Advanced concurrency

Preparation for Lab 3

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

Select

```
select {
    case a := <-chanA:
        ...
    case b := <-chanB:
        ...
    ...
}
```

The select statement lets a goroutine wait on multiple communication operations.

A select blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

Slow Sender & Fast Sender

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

    strings := make(chan string)
    go slowSender(strings)

    for { // = while(true)
        select {
        case s := <-strings:
            fmt.Println("Received a string", s)
        case i := <-ints:
            fmt.Println("Received an int", i)
        }
    }
}
```

```
ints := make(chan int)
go fastSender(ints)

strings := make(chan string)
go slowSender(strings)
```

Start two new goroutines.

`fastSender` sends an integer every 500ms.

`slowSender` sends a string every 2s.

```
for { // = while(true)
    select {
    case s := <-strings:
        fmt.Println("Received a string", s)
    case i := <-ints:
        fmt.Println("Received an int", i)
    }
}
```

Block until a message is received from one of the channels.

In this case we print the message and block again.

Example output

```
$ go run select.go

Received an int 0
Received an int 1
Received an int 2
Received a string I am the slowSender
Received an int 3
Received an int 4
Received an int 5
Received an int 6
Received a string I am the slowSender
Received an int 7
Received an int 8
Received an int 9
Received an int 10
Received a string I am the slowSender

...
```

Timers

```
func main() {
    timer := time.After(2 * time.Second)
    fmt.Println("Timer started")
    <-timer
    fmt.Println("Timer expired")
}
```

We often want to execute code at some point in the future.

The built in timers and tickers allow us to schedule code executing in the future.

```
timer := time.After(2 * time.Second)
```

`time.After()` creates a new timer and returns a channel of type `<-chan Time`.

```
<-timer
```

A message will be sent on the channel once the specified duration has elapsed.

In this case it's 2 seconds.

For-range loop on channels

```
func main() {
    queue := make(chan string, 2)
    queue <- "one"
    queue <- "two"
    close(queue)

    for elem := range queue {
        fmt.Println(elem)
    }
}
```

```
close(queue)
```

Closing a channel indicates that no more values will be sent.

This can be useful to communicate completion to the channel's receivers.

```
for elem := range queue {
    fmt.Println(elem)
}
```

This `range` iterates over each element as it is received from queue.

We closed the `queue` channel so the iteration terminates after receiving the 2 elements.

If we didn't close the channel this for-range loop would block forever waiting to receive more messages.

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 and comments have been adapted and modified from tour.golang.org under the BSD-3 license. Copyright (c) 2011 The Go Authors.

Some of the presented code snippets and comments have been adapted and modified from gobyexample.com under the CC 3.0 license. Copyright (c) Mark McGranaghan.