Programming Assignment #2

Introduction to Computer Networks

The Assignment

The assignment will take you through a series of 5 examples and in the process introduce the APIs to allow access to system I/Os, such as reading and writing to a file. Embedded in the examples are also the statements to specify variables, to control program flow, and to debug.

Skim through the assignment. Watch the companion videos. In the meantime, try the examples out yourselves. When you are done, you should be ready to program `PA2.go` that:

(1) prompts the user for the input and output filenames
(2) reads from the input file one line at a time,
(3) prepends the line count to each line, and
(4) writes the line into the output file.

Now follow through the examples below and practice the basic APIs of Go.

1. Hello World

Fashionable learning a programming language, we start by saying hello to the world. I.e., writing a simple program that prints “hello world” on the computer screen. To do so in Go, start a file `hello-world.go` and type up the following code. Here you see `.go` is often the suffix of a Go source file.

```
package main

import "fmt"

func main() {
    fmt.Printf("hello world!\n")
}
```

To execute the code, type the following at the prompt.

```
$ go run hello-world.go
```

You should see the following and back to command prompt.

```
hello, world!
$
```

You may also compile the source code to byte code permanently by:

```
$ go build hello-world.go
```

After compilation, you should see a new file “hello-world” being created in the same direction. Now start the byte code by:

```
$ ./hello-world
```

Code walk-through:

- `package main` is always the 1st line. What it means is that there’s already a package `main` built in, which defines all the fundamental symbols (e.g., data types) and syntax (e.g., assignment and if condition). With `package main`, the compiler includes the most basic, minimum set of APIs. As a result, it is also a package that can execute by itself.
- `import` indicates the additional API sets to include. C/C++ programmers call these additional API sets libraries. In Go, they are called the packages. In this example, `fmt`, short for format, is included. `fmt` package contains APIs that generate output or take input of multiple formats for a variety of system I/Os.
- `fmt.Printf()` calls the `Printf()` API defined in the `fmt` package. This API takes in a string as the argument and, simply, prints the string on the screen. Some of you might find `Printf()` familiar. In C/C++, `printf()` function works exactly the same. Later on, you will see more signs of Go being essentially the extended and more programmer-friendly C/C++.
- `\n` in `fmt.Printf()` enforces a newline after printing the string.

2. Standard I/O

We now explore the fmt package further. Start a file `hello-whoever.go` and type up the following code.

```
package main

import "fmt"
import “os”

func main() {
    fmt.Printf("Who's there?\n")
    text := ""
    fmt.Scanf("%s", &text)

    fmt.Printf("Hello, %s\n", text)
    fmt.Println("Hello,", text)
    fmt.Fprintf(os.Stdout, "Hello, %s\n", text)
}
```

You will be prompted for a name after running the code. Type your name in and hit return. You should see the following.

```
$ go run hello-whoever.go
Who's there? polly
Hello, polly
Hello, polly
Hello, polly
$
```

Code walk-through:

- `text := ""` declares the variable `text` and assigns an empty string to it. In Go, to assign value to an existing variable, just say `=`. `:=` is to declare and assign at the same time. A way to quickly declare and initialize a variable. What’s convenient in Go is that the compiler identifies the data type automatically, looking at the initial value.
- `fmt.Scanf("%s", &text)` scans a string from the standard input (the

keyboard) and assigns it to variable `text`. `%s` means string in `fmt` APIs. Note that the string from standard input is copied to the address of `text`. That's why `&text` is used as the 2nd parameter to `fmt.Scanf()`.

