditions

- Apparently the “ctx3 done at..” message can not be printed. Why?
- Question: how can we ensure that the ctx3 fmt.Printf is printed?

```
ctx, _ := context.WithTimeout(context.Background(), timeout1)

ctx2, _ := context.WithTimeout(ctx, timeout2)
go func() {
    <-ctx2.Done()
    fmt.Printf("ctx2 done at %v\n", time.Now().Sub(startTime))
}()

ctx3, cancel := context.WithCancel(ctx)
go func() {
    <-ctx3.Done()
    fmt.Printf("ctx3 done at %v\n", time.Now().Sub(startTime))
}()

go func() {
    <-handleSigs()
    cancel()
    fmt.Printf("signal in at %v\n", time.Now().Sub(startTime))
}()

<-ctx.Done()
```

These lines may not have time to be executed

Program ends shortly after this

# 1. Exit Conditions

- Question: how can we ensure that the ctx3 fmt.Printf is printed?
  - Done channel solution:  
<https://fsmolt.comp.nus.edu.sg/z/oYTP9M>
  - Waitgroup solution:  
<https://fsmolt.comp.nus.edu.sg/z/KnzbP6>

```
// wg initialization
var wg sync.WaitGroup
wg.Add(1)

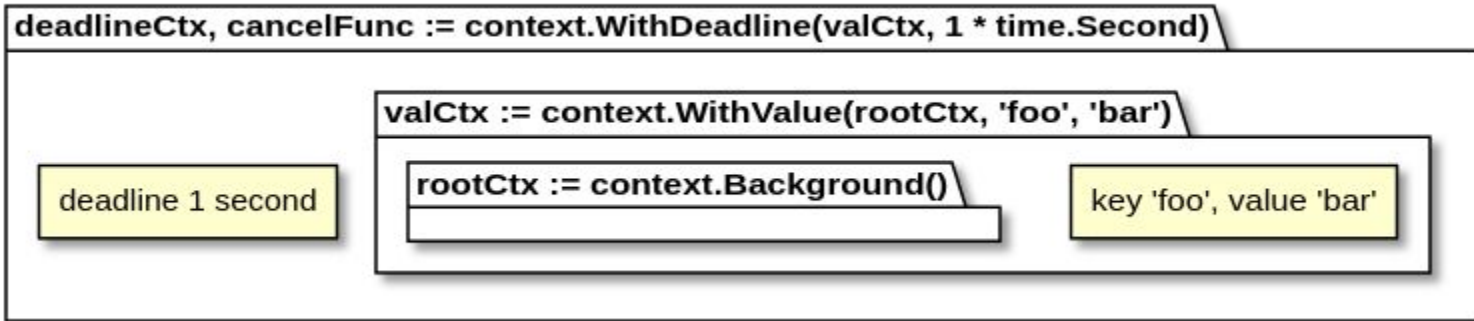
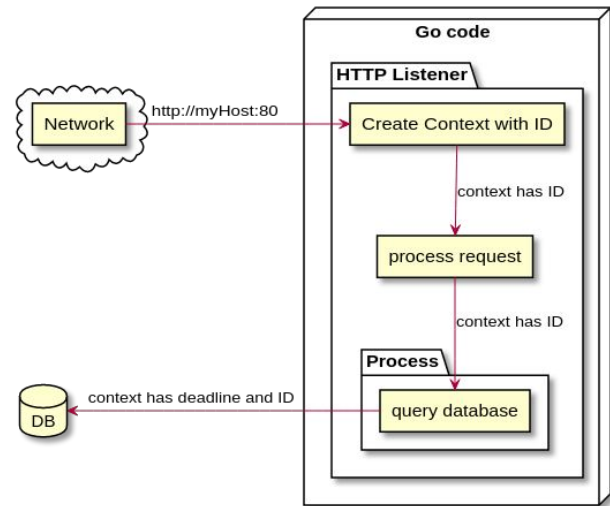
ctx3, cancel := context.WithCancel(ctx)
go func() {
    // Waitgroup done on exit
    defer wg.Done()
    <-ctx3.Done()
    fmt.Printf("ctx3 done at %v\n", time.Now().Sub(startTime))
}()

go func() {
    <-handleSigs()
    cancel()
    fmt.Printf("signal in at %v\n", time.Now().Sub(startTime))
}()

<-ctx.Done()
fmt.Printf("ctx done at %v\n", time.Now().Sub(startTime))
// Wait for ctx3 goroutine to exit
wg.Wait()
```

# Why does this question matter?

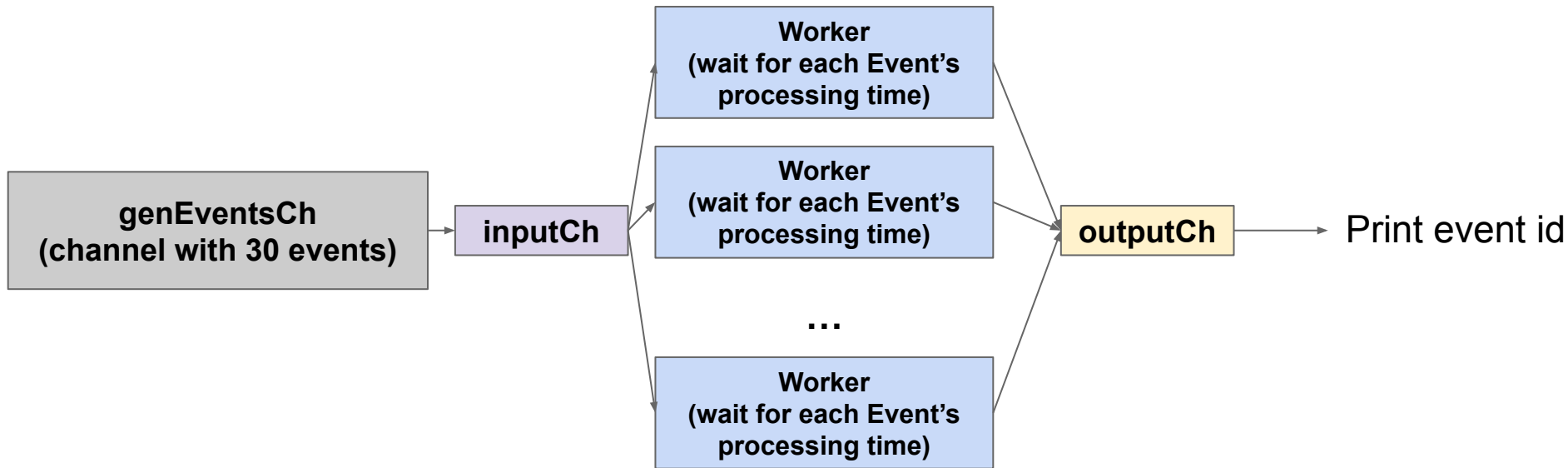
- Context trees are a powerful tool to manage the **context** of goroutines (including **timeouts** and **cancellations**)
- Context's **WithValue()** can also be used to pass **key-value pairs** to **child contexts**!
- Still... many gotchas await if you are not careful.



