“Concurrency in Go” Notes

by Arnav Kumar

Web: https://arnavcs.github.io

Concurrency in Go is a publication by O’Reilly Media Inc. written by Katherine Cox-Buday. This is a collection of notes that I make about the text as I read it, and is not a summary or recreation of the text, but rather a reference for anyone who has already read the text. As such, please read the text to gain a better understading of the contents.

---

1 Concurrency Ideas

Color Scheme Key
Design Pattern
Definition
Note

Amdahl’s Law

Amdahl’s Law models the improved performance of a fixed task when the resources are improved. In parallel computing, it is used to predict the speedup of using multiple processors. The relation is given as follows:

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

S_{latency}	the theoretical speedup of the whole program
s	the speedup of the part of the task from improved resources
p	the proportion of the execution time that benefits from the improved resources

Race Conditions

A race condition is when two or more operations must execute in the correct order, but the program leaves the order of execution unspecified.

Data Race

A data race is a race condition in which two concurrent operations attempt to read the same data at an unspecified time (namely one that could potentially conflict). In the following example, the program is not given a specified evaluation order, so the code that follows may execute before, during, or even after the goroutine. As such, the output is indeterminate.

```
var data int
go func() { data++ }()
fmt.Printf("%v\n", data)
```

Atomicity

An atomic operation is indivisible or uninteruptable in the context in which it is operating. For example, the statement `i++` consists of 3 atomic operations: retriving, incrementing, and storing the value of `i`.

Critical Selection	A critical selection is a section of code that requires exclusive access to a shared resource. In the following code, the `fmt.Printf()` and the goroutine are both critical selections.
	`var data int`
	`go func() { data++ }()`
	`fmt.Printf("%v\n", data)`

Memory Access Synchronization	To solve the problem of multiple critical selections, only enable one critical selection to access the same shared resource at a time. This can be achieved, for example, with a mutex.
Deadlock	A deadlock is a state in which all concurrent processes are waiting on each other. A deadlock can be identified by the Coffman Conditions.
Coffman Conditions	There are 4 Coffman Conditions that detect, prevent, and correct deadlocks. The conditions are as follows: **Mutual Exclusion**: A concurrent process must hold exclusive rights to a resource at any one time. **Wait For Condition**: A concurrent process must hold a resource and be waiting for another resource. **No Preemption**: A resource held by a concurrent process can only be released by that process. **Circular Wait**: A process must be waiting on a chain of processes which is circular (meaning that the process is directly or indirectly waiting on itself to give a result).
Livelock	A livelock is when the current concurrent processes are performing operations, but these operations do not terminate or move the program closer to termination.
Starvation	Starvation is a superset of a livelock or deadlock where, more generally, a concurrent process does not recieve access to the resources it needs. A common example is having a "greedy worker" hold on to access to the resource, while a "polite worker" does not, and thus has less access to the resource: it is starved.
"Finding a Balance"	What should the range of a memory lock be? Should it be broad and cover multiple critical selections, or should each critical selection get its own lock? It is important to strike a balance in answering this question because memory access synchronization is expensive, but you also want to avoid writing greedy processes to mitigate starvation.
OS Threads	OS threads are a primitve at the OS context that can be used to run pocesses concurrently. The operating system is responsible for creating and managing the threads. The threads all have access to a shared resource space.
Green Threads	Green threads are threads that are managed by a program's runtime.
Coroutines	Concurrent subroutines that are non-preemptive (meaning that they can't be interrupted) are called coroutines. They feature multiple points to suspend or reenter computation.

`M:N` Scheduler	A `M:N` scheduler is the mechanism that Golang uses to host goroutines and it consists of mapping `M` green threads onto `N` OS threads.
Fork-Join Model	The model that Golang follows for concurrency, a fork-join model is one in which a child branch can fork off from parent to be run concurrently. After the termination of the child branch, it is joined back to the parent branch at a join point.
Thread Pools	Thread pools are a software design pattern that maintains a collection of threads to map incoming tasks to threads for concurrent execution.
Concurrency vs. Parallelism	Parallelism is a property of a machine to be able to run two tasks simulatiously in the considered context. On the other hand, concurrency refers to when two processes have a lifespan that overlaps. In this sense, you could have a concurrent program running on a single thread where multiple threads are simluated. It is also possible that the concurrent processes run in parallel. Concurrency is a property of the code, and parallelism is a property of the execution of the code.
Process	A process is a portion of code that requires input to run and produces an output that is consumed by another process. The input and output of a process is called communication between processes.
Communicating Sequential Processes (CSP)	CSP is the name of a paper, programming language, and the idea of a descibing programs as processes which are sequential and communicate. Used in the paper describing CSP, the CSP language supported the use of `!` and `?` to send input into and read output from a process respectively. In addition, it supported guarded commands. This is the style of concurrency programming that Golang's channels are based on.

Guarded Command	When a statement should not be executed if another statement was false or a command exited, it is a guarded command. The CSP example below denotes a process `a`, from which a character `c` is continually read (while there is something to be read), and then inputted into the process `b`.
	
