CS3211 Tutorial 6

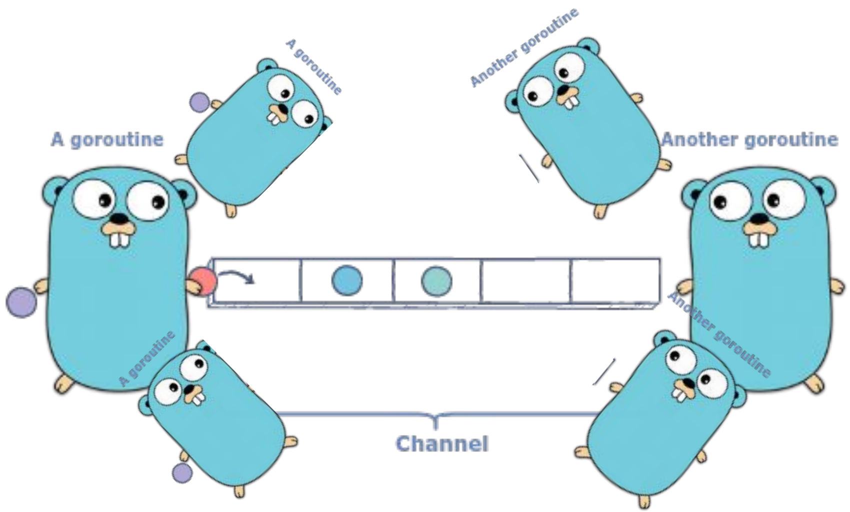
Advanced Go concurrency patterns

What we've covered

- **Motivation:** concurrency without shared memory

- Channels
  - Unbuffered vs buffered
  - Pushing into a single channel vs using per-goroutine channels
  - etc

- Waitgroups
  - Handling synchronized exits from multiple goroutines
  - etc



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

(maybe some useful techniques for Assignment 2?)

Exit Conditions and Context

1. Exit Conditions

- We have **many goroutines** that can run at the same time

- How do we manage their **myriad exit conditions?**
    - **Note:** you **cannot forcibly (non-cooperatively) kill a goroutine from another goroutine!**

1. Exit Conditions

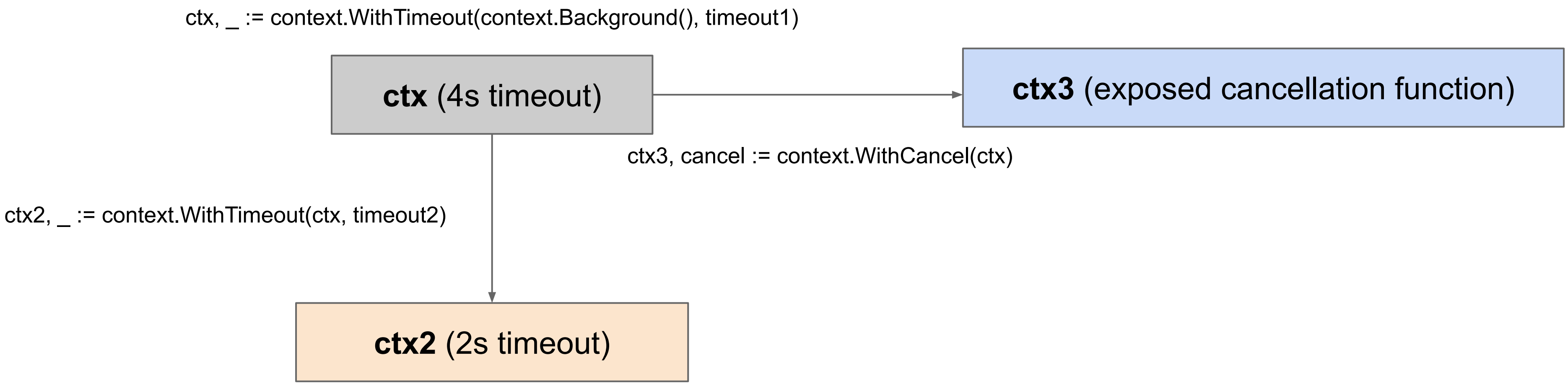
- Imagine if we have N goroutines

- **Goroutine 1 (main):** must **stop** after **4 seconds**
  - E.g., network request that you know cannot take longer than 4 seconds
- **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

