CS3211 Tutorial 5

Goroutines and Channels
Simon

Adapted From Sriram's Slides

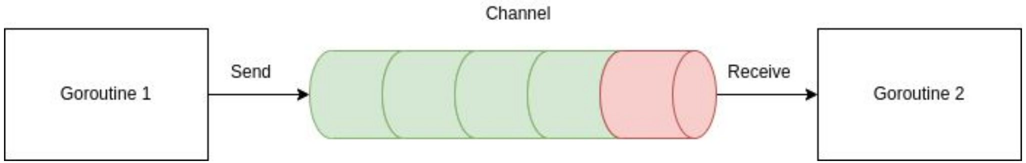
Why Go?

Message Passing as a First-Class Citizen

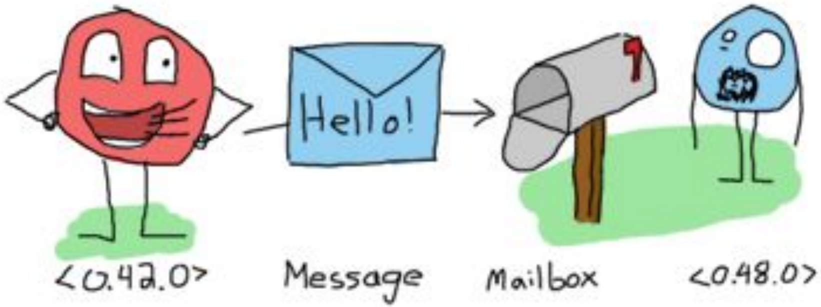
- Shared memory ⇒ problem

- So: **no shared memory ⇒ no problem?**
  - **Hah. (also P ⇒ Q <> !P ⇒ !Q)**

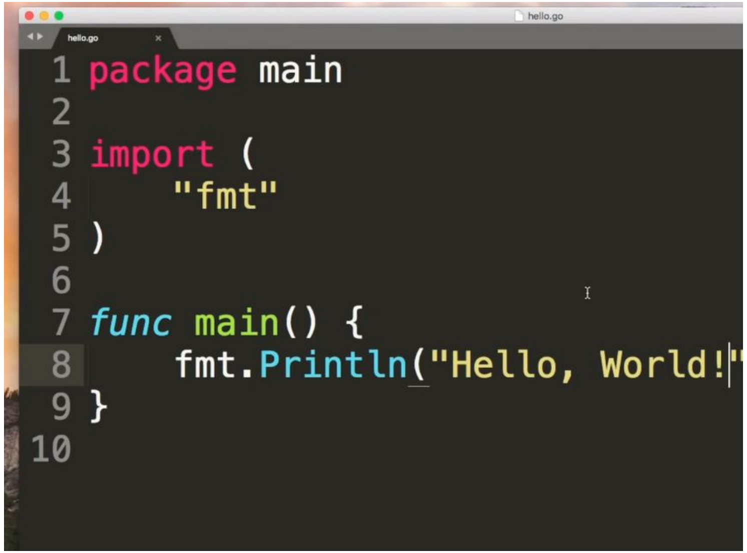
- Why do we use channel when we have mutexes?
  - Because it provides a more elegant concurrency structure (argued by CSP)
  - *Do not communicate by sharing memory; instead, share memory by communicating. (Effective Go)*
  - https://go.dev/blog/codelab-share
  - http://www.usingcsp.com/



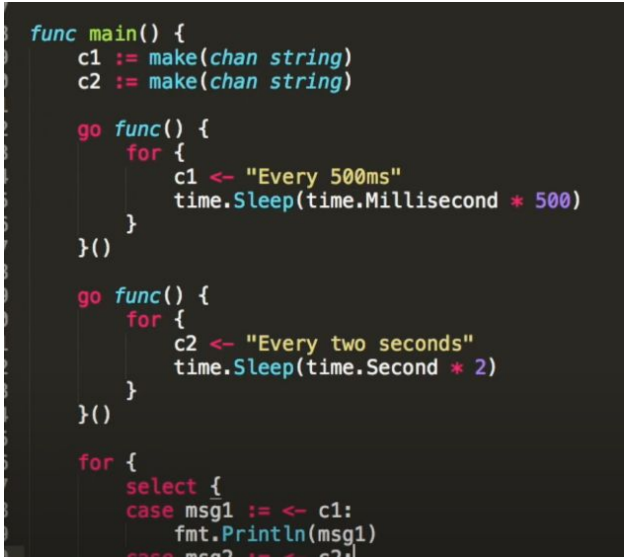
Golang



Two Useful Videos



```
package main

import (
	"fmt"
)

func main() {
	fmt.Println("Hello, World!"
}
```



```
func main() {
	c1 := make(chan string)
	c2 := make(chan string)

	go func() {
		for {
			c1 <- "Every 500ms"
			time.Sleep(time.Millisecond * 500)
		}
	}()

	go func() {
		for {
			c2 <- "Every two seconds"
			time.Sleep(time.Second * 2)
		}
	}()

	for {
		select {
		case msg1 := <- c1:
			fmt.Println(msg1)
```

[Learn Go in 12 Minutes]

[Concurrency in Go]

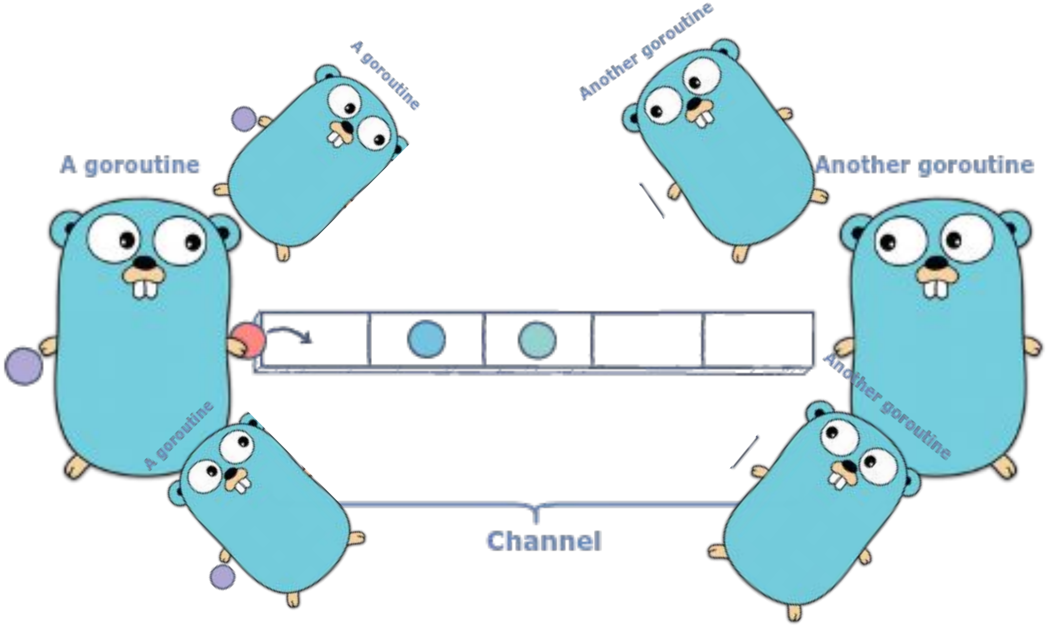
Go’s Special Sauce: Inbuilt...

- Channels

- Supporting characters (non-exhaustive)
  - Waitgroups
  - Range, (v, ok)...
  - select, select-default
  - Defer

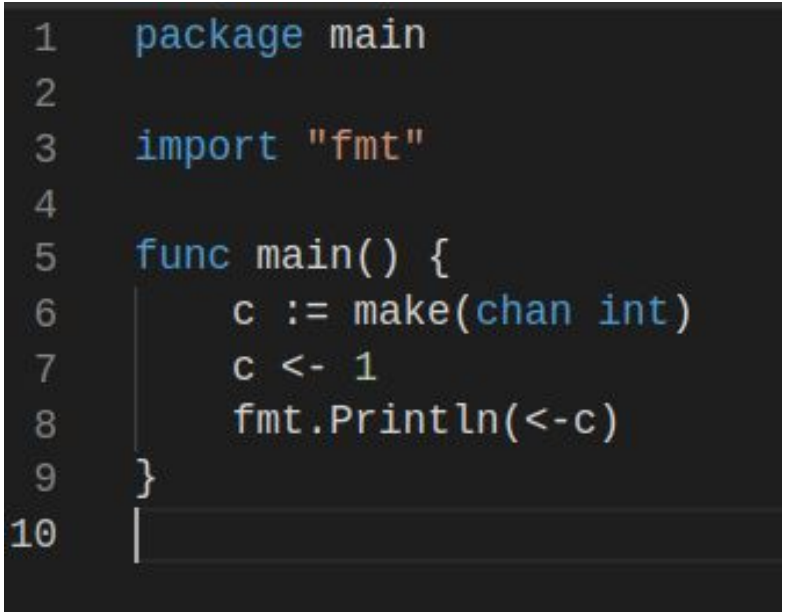
Go Channels

- Essentially, a native thread-safe **multi-producer, multi-consumer "queue"!**



Exploring Channels

- What is the output of this program? [p]



```go
package main

import "fmt"

func main() {
	c := make(chan int)
	c <- 1
	fmt.Println(<-c)
}
```

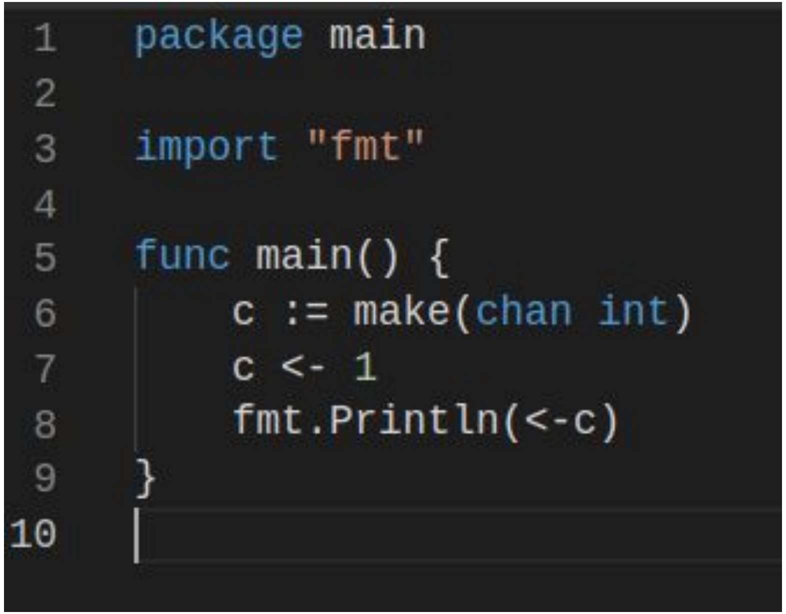
Exploring Channels

- What is the output of this program? [p]

- Deadlock!
  - Main() blocks on channel send since noone to receive yet

  - Any simple ways to avoid this?