# Fan-out, Fan-in

# Understanding the fan out-in pattern

- Work is distributed through an **input channel** that every goroutine reads from
- Outputs sent to **one output channel**, which is read from, and event id printed
- Does this sound familiar? What pattern is this usually called?





## 2.1 Fan in/out

- As in Go: the devil is in the details
- Look at the spaghetti code in the worker routine
- Why do we have so many `<-done` calls?

```
30 func (w *worker) start(  
31     done <-chan struct{},  
32     fn EventFunc, wg *sync.WaitGroup,  
33 ) {  
34     go func() {  
35         defer wg.Done()  
36         for {  
37             select {  
38                 case e, more := <-w.inputCh:  
39                     if !more {  
40                         return  
41                     }  
42                     select {  
43                         case w.outputCh <- fn(e):  
44                             case <-done:  
45                                 return  
46                     }  
47                     case <-done:  
48                         return  
49                 }  
50             }  
51         }()  
52     }
```

## 2.1 Fan in/out

- Why do we have so many `<-done` calls?
  - Each blocking call is an opportunity for deadlock
  - `<-done` gives goroutines the chance to be stopped at any point

```
30 func (w *worker) start(  
31     done <-chan struct{},  
32     fn EventFunc, wg *sync.WaitGroup,  
33 ) {  
34     go func() {  
35         defer wg.Done()  
36         for {  
37             select {  
38                 case e, more := <-w.inputCh:  
39                     if !more {  
40                         return  
41                     }  
42                 select {  
43                     case w.outputCh <- fn(e):  
44                         case <-done:  
45                             return  
46                 }  
47                 case <-done:  
48                     return  
49             }  
50         }  
51     }()  
52 }
```

## 2.1 Fan in/out

- What happens if we do not check for !more? [p]

```
30 func (w *worker) start(  
31     done <-chan struct{},  
32     fn EventFunc, wg *sync.WaitGroup,  
33 ) {  
34     go func() {  
35         defer wg.Done()  
36         for {  
37             select {  
38                 case e, more := <-w.inputCh:  
39                     if !more {  
40                         return  
41                     }  
42                 select {  
43                     case w.outputCh <- fn(e):  
44                     case <-done:  
45                         return  
46                     }  
47                 case <-done:  
48                     return  
49             }  
50         }  
51     }()  
52 }
```

## What happens if we do not check for !more?

```
30 func (w *worker) start(  
31     done <-chan struct{},  
32     fn EventFunc, wg *sync.WaitGroup,  
33 ) {  
34     go func() {  
35         defer wg.Done()  
36         for {  
37             select {  
38                 case e, more := <-w.inputCh:  
39                     if !more {  
40                         return  
41                     }  
42                 select {  
43                     case w.outputCh <- fn(e):  
44                     case <-done:  
45                         return  
46                     }  
47                 case <-done:  
48                     return  
49             }  
50         }  
51     }()  
52 }
```

Programs runs forever

Read from a closed input  
channel, run time error

Other runtime error

Compile-time error

None of the above



## 2.1 Fan in/out

- What happens if we do not check for !more?

This wg.Wait in the main goroutine never exits, so we never close the output channel, so “infinite loop” of data

```
wg.Wait()
```

```
// Close outputCh and wait for reader to finish reading  
close(outputCh)
```

<https://fsmbolt.comp.nus.edu.sg/z/Wz5qG3>

```
30 func (w *worker) start(  
31     done <-chan struct{},  
32     fn EventFunc, wg *sync.WaitGroup,  
33 ) {  
34     go func() {  
35         defer wg.Done()  
36         for {  
37             select {  
38                 case e, more := <-w.inputCh:  
39                   
40                 We never exit this  
41                 goroutine so we  
42                 never call wg.Done  
43                   
44                 select {  
45                     case w.outputCh <- fn(e):  
46                     case <-done:  
47                         return  
48                 }  
49                 case <-done:  
50                     return  
51             }  
52         }  
53     }()  
54 }
```

After inputCh is closed, we read 0 on each read!