https://fsmbolt.comp.nus.edu.sg/z/8oj3Te

1. Exit Conditions

- Solution: we can build a **context tree!**



1. Exit Conditions

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

```
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?**

- **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.

- 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

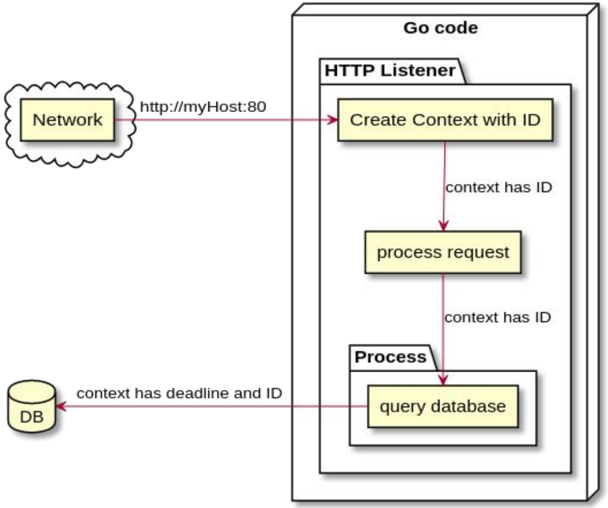
1. Exit Conditions

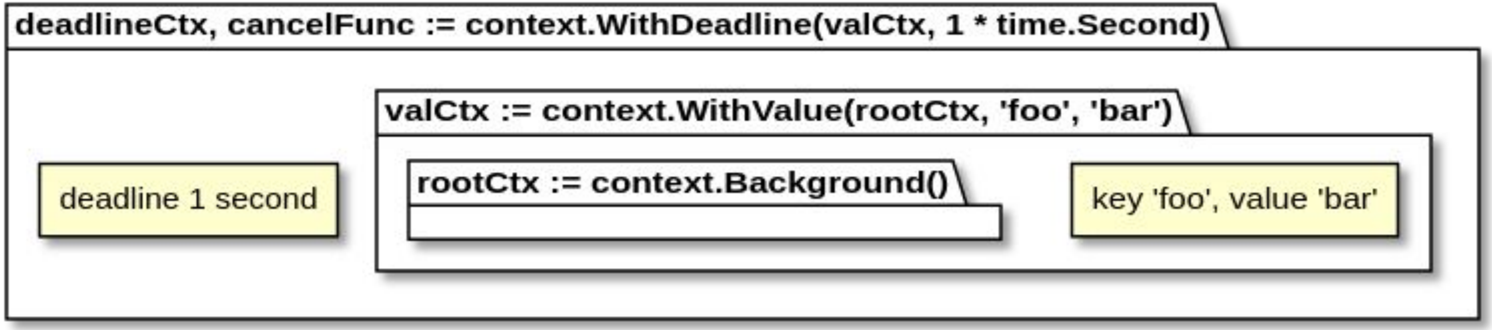
- Question: how can we ensure that the ctx3 fmt.Printf is printed?
  - Done channel solution: https://fsmbolt.comp.nus.edu.sg/z/oYTP9M
  - Waitgroup solution: https://fsmbolt.comp.nus.edu.sg/z/KnzbP6