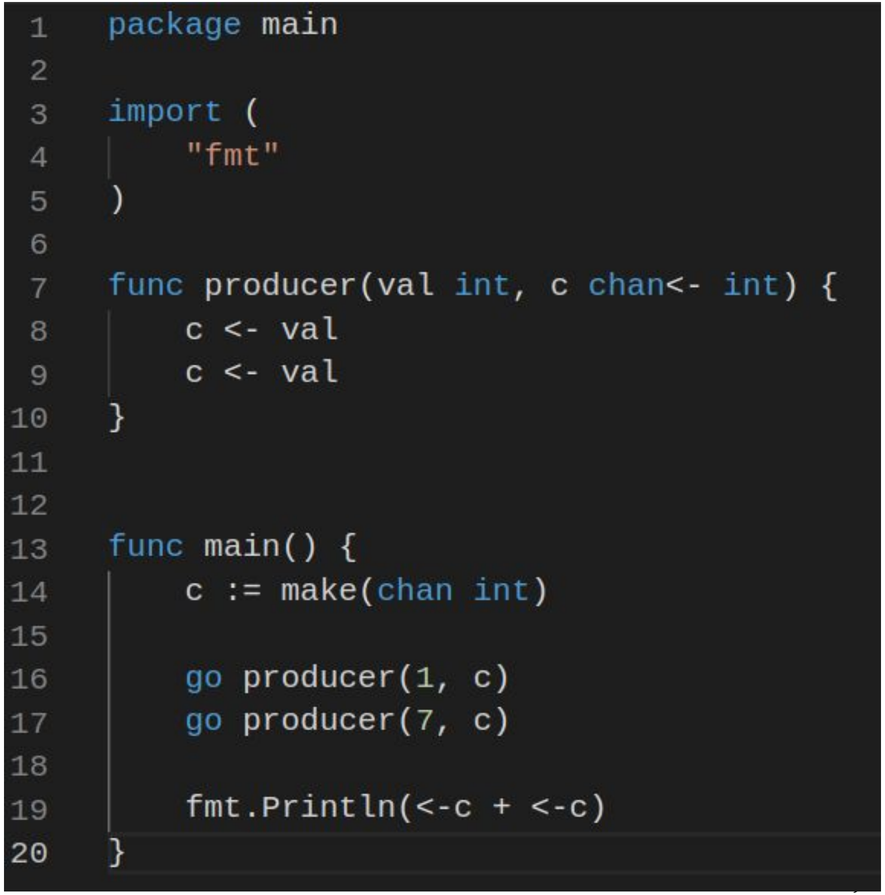
https://fsmbolt.comp.nus.edu.sg/z/nMP766



```
package main

import "fmt"

func main() {
	c := make(chan int)
	c <- 1
	fmt.Println(<-c)
}

```

Exploring Channels

- How many possible outputs does this program have, if any? [p]



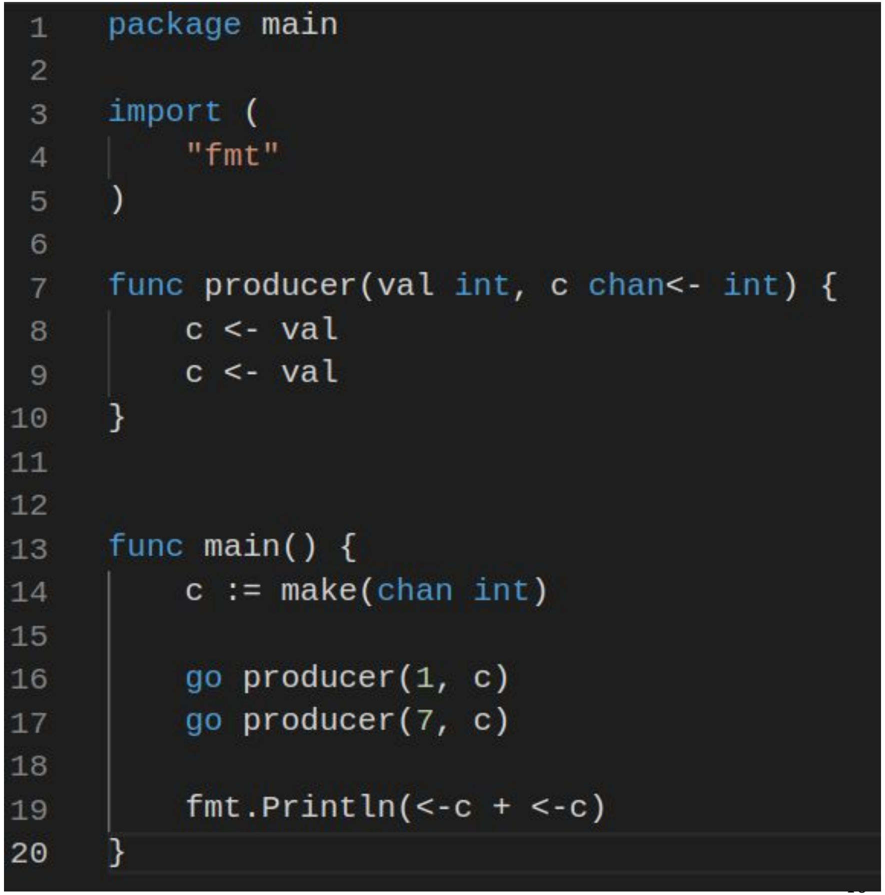
```go
package main

import (
    "fmt"
)

func producer(val int, c chan<- int) {
    c <- val
    c <- val
}


func main() {
    c := make(chan int)

    go producer(1, c)
    go producer(7, c)

    fmt.Println(<-c + <-c)
}
```

Exploring Channels

- How many possible outputs does this program have, if any? [p]

- 3 possible outputs
  - 1 + 1
  - 1 + 7 (or 7 + 1)
  - 7 + 7

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



```go
package main

import (
	"fmt"
)

func producer(val int, c chan<- int) {
	c <- val
	c <- val
}


func main() {
	c := make(chan int)

	go producer(1, c)
	go producer(7, c)

	fmt.Println(<-c + <-c)
}
```

Why does this matter?

- Channels are not an automatic solution to problems

- They can also be complex in their own ways!

Tutorial Q1.1 – 1.2

1.1 Concurrent Counter: Waitgroups

- Task: multiple threads adding to a counter

- Initial answer: https://fsmbolt.comp.nus.edu.sg/z/ch951W

- **Problem: We're not waiting for all goroutines to finish!**

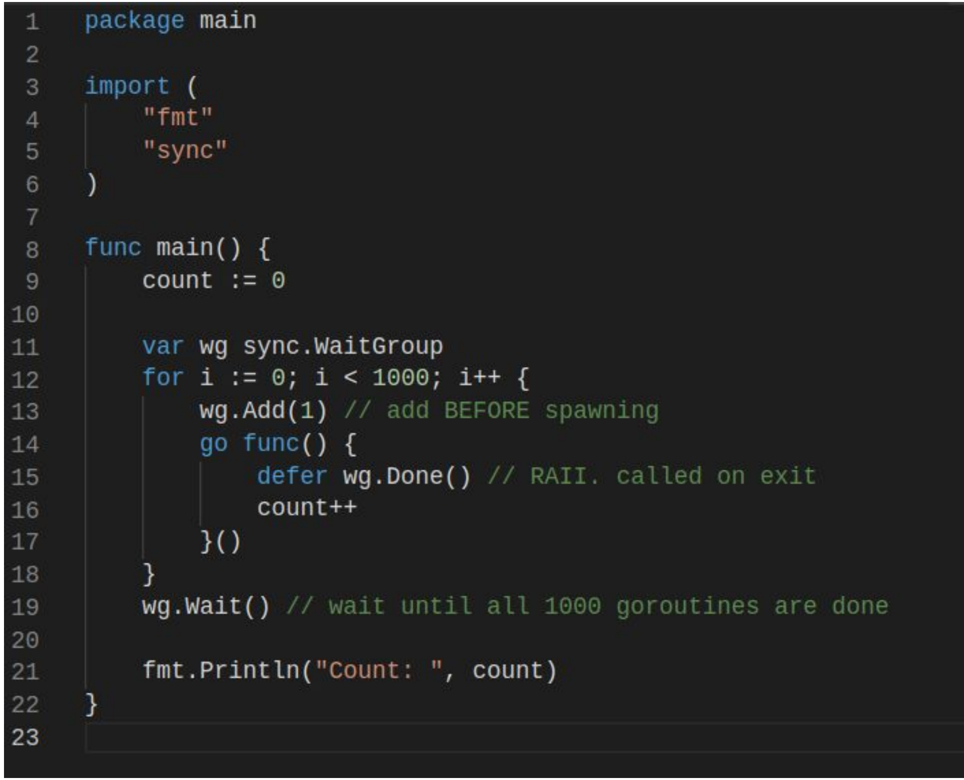
1.1 Concurrent Counter

https://fsmbolt.comp.nus.edu.sg/z/r347WE

- Concept of **sync.Waitgroup**

- A way for a thread to wait for some number of threads to finish (but more general than this)

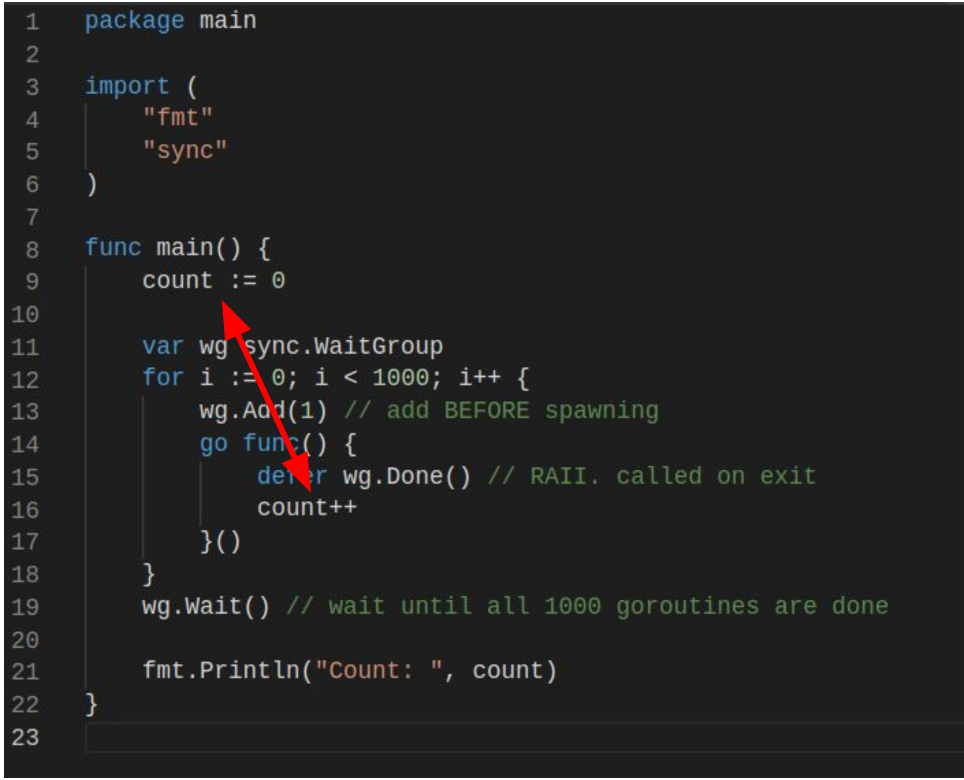
- **Problem?**



```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    count := 0

    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1) // add BEFORE spawning
        go func() {
            defer wg.Done() // RAII. called on exit
            count++
        }()
    }
    wg.Wait() // wait until all 1000 goroutines are done

    fmt.Println("Count: ", count)
}
```

1.1 Concurrent Counter

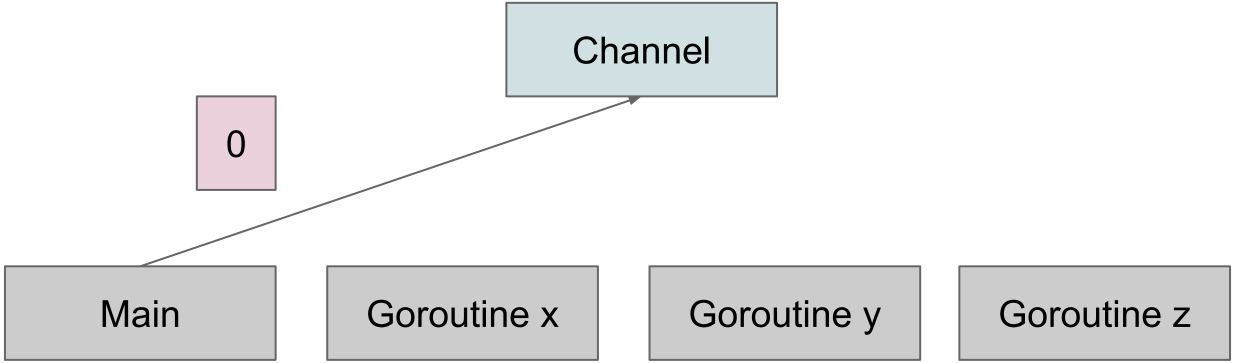
- Concept of **sync.Waitgroup**
- A way for a thread to wait for some number of threads to finish (but more general than this)
- **Problem? Straightforward data race: count++-ed by reference across threads**



```go
package main

import (
	"fmt"
	"sync"
)

func main() {
	count := 0

	var wg sync.WaitGroup
	for i := 0; i < 1000; i++ {
		wg.Add(1) // add BEFORE spawning
		go func() {
			defer wg.Done() // RAII. called on exit
			count++
		}()
	}
	wg.Wait() // wait until all 1000 goroutines are done

	fmt.Println("Count: ", count)
}
```