- `fmt.Printf("Hello, %s\n", text)` prints the content inside the double quotes. The `%s` part will be replaced by the value of `text`.
- `fmt.Println("Hello,", text)` prints the same thing. The syntax of `Println()` is different from that of `Printf()`. It outputs all parameters separated by `,`). `Println()` is pronounced print line, as it enforces a new line after execution, i.e., no need of adding `\n` at the end such as `Printf()` does.
- An interesting and important alternative is `fmt.Fprintf(os.Stdout, "Hello, %s\n", text)`. `Fprintf()` means printing to a file in fact. The first parameter of the API asks for the handle/pointer to the file. In Unix, a file is also an I/O, just like the display and keyboard. Therefore, one can think of the display as the standard output file and keyboard the standard input file. Writing to the display is equivalent of writing to a file at `os.Stdout`. `os.Stdout` and `os.Stdin`, the handles of standard input and output are provided conveniently by the `os` package. Note the package is imported upfront in this example. `fmt.Fprintf(os.Stdout,)` is equivalent of `fmt.Printf()`. `fmt.Fscanf(os.Stdin,)` is equivalent of `fmt.Scanf()`.

3. File I/O

Next is to access a (real) file. Start `file-access.go` in the editor and type up the following.

```
package main

import "fmt"
import "os"

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {
    f, err := os.Open("hello-world.go")
    check(err)

    word1, word2 := "", ""
    fmt.Fscanln(f, &word1, &word2)
    fmt.Printf("%s %s\n", word1, word2)

    for i := 2; i <= 5; i++ {
        word1, word2 = "", ""
        fmt.Fscanln(f, &word1, &word2)
        fmt.Println(word1, word2)
    }

    f.Close()
}
```

Below is what you will see running the code. It scans from the `hello-world.go` file (which you've created earlier) one line at a time and records two strings per line.

```
$ go run file-access.go
package main
```

```
import "fmt"

func main()
$
```

Code walk-through:

- Let's zoom in first to `main()`, `os.Open()` opens a file, provided the filename. You might be able to infer that `Open()` is an API defined in the `os` package. What's a bit new is that the API returns two parameters and they are assigned to `f` and `err`. `f` is the variable tracking the handle of the file opened. `err` holds the error message in case of failure.
- `check()` is a function defined in the code itself. See the `func check(e error)` code block right above `func main()`. `func` here is the keyword in Go to begin defining your own function. `(e error)` indicates that the function takes in one parameter `e` of type `error`. If `e` is not `nil`, call `panic()`, which is one of the fundamental APIs defined in the `main` package. What it does is to show the error message and to force-quit the execution.
- `fmt.Fscanln()` scans a line from a file. The first parameter is the file handle. The rest are to hold the strings, separated by a space, in the line. `word1` and `word2` there will hold only two words in a line.
- In the `for` loop, scanning of the file repeats 4 more times. One can see that line 5 in `hello-world.go` consists of 3 strings. `{` at the end will be left out. Although `Fscanln()` reading from a file works just like `Scanln()` reading from `os.Stdin`, it will not be general to textual file scanning, where the number of strings is very likely different from line to line.
- Try changing the exit condition of the `for` loop to `i <= 6` and run the code again. The compiler does not complain but the program fails to print the next line expected. `Fscanln()` is expecting to read from a new line, but sees `{` in the middle of a line instead. You see `Fscanln()` is good for files that are well structured, i.e., the number of strings per line is fixed. It unfortunately does not serve all files in general, particularly the textual files. This leads us to `bufio`, a package to treat I/Os as general byte streams.
- Before we move on, the last line `f.Close()` is simply to close the file.

4. Buffer Input

`bufio` is a must-learn package. The example here shows how one uses `bufio` to read from a file (i.e., a byte stream buffer). We will see how to write to a file right after this. Now, start `bufio-read-file.go` in the editor and type up the following.

```
package main

import "fmt"
import "os"
import "bufio"

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {
    f, err := os.Open("hello-world.go")
    check(err)

    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        fmt.Println(scanner.Text())
    }

    f.Close()
}
```

Below is what you will see running the code. It scans from the `hello-world.go` file line by line, and prints on the screen till the end of file.

```
$ go run bufio-read-file.go
package main

import "fmt"
```

```
func main() {
    fmt.Printf("hello, world!\n")
}
$
```

Code walk-through:

- `scanner := bufio.NewScanner(f)` is where the program differs significantly (vs. the previous example using `Fscanln()`). `NewScanner()` is an API defined in the `bufio` package. It converts a regular I/O (`f` in this case) to a buffer I/O – a `Scanner` object in this case. By doing so, `f` can be accessed via a rich variety of APIs. Note that `bufio` needs to be imported before one can call `bufio.NewScanner()`. The variable `scanner` is initialized and declared at the same time as a `Scanner` buffer I/O.
- `scanner.Scan()` scans from `f`. The default is to scan one line at a time. One can configure it to scan one word or one byte at a time. More details can be found in Go documentation, in particular the `Scanner` section of `bufio`. When the `scanner.Scan()` reaches the end of file, it returns `false` (`true` otherwise).
- The `for` loop there essentially calls `Scan()` repeatedly until the end of the file.
- `scanner.Text()` converts a byte stream to a string so it can be printed using `fmt.Println()`.

5. Buffer Output

The final example shows how one uses `bufio` to write to a file. Start `bufio-write-file.go` in the editor and type up the following.

```
package main

import "fmt"
import "os"
import "bufio"

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {
    f, err := os.Create("PA2-output.txt")
    check(err)
    defer f.Close()

    writer := bufio.NewWriter(f)
    len, _ := writer.WriteString("This is a test!")
    fmt.Println(len)
    writer.Flush()
}
```

Below is what you will see running the code. It opens a file named `PA2-outout.txt` and writes a line `This is a test!` to it. `15` is the length of the line.

```
$ go run bufio-write-file.go
15
$
```

Code walk-through:

- `os.Create()` is the API to open a file non-existing yet.

- `defer f.Close()` defers execution of `f.Close()` to the end of program. This is often used in programs involving I/O access. `Close()` is necessary to shut the I/O opened by `os.Open()` or `os.Create()`. But one often forgets about closing after hours of coding/debugging. Typing the closing line up as soon the I/O is opened and prepending it with `defer` prevents the bug entirely.
- `bufio.NewWriter(f)` converts a regular I/O (`f`) to a buffer I/O (`writer`), much like `bufio.NewScanner()`.
- `writer.WriteString()` writes a string through the `writer` to `f`.
- `writer.WriteString()` returns two parameters, length of the string and error message. `_` is used in place of the error message. This is a way to ignore a certain returned parameter if it is not going to be used later anyway.
- `writer.Flush()` is to enforce the string temporarily stored on the system memory to the file on the disk.

6. PA2.go

Now, start `PA2.go` and make sure it:

(1) prompts the user for the input and output filenames
(2) reads from the input file one line at a time,
(3) prepends the line count to each line, and
(4) writes the line into the output file.

To help you verify your implementation, polly has made the compiled byte code of her `PA2.go` available here: http://homepage.ntu.edu.tw/~pollyhuang/teach/intro-cn-pa/PA2/PA2. Login to the workstation and download the byte code to your account:

```
$ curl homepage.ntu.edu.tw/~pollyhuang/teach/intro-cn-pa/PA2/PA2
> pollys-PA2
```

The `curl` command downloads the byte code and saves it as a file, filename `pollys-PA2`. You'll need to change the permission to allow user to execute the file by the following before trying it out:

```
$ chmod u+x pollys-PA2
$ ./pollys-PA2
```

Cross compare execution result of your `PA2.go` to the outcome of executing `pollys-PA2`. If they work the same, you will be done and safe.

7. More Go Examples

If you find extra time at hand, try fill in the other basic syntax not covered in the examples here. https://gobyexample.com/ provides an extensive set of examples. The ones listed below are very fundamental. You are strongly encouraged to try them out: Hello World, Values, Variables, Constants, For, If/Else, Switch, Arrays, Slices, Functions, Multiple Return Values

8. Go Documentation

For details and other APIs in the packages we've touched upon so far, visit these pages:

`fmt` : https://golang.org/pkg/fmt/
`os` : https://golang.org/pkg/os/
`bufio`: https://golang.org/pkg/bufio/

9. Submit your PA2

`ssh` to the `140.112.42.221` workstation. At the team account's home directory, create a directory `PA2`. Upload your `PA2.go` to directory `PA2`. Test your `PA2.go` again on the workstation just to make sure it's working as expected.