go routines

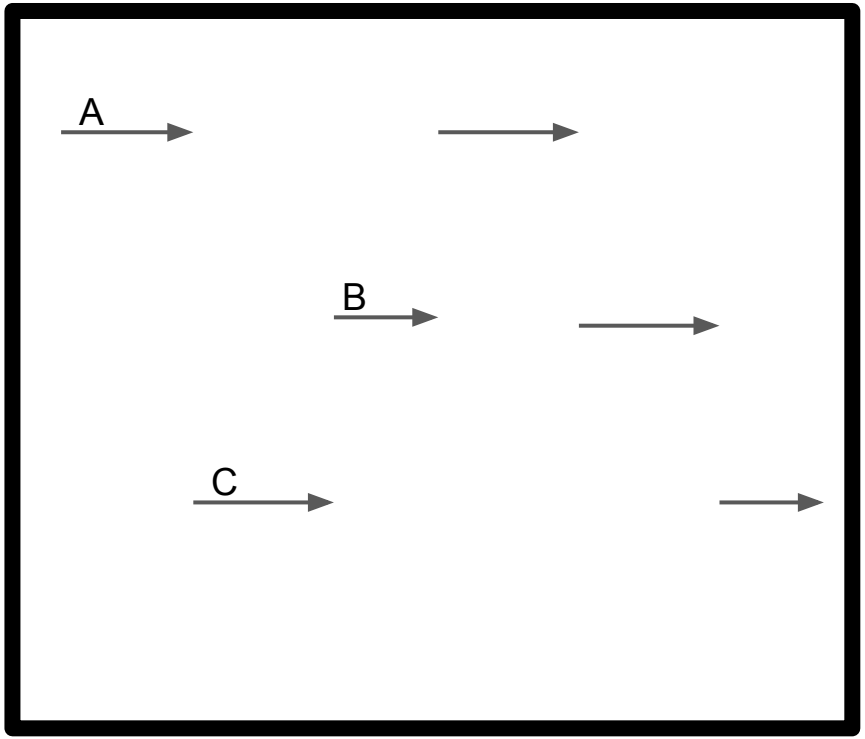
concurrency and parallelism

concurrency and parallelism

“**Concurrency** is the composition of independently executing processes, while **parallelism** is the simultaneous execution of (possibly related) computations. **Concurrency** is about dealing with lots of things at once. **Parallelism** is about doing lots of things at once.”