So we keep sending to output channel

# Serializing fan-in events

## 2.2 Fan in/out

<https://fsmbolt.comp.nus.edu.sg/z/fs4x7c>

- Turns out that the events are **not printed in order** because.. they're just pushed in whatever order they finish!
- Please suggest some ways that we can make this output serially
- Tons of possible solutions so **let's get creative!**

```
Program returned: 0
Program stdout
Event id: 7
Event id: 8
Event id: 12
Event id: 10
Event id: 6
Event id: 13
Event id: 9
Event id: 11
Event id: 4
Event id: 2
Event id: 17
Event id: 3
Event id: 5
Event id: 22
Event id: 24
Event id: 15
Event id: 21
Event id: 1
Event id: 23
Event id: 14
Event id: 18
Event id: 16
Event id: 28
Event id: 25
Event id: 19
Event id: 20
Event id: 27
Event id: 26
Event id: 30
Event id: 29
```

Go 1.20 i - 298ms

## 2.2 Fan in/out

- Joke solution: Sleepsort

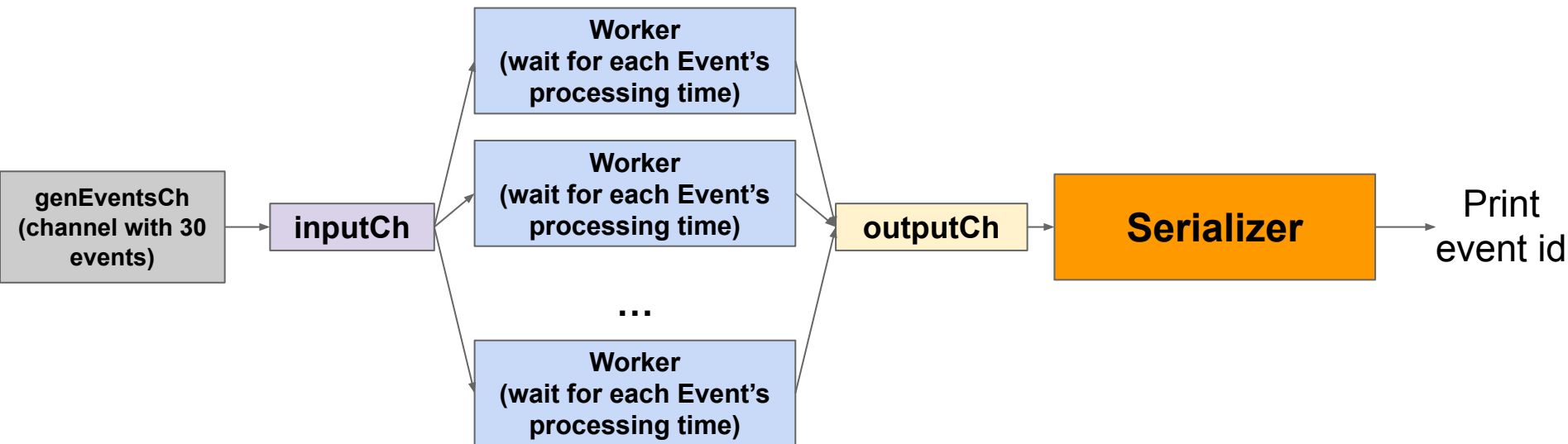
<https://fsmbolt.comp.nus.edu.sg/z/roGPjM>

- Why does this “work”?



## 2.2 Fan in/out

- My “**Serializer**” solution
- A goroutine that **re-orders the events** as they come in
- **How?**

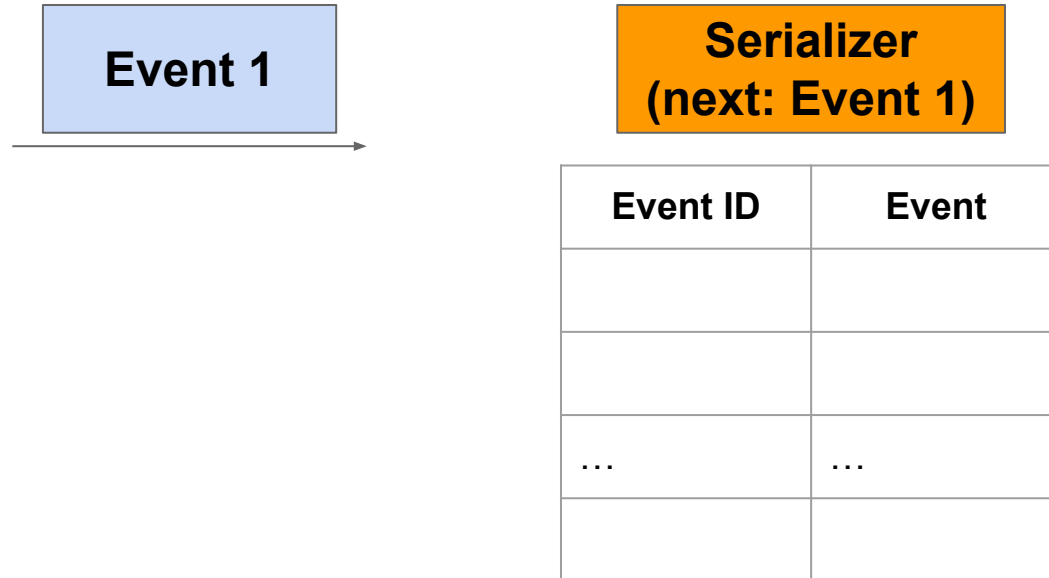


## 2.2 Fan in/out

- One option is just to
  - Read all events
  - Sort by id when the output channel is closed
  - Print them out
- **But this is pretty lame - why?**
  - **All events are printed only when the last event arrives - very slow.**

