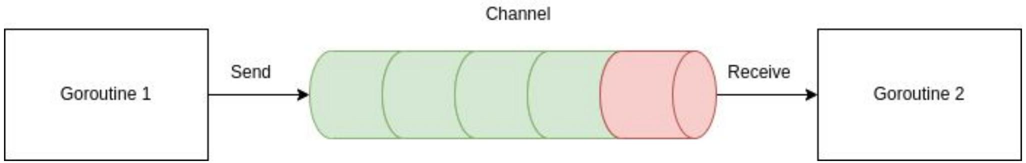
Tutorial 05 - Go

Modified from Walter’s and Sriram’s

Why Go?

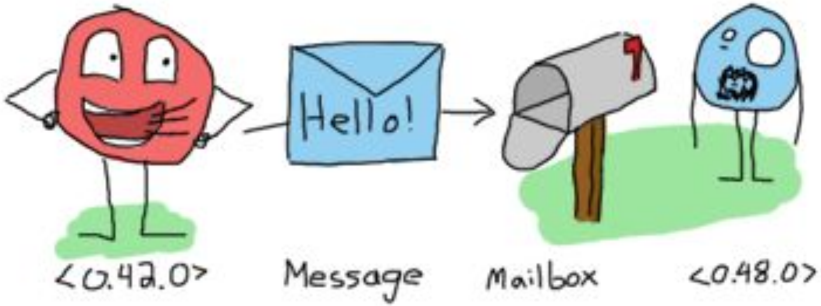
Message Passing as a First-Class Citizen

- **Shared memory ⇒ problem**



Golang

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



- However, some **highly scalable** and **correctness-focused** languages focus on **message passing**
    - Some *force* message passing
        - E.g., Elixir

Elixir

Today’s lesson plan

Go through common mistakes when it comes to using Go

- go threads, wait group, defer, channels

Focusing on Go’s concurrency, rather than the language

Common Mistakes

What’s The Output?

```go
func f() {
    time.Sleep(time.Second)
    fmt.Println("hello world")
}

func main() {
  for i := 0; i < 10; i++ {
    go f();
  }
}
```

Spawn, No Join

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

What's The Output?

```go
func f() {
	time.Sleep(time.Second)
	fmt.Println("hello world")
}

func main() {
	var wg sync.WaitGroup
	for i := 0; i < 10; i++ {
		go func() {
			wg.Add(1)
			f()
			wg.Done()
		}()
	}
	wg.Wait()
}
```

https://fsmbolt.comp.nus.edu.sg/z/edoo39

"New feature" : WaitGroup

Done() is safe,
Add(int) is not

What’s The Output?

```go
func f(i int) {
	time.Sleep(time.Second)
	fmt.Println(i)
}


func main() {
	var wg sync.WaitGroup
	for i := 0; i < 10; i++ {
		wg.Add(1)
		go func() {
			f(i)
			wg.Done()
		}()
	}
	wg.Wait()
}
```

https://fsmbolt.comp.nus.edu.sg/z/fyMex9

Variable not captured

What’s The Output?

```go
func f(i int) {
	time.Sleep(time.Second)
	fmt.Println(i)
}

func main() {
	var wg sync.WaitGroup
	for i := 0; i < 10; i++ {
		wg.Add(1)
		go func(i int) {
			defer wg.Done()
			f(i)
		}(i)
	}
	wg.Wait()
}
```

“New feature” : defer

https://fsmbolt.comp.nus.edu.sg/z/z8q4d4

What’s The Output?

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

“New feature” : chan

Queue is by default unbuffered

https://play.golang.com/p/XbZjZrMQdcN

How can we solve this?

Solution 1 - Asynchronous Send

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

Solution 2 - Buffered Channel

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

How much buffer fixes deadlock?

Solution 2 - Why Buffered Channel is always the solution

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

Extra: Exploring Channels

- How many possible outputs does this program have, if any?
- What if it is buffered?
- 3 possible outputs
- Buffer does not make a difference since program does not depend on FIFO property

```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)
}
```

What’s The Output?

```go
func f(i int) {
	time.Sleep(time.Second)
	fmt.Println(i)
}

func main() {
	var wg sync.WaitGroup
	for i := 0; i < 10; i++ {
		wg.Add(1)
		go func(i int) {
			defer wg.Done()
			f(i)
		}(i)
	}
	wg.Wait()
}
```

https://fsmbolt.comp.nus.edu.sg/z/z8q4d4

Var i is just copied around, not very useful

What if we want some true concurrent operation on some variable?

Message Passing

Why Not Shared Memory?

Memory Safety Issues:

- Use after free
- Double free
- Data races
- Stale atomic value
- ABA problem
- …

How to Solve Shared Memory Issues?

CS3211 Part 1

1. Test all interleavings, reasoning

CS3211 Part 2

2. Don’t share memory
3. Prevent unsafe access patterns (WW, WR, RW, ~~RR~~)

How to Solve Shared Memory Issues?

