Go Routines

Concurrency: The original magic of Go

Admin

- Please read chapter in the Learning Go book for this section.
- Podcast?

Concurrency

- **Someone who has taken Operating Systems**
  - Or done it in real life
  - Remind us what Concurrency is

Concurrency

- **Someone who has taken Operating Systems**
  - Or done it in real life
  - Remind us what Concurrency (in programming) is
  - Break a program/process up in to independent communicating (sub)processes.
    - *May or may not end up being parallel*
    - Depends on hardware and algorithm

Processes and Threads

- **Same group**
  - What are processes and threads?

Processes and Threads

- **Same group**
  - What are processes and threads?
  - Process:
    - the code and data for a running program,
    - in most programs this includes a virtual memory that pretends it is contiguous and starts from memory address 0x0
  - Thread:
    - A ‘light weight’ process
    - Has its own stack – but shares heap memory

Concurrency

- **Don’t over use concurrency in real life**
  - Lots of people did in the early days of go
  - Just like not everyone is Google https://programmerfriend.com/not-google/
  - And not everything should be microservices https://ciarancodes.com/not-everything-needs-to-be-microservices-seriously-stop-it/
  - We don’t need everything to be concurrent processes
    - Let’s talk a bit about what is a good problem for concurrency and what might not be

Costs

- **Again for the folks who have finished Operating Systems**
  - What is cost of a function call?
  - What is the cost of a ‘context switch’ (in the operating systems sense)
  -

Costs

- **Again for the folks who have finished Operating Systems**
  - What is cost of a function call?
    - Push stack frame for function
    - Move values from registers to stack and vice versa
  - What is the cost of a ‘context switch’ (in the operating systems sense)
    - All of the above – and likely adjust the cache to reflect the working memory of the new program
    - Work with new stack for new program

Concurrency Costs

- **Concurrency Costs**
  - Depending on thread model
  - Context switching
  - Some threads being blocked waiting on output from others
  - Cost of communication mechanisms between threads
- **Some of our academic examples will ignore this to keep the examples small**
  - But keep in mind for ‘real life’
  - Reference Steve and his tendency to ‘try it’

Kernel Threads vs Green Threads

- **What is the difference between Kernel Threads and Green threads?**

Kernel Threads vs Green Threads

- **What is the difference between Kernel Threads and Green threads?**
  - Kernel threads are managed by the OS
    - Includes context swap (swapping values out of registers.
  - Green threads are done in user space
    - Emulate OS-threads
    - Can’t do parallelism properly
      - Useful for async tasks (Waiting for I/O etc)
    - But no context switch either

Go Routines

- **Go and puns: Go's Threads are called 'Go Routines'**
  - Co-routine pun
- **Go process starts**
  - Go runtime spawns X Kernel threads
  - Then spawns green threads within them
    - Initially with very small stack so that there is very little overhead till you need them
  - Very efficient – 1000 go routines is possible without causing a modern computer to *thrash*
    - Thrash? Lucky Volunteer?

Go routines

- **Turn any function into a go routine by putting the go keyword in front of the call**
  - Pass parameters as usual
  - No return values are possible
    - And go routines don’t print to standard output without extra work

```
func main() {
	for i := 1; i <= 5; i++ {
		fmt.Println("i =", 100/i)
	}
}
```

    - If we make fmt.Println a go routine nothing printed – let’s try

Concurrency Communication Problems

- When we write concurrently, what is the problem

```
package main

import (
	"fmt"
	"os"
	"time"
)

func main() {
	for i := 1; i <= 5; i++ {
		go badWrite("output.txt", fmt.Sprintf("the number for this write is %d", 100/i))
		fmt.Println(fmt.Sprintf("the number for this write is %d", 100/i))
	}
	time.Sleep(10 * time.Second) //sleep to make sure go routines run before program ends.
}

func badWrite(fileName string, text string) {

	file, err := os.OpenFile(fileName, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
	if err != nil {
		fmt.Println("Error opening file:", err)
	}
	_, err = file.WriteString(fmt.Sprintf("%s\n", text)) //first param is number of bytes written
	if err != nil {
		fmt.Println("Error writing to file:", err)
	}
	file.Close()

}
```

- Let’s put this in goland, share on github and try it out.

Traditional solution

- **What are the traditional solutions to this ‘synchronization’ problem?**
  - Lucky Volnunteer™?

Traditional solution

- **What are the traditional solutions to this ‘synchronization’ problem?**
  - Mutexes
  - Semaphores
  - monitors/intrinsic locks (used in java)
- **Again post OS people: What happens to a thread that wants to write when the place it need to write to is already busy**
  - With proper synchronization like above?

Traditional solution

- **What are the traditional solutions to this ‘synchronization’ problem?**
  - Mutexes
  - Semaphores
  - monitors/intrinsic locks (used in java)
- **Again post OS people: What happens to a thread that wants to write when the place it need to write to is already busy**
  - With proper synchronization like above?
    - Sender/writer thread goes to sleep.

