Data races, Deadlocks & Dining Philosophers

DD1396
Ric Glassey
glassey@kth.se

Concepts

Data races and deadlocks in Go

Simple idiomatic solutions

Solving the dining philosophers problem

Data Races

Race condition

Task 1	Task 2	Account Balance
		1000 Kr
Read Balance		1000 Kr
	Read Balance	1000 Kr
Decrease Balance 600 Kr		1000 Kr
	Decrease Balance 700 Kr	1000 Kr
Update		400 Kr
	Update	**-300 Kr**

Race condition

Two or more tasks try to **concurrently access and modify** the same memory location

Two approaches:

1 - **Use locks** to protect shared memory

2 - **Use channels** to send memory

Don’t communicate by sharing; share memory by communicating

Undefined behaviour

```
// This function has a data race and may print "1", "2", or something else
func main() {
    wait := make(chan struct{})
    n := 0
    go func() {
        n++ // read, increment, write
        close(wait)
    }()
    n++ // conflicting access
    <-wait
    fmt.Println(n) // Output: UNSPECIFIED
}
```

Demo: Undefined outcomes

Sharing by communication

```
// This is the preferred way to handle concurrent data access in Go:
func sharingIsCaring() {
    fmt.Println("Good (share memory by communicating):")

    ch := make(chan int)
    go func() {
        n := 0 // A local variable is only visible to one thread
        n++
        ch <- n // The data leaves one thread...
    }()

    n := <-ch // ...and arrives safely in another thread
    n++
    fmt.Println(n) // Output: 2
}
```

Deadlocks

Deadlock in Go

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

Unbuffered channel - blocks on send and read ops

Send 1 into ch, wait for read

Never get to this line!

Spot the deadlock

```
func Publish(text string, delay time.Duration) (chan struct{}) {
    ch := make(chan struct{})
    go func() {
        time.Sleep(delay)
        // fetch the latest news
        fmt.Println("BREAKING NEWS:", text)
   }()
    return ch
}

func main() {
    wait := Publish("Channels let goroutines communicate.", 5*time.Second)
    fmt.Println("Waiting for the news...")
    <-wait
    fmt.Println("The news is out, time to leave.")
}
```

Go run reports deadlock

```
% go run deadlock2.go
Waiting for the news...
BREAKING NEWS: Channels let goroutines communicate.
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()

/Users/ric/Dropbox/kth/teaching/dd1396-palinda/lectures/lecture-02/code
/deadlock2/deadlock2.go:21 +0xbc
exit status 2
```

