Concurrent Programming with Go

Concepts of Programming Languages

15 November 2018

Johannes Weigend (QAware GmbH)
University of Applied Sciences Rosenheim

Why Concurrent Programming?

- Computer clock rates do not get higher anymore (since 2004!)
- But Moores Law is still valid (Multicore!)

The modern world is parallel

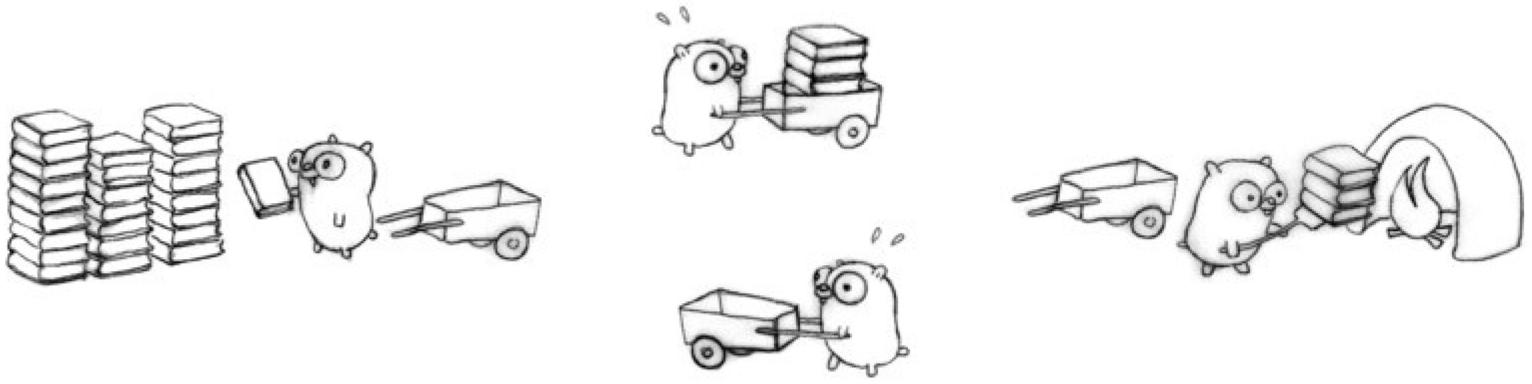
Multicore.

Networks.

Clouds of CPUs.

Loads of users.

Concurrent Programming with Go



- Don't communicate by sharing memory; share memory by communicating (Rob Pike)

Go provides:

- concurrent execution (goroutines)
- synchronization and messaging (channels)
- multi-way concurrent control (select)
- low level blocking primitives (locks) - Usually not needed!

Goroutines

A goroutine is a function running independently in the same address space as other goroutines

```
f("hello", "world") // f runs; we wait
```

```
go f("hello", "world") // f starts running
g() // does not wait for f to return
```

Like launching a function with shell's & notation.

Goroutines are not threads

(They're a bit like threads, but they're much cheaper.)

Goroutines are multiplexed onto OS threads as required.

When a goroutine blocks, that thread blocks but no other goroutine blocks.

Channels

- Go routines can use channels for safe communication
- Construct a channel

```go
c := make(chan int)     // buffer size = 0
c := make(chan int, 10) // buffer size = 10
```

- Send to channel

```go
c <- 1
```

- Read from channel

```go
x = <- c
```

- size = 0 (=default): Sender blocks until a reader requests a value from the channel
- size = n: Sender is not blocked until the buffer size is reached

Channels

Channels are typed values that allow goroutines to synchronize and exchange information.

```go
timerChan := make(chan time.Time)
go func() {
    time.Sleep(deltaT)
    timerChan <- time.Now() // send time on timerChan
}()

// Do something else; when ready, receive.
// Receive will block until timerChan delivers.
// Value sent is other goroutine's completion time.
completedAt := <-timerChan
```

Ping Pong

```go
// Ball contains the number of hits.
type Ball struct{ hits int }

func main() {
    table := make(chan *Ball)
    go player("ping", table)
    go player("pong", table)

    table <- new(Ball) // game on; toss the ball
    time.Sleep(1 * time.Second)
    <-table // game over; grab the ball
}

func player(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
```

Run

Channel and Errors

- Channel can be closed. Readers will return immediately. Successive writes will cause panic.

```
close(c)
```

- If a channel was closed, the reader gets "false" as return code (second return value)

```
x, rc := <-c
```

- Reading from a channel until closed

```
for {
    x, ok := <-c
    if !ok {
        break
    }
    // do something with x
}
// channel closed
```