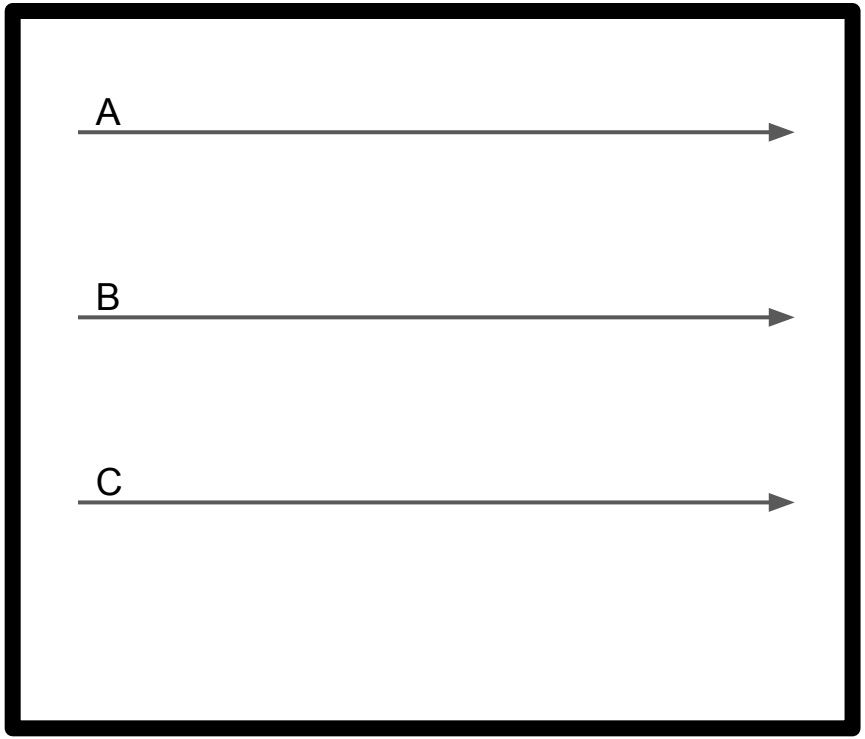
concurrency

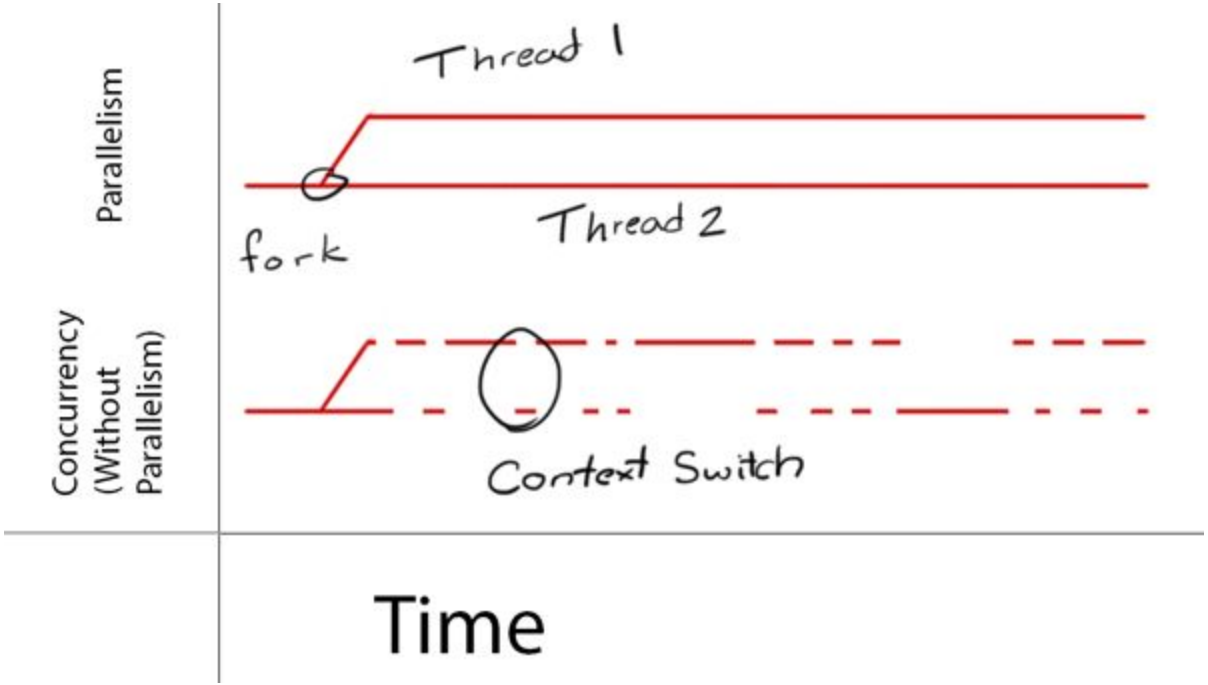
doing many things, but only one at a time
“multitasking”