Deadlock in Go, resolved

Dining Philosophers

Problem

We have a dining table with

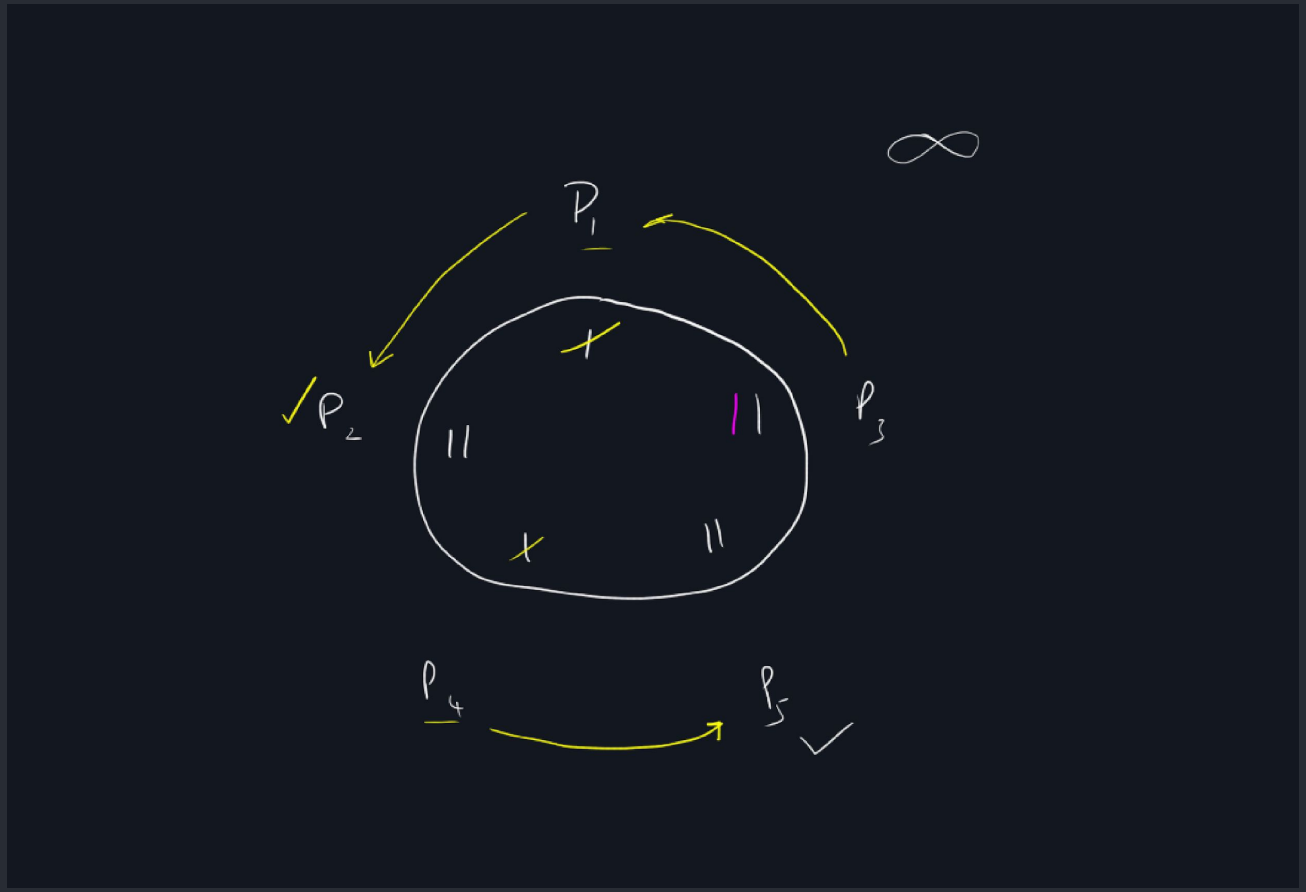
- n philosophers
- n bowls of rice
- n chopsticks

But they can only eat if they have two chopsticks. No one should starve.

Classic illustration of:

- Deadlock
- Resource starvation
- Data races

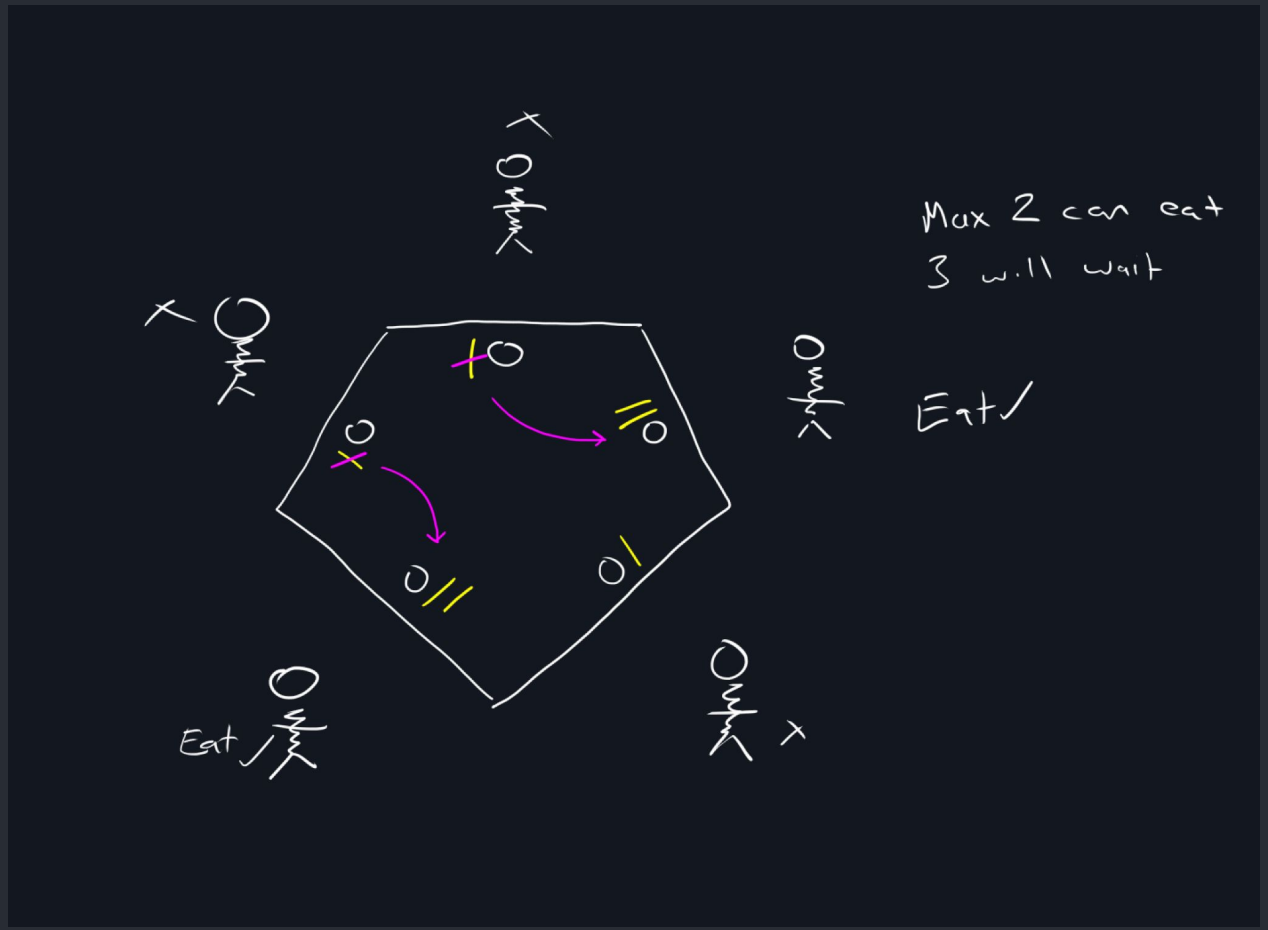
Draw: Dining problem



Deadlocks can occur when a cycle of philosophers is waiting on each others chopsticks.

Data races can occur where multiple philosophers think they have access to chopstick

Starvation can occur by some getting less than others over time



Deadlocks can occur when a cycle of philosophers is waiting on each others chopsticks.

Data races can occur where multiple philosophers think they have access to chopstick