```go
// 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.**



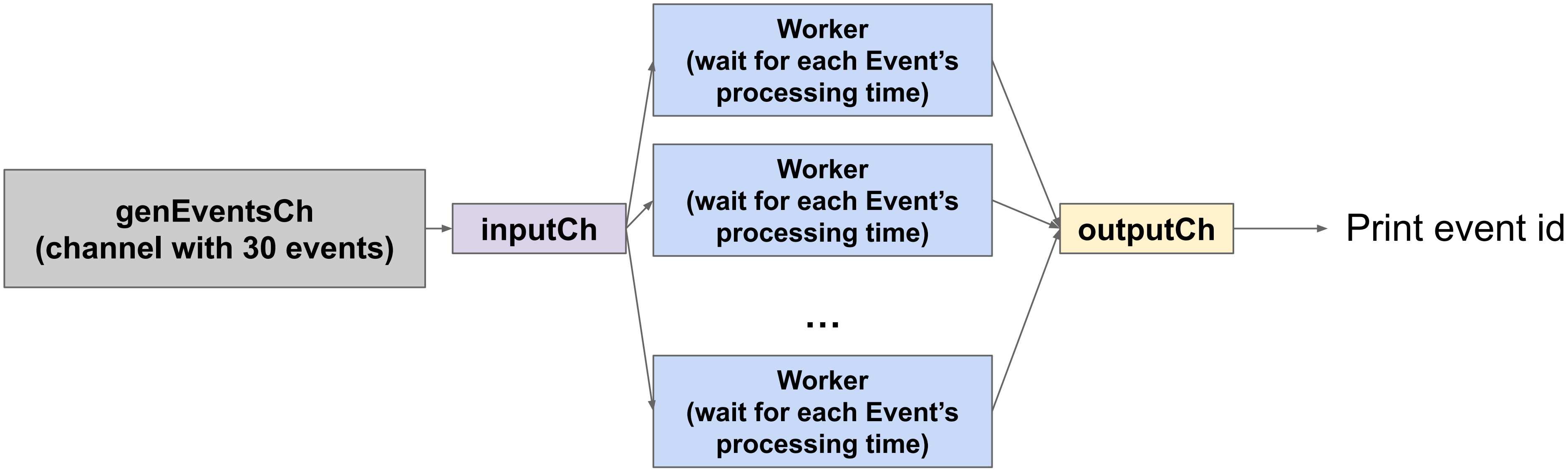


Fan-out, Fan-in

Understanding the fan out-in pattern

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

- **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



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?

```go
func (w *worker) start(
	done <-chan struct{},
	fn EventFunc, wg *sync.WaitGroup,
) {
	go func() {
		defer wg.Done()
		for {
			select {
			case e, more := <-w.inputCh:
				if !more {
					return
				}
				select {
				case w.outputCh <- fn(e):
				case <-done:
					return
				}
			case <-done:
				return
			}
		}
	}()
}
```

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

```go
func (w *worker) start(
	done <-chan struct{},
	fn EventFunc, wg *sync.WaitGroup,
) {
	go func() {
		defer wg.Done()
		for {
			select {
			case e, more := <-w.inputCh:
				if !more {
					return
				}
				select {
				case w.outputCh <- fn(e):
				case <-done:
					return
				}
			case <-done:
				return
			}
		}
	}()
}
```

2.1 Fan in/out

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

```go
func (w *worker) start(
	done <-chan struct{},
	fn EventFunc, wg *sync.WaitGroup,
) {
	go func() {
		defer wg.Done()
		for {
			select {
			case e, more := <-w.inputCh:
				if !more {
					return
				}
				select {
				case w.outputCh <- fn(e):
				case <-done:
					return
				}
			case <-done:
				return
			}
		}
	}()
}
```

2.1 Fan in/out

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

```go
func (w *worker) start(
    done <-chan struct{},
    fn EventFunc, wg *sync.WaitGroup,
) {
    go func() {
        defer wg.Done()
        for {
            select {
            case e, more := <-w.inputCh:
                [redacted]
                select {
                case w.outputCh <- fn(e):
                case <-done:
                    return
                }
            case <-done:
                return
            }
        }
    }()
}
```