parallelism

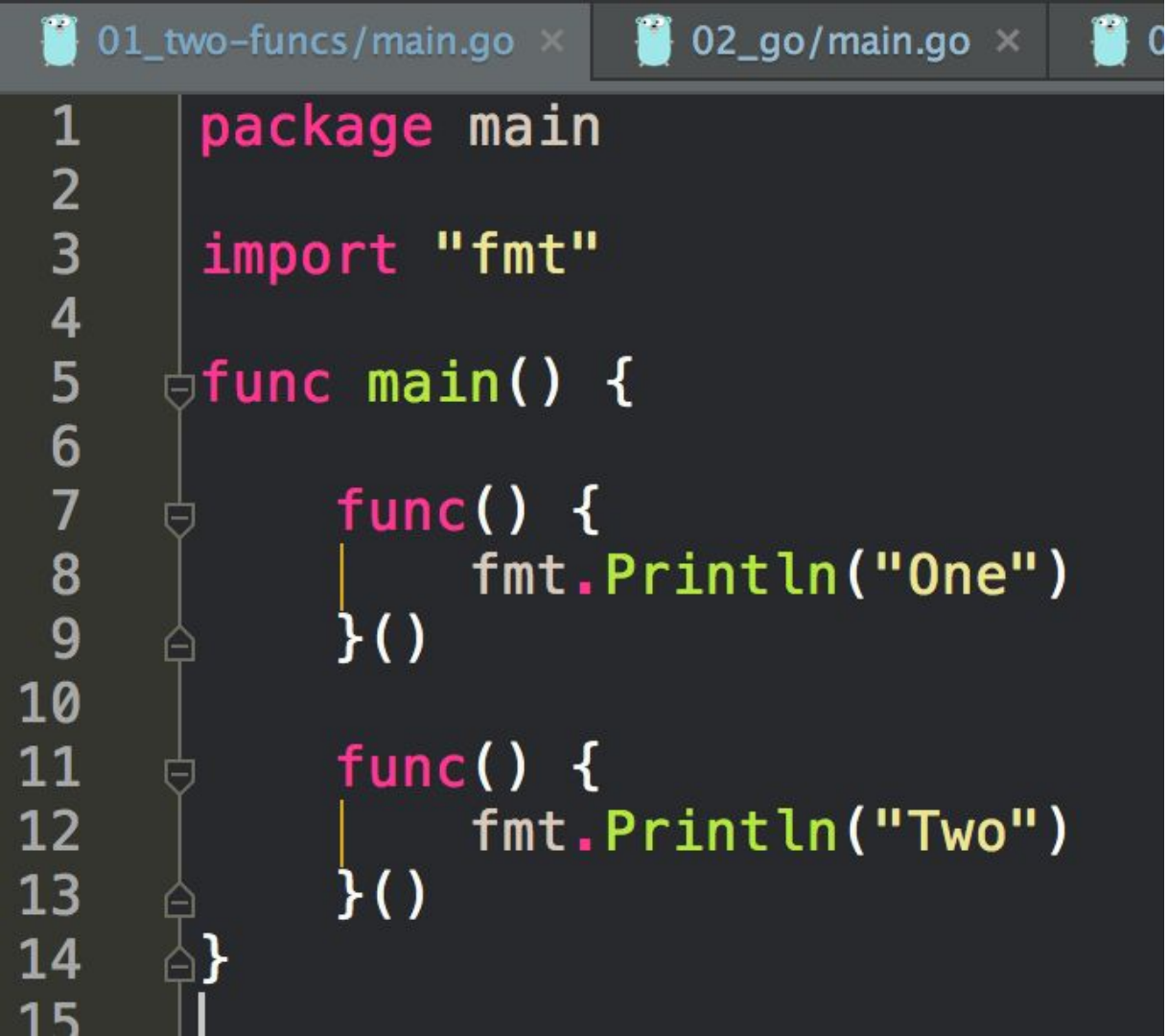
doing many things at the same time





http://cdn1.raywenderlich.com/wp-content/uploads/2014/01/Concurrency_vs_Parallelism.png

go



these files are all from
pluralsight's concurrency training
http://www.pluralsight.com/courses/go-concurrent-programming

02_go/main.go × 03_go_sleep/main.go × 04_

```go
package main

import "fmt"

func main() {

    go func() {
        fmt.Println("One")
    }()

    go func() {
        fmt.Println("Two")
    }()
}

```

03_go_sleep/main.go × 04_go_sleep_loop/main.go ×

```go
package main

import (
    "fmt"
    "time"
)

func main() {

    go func() {
        fmt.Println("One")
    }()

    go func() {
        fmt.Println("Two")
    }()

    dur, _ := time.ParseDuration("1s")
    time.Sleep(dur)
}
```

main.go

```
4       "fmt"
5       "time"
6   )
7
8   func main() {
9
10      godur, _ := time.ParseDuration("10ms")
11
12      go func() {
13          for i := 0; i < 100; i++ {
14              fmt.Println("One")
15              time.Sleep(godur)
16          }
17      }()
18
19      go func() {
20          for i := 0; i < 100; i++ {
21              fmt.Println("TwoTwo")
22              time.Sleep(godur)
23          }
24      }()
25
26      dur, _ := time.ParseDuration("1s")
27      time.Sleep(dur)
28  }
```

