

Switching between goroutines

This lesson explains how to transfer control between different goroutines via a channel.

WE'LL COVER THE FOLLOWING ^

- Waiting between a number of goroutines
- Nulling idiom
- Server backend pattern

Waiting between a number of goroutines

Getting the values out of different, concurrently executing goroutines can be accomplished with the `select` keyword, which closely resembles the `switch` control statement, and is sometimes called the *communications switch*. It acts like an *are you ready* polling mechanism. The `select` listens for incoming data on channels, but there could also be cases where a value is not sent on a channel:

```
select {
  case u:= <- ch1:
    ...
  case v:= <- ch2:
    ...
  default: // no value ready to be received
    ...
}
```

The `default` clause is optional. The *fall through* behavior, like in the normal switch, is not permitted. A `select` is terminated when a break or return is executed in one of its cases. What `select` does is, it chooses which of the multiple communications listed by its cases can proceed.

- If all are blocked, it waits until one can proceed.

- When none of the channel operations can proceed, and the `default` clause is present, then this is executed because the default is always runnable, or ready to execute.
- If multiple can proceed, it chooses one at random.

Using a send operation in a `select` statement with a `default` case guarantees that the `send` will be non-blocking! If there are no cases, the `select` blocks execution forever.

The `select` statement implements a kind of *listener* pattern, and it is mostly used within an (infinite) loop. When a certain condition is reached, the loop is exited via a `break` statement.

Look at the following program:

```
package main
import (
    "fmt"
    "time"
    "runtime"
)

func main() {
    runtime.GOMAXPROCS(2)
    ch1 := make(chan int)
    ch2 := make(chan int)
    go pump1(ch1)
    go pump2(ch2)
    go suck(ch1, ch2)
    time.Sleep(1e9)
}

func pump1(ch chan int) {
    for i:=0; ; i++ {
        ch <- i*2
    }
}

func pump2(ch chan int) {
    for i:=0; ; i++ {
        ch <- i+5
    }
}

func suck(ch1 chan int, ch2 chan int) {
    for {
        select {

            case v:= <- ch1:
                fmt.Printf("Received on channel 1: %d\n", v)
```

```

        case v:= <- ch2:
            fmt.Printf("Received on channel 2: %d\n", v)
    }
}
}

```



Goroutine with Select Statement

In above program, there are *two* channels `ch1` and `ch2` (defined at **line 10** and **line 11**) and 3 goroutines `pump1()` (started at **line 12**), `pump2()` (started at **line 13**) and `suck()` (started at **line 14**): a typical *producer-consumer* pattern.

Now, look at the header of the function `pump1()` at **line 18**. In an infinite loop at **line 20**, `ch` is filled with integers (`i*2`). Similarly, look at the header of the function `pump2()` at **line 24**. In an infinite loop at **line 26**, `ch` is filled with integers (`i+5`).

The `suck()` function polls for input also in a non-ending loop (from **line 31** to **line 38**), and takes in the integers from `ch1` and `ch2` in the `select` clause (**line 32**), and outputs them. If the *first* case : `case v:= <- ch1:` is *true* then we get an output from **line 35**. But, if the *second* case : `case v:= <- ch2:` is *true* then we get an output from **line 37**.

At **line 15**, the program is terminated in the `main()` function after 1 second.

Nulling idiom

A channel that gets the value *nil* blocks further reading from it. This is applied in the so-called **nulling idiom**, which can be applied when reading from a number of channels inside a select statement. As soon as all the values in a channel have been read, we set the channel to *nil*, effectively blocking further reading from it:

Environment Variables



Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```

package main
import "fmt"
import "os"
import "strconv"

func iter(b int, c chan int) {
    for i := 0; i < b; i++ {
        c <- i // put integers from 0 to n-1 on channel c
    }
    close(c)
}

func main() {
    n, _ := strconv.Atoi(os.Args[1]) // line arg. converted to integer
    a := make(chan int)
    b := make(chan int)
    go iter(n, a)
    go iter(n, b)
    for a != nil || b != nil {
        select {
            case x, ok := <- a: // takes int value from channel a
                if ok {
                    go fmt.Println(x)
                } else { // if channel a is closed
                    a = nil
                }
            case y, ok := <- b: // takes int value from channel b
                if ok {
                    fmt.Println(y)
                } else { // if channel b is closed
                    b = nil
                }
        }
    }
}

```

Click the **RUN** button, and wait for the terminal to start. Type `go run main.go 10` and press ENTER.

Note: You can also try any other number as a command-line argument instead of **10** to check the variation.

At **line 14**, the command-line argument is converted to an integer `n`, ignoring possible errors. At **line 15** and **line 16**, we make *two* channels `a` and `b`, respectively. Then, at **line 17**, we start a goroutine with the `iter()` function, passing `n` and channel `a`. Similarly, at **line 18**, we start a goroutine with the `iter()` function, passing `n`, and channel `b`.

The `iter()` function is defined from **line 6** to **line 11**. It puts integers from **0** to `n-1` on its channel and then closes it.

Back in `main()`, a for-loop is started at **line 19**, which continues as long as one of the *two* channels has a value different from `nil`. The `select` at **line 20** chooses between two cases:

- `x, ok := <- a`: takes an integer value `x` from channel `a`. If it's *true*, then `x` will be printed.
- `y, ok := <- b`: takes an integer value `y` from channel `b`. If it's *true*, then `y` will be printed.

If reading of the channel fails because it is closed, `ok` will be false, and the channel will get the value `nil`. When both channels are closed, they become `nil`, and the for-loop stops.

Server backend pattern

Often a server is implemented as a background goroutine which loops forever and processes values of channels via a `select` from within that loop:

```
// Backend goroutine.
func backend() {
    for {
        select {
            case cmd := <-ch1:
                // Handle ...
            case cmd := <-ch2:
                ...
            case cmd := <-chStop:
                // stop server
        }
    }
}
```

Other parts of the application send values on the channels `ch1`, `ch2`, and so on. A `stop` channel is used for a clean termination of the server process. Another possible (but less flexible) pattern is that all clients post their request on a `chRequest`, and the backend routine loops over this channel, processing requests according to their nature in a `switch`:

```
func backend() {
    for req := range chRequest {
```

```
for req := range chReq {
    switch req.Subject() {
        case A1: // Handle case ...
        case A2: // Handle case ...
        default:
            // Handle illegal request ...
            // ...
    }
}
}
```

Now that you're familiar with organizing channel reception via the select construct, the next lesson brings you a challenge to solve.