Starvation can occur by some getting less than others over time

Video

Solving with Go

Model philosophers as **structs**

- Name, chopstick status, neighbour

Model philosopher **behaviour** as functions

- `Think( ) → Get Two Chopsticks( ) → Eat( ) → Done( )`

Model chopstick status as a **channel** and share with neighbour

- Simple buffered boolean channel

Chopstick channel semantics

My chopstick is available to others:

```
phil.chopstick <- true
```

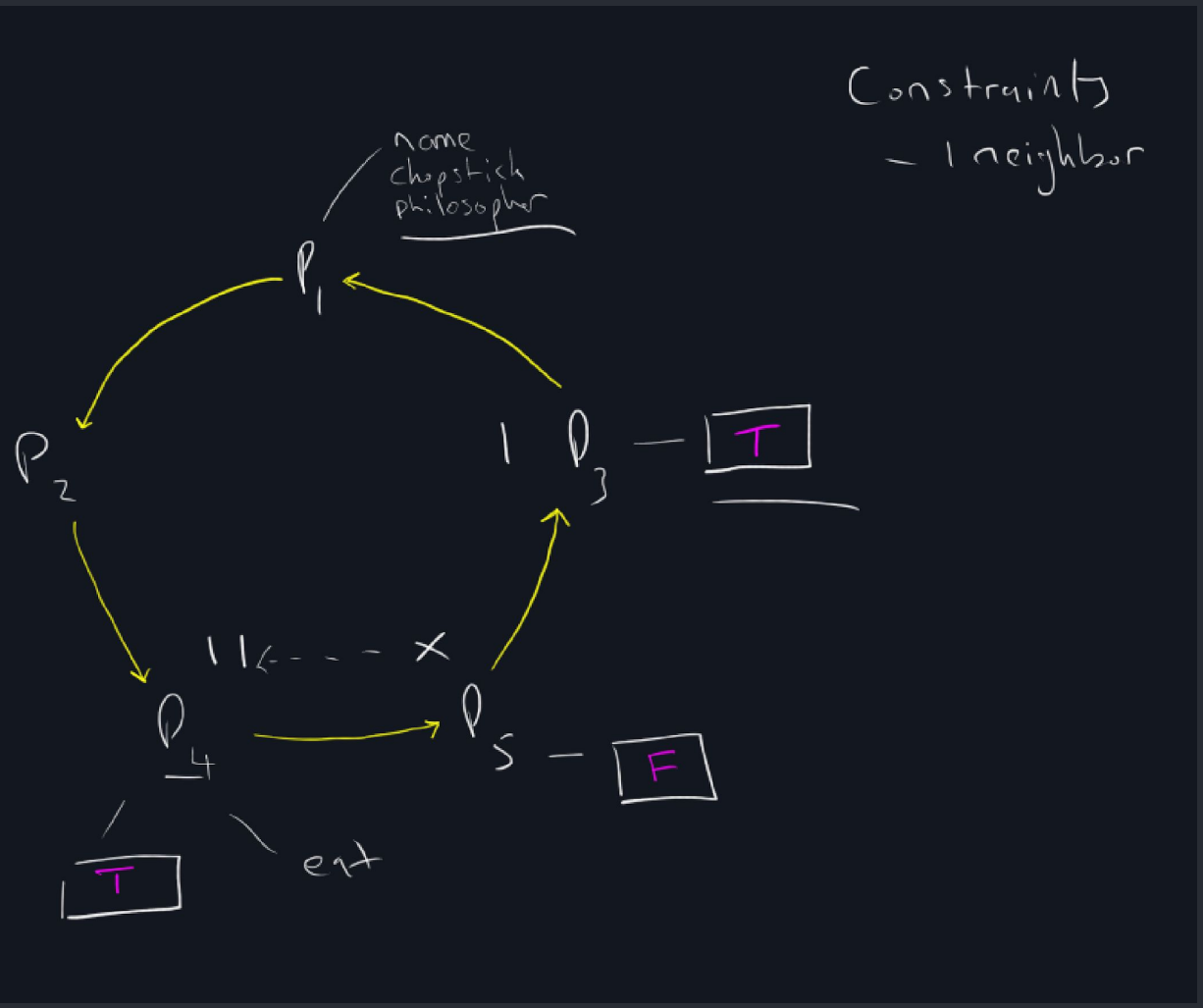
Wait for my own chopstick

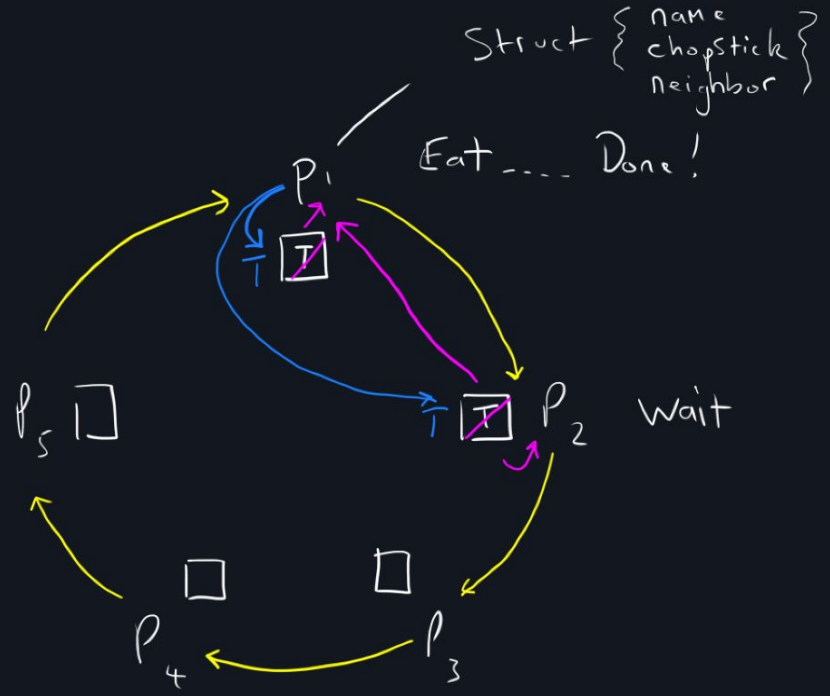
```
<- phil.chopstick
```

Wait for my neighbour’s chopstick

```
<- phil.neighbour.chopstick
```

Draw: Go solution


Constraints
- 1 neighbor
name
chopstick
philosopher
P1
P2
P3
T
P4
P5
F
T
eat



Philosophers as structs and chopstick status modelled as a boolean channel

Video

Implementation in Go

#1 Model Philosophers

```
type Philosopher struct {
    name      string
    chopstick chan bool
    neighbor  *Philosopher
}

func makePhilosopher(name string, neighbor *Philosopher) *Philosopher {
    // factory function to create and return a philosopher
    phil := &Philosopher{name, make(chan bool, 1), neighbor}

    // chopstick channel defaults to being available
    phil.chopstick <- true
    return phil
}
```