Terminal

```
04_go_sleep_loop $ go run main.go
One
TwoTwo
One
TwoTwo
One
TwoTwo
One
TwoTwo
One
TwoTwo
One
TwoTwo
One
```

exercise

create a program that launches multiple threads

channels

01/main.go × 02/main.go × 03_buffered-channel/main.go × 04/main.go ×

```go
package main

import "fmt"

func main() {
    ch := make(chan string)
    fmt.Println(<-ch)   // program waits here to drain the channel
                        // since there is nothing to receive from the channel
                        // our program is in a deadlock
}
```

print-hello

```go
package main

import (
    "fmt"
)

func main() {
    ch := make(chan string)

    ch <- "Hello"   // program waits here until channel is drained
                    // since our main thread is waiting on the above line
                    // the main thread never gets to the statement below that drains the channel
                    // our program is in a deadlock
    fmt.Println(<-ch)
}

/*
we can give our channel
the capacity to store messages
which will allow the sender & receiver to not have to wait on each other
we'll see this in the next file, 03
this is known as creating a "buffered" channel
*/
```

print-hello

print-hello


03_buffered-channel/main.go
04/main.go
1 package main
2
3 import (
4 "fmt"
5)
6
7 func main() {
8 ch := make(chan string,1)
9
10 ch <- "Hello"
11
12 fmt.Println(<-ch)
13 }

print-hello

main.go

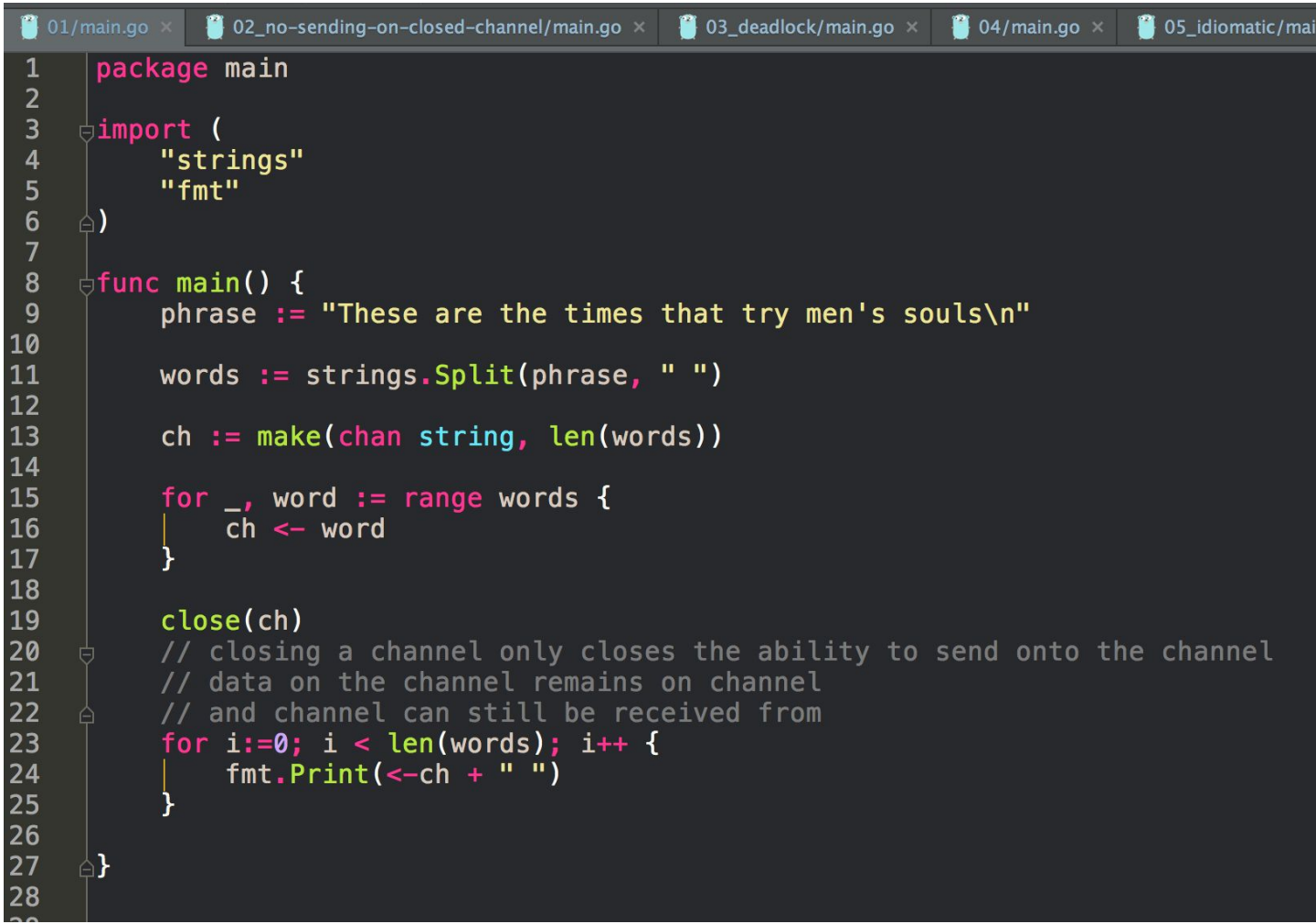
```go
package main

import (
    "fmt"
)

func main() {
    ch := make(chan string,1)

    ch <- "Hello"

    fmt.Println(<-ch)

    ch <- "Go"

    fmt.Println(<-ch)
}
```