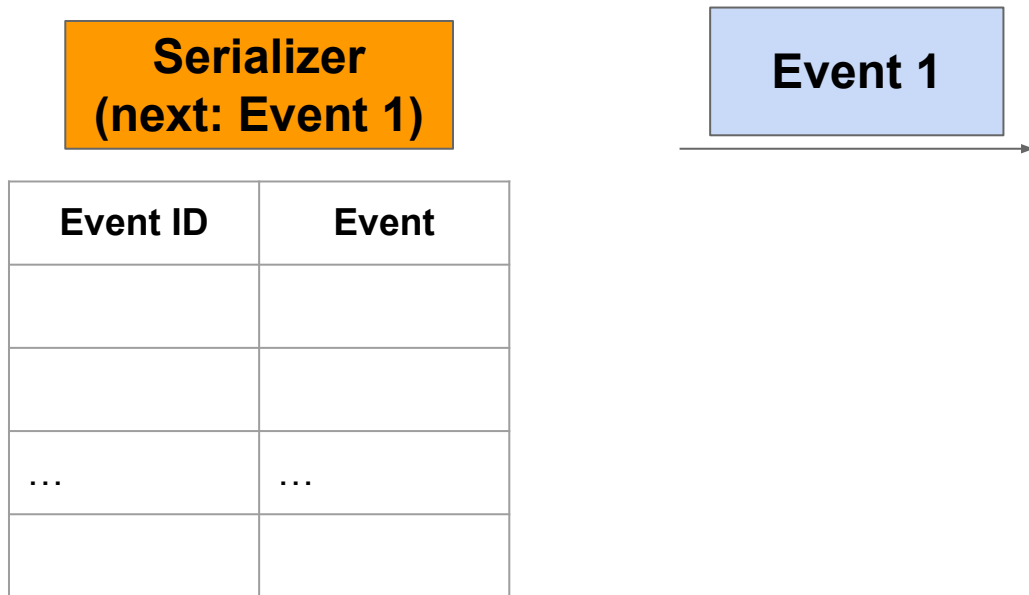
## 2.2 Fan in/out

- Better serializer: **print events when they arrive in-order**
- Store events that are out of order, send them later as necessary



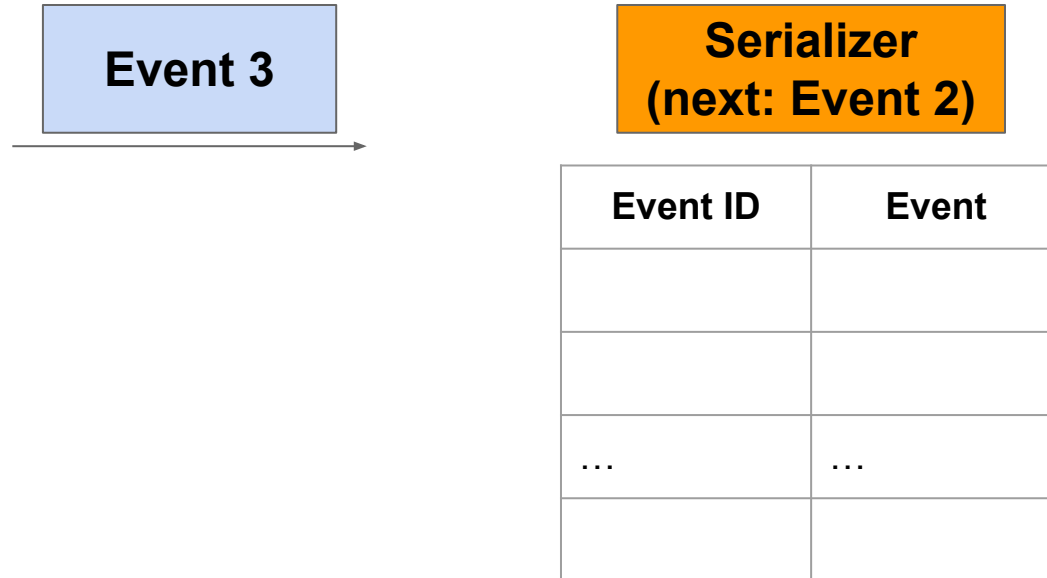
## 2.2 Fan in/out

- Better serializer: **print events when they arrive in-order**
- Store events that are out of order, send them later as necessary



## 2.2 Fan in/out

- Better serializer: **print events when they arrive in-order**
- Store events that are out of order, send them later as necessary



## 2.2 Fan in/out

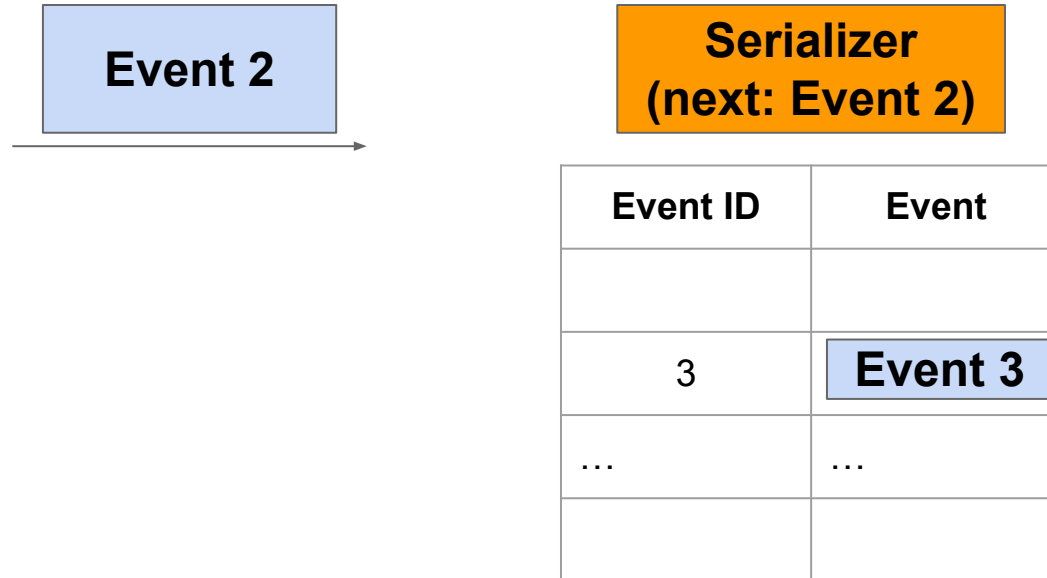
- Better serializer: **print events when they arrive in-order**
- Store events that are out of order, send them later as necessary

**Serializer**  
**(next: Event 2)**

Event ID	Event
3	<b>Event 3</b>
...	...