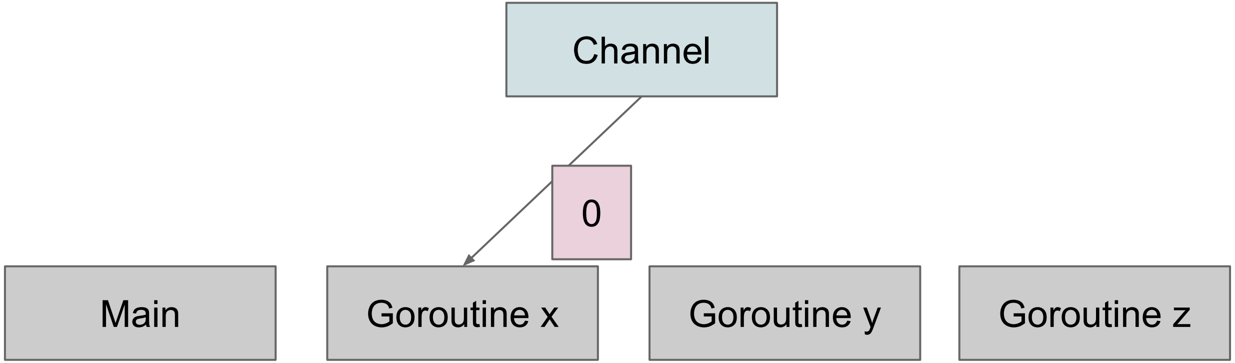
1.1 Concurrent Counter: Solution 1

- **“Channel solution”:** have an **exclusive owner** of count each time



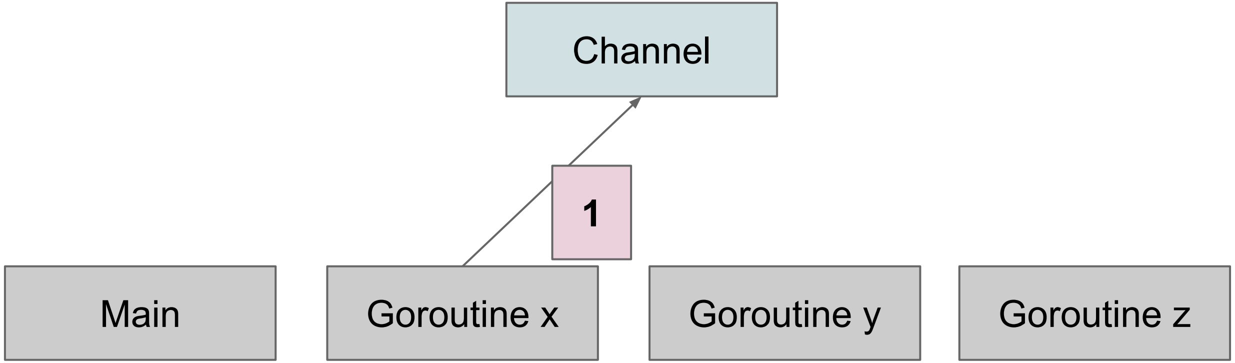
1.1 Concurrent Counter: Solution 1

- **"Channel solution":** have an **exclusive owner** of count each time



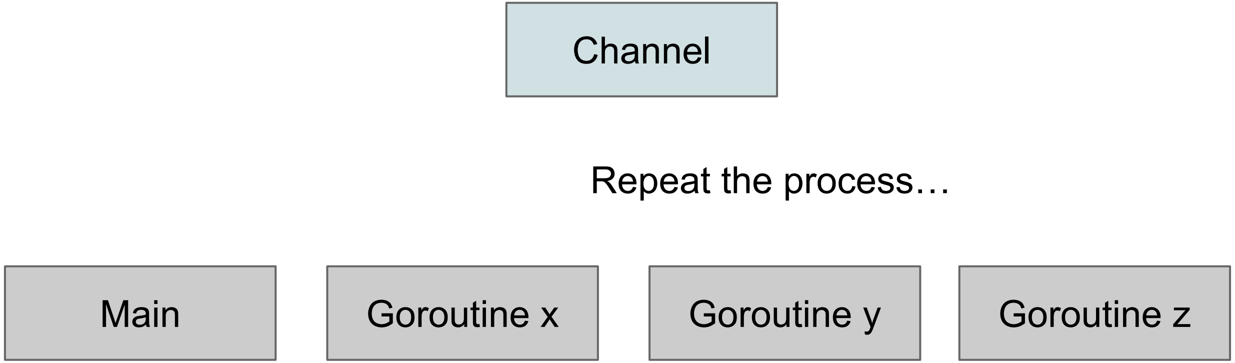
1.1 Concurrent Counter: Solution 1

- **“Channel solution”:** have an **exclusive owner** of count each time



1.1 Concurrent Counter: Solution 1

- **“Channel solution”:** have an **exclusive owner** of count each time



Channel

Repeat the process…

Main

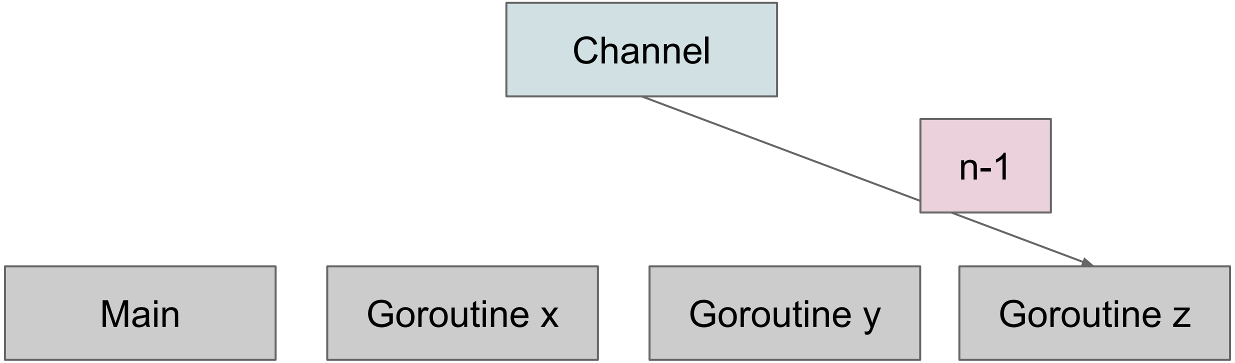
Goroutine x

Goroutine y

Goroutine z

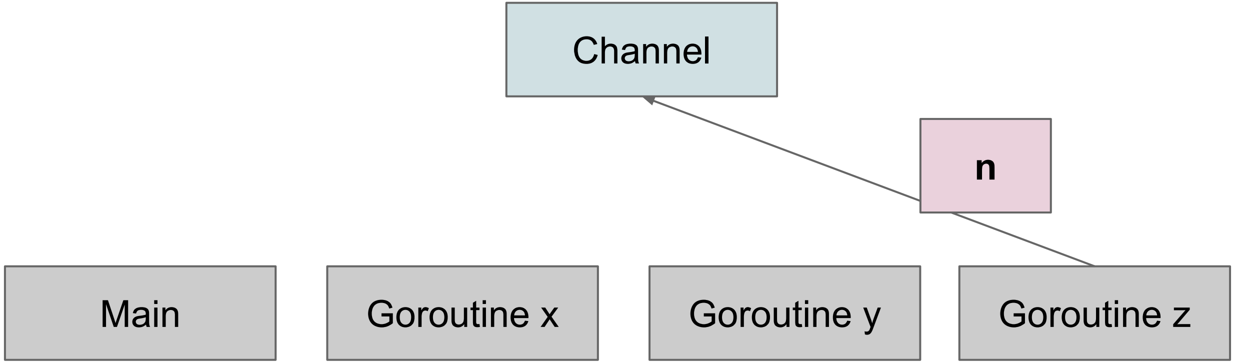
1.1 Concurrent Counter: Solution 1

- **“Channel solution”:** have an **exclusive owner** of count each time



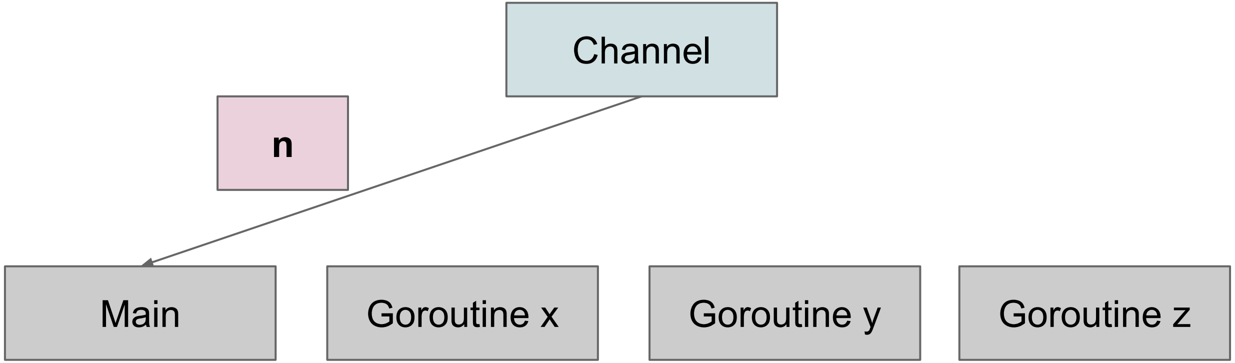
1.1 Concurrent Counter: Solution 1

- **“Channel solution”:** have an **exclusive owner** of count each time



1.1 Concurrent Counter: Solution 1

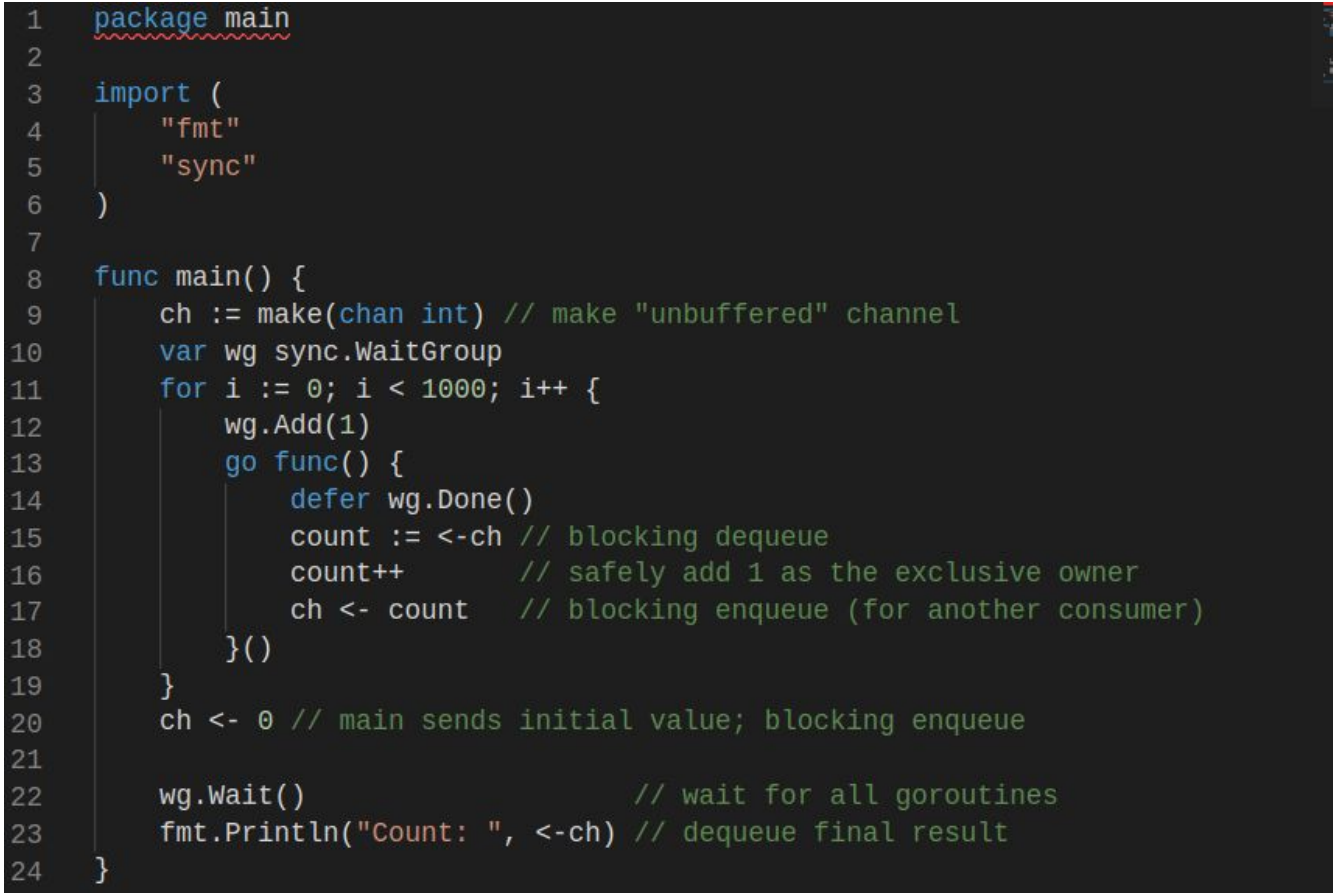
- **"Channel solution":** have an **exclusive owner** of count each time



Also, any non-correctness-related issues with this situation?

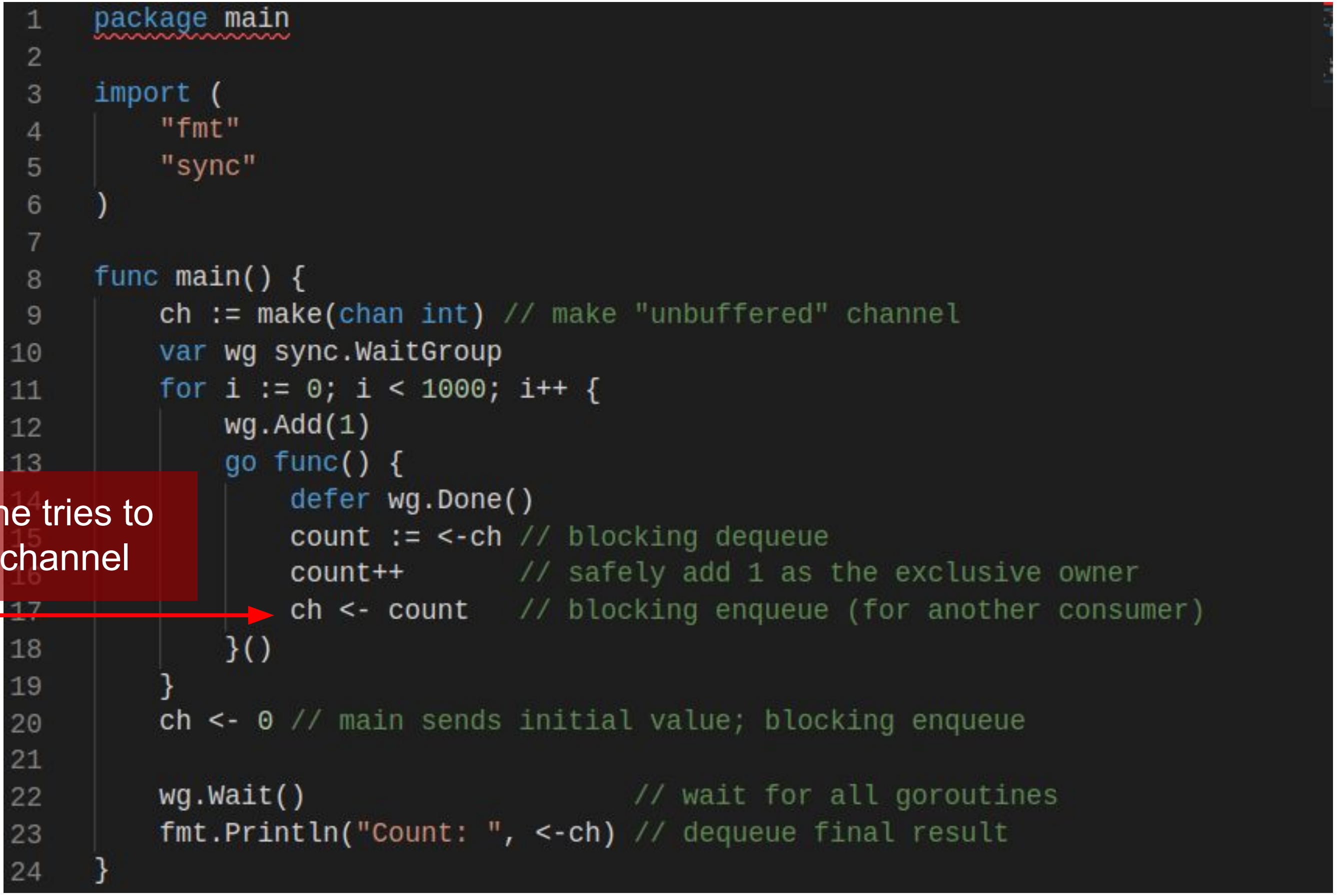
https://fsmbolt.comp.nus.edu.sg/z/ao13f7

Which line is stuck (deadlock)?



```
 1  package main
 2
 3  import (
 4      "fmt"