CS3211 Part 1

1. Test all interleavings

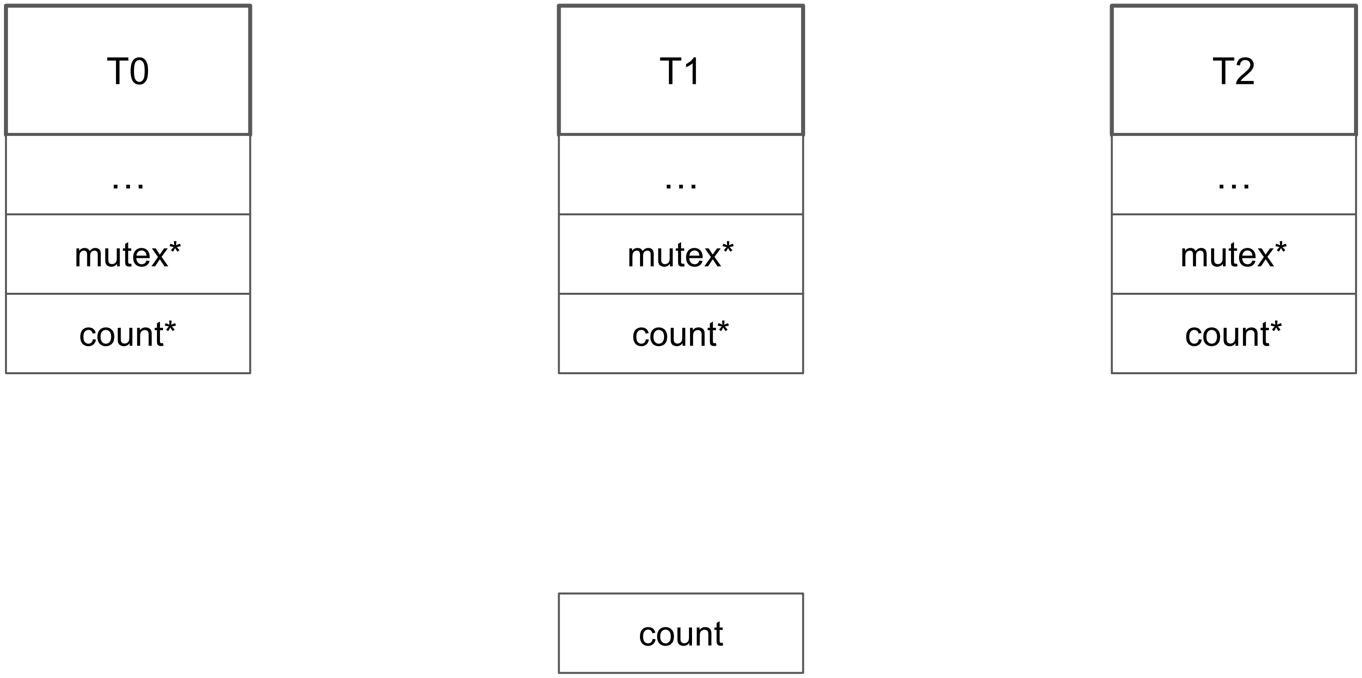
CS3211 Part 2

2. Don't share memory **Go channels**
3. Prevent unsafe access patterns (WW, WR, RW, ~~RR~~) **Rust borrow checker**

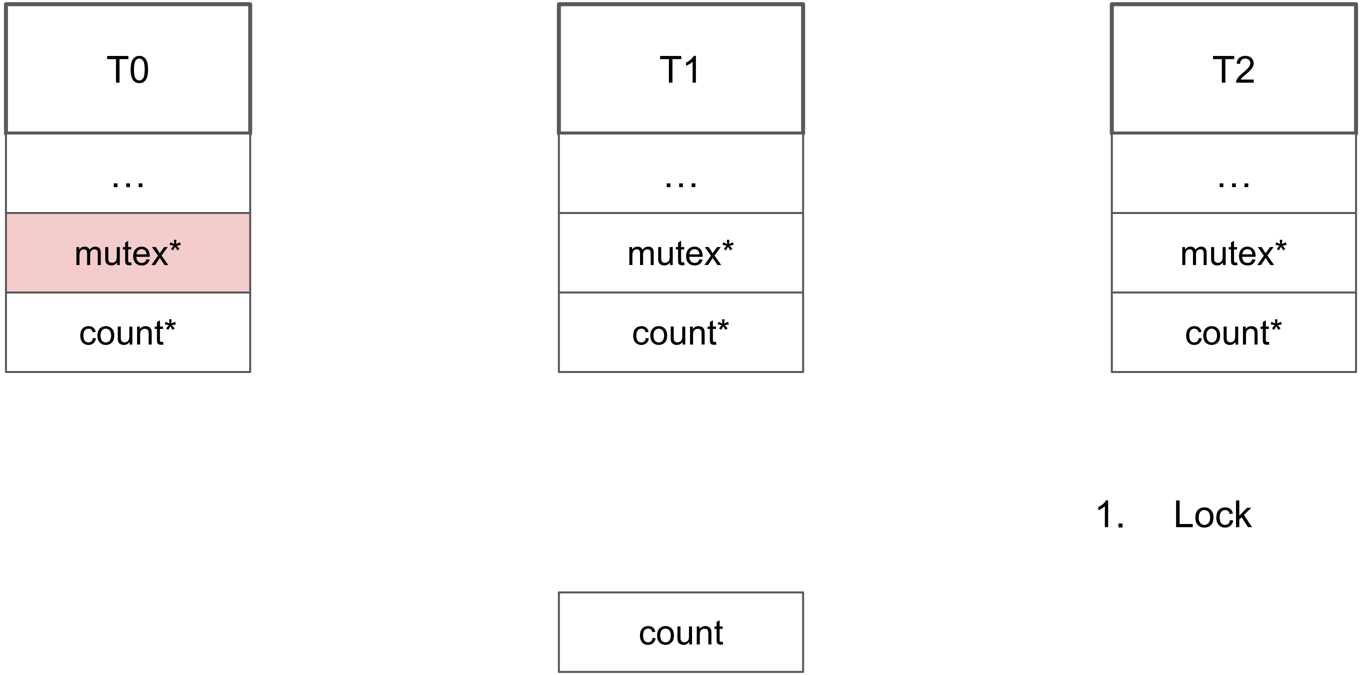
Concurrent Counter

…with channels

Recall: Mutex Impl (Shared Mem)