buffering

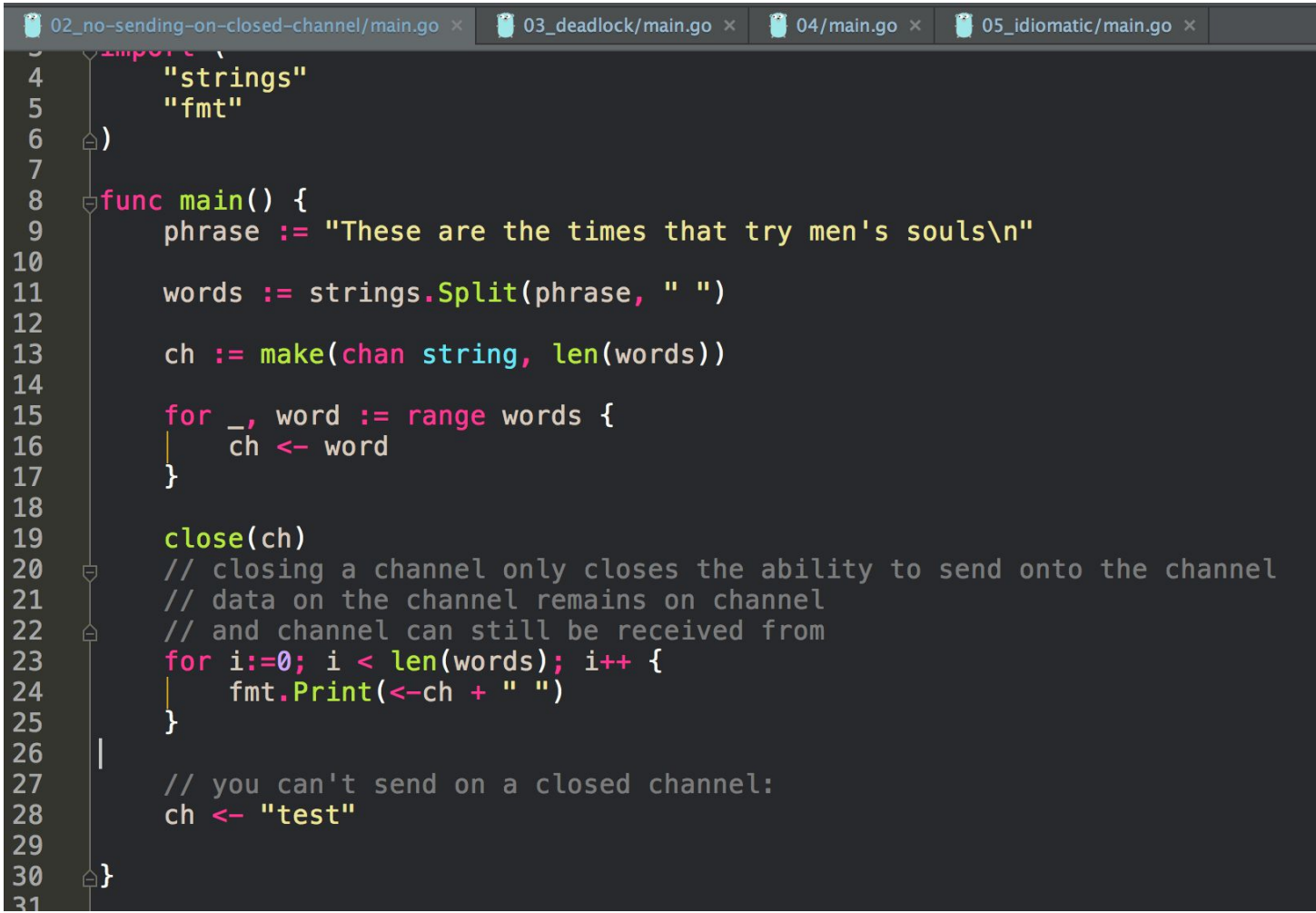
main.go

```go
package main

import (
    "strings"
    "fmt"
)

func main() {
    phrase := "These are the times that try men's souls\n"

    words := strings.Split(phrase, " ")

    ch := make(chan string, len(words))

    for _, word := range words {
        ch <- word
    }

    for i:=0; i < len(words); i++ {
        fmt.Print(<-ch + " ")
    }

}
```