 5      "sync"
 6  )
 7
 8  func main() {
 9      ch := make(chan int) // make "unbuffered" channel
10      var wg sync.WaitGroup
11      for i := 0; i < 1000; i++ {
12          wg.Add(1)
13          go func() {
14              defer wg.Done()
15              count := <-ch // blocking dequeue
16              count++       // safely add 1 as the exclusive owner
17              ch <- count   // blocking enqueue (for another consumer)
18          }()
19      }
20      ch <- 0 // main sends initial value; blocking enqueue
21
22      wg.Wait()                      // wait for all goroutines
23      fmt.Println("Count: ", <-ch) // dequeue final result
24  }
```

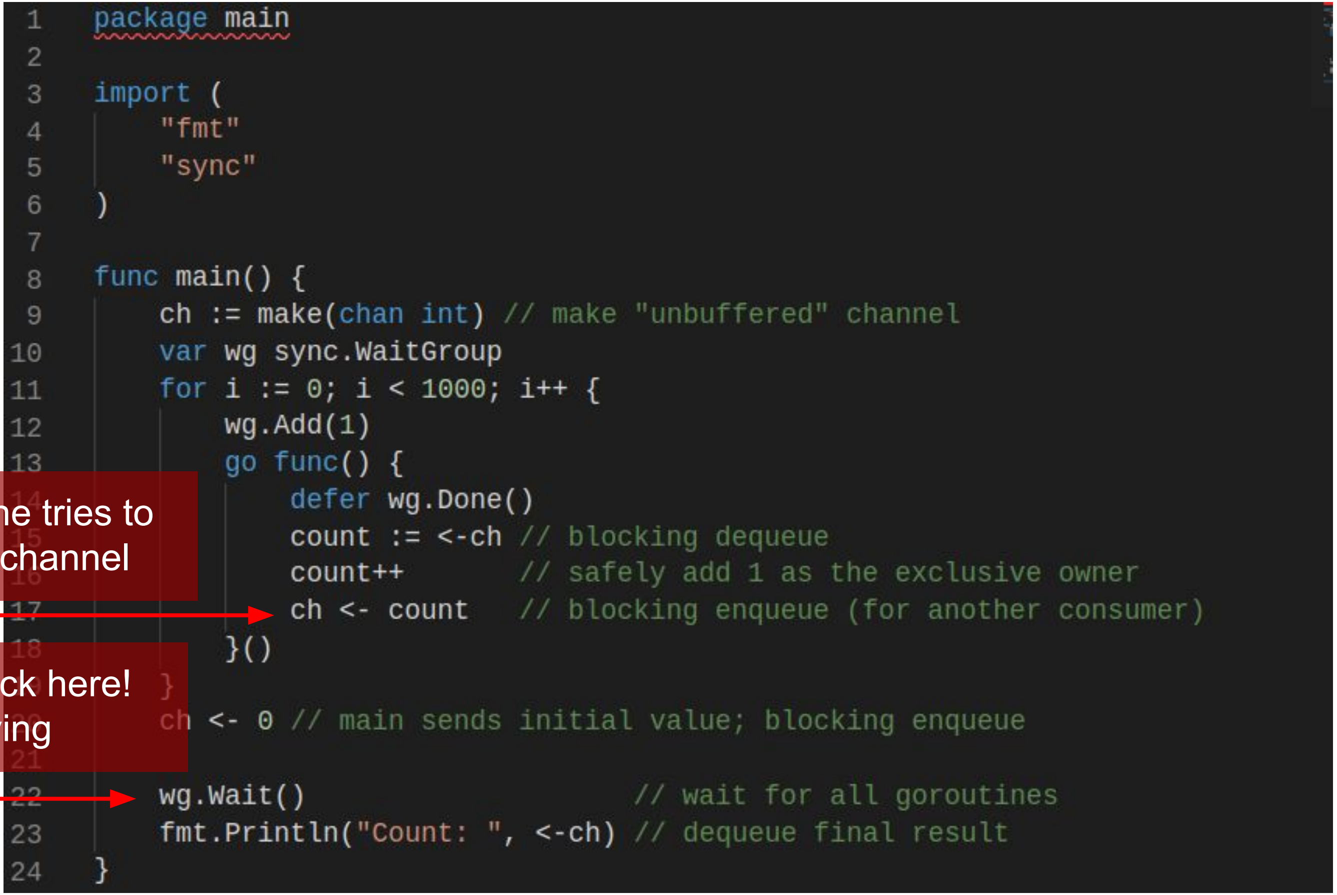
Which line is stuck (deadlock)?



```go
package main

import (
	"fmt"
	"sync"
)

func main() {
	ch := make(chan int) // make "unbuffered" channel
	var wg sync.WaitGroup
	for i := 0; i < 1000; i++ {
		wg.Add(1)
		go func() {
			defer wg.Done()
			count := <-ch // blocking dequeue
			count++       // safely add 1 as the exclusive owner
			ch <- count   // blocking enqueue (for another consumer)
		}()
	}
	ch <- 0 // main sends initial value; blocking enqueue

	wg.Wait()                          // wait for all goroutines
	fmt.Println("Count: ", <-ch) // dequeue final result
}
```

1000th goroutine tries to write 1000 to channel

Which line is stuck (deadlock)?