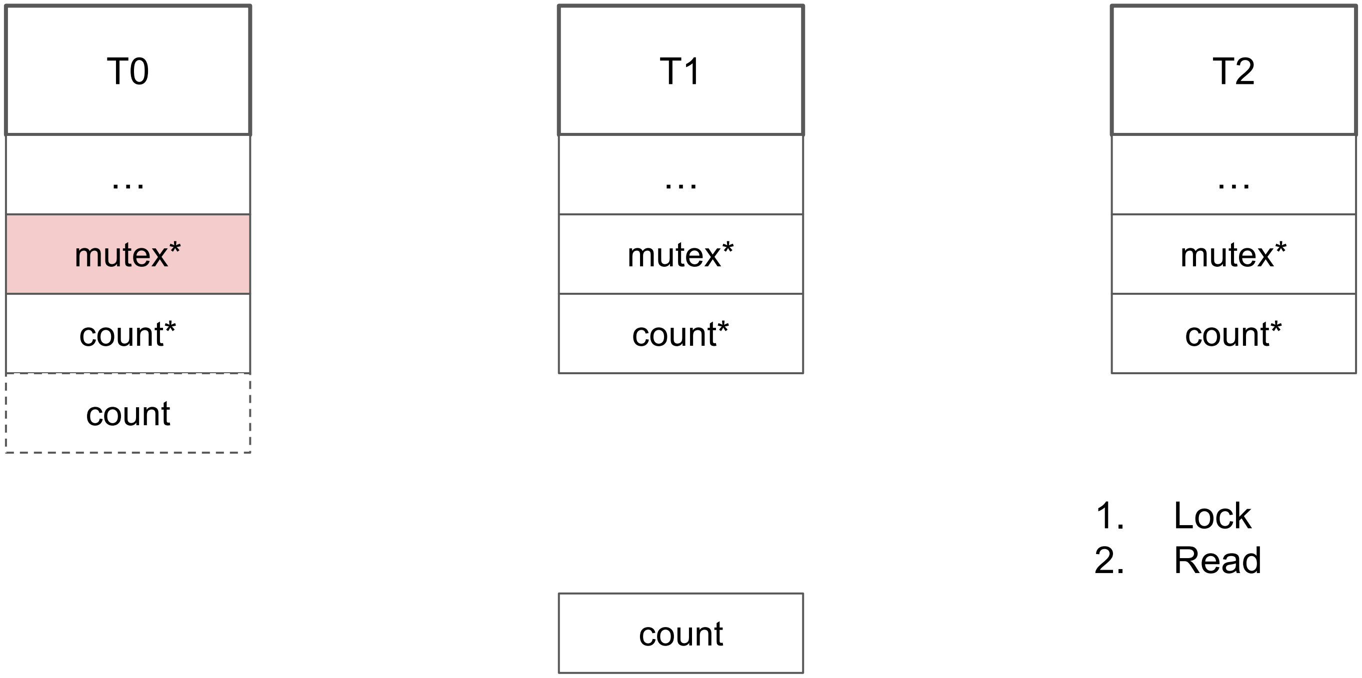


Recall: Mutex Impl



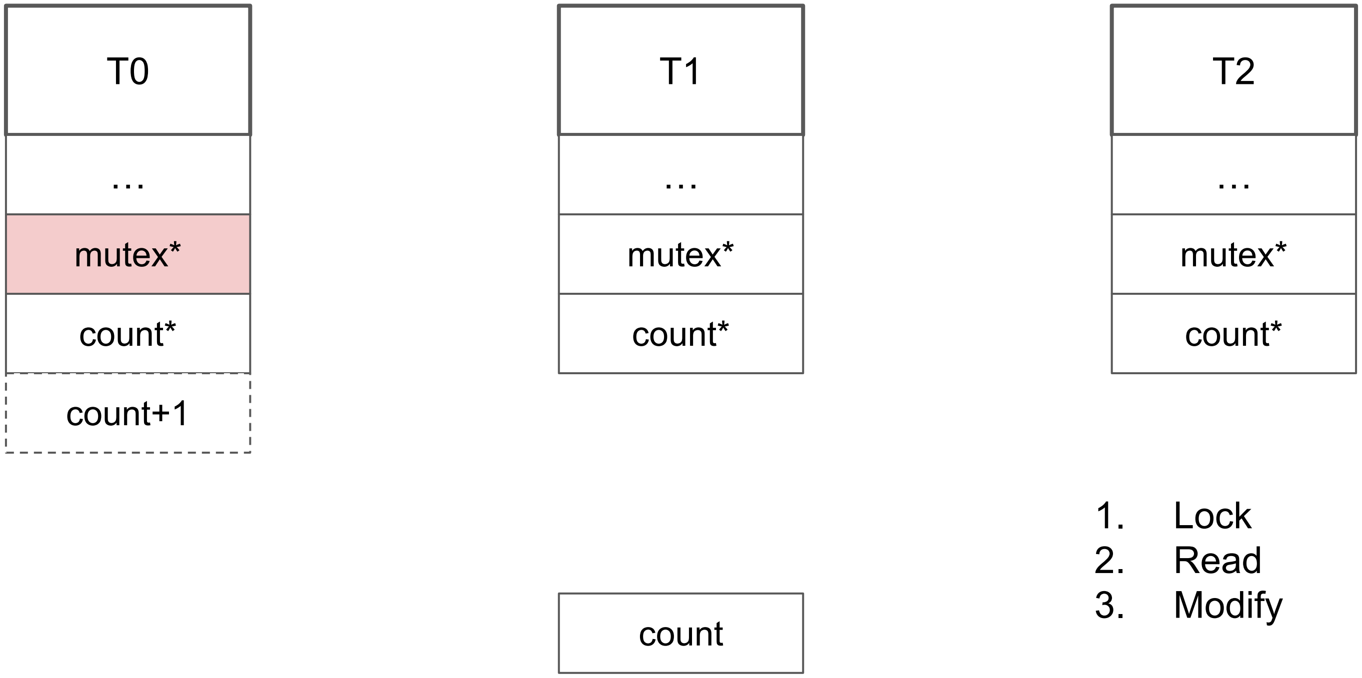
1. Lock

Recall: Mutex Impl



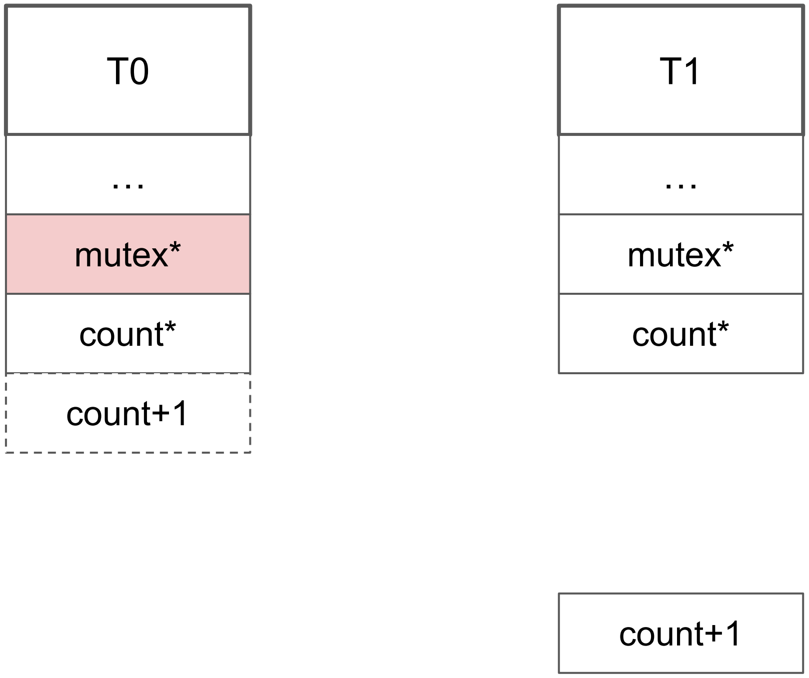