	`*[c:character;`
	`a?c -> b!c]`

Process Calculus	Process calculus is a mathematical way to model concurrent systems and analyze their properties.
Should I use CSP style or OS threads?	The CSP style has certain advantages that it comes with, and more generally, the Golang developing team suggest to use the CSP style over primitves like `sync.Mutex`, but there are certain guidelines outlined that help determine when you should use channels or OS thread primitives. Follow the first applicable statement. 1. If your code is performance critical, use primitives 2. If you are trying to transfer ownership of data, use channels 3. If you are trying to guard the internal structure of a struct, use primitives 4. If you are coordinating multiple pieces of logic, use channels 5. Use primitives

Mutex	Mutex stands for "mutual exclusion" and enables a way to express exclusive access to a shared resource. A mutex is often used for critical selections.
Object Pool	This pattern is a way to create a fixed number of objects for use, and is especially useful for objects that are computationally expensive or objects that will take a lot of memory.

2 Golang Features

Color Scheme Key
Concept / Syntax
Type
Function
Keyword

Variable Declaration and Initiation

`func`

This keyword can be used to create named functions, closures, or anonymous functions. A named function example is show below.

```
func helloWorld(numTimes int) {
    for ; numTimes > 0; numTimes-- {
        fmt.Printf("Hello World!\n")
    }
}
```

An anonymous version of the same function is also shown below.

```
var f := func(numTimes int) {
    for ; numTimes > 0; numTimes-- {
        fmt.Printf("Hello World!\n")
    }
}
```

`go`

Creates a *goroutine* that runs the function, method, or closure concurrently by multiplexing onto OS threads. Each goroutine is a special class of coroutine where you do not have to manually describe the suspension and resuming of the routine. At runtime, Golang automatically suspends goroutines when they are blocked and resumes them when they are unblocked. Goroutines use the fork-join model for concurrency and during runtime, a `M:N` scheduler is used. See the following example using goroutines modified from the textbook that uses closures to print `"go"`, `"rust"`, and `"c"` concurrently in an unspecified order.

```
var wg sync.WaitGroup
for _, lang := range []string{"go", "rust", "c"} {
    wg.Add(1)
    go func(l string) {
        defer wg.Done()
        fmt.Println(l)
    }(lang)
}
wg.Wait()
```

<table>
<tr><td><code>defer</code></td><td>Defers the execution of the statement to the end of the function. In the following example, the mutex isn't unlocked until the after the value of <code>data</code> increments.<pre>
var data int = 0
var mu sync.Mutex
func inc() {
    mu.Lock()
    defer mu.Unlock()
    data++
}()
</pre></td></tr>
<tr><td><code>struct</code></td><td></td></tr>
<tr><td><code>interface</code></td><td></td></tr>
<tr><td><code>sync.WaitGroup</code></td><td>A waitgroup stops the execution of certain code past a point untill all processes being waited on are completed. It supports methods such as <code>.Add(int)</code>, <code>.Wait()</code>, and <code>.Done()</code>. You should use a <code>WaitGroup</code> when you do not care about the results of the concurrent operations or have another mean to collect the results. If that is not the case, use a <code>select</code> statement with channels. See the exmaple below which prints the number from <code>0</code> to <code>n-1</code> in some unspecified order.<pre>
const n = 3
var wg sync.WaitGroup
wg.Add(n)
for i := 0; i < n; i++ {
    go func(wg *sync.WaitGroup, i int) {
        defer wg.Done()
        fmt.Println(i)
    }(&wg, i)
}
wg.Wait()
</pre></td></tr>
<tr><td><code>sync.Mutex</code></td><td>A mutex type that supports the <code>.Lock()</code>, <code>.TryLock()</code>, and <code>.Unlock()</code> methods. These methods declare exclusive access to the shared resource that the mutex represents. By convention, a mutex unlock statement is in a <code>defer</code> statement to avoid <code>panic</code>ing meaning that the mutex is not unlocked.</td></tr>
<tr><td><code>sync.RWMutex</code></td><td>This form of mutex requires the specification of the type of access desired. An arbitrary number of readers are allowed to read the same resource granted that there are no writers. In exchange for the greater control over the memory (and potentially less opportunity for starvation), it gives lower performance than <code>sync.Mutex</code> for a small number of readers. When the number of readers is high, though, it's performance is noticible. The supported methods are those from <code>sync.Mutex</code>, and the additional <code>.RLock()</code>, <code>.TryRLock()</code>, <code>.RUnlock()</code>, and <code>.RLocker()</code>.</td></tr>
</table>

`sync.Cond`	A `sync.Cond` is a "rendevous point" for goroutines waiting for an event (an signal between two or more goroutines that carries no information). The instantiation of a `Cond` is done with `sync.NewCond` which takes a `sync.Locker` interface (accessible with `.L`). Additionally, the methods `.Broadcast()`, `.Signal()`, and `.Wait()` are avaliable to be used. Consider the following function from the textbook that "subscribes" a function to a `Cond`, running the function once when the `Cond` first broadcasts.
	<pre>subscribe := func(c *sync.Cond, f func()) {
	    var goroutineRunning sync.Waitgroup
	    goroutineRunning.Add(1)
	    go func() {
	        goroutineRunning.Done()
	        c.L.Lock()
	        defer c.L.Unlock()
	        c.Wait()
	        f()
	    }()
	    goroutineRunning.Wait()
	}</pre>

`sync.Once`	A variable, `once`, of type `sync.Once` will support the `.Do(func())` method which will only execute the passed function once regardless of what if a different function is passed.
`sync.Pool`	A pool object is an implemenation of an object pool. It can be instantiated by specifying the `New` field which is a thread safe member variable function that creates a new object in the pool. the `Pool` also supports the methods `.Get()`, and `.Put(object)`. Make no assumptions about the state of the instance you get back from `.Get()`, but objects in the `Pool` should be roughly uniform in makeup.
Channels	
`make()`	
`new()`	
Type Assertions	