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

We never exit this goroutine so we never call wg.Done

So we keep sending to output channel

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

```go
wg.Wait()

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

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

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
```

Go 1.20 - 298ms

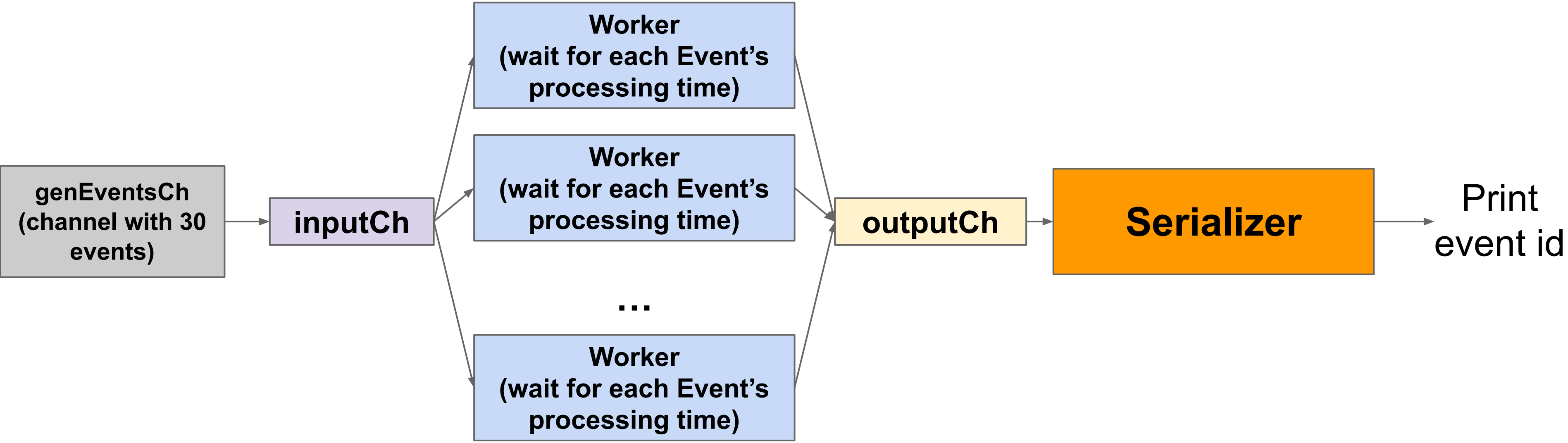
2.2 Fan in/out

- 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

Event ID	Event
…	…

https://fsmbolt.comp.nus.edu.sg/z/zq7x49

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 1)

Event 1

Event ID	Event
…	…

2.2 Fan in/out

https://fsmbolt.comp.nus.edu.sg/z/zq7x49

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

Event 3

Serializer (next: Event 2)

Event ID	Event
…	…

https://fsmbolt.comp.nus.edu.sg/z/zq7x49

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	Event 3
…	…

https://fsmbolt.comp.nus.edu.sg/z/zq7x49

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

Event 2 →

Serializer (next: Event 2)