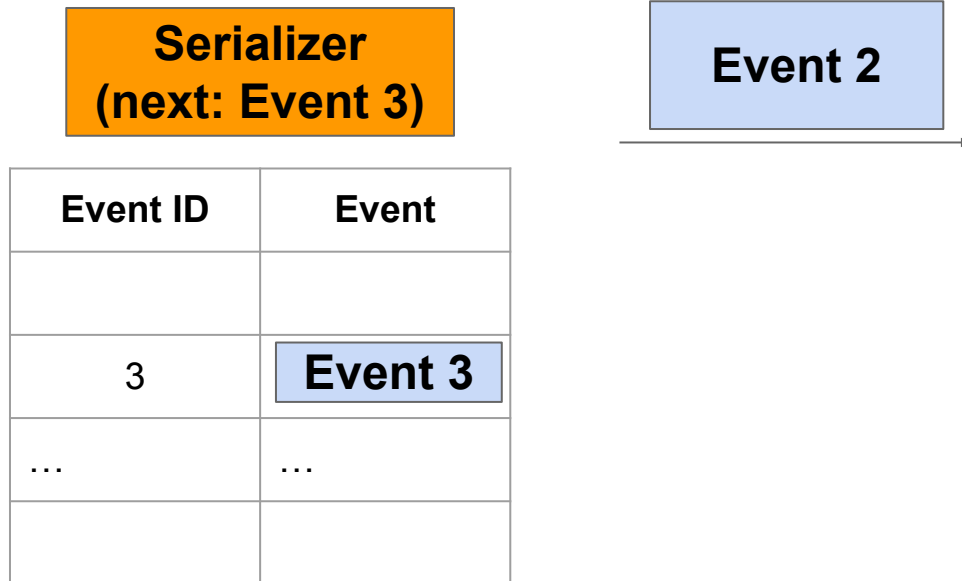
## 2.2 Fan in/out

- Better serializer: **print events when they arrive in-order**
- Store events that are out of order, send them later as necessary



## 2.2 Fan in/out

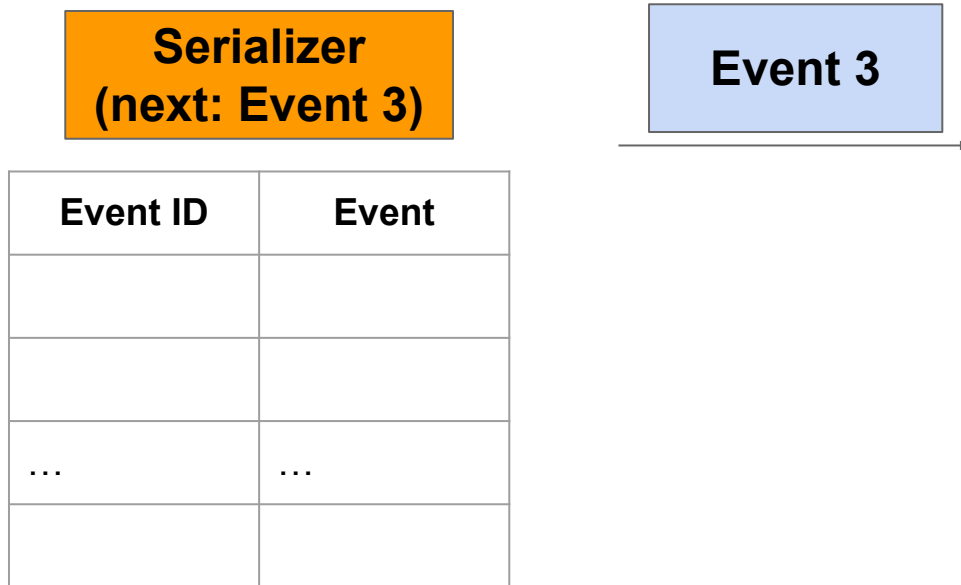
- Better serializer: **print events when they arrive in-order**
- Store events that are out of order, send them later as necessary





## 2.2 Fan in/out

- Better serializer: **print events when they arrive in-order**
- Store events that are out of order, send them later as necessary