1. Lock
2. Read

Recall: Mutex Impl



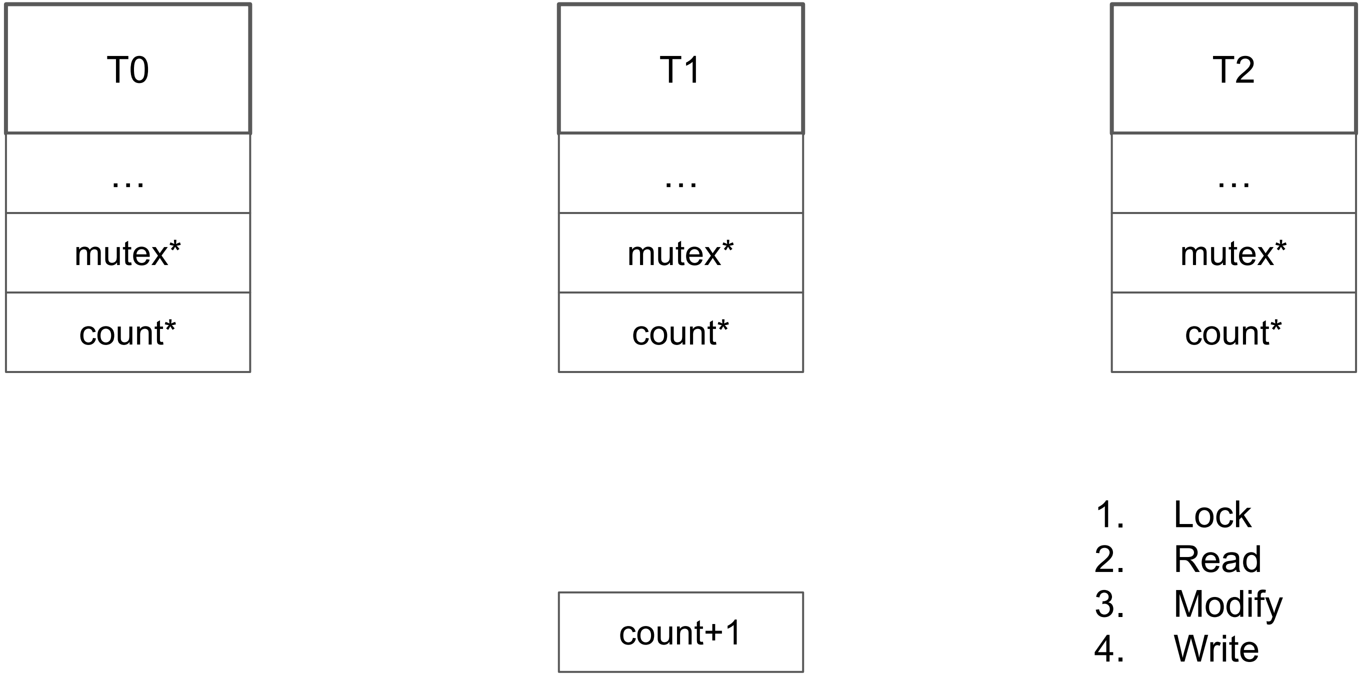
1. Lock
2. Read
3. Modify

Recall: Mutex Impl



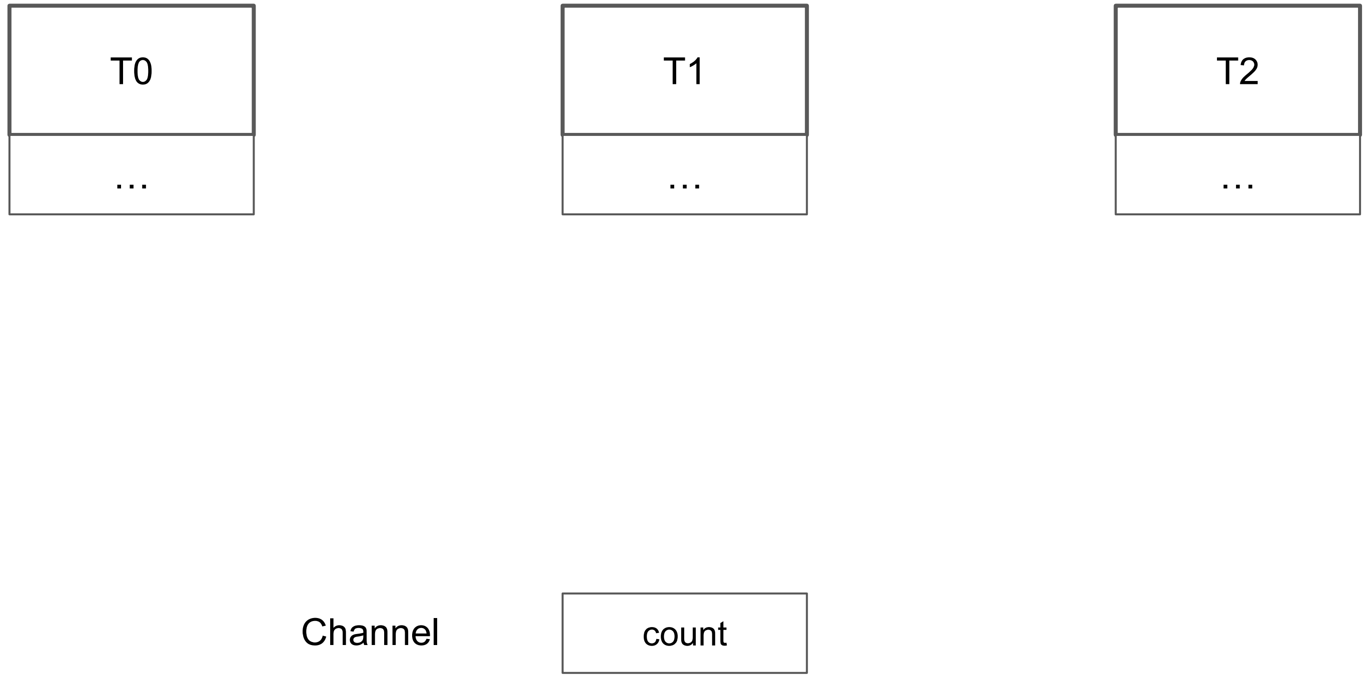
1. Lock