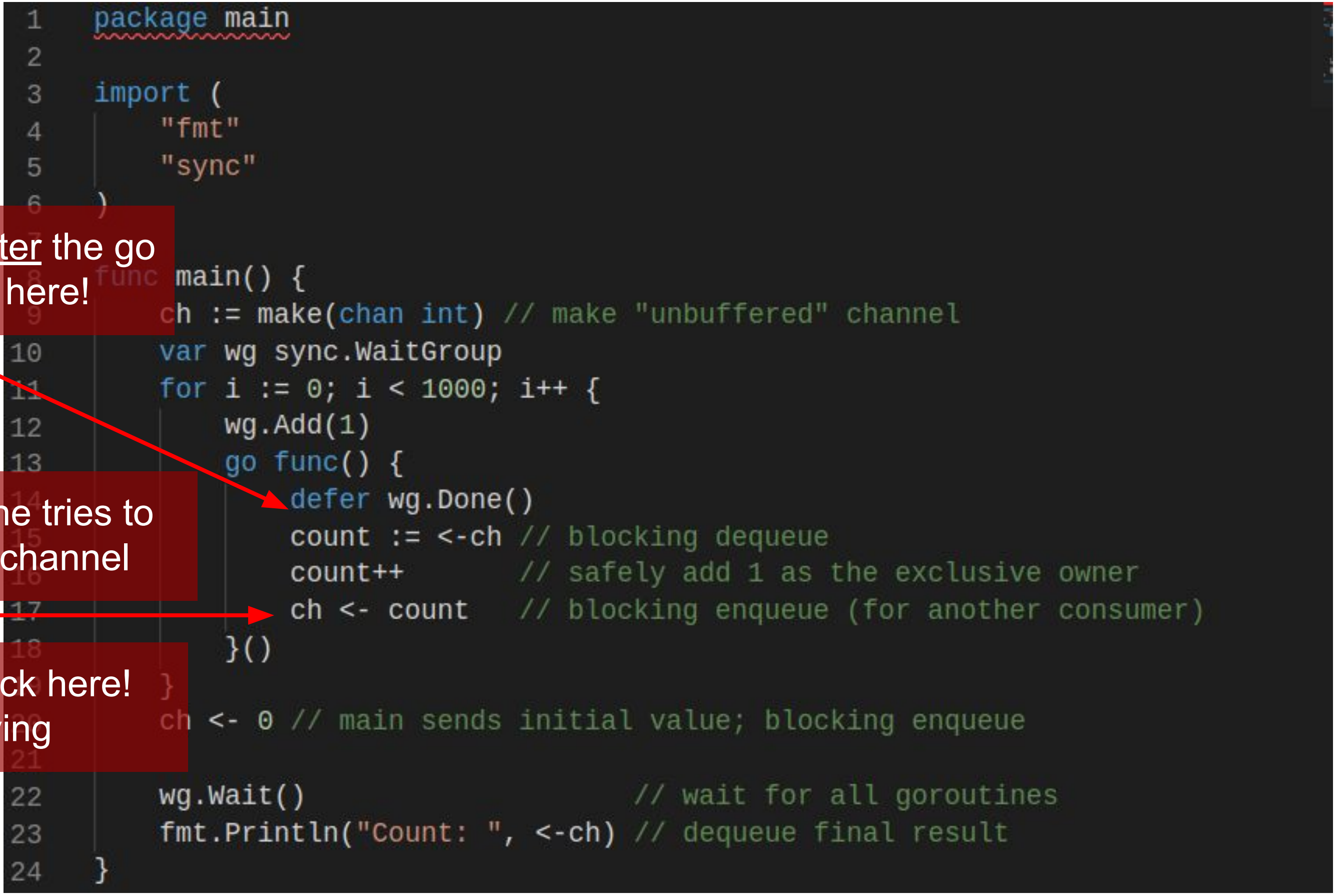


```
package main

import (
    "fmt"
    "sync"
)

func main() {
    ch := make(chan int) // make "unbuffered" channel
    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            count := <-ch // blocking dequeue
            count++       // safely add 1 as the exclusive owner
            ch <- count   // blocking enqueue (for another consumer)
        }()
    }
    ch <- 0 // main sends initial value; blocking enqueue

    wg.Wait()                      // wait for all goroutines
    fmt.Println("Count: ", <-ch) // dequeue final result
}
```

1000th goroutine tries to write 1000 to channel

But main is stuck here! Not receiving

Which line is stuck (deadlock)?



```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    ch := make(chan int) // make "unbuffered" channel
    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            count := <-ch // blocking dequeue
            count++       // safely add 1 as the exclusive owner
            ch <- count   // blocking enqueue (for another consumer)
        }()
    }
    ch <- 0 // main sends initial value; blocking enqueue

    wg.Wait()                      // wait for all goroutines
    fmt.Println("Count: ", <-ch) // dequeue final result
}
```

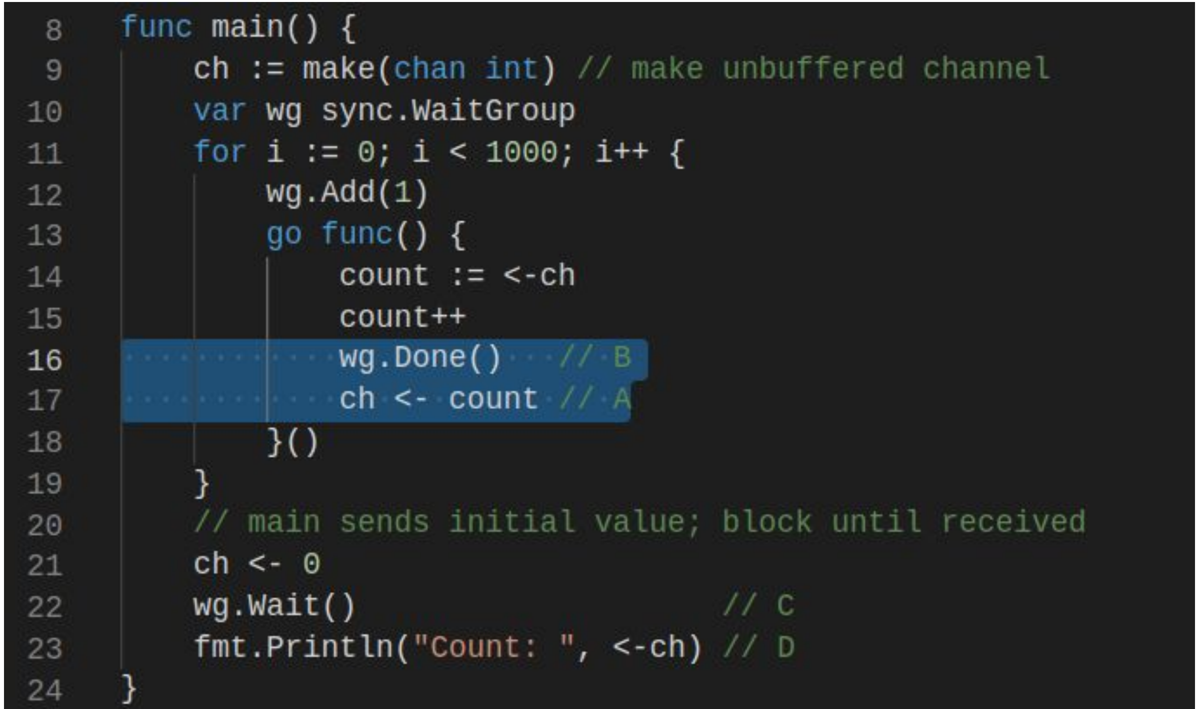
Defer only runs after the go func() is done here!

1000th goroutine tries to write 1000 to channel

But main is stuck here! Not receiving

1.2 Solution 1: wg.Done() before channel send

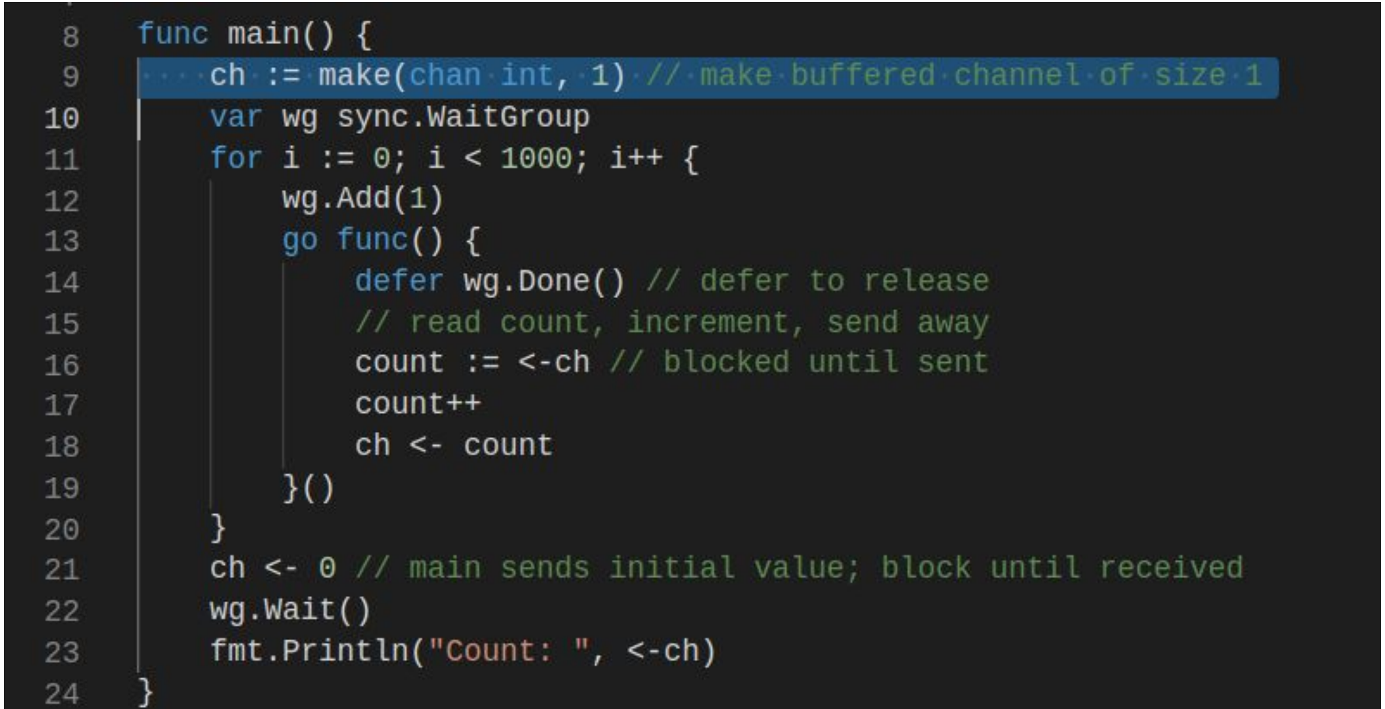
- “Natural” use of *defer* really messed us up!
- No easy day in concurrency land...



```go
func main() {
    ch := make(chan int) // make unbuffered channel
    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            count := <-ch
            count++
            wg.Done()   // B
            ch <- count // A
        }()
    }
    // main sends initial value; block until received
    ch <- 0
    wg.Wait()                    // C
    fmt.Println("Count: ", <-ch) // D
}
```