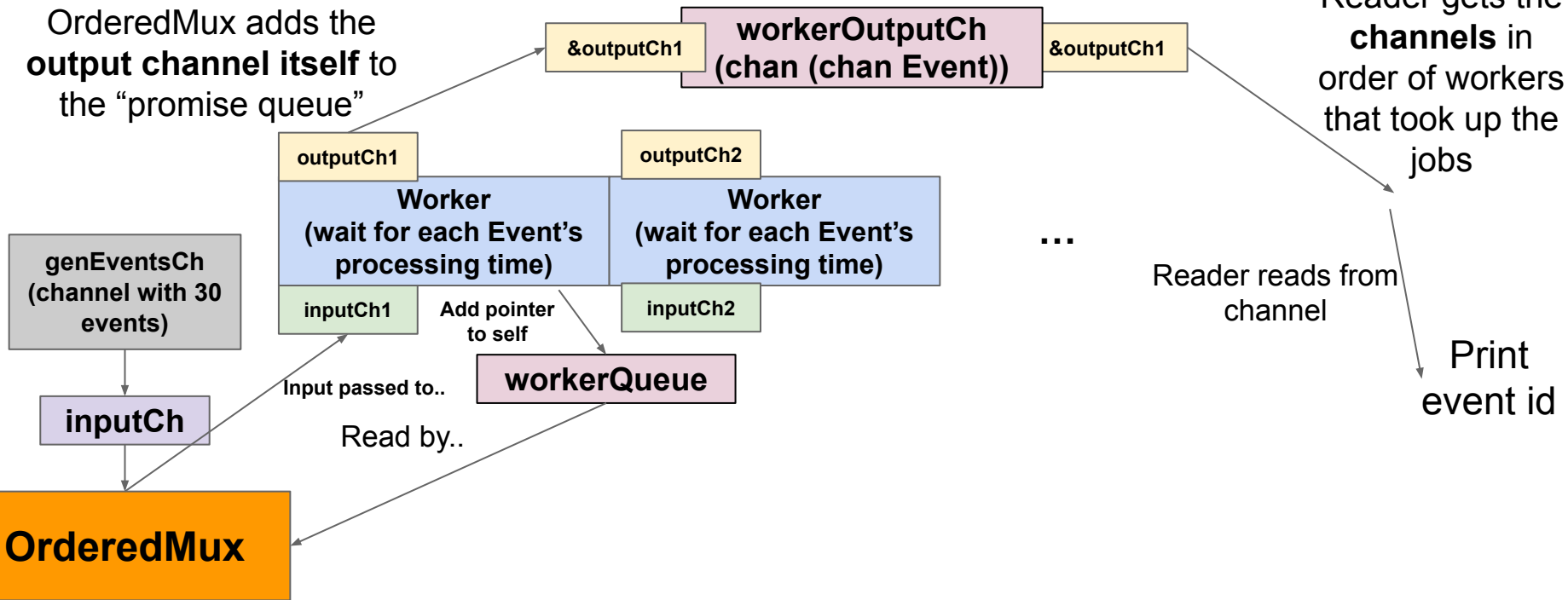
# Higher order channels and “promises”

## 2.3 Higher-order channels

<https://fsmbolt.comp.nus.edu.sg/z/h58E6j>

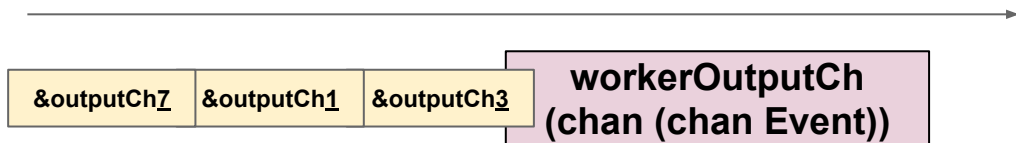
- Goroutine “orderedmux” passes **channels to be read from**, in order, to reader

OrderedMux adds the **output channel itself** to the “promise queue”



## 2.3 Higher-order channels

- To reiterate: why does this work?
- Imagine if:
  - Worker 3 picked up event ID 1
  - Worker 1 picked up event ID 2
  - Worker 7 picked up event ID 3
- **Channels will be read by reader goroutine in this order!**



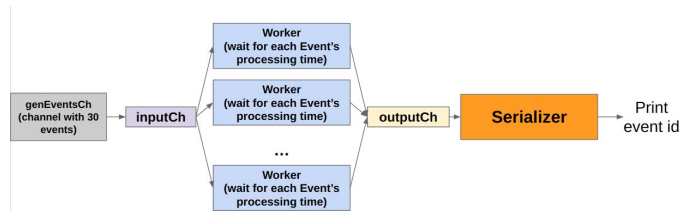
# Why does this question matter?

2.1: Highlighting complexity of exiting from cooperating threading models

2.2: Nuances of fan-out/in pattern – **easy to fan-out (distribute work)** but **not so easy to fan-in sometimes (collect results sensibly)**

2.3: You can build **complex behavior** (e.g., ordered queue) with **channels alone!**

```
30 func (w *worker) start(  
31     done <-chan struct{},  
32     fn EventFunc, wg *sync.WaitGroup,  
33 ) {  
34     go func() {  
35         defer wg.Done()  
36         for {  
37             select {  
38                 case e, more := <-w.inputCh:  
39                     if !more {  
40                         return  
41                     }  
42                     select {  
43                         case w.outputCh <- fn(e):  
44                             case <-done:  
45                                 return  
46                     }  
47                 case <-done:  
48                     return  
49             }  
50         }  
51     }()  
52 }
```



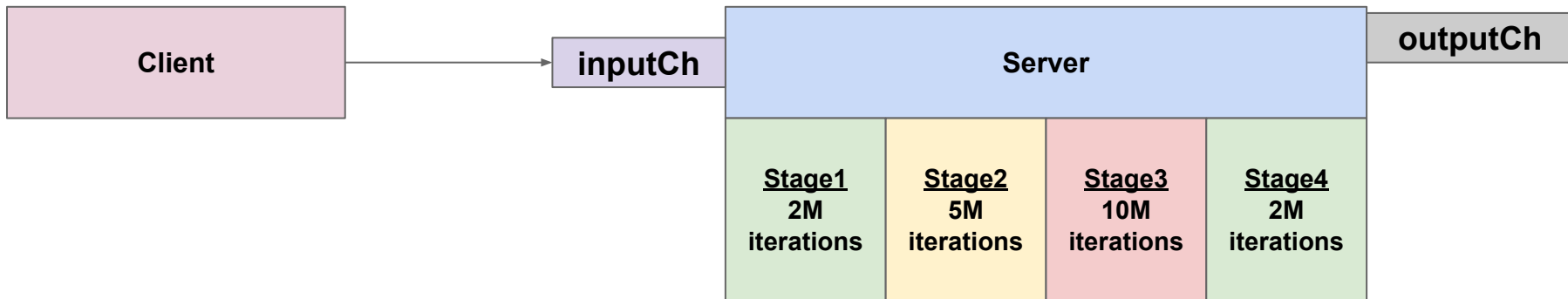
# Pipelining

# 3.1 Typical Client-Server

<https://fsmbolt.comp.nus.edu.sg/z/W4eqTG>

- Client sends requests to server
- Server processes it sequentially through multiple stages
  - Separation of responsibilities, etc