2. Read
3. Modify
4. Write

Recall: Mutex Impl

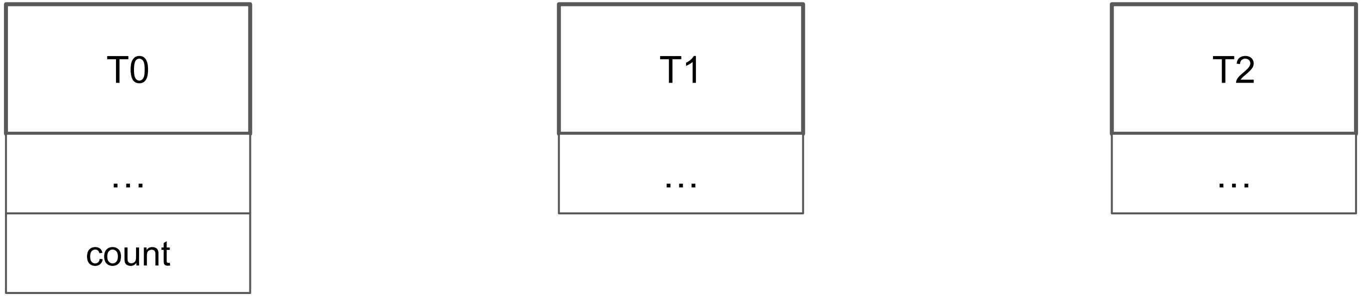


1. Lock
2. Read
3. Modify
4. Write
5. Unlock

Message Passing (Unique Ownership)