Event ID	Event
3	Event 3
…	…

https://fsmbolt.comp.nus.edu.sg/z/zq7x49

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

Event ID	Event
3	Event 3
…	…

https://fsmbolt.comp.nus.edu.sg/z/zq7x49

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

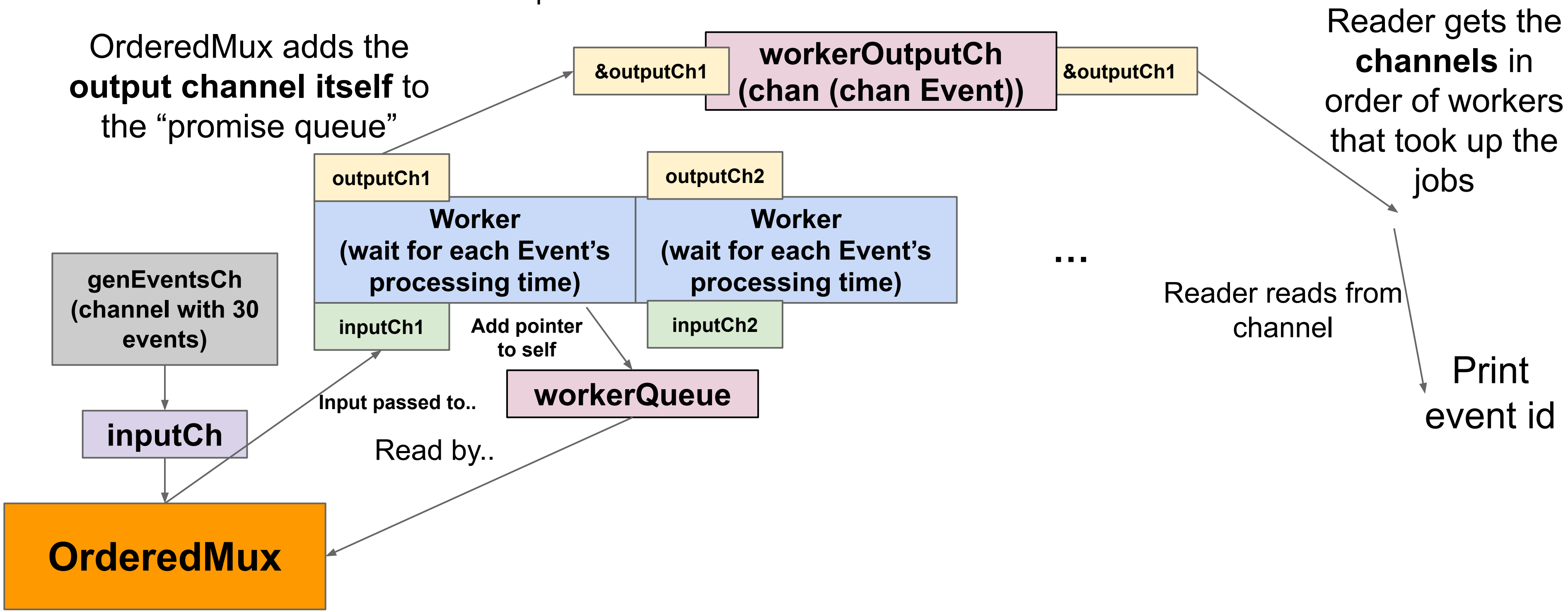
Event ID	Event
…	…

Higher order channels and “promises”

2.3 Higher-order channels

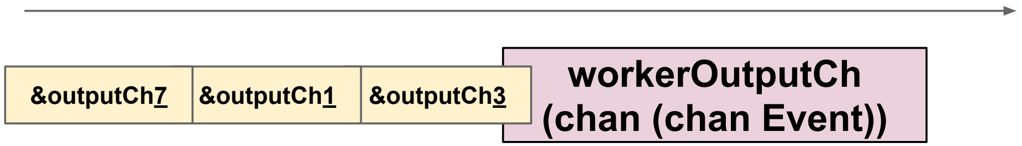
https://fsmbolt.comp.nus.edu.sg/z/h58E6j

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



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!**



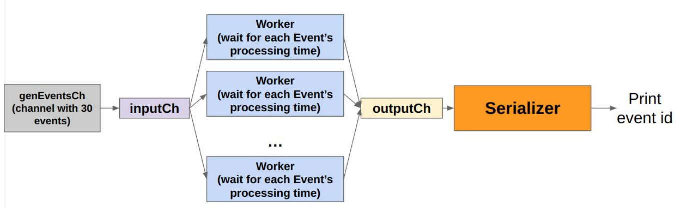
Why does this question matter?