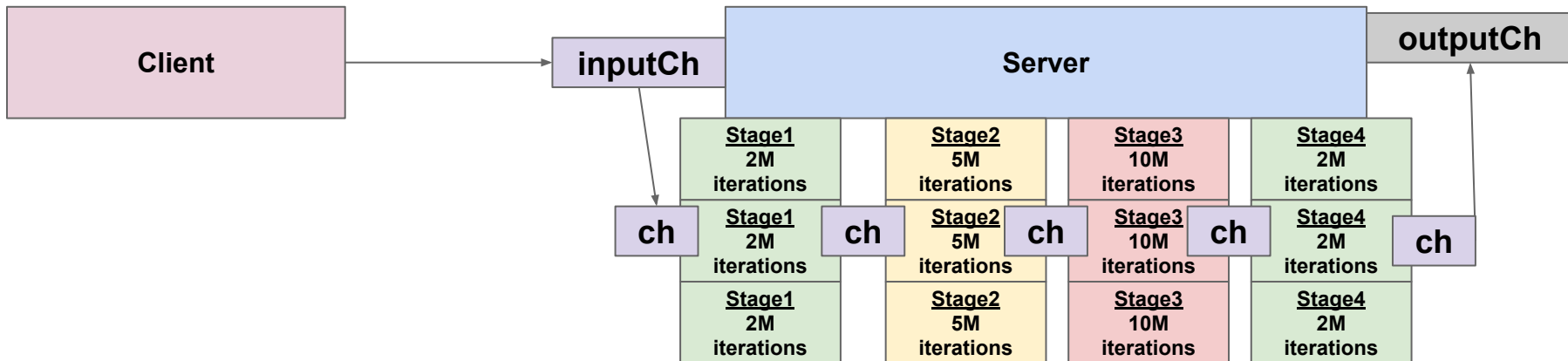
Sending requests



## 3.2 Pipelining

- Parallelism strategy: **let's pipeline the stages!**
- Goroutines in each stage read from same input channel and output to same output channel
- **If we have a limited no. of goroutines, how to allocate? [p]**

Sending requests





**If we have a limited number of goroutines, in what ratio should we allocate them to each pipeline stage?**

2 : 5 : 10 : 2

$1/2 : 1/5 : 1/10 : 1/2$

1 : 1 : 1 : 1

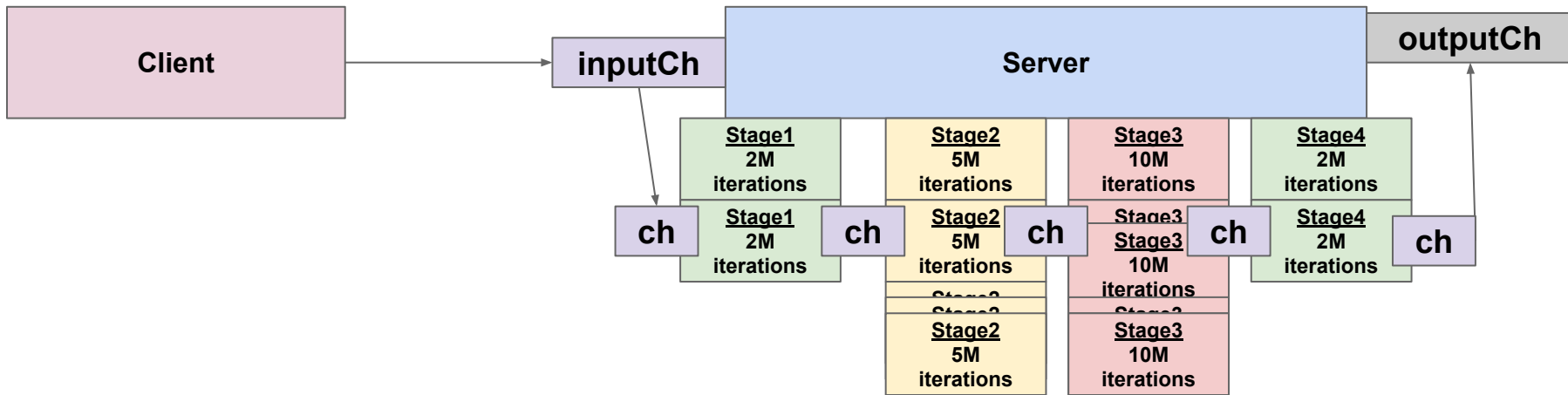
Doesn't matter



## 3.2 Pipelining

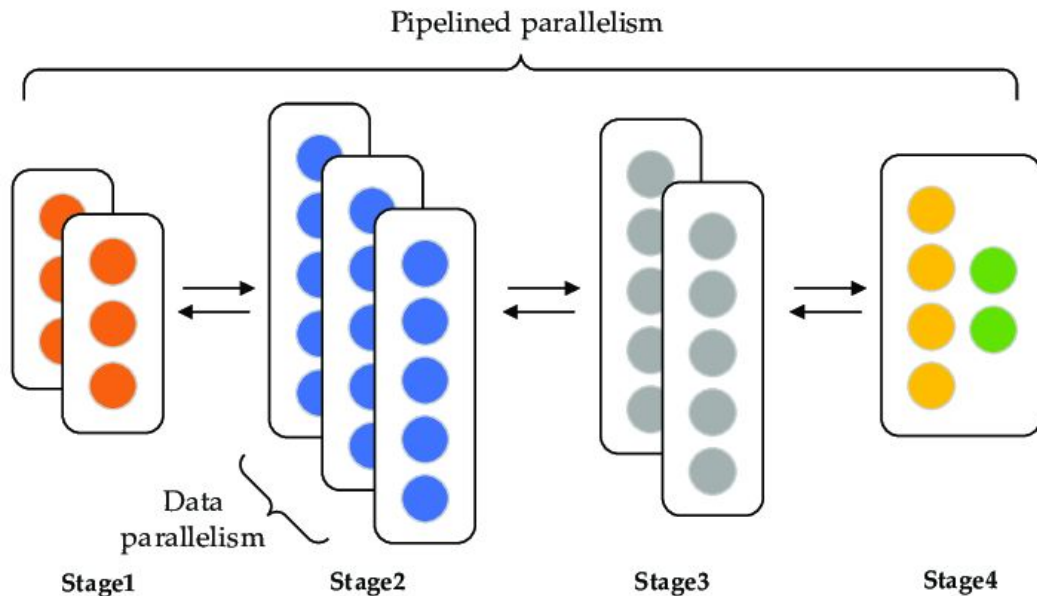
- If we have a limited no. of goroutines, how to allocate? [p]
- More intensive stages should get more resources
  - 2 : 5 : 10 : 2 will spread the load the most evenly
- Let's change no. of stages in <https://fsmbolt.comp.nus.edu.sg/z/aqqahc>

