Basic concurrency

Preparation for Lab 2

You can quickly run the code from this lecture by pasting the program into the Go Playground: play.golang.org

Hello World 2.0

```
package main

import (
    "fmt"
    "time"
)

func say(something string) {
    fmt.Println(something)
}

func main() {
    go say("Hello " + "World")
    time.Sleep(1 * time.Second)
}
```

```
go say("Hello " + "World")
```

Start a new *goroutine* running the function `say()`.

The evaluation of `"Hello " + "World"` happens in the **current** goroutine.

The execution of `say()` happens in a **new** goroutine.

Goroutine

A goroutine is a **green thread**.

It is an application-level thread.

It is much more lightweight than an OS-thread.

Many goroutines can run on one OS-thread.

```
time.Sleep(1 * time.Second)
```

A go program terminates if the function `main()` returns.

It will not wait for other goroutines to finish.

The *hacky* way of fixing that is putting `main()` to sleep.

The *correct* way is using channels (coming up shortly).

Memory

Goroutines run in the same address space, so access to shared memory must be synchronized.

However, usually we can avoid sharing memory at all!

Channels

Channel provides a way to send a message between two goroutines.

```
channel := make(chan int)
```

Create a new channel.

```
someInt := 5
channel <- someInt
```

Send `someInt` to the channel

```
v := <-channel
```

Receive from channel and assign the received value to v.

Parallel sum

```
func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c <- sum // send sum to c
}

func main() {
    s := []int{7, 2, 8, -9, 4, 0}

    mid := len(s)/2

    leftSum := make(chan int)
    rightSum := make(chan int)

    go sum(s[:mid], leftSum)
    go sum(s[mid:], rightSum)

    x := <-leftSum
    y := <-rightSum

    fmt.Println(x, y, x+y)
}
```

```
leftSum := make(chan int)
rightSum := make(chan int)
```

Make two new channels.

Goroutines need to be void functions.

In order to return from a goroutine we need to use a channel.

```
go sum(s[:mid], leftSum)
go sum(s[mid:], rightSum)
```

Start two new goroutines.

This will allow us to compute the sums of two halves of the slice **in parallel**.

```
x := <-leftSum
y := <-rightSum
```

Receive the results from the two goroutines.

Sends and receives block until the other side is ready.

For each *send* there must be a corresponding *receive*.

Otherwise you're risking deadlock!

Note that `main()` is itself a goroutine.

Buffered channels

A buffered channel is a type of a FIFO data structure.

(First in, first out)

```
ch := make(chan int, 3) // ch = []
ch <- 1                 // ch = [1]
ch <- 2                 // ch = [1, 2]
ch <- 3                 // ch = [1, 2, 3]
fmt.Println(<-ch) // 1      ch = [2, 3]
fmt.Println(<-ch) // 2      ch = [3]
fmt.Println(<-ch) // 3      ch = []
```

Blocking

Buffered channels accept a limited number of values without a corresponding receiver for those values.

Sends to a buffered channel block only when the buffer is full.

Receives only block when the buffer is empty.

Hello World 3.0

```
package main

import "fmt"

func say(something string, done chan<- bool) {
    fmt.Println(something)
    done <- true
}

func main() {
    done := make(chan bool)
    go say("Hello " + "World", done)
    <-done
}
```

```
func main() {
    done := make(chan bool)
    go say("Hello " + "World", done)
    <-done
}
```

Use a `chan bool` to synchronise execution across goroutines.

Here we start a new goroutine and then wait for it to finish using the *receive* operation.

```
func say(something string, done chan<- bool) {
    fmt.Println(something)
    done <- true
}
```

Perform some operation (print in this case) and report the work has been done by sending `true` on the `done` channel.

```
done chan<- bool
```

Channel of type `chan<-` is send-only. Receiving from a channel of this type is illegal (a compile-time error).

Functions in Go

You need to know a few extra things about functions in Go for this lab.

Functions are values

They can be passed around just like other values.

They can be return types and arguments to other functions.

```
func compute(fn func(float64, float64) float64) float64 {
        return fn(3, 4)
}

func main() {
        hypot := func(x, y float64) float64 {
                return math.Sqrt(x*x + y*y)
        }
        fmt.Println(hypot(5, 12))       // = 13
        fmt.Println(compute(hypot))     // = 5
        fmt.Println(compute(math.Pow))  // = 81
}
```

Closures

A closure is a function value that references variables from outside its body. The closure may access and modify the referenced variables.

```
func adder() func(int) int {
        sum := 0
        return func(x int) int {
                sum += x
                return sum
        }
}

func main() {
        pos, neg := adder(), adder()
        for i := 0; i < 10; i++ {
                fmt.Println(pos(i), neg(-i))
        }
}
```

gobyexample.com

tour.golang.org

Look at the above resources to learn more about Go's syntax and features.

Acknowledgements

Some of the presented code snippets have been adapted and modified from tour.golang.org under the BSD-3 license.

Some of the presented code snippets have been adapted and modified from gobyexample.com under the CC 3.0 license.