Fan Out

- Read tasks from a channel and start parallel processing. Results will be written in a result channel.

```go
// FanOut reads from a channel and starts an async processing task.
// The result values of the tasks will be returned in the result channel
func FanOut(input chan int, task func(int, chan int)) chan int {
    result := make(chan int)
    go func() {
        for {
            x, ok := <-input
            if !ok {
                break
            }
            go task(x, result)
        }
    }()
    return result
}
```

Fan In

- Merge n channels into one

```go
// FanIn reads from N-Channels and forwards the result to the output channel.
func FanIn(channels []chan int, output chan int) {
    for i := 0; i < len(channels); i++ {
        // fan in
        go func(i int) {
            for {
                n, ok := <-channels[i]
                if !ok {
                    break
                }
                output <- n
            }
            fmt.Println("input channel closed: done.")
        }(i)
    }
}
```

Select

The `select` statement is like a `switch`, but the decision is based on ability to communicate rather than equal values.

```go
select {
    case v := <-ch1:
        fmt.Println("channel 1 sends", v)
    case v := <-ch2:
        fmt.Println("channel 2 sends", v)
    default: // optional
        fmt.Println("neither channel was ready")
}
```

Go really supports concurrency

Really.

It's routine to create thousands of goroutines in one program.
(Once debugged a program after it had created 1.3 million.)

Stacks start small, but grow and shrink as required.

Goroutines aren't free, but they're very cheap.

More information about Go and concurrency

youtu.be/f6kdp27TYZs?t=1 (https://youtu.be/f6kdp27TYZs?t=1)

Java like BlockingQueue with Channels

```go
// BlockingQueue is a FIFO container with a fixed capacity.
// It  blocks a reader when it is empty and a writer when it is full.
type BlockingQueue struct {
    channel chan interface{}
}

// NewBlockingQueue constructs a BlockingQueue with a given capacity.
func NewBlockingQueue(capacity int) *BlockingQueue {
    q := BlockingQueue{make(chan interface{}, capacity)}
    return &q
}

// Put puts an item in the queue and blocks it the queue is full.
func (q *BlockingQueue) Put(item interface{}) {
    q.channel <- item
}

// Take takes an item from the queue and blocks if the queue is empty.
func (q *BlockingQueue) Take() interface{} {
    return <-q.channel
}

// EOF
```

Java like BlockingQueue - Test

```
func TestBlockingQueue(t *testing.T) {
    bq1 := NewBlockingQueue(1)
    done := make(chan bool)
    // slow writer
    go func(bq *BlockingQueue) {
        bq.Put("A")
        time.Sleep(100 * time.Millisecond)
        bq.Put("B")
        time.Sleep(100 * time.Millisecond)
        bq.Put("C")
    }(bq1)
    // reader will be blocked
    go func(bq *BlockingQueue) {
        item := bq.Take()
        fmt.Printf("Got %v\n", item)
        item = bq.Take()
        fmt.Printf("Got %v\n", item)
        item = bq.Take()
        fmt.Printf("Got %v\n", item)
        done <- true
    }(bq1)

    <-done
}
```

