GO on the toilet

episode #14

Handling OS signals

```
func main() {
    sigs := make(chan os.Signal, 1)
    exit := make(chan struct{})
    signal.Notify(sigs, syscall.SIGTERM, syscall.SIGHUP)

    go func() {
       for {
             switch <-sigs {
             case syscall.SIGTERM:
                   fmt.Println("\rcleaning up and exiting")
                   cleanup()
                   exit <- struct{}{}
             case syscall.SIGHUP:
                   fmt.Println("\rreloading configuration")
                   reload()
             }
       }
    }()
    // something useful going on here
    <-exit
}
```

Fundamental building block of signal handling is **`Notify`** function in the **`signal`** package. It accepts channel of **`os.signal`**, and signals which will be relayed to that channel. What we do here is we listen to SIGTERM and SIGHUP signals. SIGTERM is a termination signal sent by kill command by default. SIGHUP is the hang up signal which in the early days (according to wikipedia) was used to inform the process that connection on the serial line with terminal on the other end was dropped. These days long running applications (aka daemons) like nginx or docker use it to tell the process "reload your config".

We don't listen to other signals here. For example sending SIGINT (ctrl+c) to the process will not be caught by our code and will be handled in the usual way.

Note that **`sigs`** channel has single element buffer. According to the documentation you can have any buffer size you want. Only limitation is that if the rate of incoming signals is higher than buffer size you might lose some signals. Example here would be that if **`reload()`** takes 3 seconds, and process will receive immediately SIGHUP, SIGHUP and SIGTERM signals, termination signal might get lost.

Last but not least, catching signals is happening in a different goroutine because we don't want to block anything useful (for example, running in a foreground HTTP service).