Lab 1: Golang

Spring 2021

Due: 11:59 PM, Feb 3, 2020

Contents

1 Introduction

Welcome to CS 1380! This lab will help you learn some of the basics of Go and will provide you with tools to successfully complete your projects! We'll introduce you to the Go playground, goroutines, channels, select statements, WaitGroup, and mutexes. We'll also teach you how to use some tools to improve your testing.

Note: We highly recommend that you use both this lab, Get Going with Go, and Uber's Go style guide as a reference throughout the semester when completing your projects! **If you haven't used Go before, A Tour of Go might be a good start**.

2 Go Playground

The Go Playground compiles Go programs, runs the program inside a sandbox, then returns the output. It is very useful to learn and practice Go and to test short snippets of code!

The Go Playground can be found here! Open it up and click run!

3 Goroutines, Channels, Select Statements

Goroutines

A goroutine is a function that is running concurrently alongside other code! To achieve this you want to use the keyword `go` before a function. In the example below, the function `studyDistributed` will be called and executed concurrently to the code that follows in the `main` function. Paste the code below into Go playground to see goroutines in action!

```
package main

import (
	"fmt"
	"time"
)

func main() {
	go studyDistributed()
	fmt.Println("Alex: I'm eating")
	time.Sleep(3 * time.Second)
	fmt.Println("Alex: I'm full")
}

func studyDistributed() {
	fmt.Println("Martin: I'm studying!")
	time.Sleep(2 * time.Second)
	fmt.Println("Martin: I'm done studying")
}
```

You can also run goroutines using anonymous functions!

```
func main() {
    go func() {
        fmt.Println("I'm studying!")
     }()
     // other code
}
```

Channels:

Channels allow you to send and receive values. They are used to communicate information between goroutines! You can send values into channels from one goroutine and receive those values in another goroutine. Think of channels like queues.

In order to use a channel, you must first create it! You can create a channel by calling the `make` function.

To make a channel with type `int` we do:

```
ch := make(chan int)
```

To send a value `v` into a channel `ch` (make sure `v` is the same type that we initialized the channel to take - in this example int) we do:

```
ch <- v
```

To receive a value from the channel `ch` and assign that value to `V` we do:

```
V := <-ch
```

Regular channels are *blocking*: a send will block the current goroutine / main thread until at least one *other* goroutine is receiving from it, and a receive will block the current goroutine / main thread until at least one *other* goroutine sends to it. **To prevent deadlocks**, always insure there's a matching send for any receive operation you make, and vice-versa.

Channels can also be *buffered* by providing a *buffer capacity* (like the maximum length of a queue) as the second argument to `make`. Buffered channels, unlike regular channels, allow non-blocking sends and receives in certain cases. Sends to a buffered channel block only when the buffer is full. Receives block only when the buffer is empty.

To make a buffered channel of type int with a buffer size of 100 do:

```
ch := make(chan int, 100)
```

Task 1

In the Go playground, create an **unbuffered** channel, pass in the value 5 to the channel, and then receive from the channel and print that value - it should be 5! Hint: make sure you are using a goroutine. **Screenshot your code with the output** to submit at the end of the lab.

Select Statement:

The select statement allows for a goroutine to wait on multiple communication operations! A select statement consists of several case statements, which are considered simultaneously and then individual cases are executed when ready! Here is an example below of using the `select` statement!

```
func main(){
   // code to creating channels
   // code that adds values to channels

   for i := 0; i < 2; i++ {
      select{
      case op1 := <-ch1:
         fmt.Println(op1)
      case op2 := <-ch2:
         fmt.Println(op2)
      }
   }
}
```

In this example, we use a for loop and select to await both values simultaneously, printing each one as it arrives.

Task 2

In Go playground, create three channels, pass in the values “Hello”, “World”, and “!” to channels 1, 2, and 3 respectively, and then receive from the channels in a select statement and print what each channel outputs! Note: your print lines might not be in the order “Hello World!” because the order of printing depends on the order of goroutines finishing. Again, make sure to **screenshot your code with the output**.

4 WaitGroup and Mutexes

WaitGroup

`WaitGroup` allows you to wait for a set of concurrent operations to complete and is useful when either you don't care about the result of the concurrent operation or you are collecting the results in a different manner. You will want to use Go's sync.WaitGroup with the functions `Add`, `Done`, and `Wait`. `WaitGroup` is like a concurrent-safe counter - you use the add function to indicate you have a goroutine running and therefore incrementing the counter and you use the done function to indicate the goroutine has exited and therefore decementing the counter. Finally, you use the wait function to block until the counter is 0 and therefore indicating all the goroutines have finished. Note: make sure to call add before executing the goroutine!