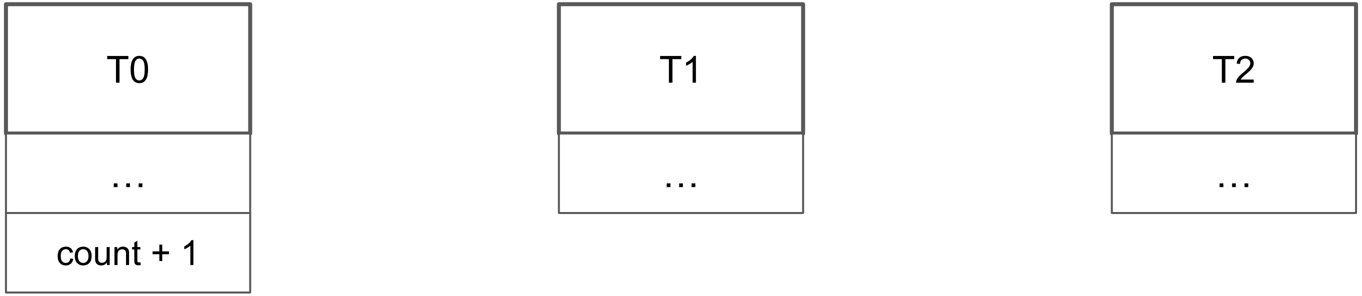
Message Passing



1. Read from ch

Channel

Message Passing



1. Read from ch
2. Modify

Channel

Message Passing

T0

…

T1

…

T2

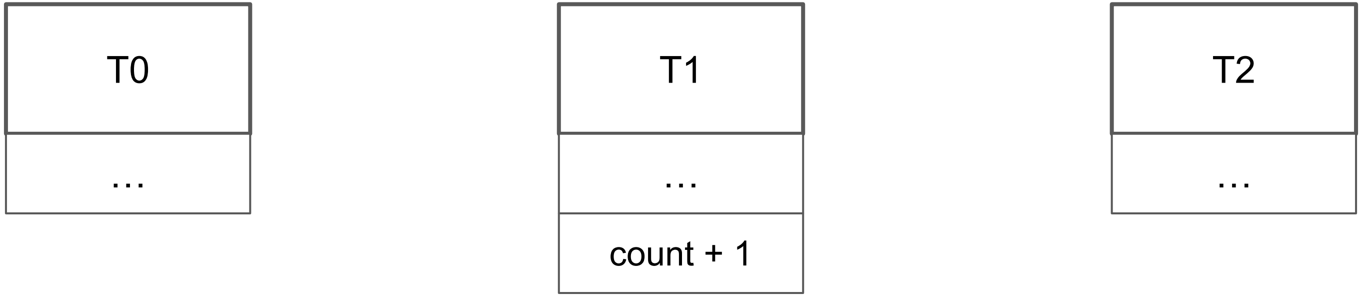
…

Channel

count + 1

1. Read from ch
2. Modify
3. Push to ch

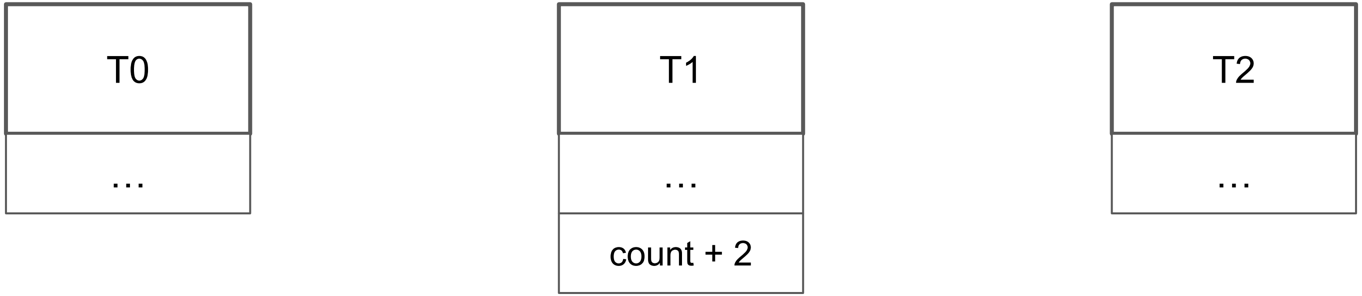
Message Passing



Channel

1. **Read from ch**
2. Modify
3. Push to ch

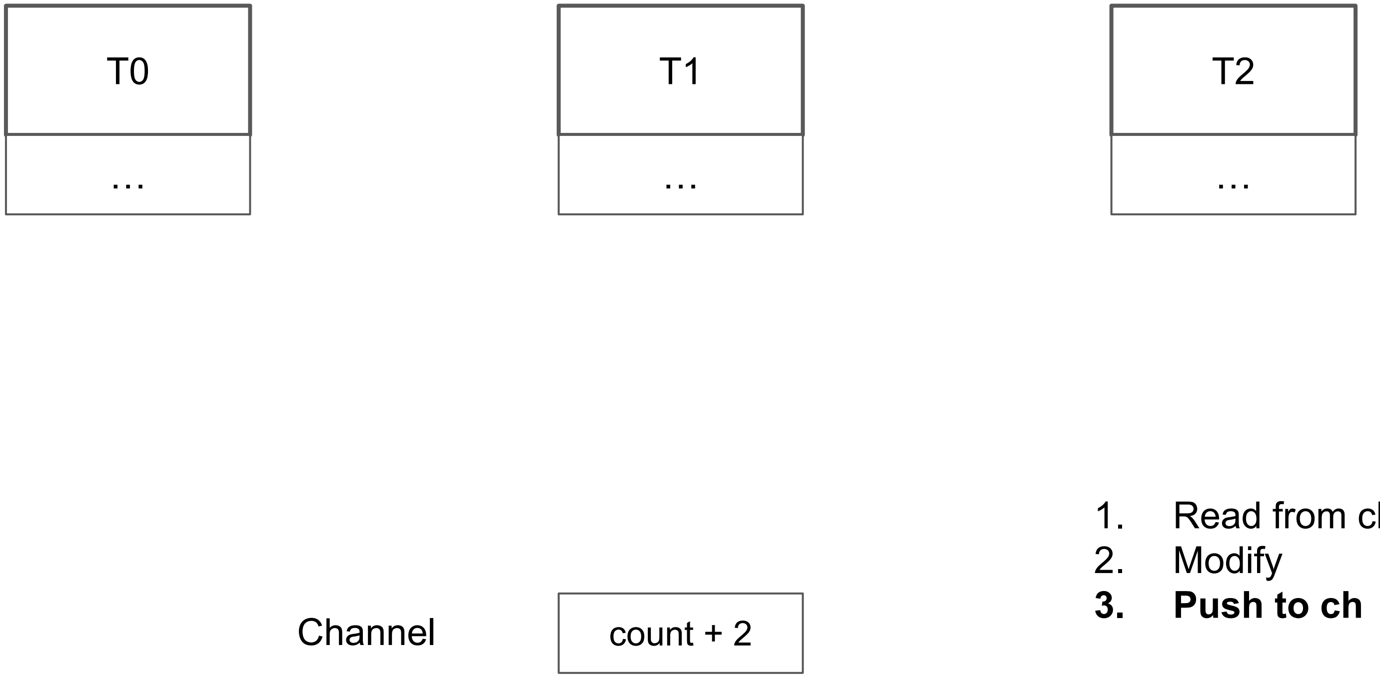
Message Passing



Channel

1. Read from ch
2. **Modify**
3. Push to ch

Message Passing



1. Read from ch
2. Modify
3. **Push to ch**

Code - Any problem?

```go
func main() {
	ch := make(chan int)
	for i := 0; i < 10; i++ {
		go func() {
			count := <-ch
			count++
			ch <- count
		}()
	}
	ch <- 0
	fmt.Println(<-ch)
}
```

