RX-M Cloud Native Consulting

Go

1. CSP and goroutines

Per the CSP documentation:

Concurrent programming is a large topic and there is space only for some Go-specific highlights here.

Concurrent programming in many environments is made difficult by the subtleties required to implement correct access to shared variables. Go encourages a different approach in which shared values are passed around on channels and, in fact, never actively shared by separate threads of execution. Only one goroutine has access to the value at any given time. Data races cannot occur, by design.

To encourage this way of thinking we have reduced it to a slogan:

> Do not communicate by sharing memory; instead, share memory by communicating.

This approach can be taken too far. Reference counts may be best done by putting a mutex around an integer variable, for instance. But as a high-level approach, using channels to control access makes it easier to write clear, correct programs.

One way to think about this model is to consider a typical single-threaded program running on one CPU. It has no need for synchronization primitives. Now run another such instance; it too needs no synchronization. Now let those two communicate; if the communication is the synchronizer, there's still no need for other synchronization. Unix pipelines, for example, fit this model perfectly. Although Go's approach to concurrency originates in Hoare's Communicating Sequential Processes (CSP), it can also be seen as a type-safe generalization of Unix pipes.

2. goroutines

To run code as a goroutine, we prefix the call with `go`. Here is an example where a long running function normally taking 10 seconds, so on the second execution we run it as a goroutine.

```
user@ubuntu:~$ mkdir -p ~user/go/src/lab-csp
user@ubuntu:~$ cd ~user/go/src/lab-csp
user@ubuntu:~/go/src/lab-csp$
```

```
user@ubuntu:~/go/src/lab-csp$ vi lab-csp.go
user@ubuntu:~/go/src/lab-csp$ cat lab-csp.go

package main

import (
        "fmt"
        "time"
)

func main() {
        f := func() {
                fmt.Println("before")
                time.Sleep(10 * time.Second)
                fmt.Println("after")
        }

        f()

        go f()
}
user@ubuntu:~/go/src/lab-csp$
```

Run our sample program.

```
user@ubuntu:~/go/src/lab-csp$ go run lab-csp.go
before
after
user@ubuntu:~/go/src/lab-csp$
```

It seems things executed but we don't have any output associated with the goroutine invocation of the function `f`.

Another issue is there is no way for the goroutine to signal its completion to the caller.

3. Channels

In order to communicate between the coroutine and the caller we leverage a channel.

To create a channel we use `make`. Here we modify our existing program to leverage a channel to return the result from the function being executed in the goroutine.

We modify the second call to `f()` by wrapping it in another anonymous function and executing it as a goroutine. A closure is created with the channel `c`. We also add a print statement to show activity beyond the goroutine and ultimately we wait at the end to read from the channel `c` before printing a final time the return value.

```
user@ubuntu:~/go/src/lab-csp$ vi lab-cspb.go
user@ubuntu:~/go/src/lab-csp$ cat lab-cspb.go

package main

import (
        "fmt"
        "time"
)

func main() {
        f := func() int {
                fmt.Println("before")
                time.Sleep(10 * time.Second)
                fmt.Println("after")
                return 0
        }

        f()

        c := make(chan int)

        go func() {
                c <- f()
        }()

        fmt.Println("do some other work before we block on <-")

        returnVal := <-c

        fmt.Println("return value of", returnVal)
}
user@ubuntu:~/go/src/lab-csp$
```

Run the program.

```
user@ubuntu:~/go/src/lab-csp$ go run lab-csp.go
before
after
do some other work before we block on <-
before
after
return value of 0
user@ubuntu:~/go/src/lab-csp$
```

We see in the previous example some basic usage of the channel, the key is to notice how things are being synchronized without the use of locks. It may not be clear if you have not used locks in your own code before.