closing channel



closing channel



closing channel

03_deadlock/main.go | 04/main.go | 05_idiomatic/main.go

```go
package main

import (
    "strings"
    "fmt"
)

func main() {
    phrase := "These are the times that try men's souls\n"

    words := strings.Split(phrase, " ")

    ch := make(chan string, len(words))

    for _, word := range words {
        ch <- word
    }

    for {
        if msg, ok := <- ch; ok {   // we check to see if channel is closed
            fmt.Print(msg + " ")
        } else {
            break
        }
    }
                                    // but we haven't closed the channel yet
                                    // so the for loop on line 19
                                    // loops through all of the words on the channel
                                    // then waits for another word to be put on the channel
                                    // and as no word is ever going to be put on the channel
                                    // program is in deadlock
}
```

closing channel

04/main.go × 05_idiomatic/main.go ×

```go
package main

import (
    "strings"
    "fmt"
)

func main() {
    phrase := "These are the times that try men's souls\n"

    words := strings.Split(phrase, " ")

    ch := make(chan string, len(words))

    for _, word := range words {
        ch <- word
    }

    close(ch) // closing the channel removes deadlock

    for {
        if msg, ok := <- ch; ok {   // when channel is closed
            fmt.Print(msg + " ")    // this for loop will no longer be waiting
        } else {                    // to receive something from channel
            break                   // the loop will break
        }
    }

}
```

closing channel

main.go

```go
package main

import (
    "strings"
    "fmt"
)

func main() {
    phrase := "These are the times that try men's souls\n"

    words := strings.Split(phrase, " ")

    ch := make(chan string, len(words))

    for _, word := range words {
        ch <- word
    }

    close(ch)

    for msg := range ch {
            fmt.Print(msg + " ")
    }
    // range knows to stop looping
    // when there is nothing left in a closed channel
}
```

exercise

create a program that demonstrates
putting a message onto a channel
and receiving that message from a channel
using a buffer

exercise

create a program that demonstrates a deadlock

exercise

create a program that uses a range loop
to loop over a closed channel
and display strings retrieved from the channel

these files are all from caleb's book
http://www.golang-book.com/books/intro/10

02/main.go × 03/main.go ×

```go
package main

import "fmt"

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
    }
}

func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

main.go

```go
package main

import (
    "fmt"
    "time"
    "math/rand"
)

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
        amt := time.Duration(rand.Intn(250))
        time.Sleep(time.Millisecond * amt)
    }
}

func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

```
7 : 8
2 : 8
0 : 9
5 : 9
1 : 7
6 : 9
9 : 9
1 : 8
2 : 9
7 : 9
4 : 8
4 : 9
```

main.go

```go
package main

import (
    "fmt"
    "time"
)

func pinger(c chan string) {
    for i := 0; ; i++ {
        c <- "ping"             // program waits here until channel is drained
    }
}

func ponger(c chan string) {
    for i := 0; ; i++ {
        c <- "pong"             // program waits here until channel is drained
    }
}

func printer(c chan string) {
    for {
        msg := <- c
        fmt.Println(msg)
        time.Sleep(time.Second * 1)
    }
}

func main() {
    var c chan string = make(chan string)

    go pinger(c)
    go ponger(c)
    go printer(c)

    var input string
    fmt.Scanln(&input)
}
```

Review

- Concurrency
  - doing many things
- Parallelism
  - doing many things at the same time
- go
  - launches thread
  - threads are virtual
- chan
  - allows communication, orders events