Danger of poorly synchronized code

- **What does this mean?**

Channels

- **The Go solution to inter thread (inter-go routine) communication is a channel**
  - Channels wrap synchronization and data sharing.
    - Possibly buffered

Channels

- **Make a channel – using make of course**

```
MyChannelVar := make(chan string)
```

  - Use make function, first argument is go keyword chan
  - Second argument/param is the type of thing you can send on channel
    - Usually common type like int, string etc.
    - But could be arbitrary type

Reading and writing

- **Use the <- operator to read or write from a channel**
  - (greater than symbol immediately followed by minus)
  - To read from a channel, put this operator to the left of the channel
  - Eg:

```
text := <-MyChannelVar
```

  - To write to the channel put this operator to the right of the channel
  - Eg:

```
MyChannelVar <- "the text here"
```

    - Read/write on unbuffered channel twice before inverse puts thread to sleep
    - If all threads are sleeping blocked then go panics on deadlock.

Directional channels

- **Make creates a bi-directional channel**
  - But often one function only reads from channel, and one only writes.
  - So make that clear by placing the <- operator next to the chan type when declaring parameters

```
func betterWrite(fileName string, receiver <-chan string) {
```

  - The above function has a channel parameter that can only be read from
    - Compiler enforced.

Let’s try

- **Let’s replace our badWrite with a ‘betterWrite’ that will use channels to make sure the data ends up in the file in the right order.**
  - We’ll
    - take our existing main function
      - create/make a channel
      - Call the new function below as a go routine, passing the channel and a new filename
      - Write the data to the channel after calling the go routine in our loop
    - Make a new function that has the signature of the betterWrite from previous slide
      - Do mostly the same as the badWrite, but get the text to write to the file from the channel

For-Range and channels

- **Often, rather than many goroutines listening to a channel**
  - Like we did in our last example
- **Will have one go routine that keeps listening to a channel as long as it is open.**

```
for text:= range receiver{
```

- **Will read from the receiver channel until that channel is closed**

Closing channels

- **If we read till closed – need close**

Closing channels

- **If we read till closed – need close**

close(MyChannelVar)

  - Most channels are written to from one thread and read from another
  - The writer should close the channel
  - Once channel is closed
    - Read will give zero value and be fine
    - Use ok version to see if value is sent zero val or created

text, ok := <-receiver

    - Write to closed channel will *panic*. (see chart on page 293)

Select and channels

- **Select for channels**
  - looks a bit like switch
  - But no {} needed for cases
- **Use to read from one of several channels**
  - Which ever has values available
  - If multiple cases possible, go randomly selects.

```
func spawn() {
 result := 0
 channel1 := make(chan int)
 channel2 := make(chan int)
 channel3 := make(chan int)
 go calculator(channel1)
 go calculator(channel2)
 go calculator(channel3)
 select {
 case partialResult := <-channel1:
          result += partialResult
          fmt.Println("Partial result from channel 1:",
partialResult)
 case partialresult2 := <-channel2:
          result += partialresult2
 case partial3 := <-channel3:
          result += partial3
          fmt.Println("Partial result from channel 3:", partial3)
 }
}
```

- **s**

Select and Channels II

- **Often want to repeat checking the select**
  - But make sure you have a ***break*** or *waitgroup* (deferred)
  - Which I don't have here

```
func spawn() {
        result := 0
        <snip>
        for {
                select {
                case partialResult := <-channel1:
                        result += partialResult
                        fmt.Println("Partial result from channel 1:",
partialResult)
                case partialresult2 := <-channel2:
                        result += partialresult2
                case partial3 := <-channel3:
                        result += partial3
                        fmt.Println("Partial result from channel 3:", partial3)
                }
        }
}
```

WaitGroups

- **When one go routine needs to wait for multiple others to finish**
  - Could close channels in each go routine
  - But if we want to keep them open can use waitgroups.
  - sync.WaitGroup
    - Important functions
      - Add
      - Done
      - Wait

WaitGroups II

- **Add**
  - Increments the wait counter
- **Done**
  - Decrements the wait counter
- **Wait**
  - Blocks the thread till the wait counter is 0 again
- **Example from**
  - https://oneuptime.com/blog/post/2026-01-23-go-waitgroup/view

```go
func main() {
	var wg sync.WaitGroup

	for i := 1; i <= 3; i++ {
		wg.Add(1)  // Increment counter before goroutine

		go func(id int) {
			defer wg.Done()  // Decrement when done

			fmt.Printf("Worker %d starting\n", id)
			time.Sleep(time.Second)
			fmt.Printf("Worker %d done\n", id)
		}(i)
	}

	wg.Wait()  // Block until counter is zero
	fmt.Println("All workers completed")
}
```

Mutex

- **Go does provide a traditional mutex**
  - As per chapter 12 in book
  - Usually use channels
  - Occasionally for many processes checking shared value but not consuming it mutexes the right thing

Reading

- **Chapter 12 in your book of course**
- **Some other links I used**
  - https://go101.org/article/channel.html
  - https://go.dev/tour/concurrency/1
  - https://gobyexample.com/channels
  - The latter two are slides, you want to follow links to the next few.

Reminder: read chapter 12