Code - Any problem?

```
func main() {
    ch := make(chan int)
    for i := 0; i < 10; i++ {
        go func() {
            count := <-ch
            count++
            ch <- count
        }()
    }
    ch <- 0
    fmt.Println(<-ch) // may read count < 10, should wait for workers
}
```

We went through similar problem earlier on.

Master should wait for workers

Code - Any problem?

```go
func main() {
    ch := make(chan int)
    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            count := <-ch
            count++
            ch <- count
        }()
    }
    ch <- 0
    wg.Wait()
    fmt.Println(<-ch)
}
```

Code - Any problem?

```
func main() {
    ch := make(chan int)
    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            count := <-ch
            count++
            ch <- count            // A
            /* wg.Done() */        // B
        }()
    }
    ch <- 0
    wg.Wait()                      // C
    fmt.Println(<-ch)              // D
}
```

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

Code - Any problem?

```
func main() {
    ch := make(chan int)
    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            count := <-ch
            count++
            ch <- count            // A
            /* wg.Done() */        // B
        }()
    }
    ch <- 0
    wg.Wait()                      // C
    fmt.Println(<-ch)              // D
}
```

Master wait for worker to be done, worker wait for master take the count

Code - Any problem?

```
func main() {
	ch := make(chan int)
	var wg sync.WaitGroup
	for i := 0; i < 10; i++ {
		wg.Add(1)
		go func() {
			defer wg.Done()
			count := <-ch
			count++
			ch <- count              // A
			/* wg.Done() */          // B
		}()
	}
	ch <- 0
	fmt.Println(<-ch)                // D
	wg.Wait()                        // C
}
```

Master should wait for workers

Code - Any problem?

```
func main() {
	ch := make(chan int)
	var wg sync.WaitGroup
	for i := 0; i < 10; i++ {
		wg.Add(1)
		go func() {
			defer wg.Done()
			count := <-ch
			count++
			ch <- count            // A
			/* wg.Done() */        // B
		}()
	}
	ch <- 0
	wg.Wait()                      // C
	fmt.Println(<-ch)              // D
}
```

Solution - Break Cycle

```go
func main() {
	ch := make(chan int)
	var wg sync.WaitGroup
	for i := 0; i < 10; i++ {
		wg.Add(1)
		go func() {
			/* remove defer */
			count := <-ch
			count++
			wg.Done()              // B
			ch <- count            // A
		}()
	}
	ch <- 0
	wg.Wait()                      // C
	fmt.Println(<-ch)              // D
}
```

Solution - Buffer

```go
func main() {
	ch := make(chan int, 1) // make buffered channel of size 1
	var wg sync.WaitGroup
	for i := 0; i < 10; i++ {
		wg.Add(1)
		go func() {
			defer wg.Done()
			count := <-ch
			count++
			ch <- count                 // A
			/* wg.Done() */             // B
		}()
	}
	ch <- 0
	wg.Wait()                           // C
	fmt.Println(<-ch)                   // D
}
```

Solution - More buffer. Will I read less than 1000?

```
func main() {
	ch := make(chan int, 2) // make buffered channel of size 2
	var wg sync.WaitGroup
	for i := 0; i < 10; i++ {
		wg.Add(1)
		go func() {
			defer wg.Done()
			count := <-ch
			count++
			ch <- count            // A
			/* wg.Done() */        // B
		}()
	}
	ch <- 0
	wg.Wait()                      // C
	fmt.Println(<-ch)              // D
}
```

No, because only 1 item passed around, more buffer or not makes no difference.

Can we do better?

MPMC Queue

Back to this. What’s the “problem”? (performance)

```
func main() {
	ch := make(chan int, 2) // make buffered channel of size 2
	var wg sync.WaitGroup
	for i := 0; i < 10; i++ {
		wg.Add(1)
		go func() {
			defer wg.Done()
			count := <-ch
			count++
			ch <- count              // A
			/* wg.Done() */          // B
		}()
	}
	ch <- 0
	wg.Wait()                        // C
	fmt.Println(<-ch)                // D
}
```

Producer

```go
func producer(done chan struct{}, q chan<- int) {
    for {
        select {
        case q <- 1: // keeps incrementing...
        case <-done: // until stopped (channel closed)
            return
        }
    }
}
```

Consumer

```go
func consumer(done chan struct{}, q chan int) {
    sum := 0
    for {
        select {
        case num := <-q:
            sum += num
        case <-done:
            // TODO send local sum
            return
        }
    }
}
```

Consumer

```go
func consumer(done chan struct{}, q chan int, sumCh chan int) {
	sum := 0
	for {
		select {
		case num := <-q:
			sum += num
		case <-done:
			sumCh <- sum
			return
		}
	}
}
```

Consumer <-> Main

```go
func consumer(
        done chan struct{},
        q chan int,
        sumCh chan int) {
    sum := 0
    for {
        select {
        case num := <-q:
            sum += num
        case <-done:
            sumCh <- sum
            return
        }
    }
}
```

```go
func main() {
    sumCh := make(chan int)

    // spawn producers & consumers

    sum := 0
    for subSum := range sumCh {
        sum += subSum
    }
    fmt.Println("Sum: ", sum)
}
```

Any problem?

