Starting an External Command or Program

This lesson provides an explanation about how to restart a program externally in case of a panic.

WE'LL COVER THE FOLLOWING

- Restart program if `panic()` occurs:

The `os` package contains the function `StartProcess` to call or start external OS commands or binary executables; its 1st argument is the process to be executed, the 2nd can be used to pass some options or arguments, and the 3rd is a struct that contains basic info about the OS-environment. It returns the *process* id (pid) of the started process, or an error if it failed.

The `exec` package contains the structures and functions to accomplish the same task more easily; most important are `exec.Command(name string, arg ...string)` and `Run()`. The function `exec.Command(name string, arg ...string)` needs the name of an OS command or executable and creates a `Command` object, which can then be executed with `Run()` which uses this object as its receiver. The following program (which only works under Linux because Linux commands are executed) illustrates their use:

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"
    "os/exec"
        "os"
)

func main() {
```

```
// 1) os.StartProcess //
/*********************/
/* Linux: */
        env := os.Environ()
        procAttr := &os.ProcAttr{
                        Env: env,
                        Files: []*os.File{
                                os.Stdin,
                                os.Stdout,
                                os.Stderr,
                        },
                }
        // 1st example: list files
        pid, err := os.StartProcess("/bin/ls", []string{"ls", "-l"}, procAttr)
        if err != nil {
                        fmt.Printf("Error %v starting process!", err)  //
                        os.Exit(1)
        }
        fmt.Printf("The process id is %v", pid)
        // 2nd example: show all processes
        pid, err = os.StartProcess("/bin/ps", []string{"-e", "-opid,ppid,comm"}, procAttr)
        if err != nil {
                        fmt.Printf("Error %v starting process!", err)  //
                        os.Exit(1)
        }
        fmt.Printf("The process id is %v", pid)
/* Output 1st:
The process id is &{2054 0}total 2056
-rwxr-xr-x 1 ivo ivo 1157555 2011-07-04 16:48 Mieken_exec
-rw-r--r-- 1 ivo ivo    2124 2011-07-04 16:48 Mieken_exec.go
-rw-r--r-- 1 ivo ivo   18528 2011-07-04 16:48 Mieken_exec_go_.6
-rwxr-xr-x 1 ivo ivo  913920 2011-06-03 16:13 panic.exe
-rw-r--r-- 1 ivo ivo     180 2011-04-11 20:39 panic.go
*/

// 2) exec.Run //
/***************/
// Linux:  OK, but not for ls ?
// cmd := exec.Command("ls", "-l")  // no error, but doesn't show anything ?
// cmd := exec.Command("ls")            // no error, but doesn't show anything ?
        cmd := exec.Command("gedit")  // this opens a gedit-window
        err = cmd.Run()
        if err != nil {
                fmt.Printf("Error %v executing command!", err)
                os.Exit(1)
        }
        fmt.Printf("The command is %v", cmd)
// The command is &{/bin/ls [ls -l] []  <nil> <nil> <nil> 0xf840000210 <nil> true [0xf84000ea
}
// in Windows: uitvoering: Error fork/exec /bin/ls: The system cannot find the path specifie
```

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

Starting an external command or program can be done in several ways (the ways to do that are not platform-specific, but the examples in this program only work on Unix-like machines):

- The first way is using `os.StartProcess`, as at **line 22**. This accepts a folder where to start the command, a slice of strings with the command and its attributes (here: `ls -l`) and a variable, which is a pointer to `os.ProcAttr`. A possible error causes a print of the error and an exit of the program (from **line 23** to **line 26**). If the command is executed successfully, we get back its process-id `pid` at **line 22**, which is printed at **line 27**. In the 2nd example, with `os.StartProcess`, we launch the `ps` command, printing its `pid` at **line 34**.

- The second way is using `exec.Command`. Here, we use this at **line 37** to make a `Command` struct, and then run the command at **line 38**, and print it out at **line 43**. Specifically, here, we launch a `gedit` text-editor window, which you will not be able to see in the course window. The usual error-handling is done from **line 51** to **line 54**.

Restart program if `panic()` occurs:

The following code snippet recovers from the panic and restarts the program. If this fails the error is logged:

```go
package main
import (
"os"
"os/exec"
"log"
)

func ultraCrazyFunction() {
    prog := os.Args[0]
    e := recover()
    if e != nil {
      // In case of panic, try to restart the entire application:
      cmd := exec.Command(prog)

      err := cmd.Run()
      // If that fails, log the error
      if err != nil {
        log.Fatal(err.Error())
      }
    }
  }

func main() {
  defer ultraCrazyFunction()
}
```

That’s it about handling errors. An excellent way to debug a program is to test it. Go provides the support of testing an application before running it. Let’s see how to check a program for errors in the next lesson.