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!**

```go
func (w *worker) start(
    done <-chan struct{},
    fn EventFunc, wg *sync.WaitGroup,
) {
    go func() {
        defer wg.Done()
        for {
            select {
            case e, more := <-w.inputCh:
                if !more {
                    return
                }
                select {
                case w.outputCh <- fn(e):
                case <-done:
                    return
                }
            case <-done:
                return
            }
        }
    }()
}
```

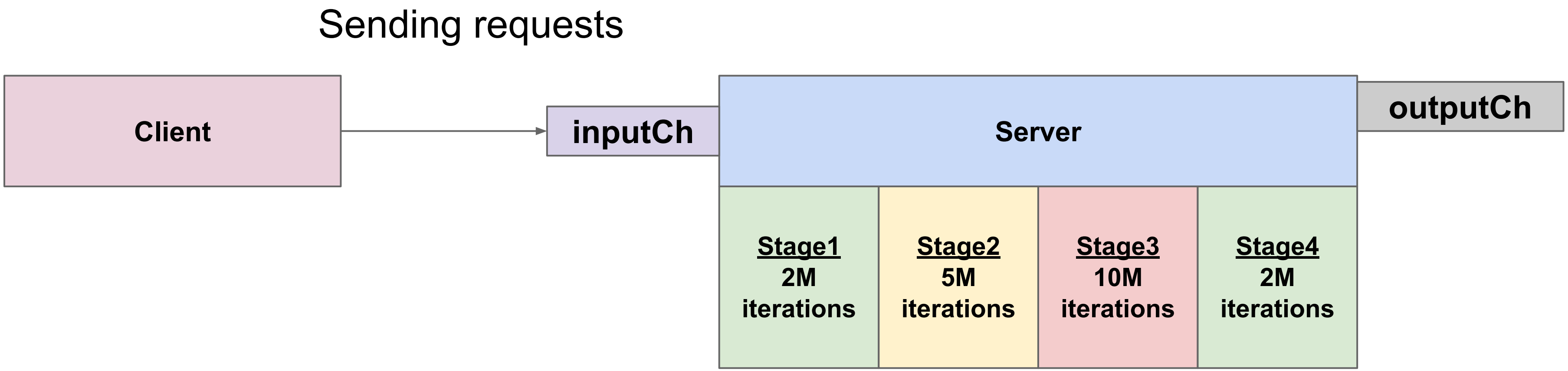


Pipelining

3.1 Typical Client-Server

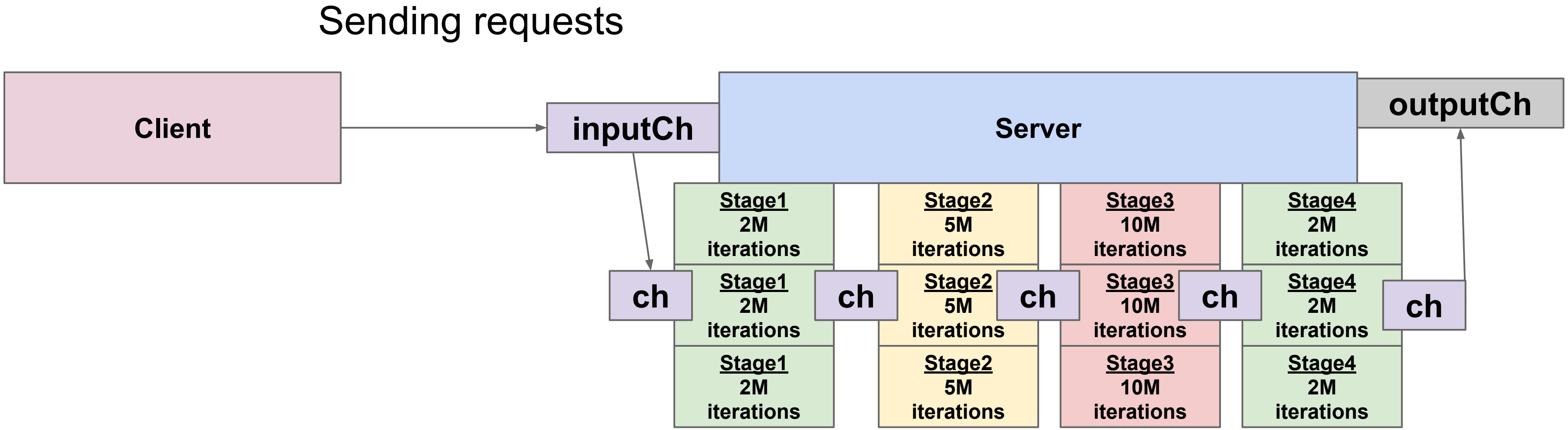
https://fsmbolt.comp.nus.edu.sg/z/W4eqTG