Lets look at another example where we try simulate using all the CPUs available to us. Here is one way to see how many CPUs you have (under Linux):

```
user@ubuntu:~/go/src/lab-csp$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             2
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 94
Model name:            Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz
Stepping:              3
```

```
CPU MHz:               2711.726
BogoMIPS:              5423.45
Hypervisor vendor:     VMware
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0,1
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts mmx fxsr sse sse2 ss
syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts nopl xtopology tsc_reliable nonstop_tsc aperfmperf eagerfpu pni
pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm
3dnowprefetch epb fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 invpcid rtm rdseed adx smap xsaveopt dtherm ida arat pln pts hwp
hwp_notify hwp_act_window hwp_epp
user@ubuntu:~/go/src/lab-csp$
```

The key field of course is the **CPU(s):** count.

```
user@ubuntu:~/go/src/lab-csp$ lscpu | grep ^CPU\(s\): | awk '{print $2}'
2
user@ubuntu:~/go/src/lab-csp$
```

Now lets create a program that uses the number of CPUs to run some code, and then wait for all of them to complete in the driver (main) program.

```
user@ubuntu:~/go/src/lab-csp$ vi lab-cspc.go
user@ubuntu:~/go/src/lab-csp$ cat lab-cspc.go

package main

import (
        "fmt"
        "runtime"
        "time"
)

var numCPU = runtime.NumCPU()

// Apply the operation to v[i], v[i+1] ... up to v[n-1].
func DoSome(cpu int, c chan int) {
        fmt.Println("Doing something on ", cpu)
        time.Sleep(2 * time.Second)
        c <- 1 // signal that this piece is done
}

func DoAll() {
        c := make(chan int, numCPU) // Buffering optional but sensible.

        for i := 1; i <= numCPU; i++ {
                go DoSome(i, c)
        }
        // Drain the channel.
        for i := 0; i < numCPU; i++ {
                <-c // wait for one task to complete
        }
        // All done.
        fmt.Println("I only run once all the farmed out CPU work is done")
}

func main() {
        DoAll()
}
user@ubuntu:~/go/src/lab-csp$
```

In the function `DoAll()`, we create a channel (c), then we create a goroutine that executes `DoSome()` and pass the channel to it. These means we can pass channels as first class citizens. In each instance of the goroutine, we do some work and send a result back to the channel. Back in the `DoAll()` function, we wait for the same number of results to return via the channel. In our case, we don't what the result is, we just want to make sure we get the same count. Finally, there is a Println that is represents further computation that was blocked until all the *numCPU* goroutines finished.

With that understanding, take your program for a run.

```
user@ubuntu:~/go/src/lab-csp$ go run lab-cspc.go
Doing something on  2
Doing something on  1
I only run once all the farmed out CPU work is done
user@ubuntu:~/go/src/lab-csp$
```

There is more to goroutines and channels, but this hopefully gives a feel to doing synchronization without locking.

4. Challenge

Create a simple ping pong game using goroutines and channels. Two players (goroutines) send the ball back and forth via a channel (table). Create a function play() that takes two ints - the chance out of 100 that each player will hit the ball successfully on a given round (e.g. a highly skilled player might have 90). These probabilities should be used to determine if a given player hits the ball in the function hits(). Set up a loop that runs until a player fails to hit the ball. Print out a count each iteration and a message declaring who won and who lost. Use fmt.Scanln() so that the user must press enter between turns and to start the game. Here is some starter code:

```
package main

import "fmt"
import "math/rand"
import "time"

func hits(probability int) bool {
        // given a probability, use randomness to determine if the player hits the ball
        // go here: https://gobyexample.com/random-numbers to learn more about the random package
}

func play(player0prob, player1prob int) {
        // make a channel
        fmt.Println("Press enter to begin")
        var input string
        fmt.Scanln(&input)
        playing := true
        // initialize other useful variables
        for playing {
                // print out a rally count
                // check if the player hit the ball
                // if they did, send a message through the channel and print it out
                // make the user press enter before continuing to player1's turn
        }
        // print out who won

}
func main() {
        play(94, 66)
        // a player that is very good and a player that is okay
}
```

Congratulations you have completed the lab!!