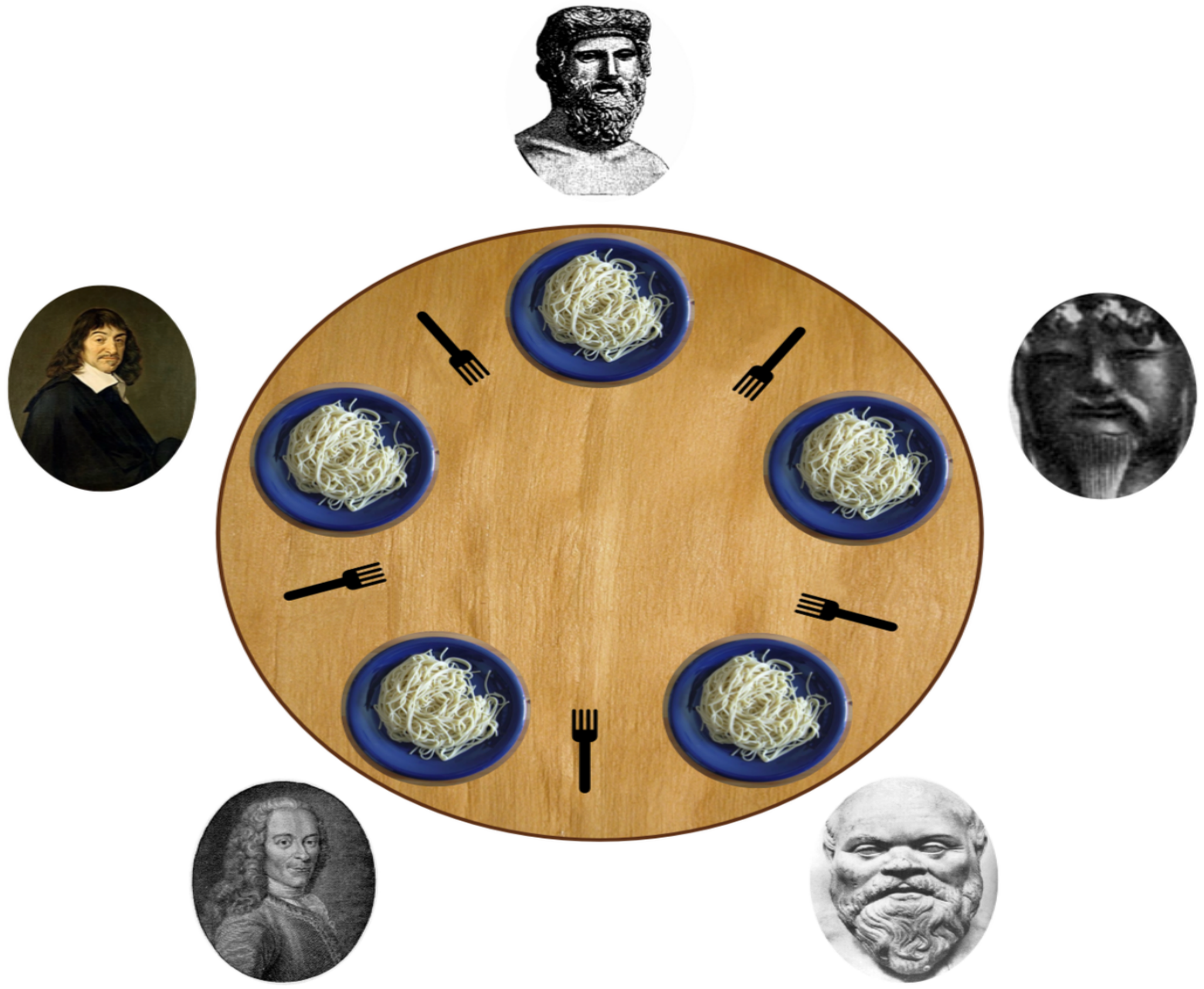
Java like BlockingQueue with Locks (Low Level)

```go
// BlockingQueue is a FIFO container with a fixed capacity.
// It  blocks a reader when it is empty and a writer when it is full.
type BlockingQueue struct {
    m        sync.Mutex
    c        sync.Cond
    data     []interface{}
    capacity int
}

// NewBlockingQueue constructs a BlockingQuee with a given capacity.
func NewBlockingQueue(capacity int) *BlockingQueue {
    q := new(BlockingQueue)
    q.c = sync.Cond{L: &q.m}
    q.capacity = capacity
    return q
}

// A1
```

Java like BlockingQueue with Locks (Low Level)

```go
// Put puts an item in the queue and blocks it the queue is full.
func (q *BlockingQueue) Put(item interface{}) {
    q.c.L.Lock()
    defer q.c.L.Unlock()

    for q.isFull() {
        q.c.Wait()
    }
    q.data = append(q.data, item)
    q.c.Signal()
}

// Take takes an item from the queue and blocks if the queue is empty.
func (q *BlockingQueue) Take() interface{} {
    q.c.L.Lock()
    defer q.c.L.Unlock()

    for q.isEmpty() {
        q.c.Wait()
    }
    result := q.data[0]
    q.data = q.data[1:len(q.data)]
    q.c.Signal()
    return result
}

// A2
```

Dining Philosophers



Dining Philosophers - Hints

- Never grab one fork and wait for the other. This is a deadlock situation.
- If you cant get the second fork, you should immediately release the first one.
- The table itself should be a Go Routine and return the forks to a requesting philosopher, this makes synchronization easy (the table is single threaded)
- The philosopher loop looks like this:

```go
// Main loop
func (p *Philosopher) run() {
    for {
        p.takeForks()
        p.eat()
        p.putForks()
        p.think()
    }
}
```

Wrong Solutions

There are many wrong solution on the web.
Most of them share the problem that the Philosopher picks up the left fork (implemented with channels or locks) and immediately the right fork.
The problem arises, when the second fork is in use. There is a potential deadlock, when all Philosophers wait on the second fork.
In theory a deadlock occurs if there is a cycle in the Resource Allocation Graph.

play.golang.org/p/rXCotNNY24 (https://play.golang.org/p/rXCotNNY24)

Summary

- With Go you can solve sync problems with channels
- Channels use Message Passing instead of locks
- Go has a low level lock API, but this is seldom needed
- It is possible to port all classes from java.util.locking easily

Thank you

Johannes Weigend (QAware GmbH)
University of Applied Sciences Rosenheim
johannes.weigend@qaware.de (mailto:johannes.weigend@qaware.de)
http://www.qaware.de (http://www.qaware.de)
@johannesweigend (http://twitter.com/johannesweigend)