Here is a simple example of using `WaitGroup` (Try it out in Go Playground):

```
package main

import (
	"fmt"
	"sync"
	"time"
)

func main() {
	var wg sync.WaitGroup
	wg.Add(1)
	go func() {
		time.Sleep(1 * time.Second)
		fmt.Println("goroutine #1 finished")
		wg.Done()
	}()
	wg.Add(1)
	go func() {
		time.Sleep(2 * time.Second)
		fmt.Println("goroutine #2 finished")
		wg.Done()
	}()
	wg.Wait()
	fmt.Println("Both finished!")
```

```
}
```

In this example, "Both Finished!" will print only after both goroutines are complete!

Mutexes

Mutexes provide a concurrent-safe way to express exclusive access to shared resources. As the coder, you are responsible for coordinating access to memory by guarding the access to it with mutexes. This is necessary when you have several goroutines that share and update the same values. For example, if you have two goroutines that both increment or decrement a common value.

A data race occurs when more than one goroutine is accessing a value at once. Data races are a big problem as, each time you execute your code, it executes differently. To prevent data races you can use mutexes! Additionally, Go has a race detector that you can use by adding the `-race` flag in the command line (ie `go run -race myProgram.go`). We recommend using this flag to check for race conditions!

In order to prevent data races, you will want to use Go's sync.Mutex with the functions `Lock` and `Unlock`. By using lock and unlock around code that either manipulates or accesses the common value, you ensure that only one goroutine accesses the value at a time.

To set up a mutex you can either initialize the mutex with:

```
var m sync.Mutex
```

Or you can create a struct with an element for the mutex and elements for each variable that is shared. For example if you have the shared value val, you will want to do:

```
type MutexExample struct {
     val int
     m   sync.Mutex
}
```

Task 3

Download the code here on your own computer or on a department machine. Run it using the `-race` flag to detect a data race! Additionally, the final x value should be 500, but you might notice that it is sometimes 490 or 495 or other values. Using what we learned about WaitGroup

and Mutexes fix this code so that it prints the correct value and does not have a data race! Hint: make sure to pass pointers of your WaitGroup and Mutex! Submit **screenshots of your fixed code and the output of running the fixed code with the `-race` flag**.

5 Testing and Benchmarks

Testing will be a very important part of your projects! Tests can be divided into *functional* tests and *benchmarks*. The first verifies correctness. The second measures performance.

To write tests, you will want to create a testing file or files, which need to be named xxxx_test.go where xxxx is whatever you would like.

The general structure of your testing file should look like this:

```
package example // same package as the code you're testing

import “testing”

func TestExample(t *testing.T){ // must be named TestXxxx
    // testing code!
    // Use t.Errorf to print out error messages!
    // t.Fatalf() will kill the current test with an error message.
    // t.Skip() will skip the current test (wrap in an if condition)
    // t.Log() and t.Logf() will print helpful log messages for you
}

func BenchmarkExample(t *testing.B){ // must be named BenchmarkXxxx
    // benchmark code! A timer automatically measures how long
    // things take, starting from when this function is invoked.
    //
    // This code under perf should get run multiple times - the
    // time taken across runs will be displayed to you.
    // Use the following template to wrap your code
    for i := 0; i < b.N; i++ {
        // write the code you want profiled here
    }
    //
    // Use b.ResetTimer() to reset the timer to zero after
```

```
    // expensive initialization code you don't want measured.
    // b.StopTimer() and b.StartTimer() can also be used judicially.
    //
    // b.Skip() or b.SkipNow() will end the benchmark, as will
    // b.Fatal() and b.Fatalf().
    //
    // b.Log() and b.Logf() will print helpful log information
    //
    // Finally, if your benchmark code requires a tear down step
    // before the function can be run again, use b.Cleanup()
}
```

To run your tests and benchmarks, use the command:

```
go test -bench=.
```

The term "test coverage" describes how much the code is executed when running the tests (you want your test coverage to be high!). Go provides a nice tool to see the test coverage of your tests, we highly recommend using this tool to ensure your tests are extensive.

To check your test coverage use the -cover flag when running tests:

```
go test -cover -bench=.
```

This will run your tests and outputs your test coverage as a percentage.

To visualize your test coverage use the two following commands:

```
go test -coverprofile=coverage.out
```

```
go tool cover -html=coverage.out
```

After you run the second command, a browser window pops up which shows you your code. Each line is color coded where covered is green, uncovered is red, and uninstrumented is grey.

We also recommend using the `-v` flag! This shows you which tests are running currently and when they pass or fail. Additionally, if you just want to run a single test you can do so with `go test -run <name of test>.`

Task 4

Download this code on your own computer or on a department machine.

(a) Write tests for the code in order to get a test coverage of 100%. Run the commands that we described above to make sure the code is correct and that your coverage is at the level needed! **Please submit screenshots of your tests, your coverage percentage, and the visualization of coverage**.

(b) Write one benchmark test for this code and report the results of running the benchmark. **Please submit screenshots of your benchmark test, and your benchmark results.**

6 Turning In

Once you have finished all of the tasks you can turn in your lab by submitting your screenshots as a PDF file to Gradescope. Congrats on finishing your first assignment for CS 1380!