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



3.2 Pipelining

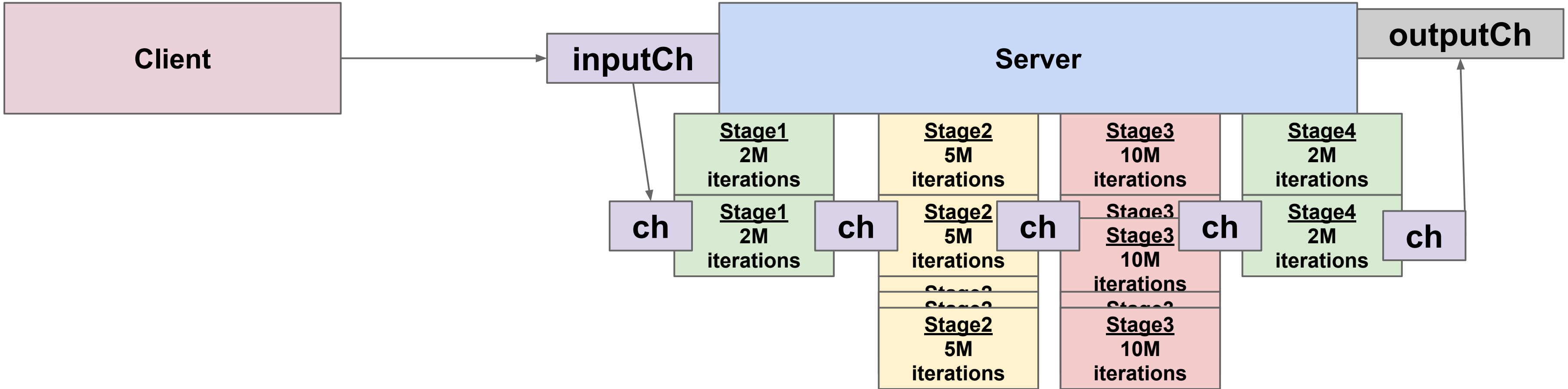
- Parallelism strategy: **let's pipeline the stages!**
- Goroutines in each stage read from same input channel and output to same output channel
- **If we have a limited no. of goroutines, how to allocate?**