1.2 Solution 2: buffered channel

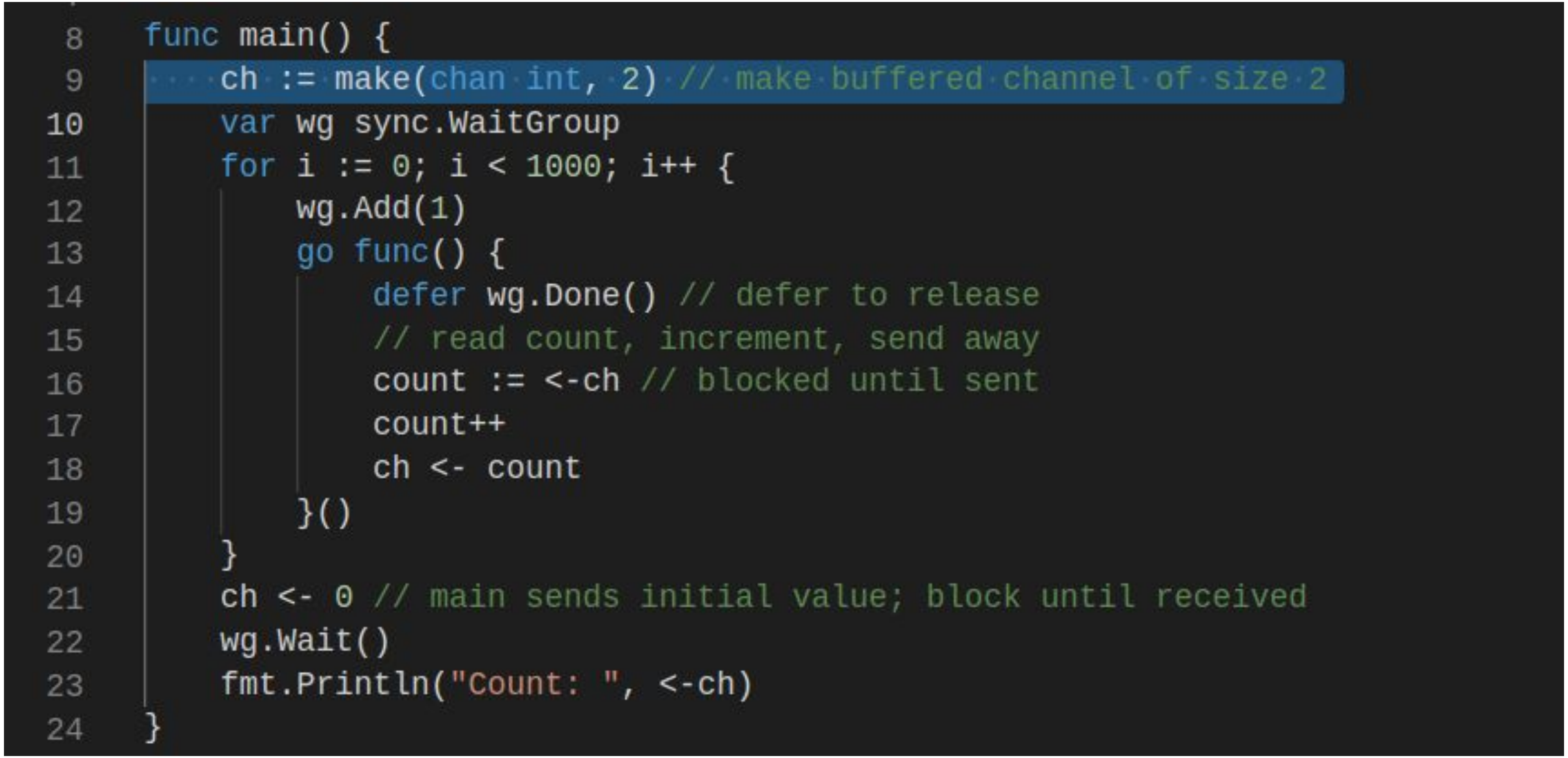
- Scary to generalize this...



```go
func main() {
    ch := make(chan int, 1) // make buffered channel of size 1
    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done() // defer to release
            // read count, increment, send away
            count := <-ch // blocked until sent
            count++
            ch <- count
        }()
    }
    ch <- 0 // main sends initial value; block until received
    wg.Wait()
    fmt.Println("Count: ", <-ch)
}
```

1.2 Solution 2: buffered channel

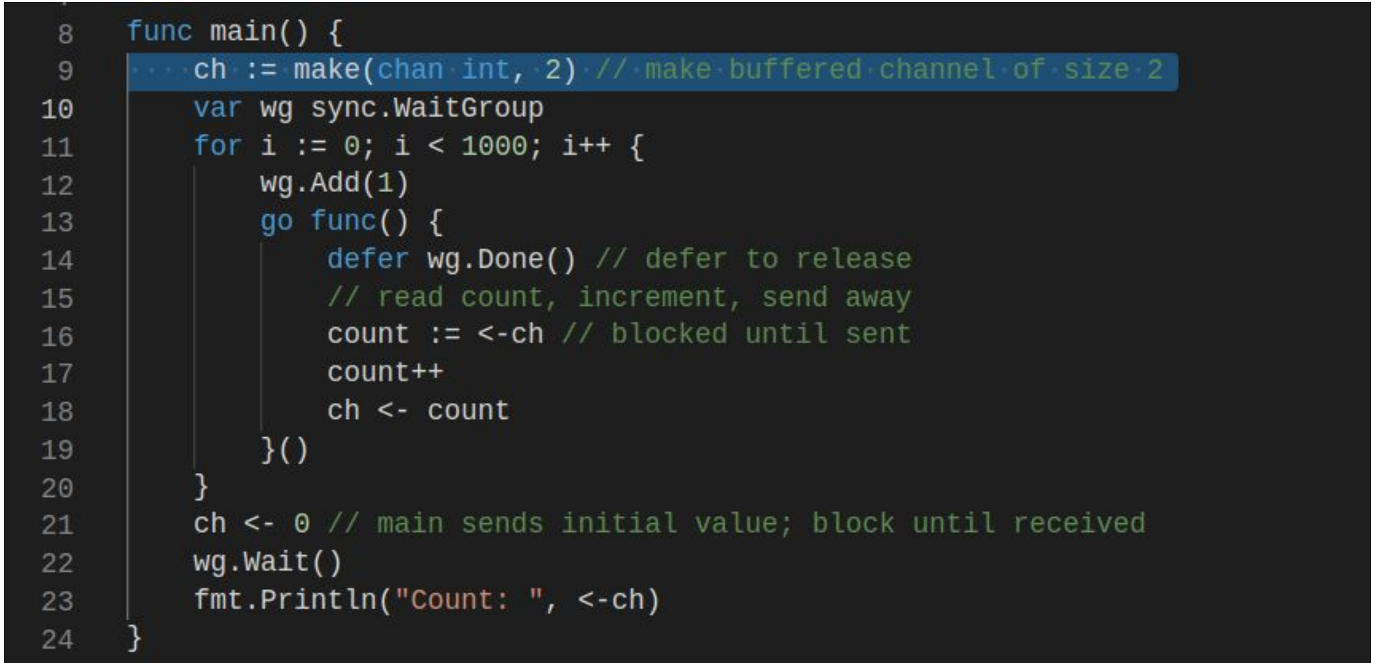
- What happens here? [p]



```go
func main() {
    ch := make(chan int, 2) // make buffered channel of size 2
    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done() // defer to release
            // read count, increment, send away
            count := <-ch // blocked until sent
            count++
            ch <- count
        }()
    }
    ch <- 0 // main sends initial value; block until received
    wg.Wait()
    fmt.Println("Count: ", <-ch)
}
```

1.2 Solution 2: buffered channel

- What happens here? [p] https://fsmbolt.comp.nus.edu.sg/z/cTWvz5
- Thankfully, still correct (outputs 1000). But like.. why?

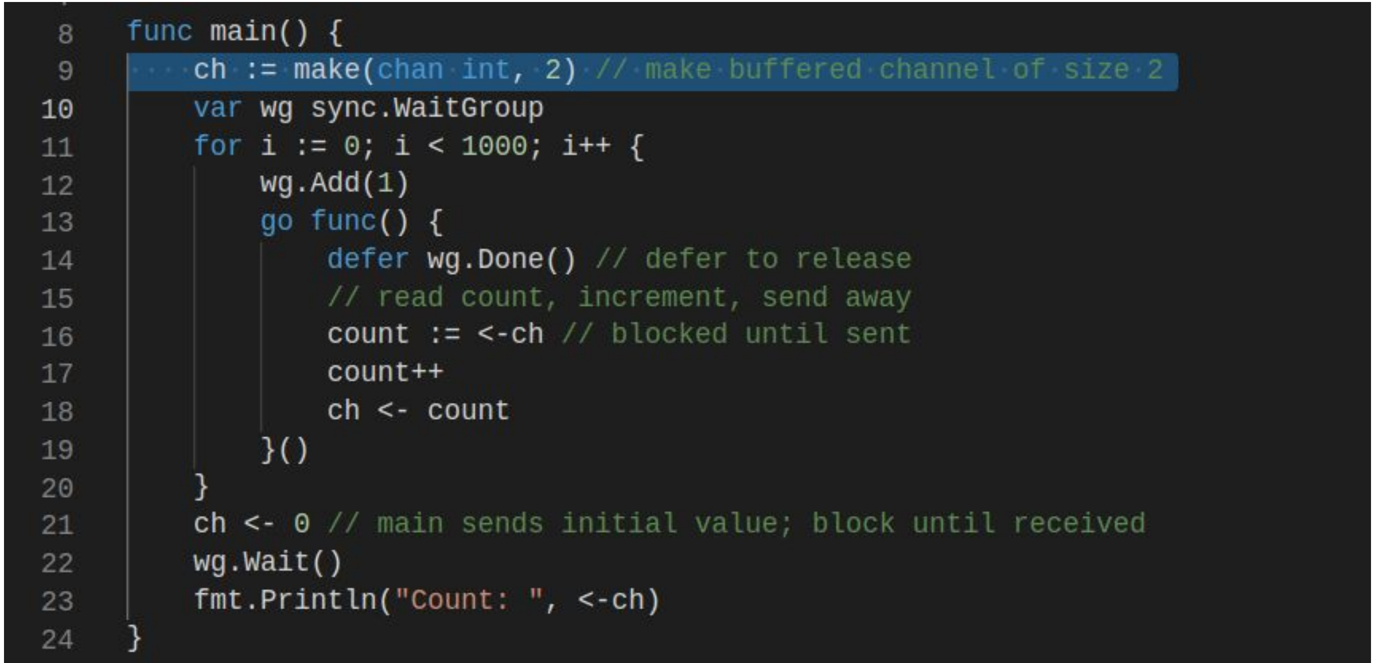


```go
func main() {
    ch := make(chan int, 2) // make buffered channel of size 2
    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done() // defer to release
            // read count, increment, send away
            count := <-ch // blocked until sent
            count++
            ch <- count
        }()
    }
    ch <- 0 // main sends initial value; block until received
    wg.Wait()
    fmt.Println("Count: ", <-ch)
}
```

1.2 Solution 2: buffered channel

- What happens here? [p]
- Thankfully, still correct. Max 1 value in the queue based on our design, but not restricted by the channel. Had to reason about it...

https://fsmbolt.comp.nus.edu.sg/z/cTWvz5



```go
func main() {
    ch := make(chan int, 2) // make buffered channel of size 2
    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done() // defer to release
            // read count, increment, send away
            count := <-ch // blocked until sent
            count++
            ch <- count
        }()
    }
    ch <- 0 // main sends initial value; block until received
    wg.Wait()
    fmt.Println("Count: ", <-ch)
}
```

Tutorial Q1.3

1.3 *Independently* producing / merging counts

- We don’t want each thread to take *exclusive* access of a global count **(this is basically a mutex implemented by channels!)**
- **How can we parallelize adding to the final count?**

1.3 *Independently* producing / merging counts

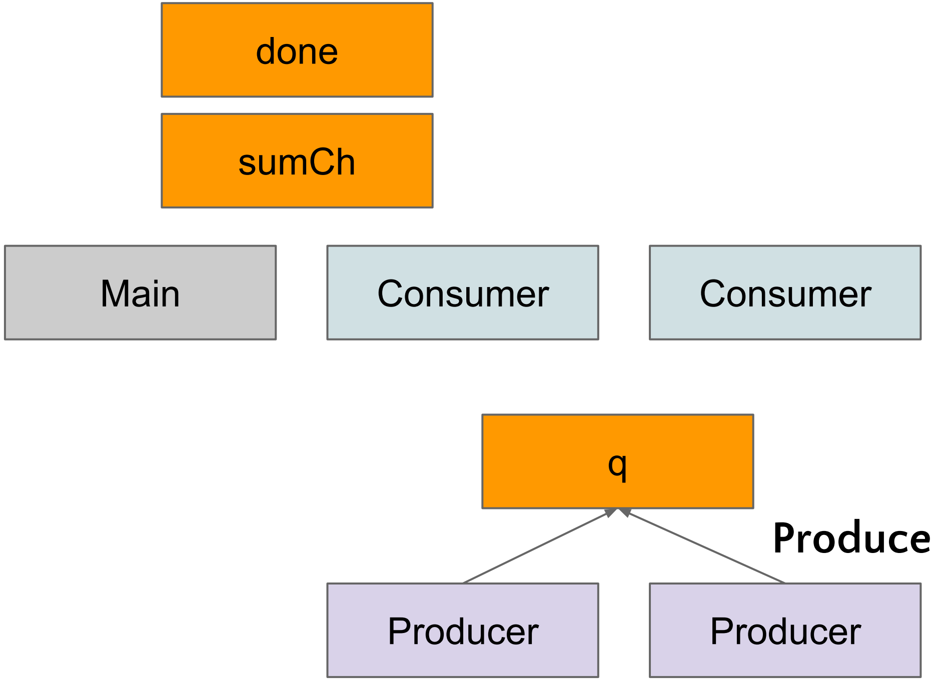
- We don't want each thread to take *exclusive* access of a global count **(this is basically a mutex implemented by channels!)**
- **How can we parallelize adding to the final count?**
- **Step 1: Consumers read from producers and update local counts https://fsmbolt.comp.nus.edu.sg/z/7Yh9sc**