Consumer <-> Main

```go
func consumer(
        done chan struct{},
        q chan int,
        sumCh chan int) {
    sum := 0
    for {
        select {
        case num := <-q:
            sum += num
        case <-done:
            sumCh <- sum
            return
        }
    }
}
```

```go
func main() {
    sumCh := make(chan int)

    // spawn producers & consumers
    // close(done)
    sum := 0
    for subSum := range sumCh {
        sum += subSum
    }
    fmt.Println("Sum: ", sum)
}
```

Close the channel ‘done’!

Never ends since channel ‘sumCh’ is open!

Solution 1 - Close > Read

```go
func consumer(
        done chan struct{},
        q chan int,
        sumCh chan int) {
    sum := 0
    for {
        select {
        case num := <-q:
            sum += num
        case <-done:
            sumCh <- sum
            return
        }
    }
}
```

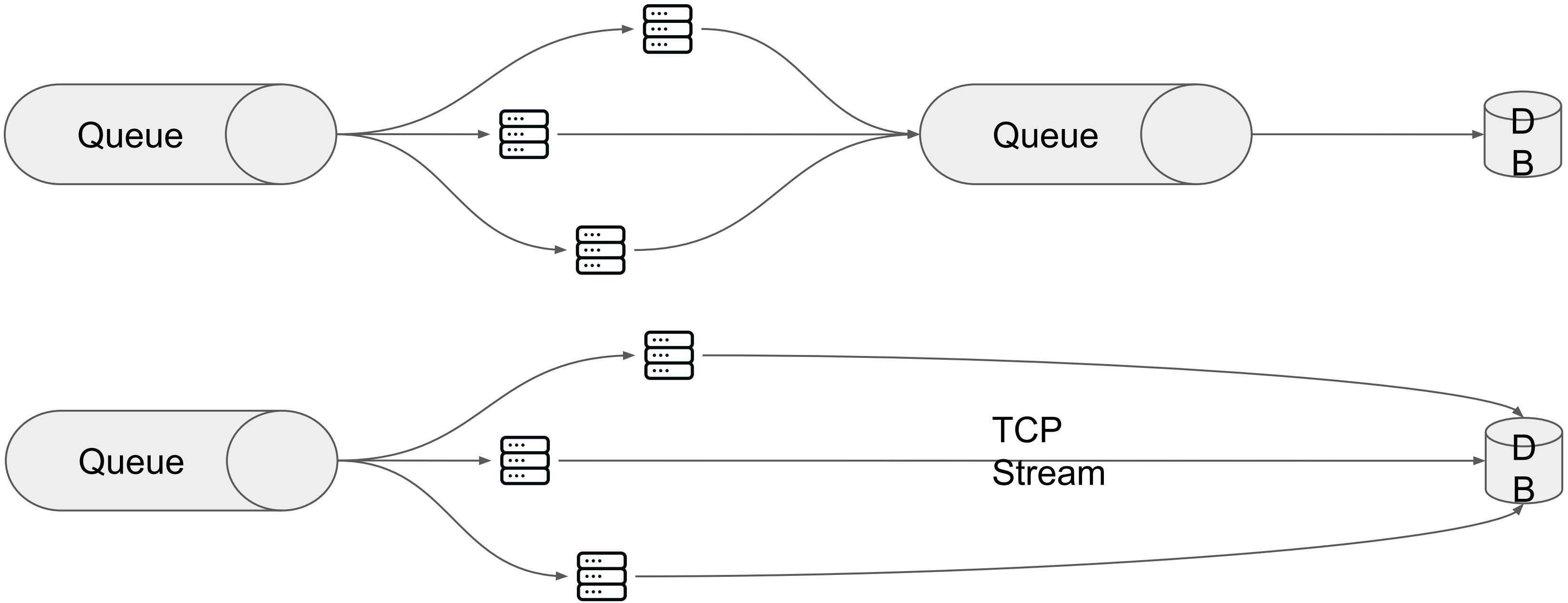
```go
func main() {
    sumCh := make(chan int)

    // spawn producers & consumers
    // close(done)
    sum := 0
    for i := 0; i < NCon; i++ {
        sum += <-sumCh
    }
    close(sumCh)
    fmt.Println("Sum: ", sum)
}
```

Read exactly N times

WARN: May panic if producer sends >1x
Idiomatic for writer to close
But we must ensure only last writer closes

Solution 2 - Fan-In vs Individual Queue



Solution 2 - Fan-In vs Individual Queue

```go
func consumer(
        done chan struct{},
        q chan int,
        sumCh chan int) {
    sum := 0
    for {
        select {
        case num := <-q:
            sum += num
        case <-done:
            sumCh <- sum
            close(sumCh)
            return
        }
    }
}
```

```go
func main() {
    sumCh := make([]chan int, NCon)

    // spawn producers & consumers

    sum := 0
    for i := 0; i < NCon; i++ {
        sum += <-sumCh[i]
    }
    fmt.Println("Sum: ", sum)
}
```

Bonus: Exploring Channels

- What is the final value of this program? (Assume that it correctly compiles)?
- There is data race

```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)
}
```

Takeaway

- “Undefined” Behavior
    - When passing args to goroutines, copy arguments (don’t share)
- “Catchable” Errors
    - Channels are synchronous by default. They can block
    - Double close / send after close on a channel panics

Notice it’s mostly deadlocks!

- Deadlocks are observable
    - eg long runtime, user QPS dropping, all threads asleep
- Deadlocks affect liveness, not safety
- This **DOES NOT MEAN** it’s always safe

See you next week!