#2 Create and link philosophers

```
func main() {
    // create our 5 philosophers, and store in a slice
    names := []string{"Arendt", "Beauvoir", "Descartes", "Kant", "Locke"}
    philosophers := make([]*Philosopher, len(names))

    var phil *Philosopher
    for i, name := range names {
        phil = makePhilosopher(name, phil)
        philosophers[i] = phil
    }
    // fix the first philosophers neighbour
    philosophers[0].neighbor = phil
```

#3 Start the dining

```
    // cont...
    // start the dining! create a thread for each philosopher
    announce := make(chan *Philosopher)
    for _, phil := range philosophers {
        go phil.dine(announce)
    }

    // announce who has finished eating
    for i := 0; i < len(names); i++ {
        phil := <-announce
        fmt.Printf("🏁 %v is done dining 👋\n", phil.name)
    }
} // end of main
```

#4 Philosopher lifecycle

```
func (phil *Philosopher) dine(announce chan *Philosopher) {
    phil.think()
    phil.getChopsticks()
    phil.eat()
    phil.returnChopsticks()
    announce <- phil
}
```

#5 Thinking and eating

```
func (phil *Philosopher) think() {
    // simply wait a random number of seconds
    // presumably having deep thoughts
    fmt.Printf("🤔 %v is thinking\n", phil.name)
    time.Sleep(time.Duration(rand.Intn(3)) * time.Second)
}

func (phil *Philosopher) eat() {
    // simply wait a random number of seconds
    // enjoying the lovely food being eaten
    fmt.Printf("🍚 %v is eating\n", phil.name)
    time.Sleep(time.Duration(rand.Intn(3)) * time.Second)
}
```

#6 Getting chopsticks

```
func (phil *Philosopher) getChopsticks() {
    // remove chopstick from own channel (i.e. hold your chopstick)
    // block until it becomes available
    <- phil.chopstick
    fmt.Printf("-- %v got their chopstick\n", phil.name)

    select {
        // if your neighbour is not using their chop stick, proceed to eating
        case <-phil.neighbor.chopstick:
            fmt.Printf("-- %v got %v's chopstick\n", phil.name, phil.neighbor.name)
            fmt.Printf("// %v has two chopsticks\n", phil.name)
            return
        // too late! make your chop stick available
        case <- time.After(time.Duration(rand.Intn(2)) * time.Second):
            phil.chopstick <- true
            phil.think()
            phil.getChopsticks()
    }
}
```

#7 Returning chopsticks

```
func (phil *Philosopher) returnChopsticks() {
    // indicates that both chopsticks are now available
    // by putting them 'back into the channel'
    phil.chopstick <- true
    phil.neighbor.chopstick <- true
}
```

Five philosophers sit down at a table.
There are five bowls of rice,
but only five chopsticks...

🤔 Locke is thinking
🤔 Arendt is thinking
-- Arendt got their chopstick
-- Arendt got Locke's chopstick
🥢 Arendt has two chopsticks
🍚 Arendt is eating
🤔 Beauvoir is thinking
🤔 Kant is thinking
-- Kant got their chopstick
-- Kant got Descartes's chopstick
🥢 Kant has two chopsticks
🍚 Kant is eating
🤔 Descartes is thinking
-- Beauvoir got their chopstick
🤔 Beauvoir is thinking
🏁 Kant is done dining 👋
🏁 Arendt is done dining 👋
-- Descartes got their chopstick

https://gits-15.sys.kth.se/inda-20/dining/blob/master/dining.go

Concept Review

Concept review

Concurrency comes with challenges (data races, deadlocks, starvation)

We must be extra careful, as they can be hard to spot

Go runtime will spot some problem, but not all

Recommended Reading

Fundamentals of Concurrent Programming

- https://yourbasic.org/golang/data-races-explained/
- https://yourbasic.org/golang/detect-data-races/
- https://yourbasic.org/golang/detect-deadlock/

Go for Java Programmers

- http://yourbasic.org/golang/go-java-tutorial/

Golang website

- http://golang.org/