3.2 Pipelining

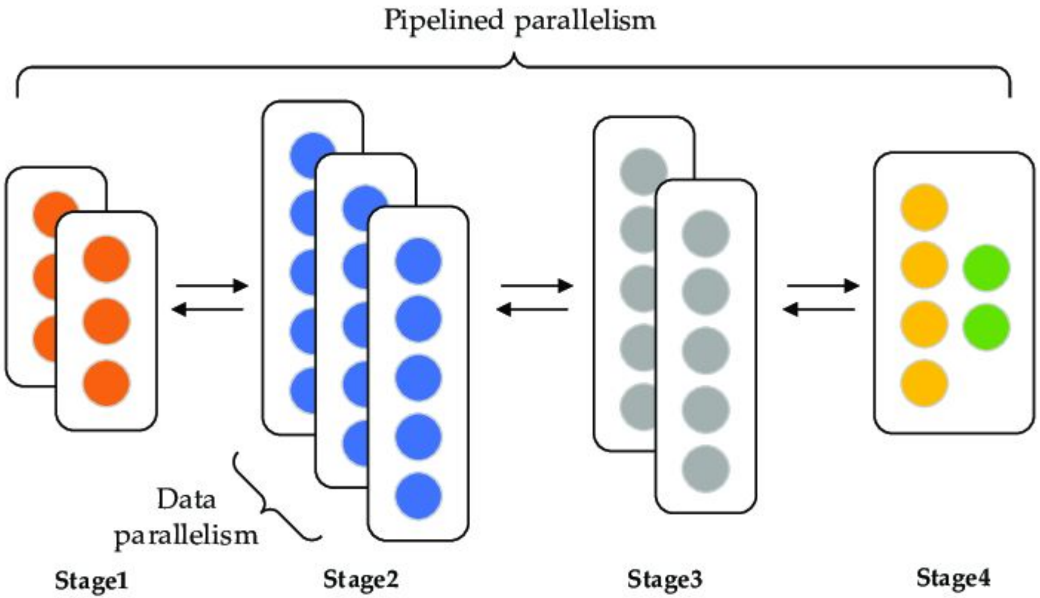
- If we have a limited no. of goroutines, how to allocate?
- More intensive stages should get more resources
  - 2 : 5 : 10 : 2 will spread the load the most evenly
- Let's change no. of stages in https://fsmbolt.comp.nus.edu.sg/z/aqqahc

Sending requests



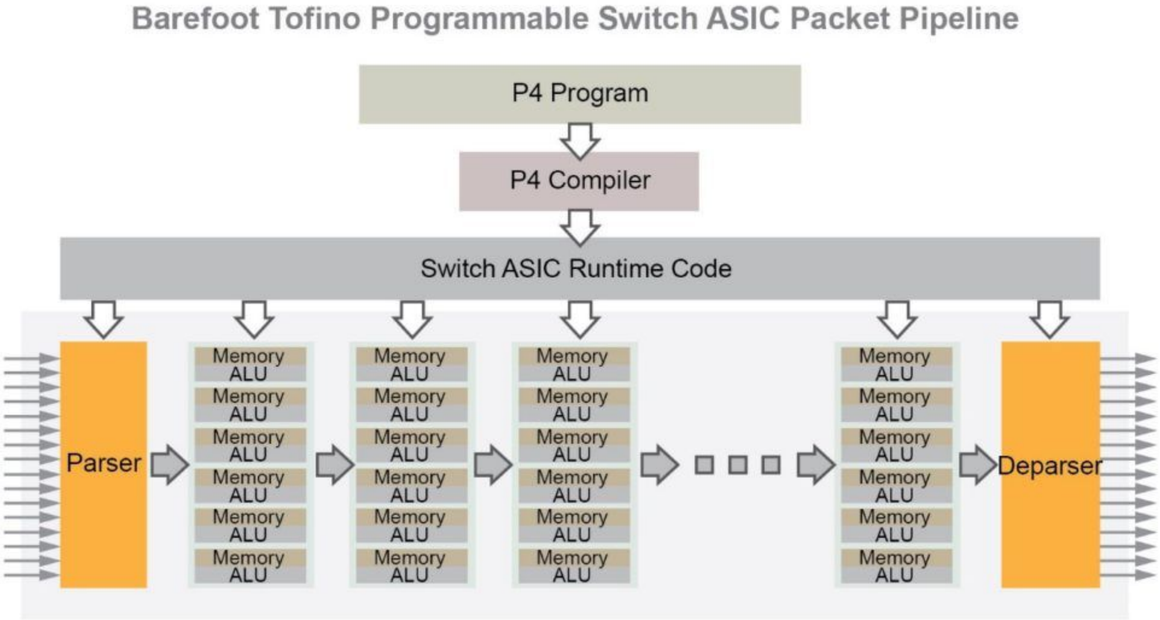
Why does this question matter?

- 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

See you next week!

Extra: issues with our <u>chan chan</u> 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.