1.3 *Independently* producing / merging counts

- We don’t want each thread to take *exclusive* access of a global count **(this is basically a mutex implemented by channels!)**
- **How can we parallelize adding to the final count?**
- **Step 1: Consumers read from producers and update local counts** **https://fsmbolt.comp.nus.edu.sg/z/7Yh9sc**
- **Step 2: Consumers send local counts to overall sum channel when production is done** **https://fsmbolt.comp.nus.edu.sg/z/nPvfsq**

1.3 *Independently* producing / merging counts

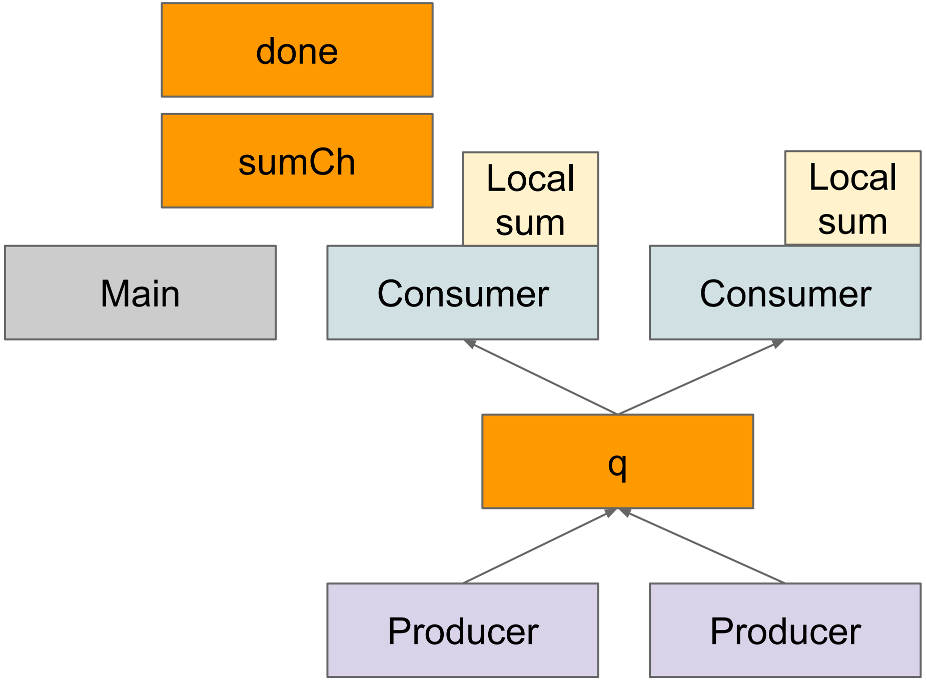
Overall idea:



Producers keep adding 1 to q

1.3 *Independently* producing / merging counts

Overall idea:

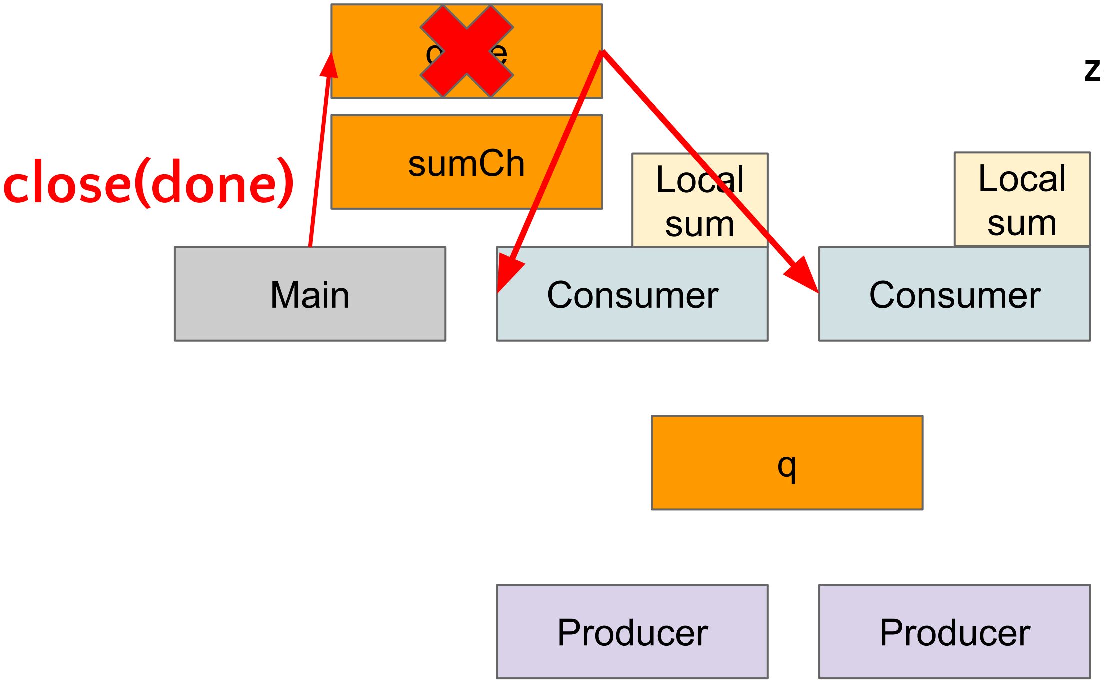


Consumers take 1s off q and add to local count

1.3 *Independently* producing / merging counts

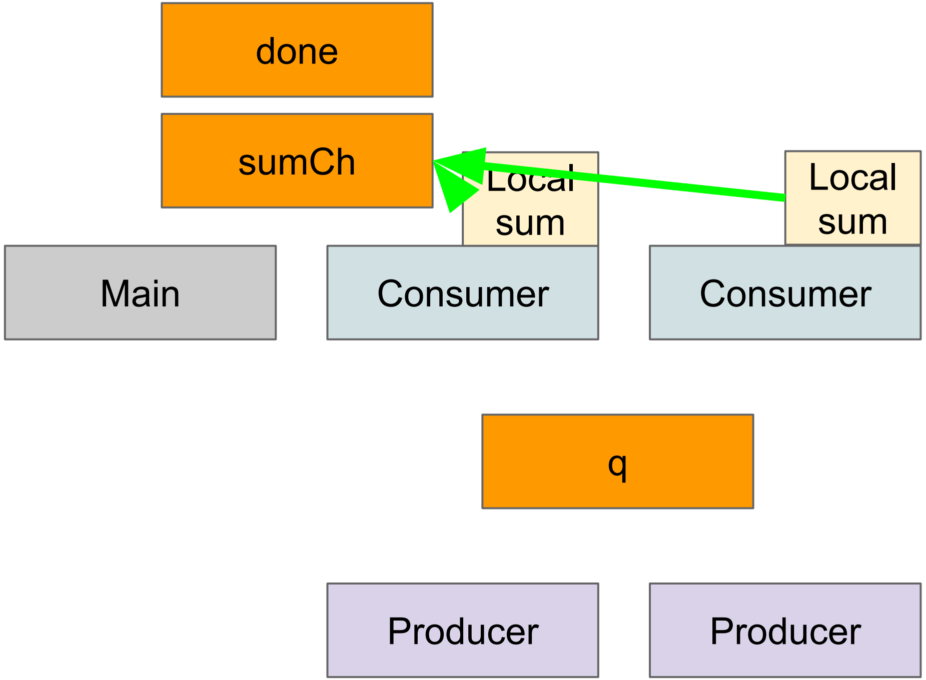
Overall idea:

When consumers see than done channel is closed… (closed channel will return zero-value of type when read)



1.3 *Independently* producing / merging counts

Overall idea:

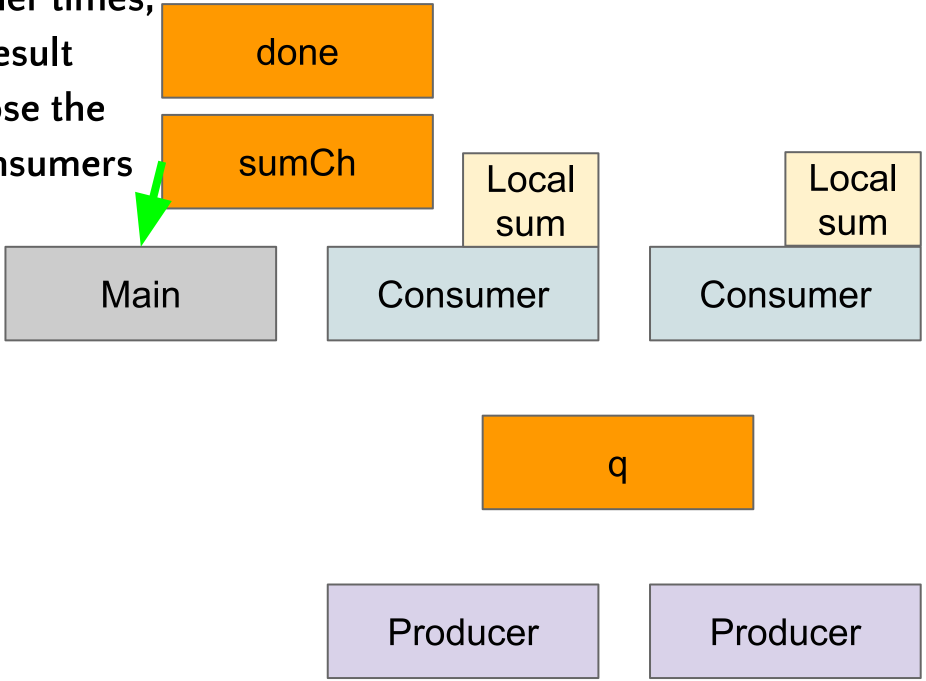


They send their local sum to the sumCh channel

1.3 *Independently* producing / merging counts

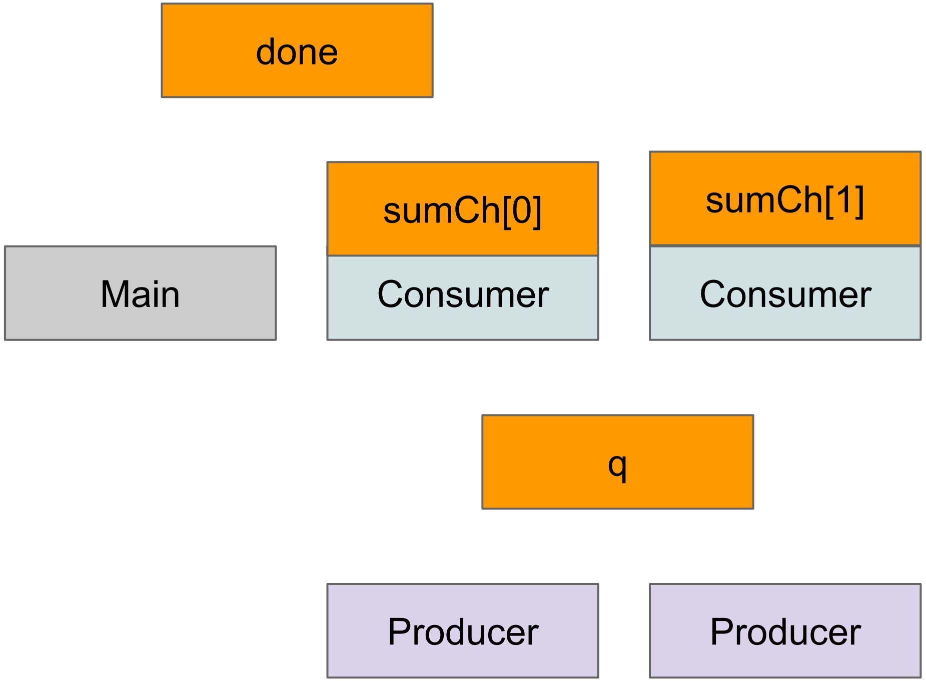
Overall idea:

Read NumConsumer times,
Add to final result
Qn: can you close the
channel from consumers



1.3 *Independently* producing / merging counts

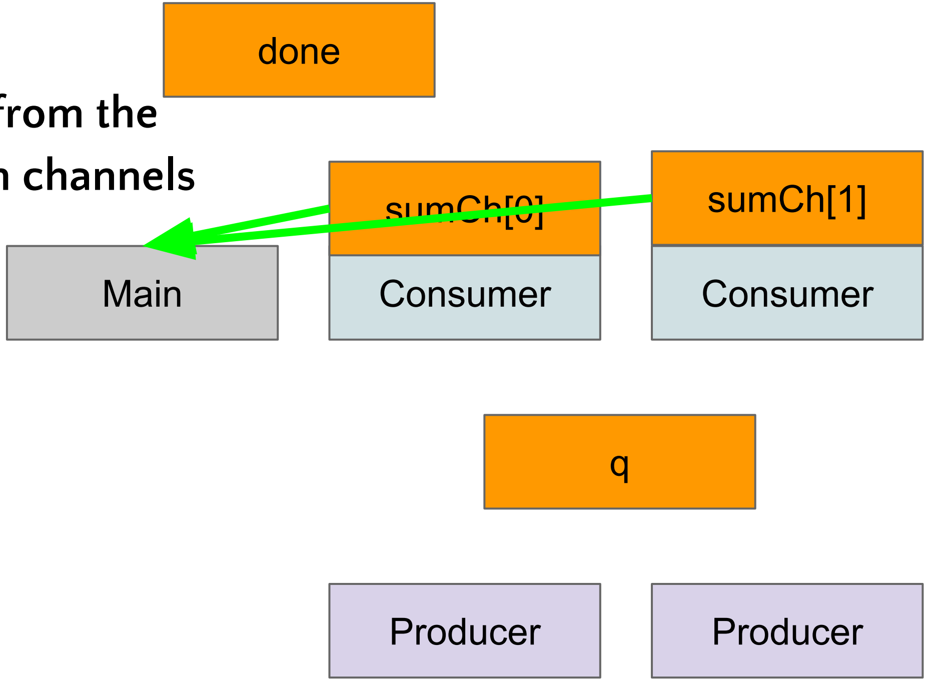
Alternate design: separate sum channels for each consumer!



1.3 *Independently* producing / merging counts

Alternate design: separate sum channels for each consumer!

Main reads from the individual sum channels



Tutorial Q2
Quick Detour to Go Patterns

Extra Qn: Is it possible to make Lock Free Queue using Go Channels?

For-Select Go Pattern

```go
for {
    select {
    case q <- 1: // keeps sending to q
    case <-done: // should exit
        return
    }
}
```

Note that Goroutines are resources too, failing to exit the Goroutines may cause memory leaks

Select Default Go Pattern

```
select {
    case q <- num:
        return true
    default:
    }
    return false
}
```

Qn:

1. What’s the difference between C++ switch case and Go select?
2. If you know Linux Select, what’s the difference with Go Select?

Context in Go Pattern

The context package is used for carrying deadlines, cancellations, and other request-scoped values across goroutines.

```
// Create common context in main goroutine
ctx, _ := context.WithTimeout(context.Background(), time.Second)
…

// Use the context in some Goroutines
   for {
        select {
        case q <- 1: ...
        case <-ctx.Done():
            return
        }
    }
```

Why does this all matter?

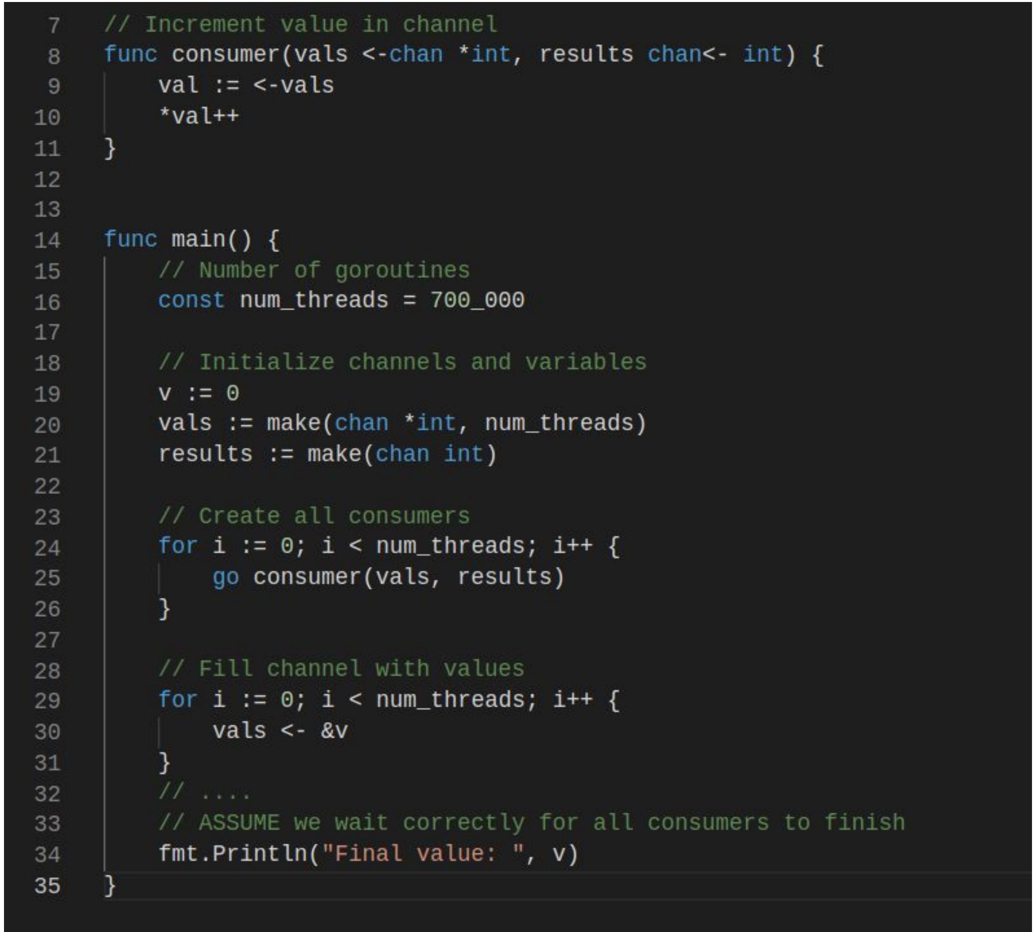
- Channel/message-passing-based thinking is extremely useful for certain kinds of problems, it provides elegant solutions to many concurrency problems

- Avoids many issues with shared memory

- **However: please look at the safety of the channels themselves, there are certain requirements that you need to adhere when you are using channels such as never close a channel twice, etc.**

Extra: Exploring Channels

- What is the final value of this program? (Assume that it correctly compiles) [p]



```go
// Increment value in channel
func consumer(vals <-chan *int, results chan<- int) {
    val := <-vals
    *val++
}


func main() {
    // Number of goroutines
    const num_threads = 700_000

    // Initialize channels and variables
    v := 0
    vals := make(chan *int, num_threads)
    results := make(chan int)

    // Create all consumers
    for i := 0; i < num_threads; i++ {
        go consumer(vals, results)
    }

    // Fill channel with values
    for i := 0; i < num_threads; i++ {
        vals <- &v
    }
    // ....
    // ASSUME we wait correctly for all consumers to finish
    fmt.Println("Final value: ", v)
}
```

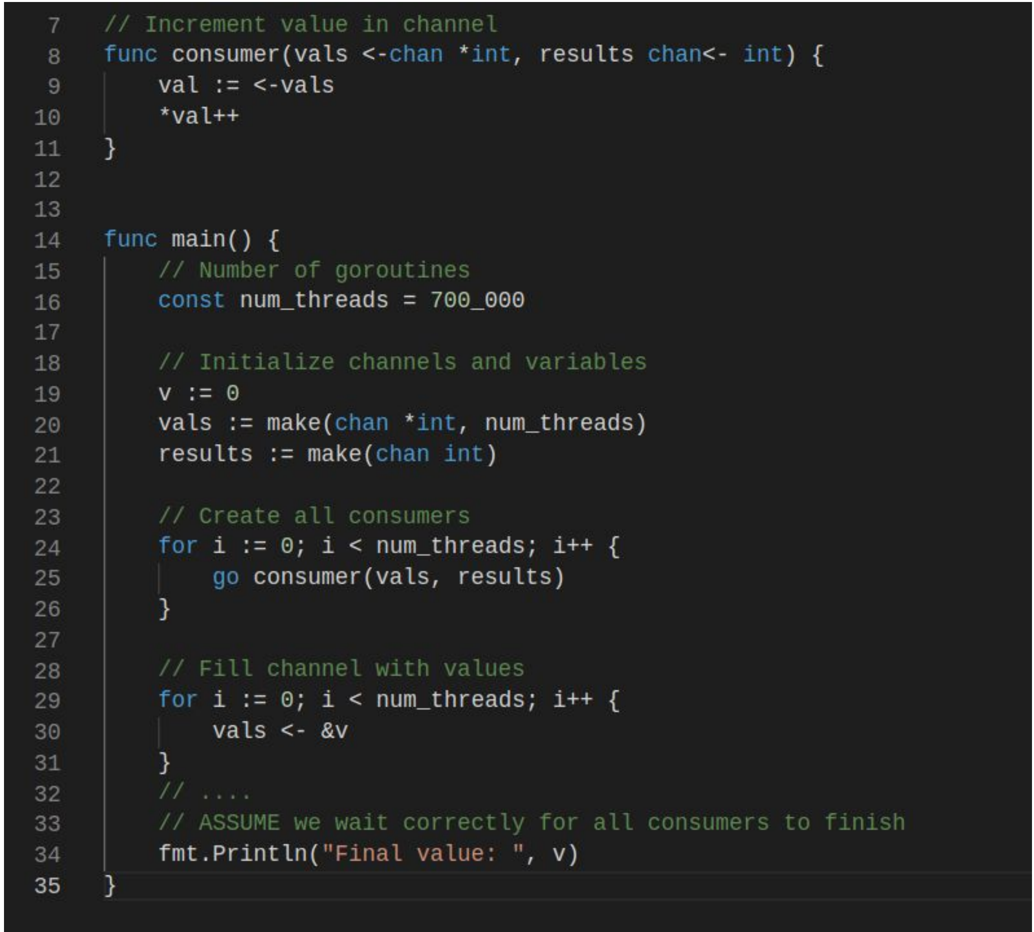
Extra: Exploring Channels

https://fsmbolt.comp.nus.edu.sg/z/xT1nEz

- What is the final value of this program? (Assume that it correctly compiles) [p]

- Unknown, much less than 700_000 is very likely
    - Data race!
    - Passed v by *reference* (ptr)

- Channels do not automatically free us from shared memory concerns!



```go
// Increment value in channel
func consumer(vals <-chan *int, results chan<- int) {
    val := <-vals
    *val++
}


func main() {
    // Number of goroutines
    const num_threads = 700_000

    // Initialize channels and variables
    v := 0
    vals := make(chan *int, num_threads)
    results := make(chan int)

    // Create all consumers
    for i := 0; i < num_threads; i++ {
        go consumer(vals, results)
    }

    // Fill channel with values
    for i := 0; i < num_threads; i++ {
        vals <- &v
    }
    // ....
    // ASSUME we wait correctly for all consumers to finish
    fmt.Println("Final value: ", v)
}
```

See you next week!