Sending requests



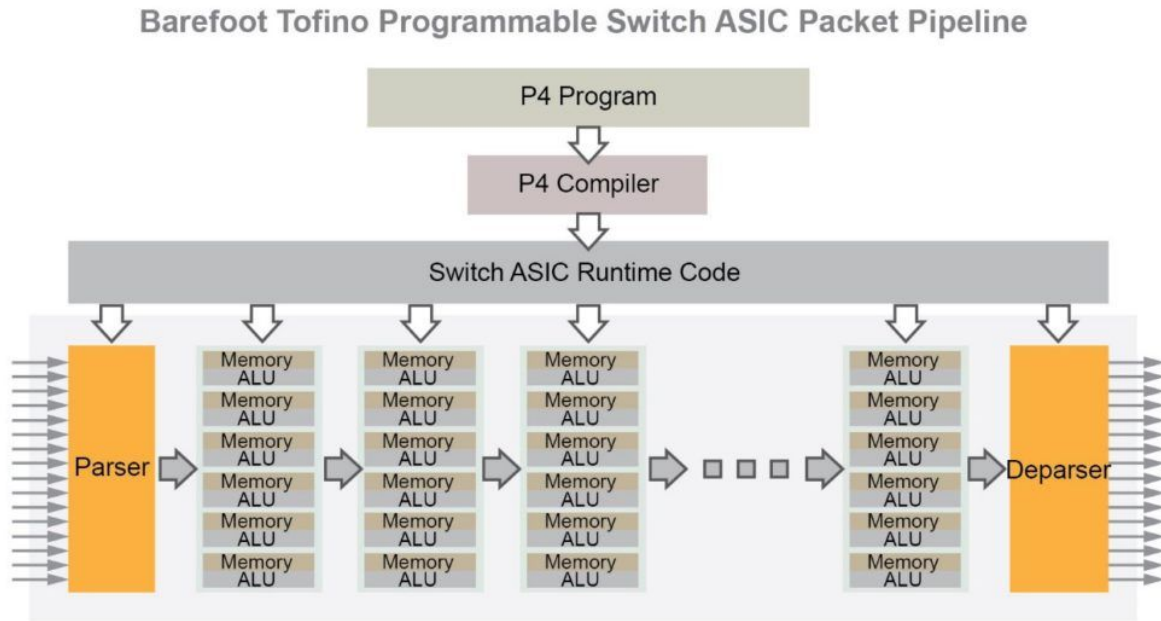
# Why does this question matter?

- Pipelining is useful in **certain contexts**: for **resource constrained** systems that need to carefully allocate parallelism resources



# Why does this question matter?

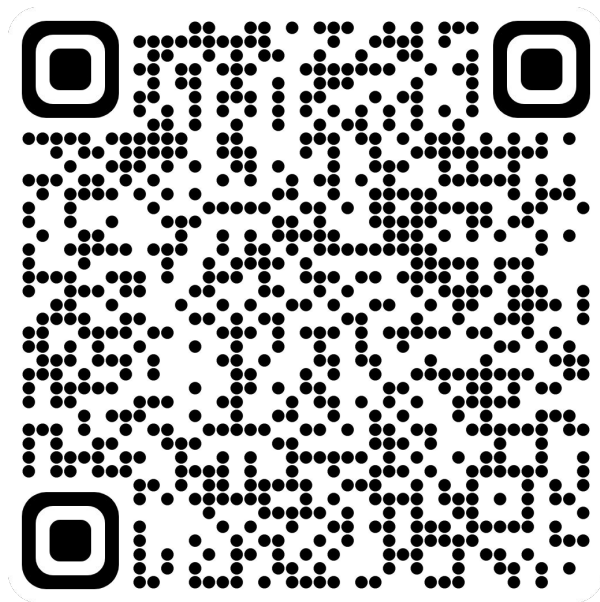
- Lots of our **crucial hardware** (e.g., network switches @ 12.4 TB/s) use **pipelining** (e.g., pipelined packet processing)!



# Summary

- **Contexts** are useful for **managing complex exit conditions** and **sharing data** across **different goroutines**
- **Fan-out** is useful for distributing intense work to many threads
- **Fan-in** is necessary to centralize results, but may need some thought
- **Pipelining** is useful for **resource-constrained** parallelism while maintaining a **separation of concerns** between stages

# Scan for attendance!



# See you next week!

Feedback: <https://bit.ly/sriram-feedback>  
Email: [sriramsami@nus.edu.sg](mailto:sriramsami@nus.edu.sg)

# Extra: issues with our chan chan code

- Any issues remaining in that code?



# Extra: issues with our chan chan code

<https://fsmbolt.comp.nus.edu.sg/z/h58E6j>

- **workerOutputCh is unbuffered! Limits max concurrency.**

OrderedMux adds the **output channel itself** to the “promise queue”

