Technical science

UDC 004.43

Malyshchenko Nataliia
Student of the
Kharkiv National University of Radio Electronics

Chernonos Mariia
Student of the
Kharkiv National University of Radio Electronics

Oliinyk Olena
Senior Lecturer of the Department of Software Engineering
Kharkiv National University of Radio Electronics

GOROUTINES IN THE CONCURRENT PROGRAMMING

Summary. *The concepts of concurrency and goroutines were reviewed. Sequential and concurrent program execution were compared.*

Key words: *concurrency, Go, goroutine, parallelism, programming, sequential.*

Every day we complete an uncountable number of tasks. Without any hesitation, we can do many of them simultaneously. In modern world we do not have enough time to perform tasks one by one. To be extremely progressive and productive, we must be concurrent and parallel. So do programs.

To fully understand the problem, we must strictly distinguish concepts of concurrency and parallelism. Concurrency is the execution of tasks in a certain time (for example, there are 4 processes and all of them in total are executed within 90

minutes in turn). An important detail is that tasks do not have to be performed at the same time, so they can be divided into smaller and interleaved ones. Parallelism is the execution of tasks at the same time (for example, there are 4 tasks, each one takes 90 minutes). The name itself implies that they run in parallel. We may say that parallelism is a subclass of concurrency: before running multiple concurrent tasks, you need to organize them properly first.

The concept of goroutines is a must-have in achieving concurrency and parallelism in Go. A goroutine is an efficient and lightweight multithreading mechanism, a function that runs concurrently with other goroutines in the same address space. It can be applied in the tasks:

- where a programmer needs asynchrony. For example, when we work with a network, a disk, a database, a mutex-protected resource, etc;
- if the execution time of the function is long enough and you can get a gain by loading other cores.

It is extremely easy to define a goroutine in a program: a programmer just has to put the «go» operator before the function call.

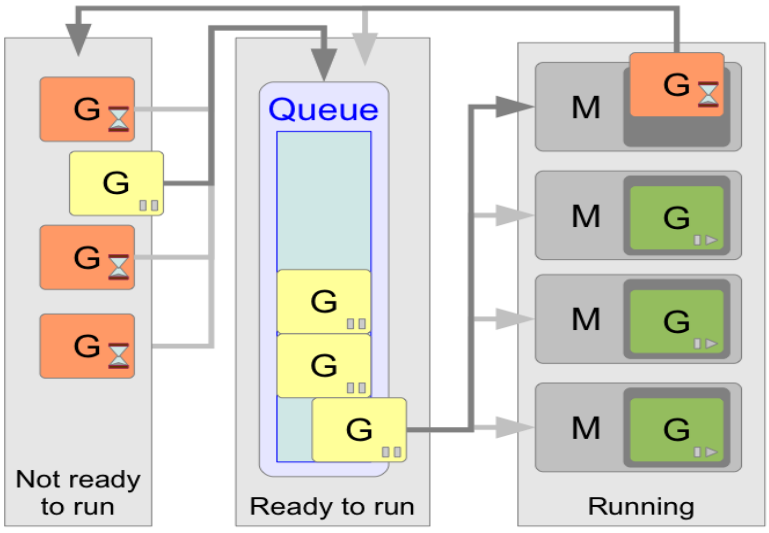

Fig. 1. Figure and description of the Go Scheduler

There is also the Go Scheduler, which distribute ready-to-run goroutines to free machines as can be seen from Figure 1 [1]. In such way we execute a concurrent program.

Ready-to-run goroutines are executed in turn order, i.e. FIFO (First In, First Out). The execution of the goroutine is interrupted only when it can no longer be executed: that is, due to a system call or the use of synchronizing objects (operations on pipes, mutexes, etc.). There are no time slots for the goroutine to work, after which it gets back into the queue. For the scheduler to do this, a programmer needs to call runtime.Gosched() themselves [2].

Because goroutines are closely connected to runtime, there is a Go package named runtime, that has very important functions to work with goroutines such as Runtime.Gosched(), as was described above, works with the scheduler and allows it to execute previous goroutines and then go to next ones. The runtime.Goexit() stops the goroutine that is currently executed and lets the code statements after the defer keyword run in the usual order. The runtime.NumGoRoutine() returns the total number of goroutines that are presented in the program. And then there are two functions that are related to the CPU, it is runtime.NumCPU(), that returns number of the CPU's cores and the runtime.GOMAXPROCS(n), that sets the number n as the number of CPU's cores that you want to use when executing your program.

As was mentioned before, the goroutines are the powerful mechanism embedded in Go, moreover, the language design implies its usage for concurrent implementations of programs. There are additional tools for creating concurrency in the Go such as the channels and select statements, but we would describe them later.

So, now we need to talk why it is better to use goroutines and not sticking to the sequential execution when designing applications. Not every task needs

concurrency, but most of them are performing better when concurrency is involved. To be more precise, every time you need to manage a few parts of the code that can be executed independently, i.e. the result of previously performed part won't affect the results of the next part's execution, the concurrency will be better solution compared to serial program composing. And even in simple tasks, such as computing the sum of two integers, with goroutines the execution time is significantly reduced.

As an illustrative example, we've written a code (created by the authors, based on [4]) that consists of two anonymous functions that are calculating the sum of two integers. In the serial version of program (see Figure 2) anonymous function #1 executes first, then goes anonymous function #2 and the last stage of executing is the outputting of elapsed time.

```
package main

import (
    "fmt"
    "time"
)

func main() {
        start := time.Now()
        func() {
                var i = 3 + 2
        fmt.Println(i)
        }()

        func() {
                var i = 3 + 4
        fmt.Println(i)
        }()

execTime := time.Since(start)
fmt.Println("Total Time For Execution: " + execTime.String())
time.Sleep(time.Second)
}
```

Fig. 2. Sequential implementation of the program

The elapsed time for this fragment of code is $28.011 * 10^{-6}$ seconds. And now we will run the concurrent code (see Figure 3), where we created a goroutine for both anonymous functions #1 and #2.

```
package main

import (
    "fmt"
    "time"
)

func main() {
        start := time.Now()
        go func() {
                var i = 3 + 2
        fmt.Println(i)
        }()

        go func() {
                var i = 3 + 4
        fmt.Println(i)
        }()

execTime := time.Since(start)
fmt.Println("Total Time For Execution: " + execTime.String())
time.Sleep(time.Second)
}
```

Fig. 3. Concurrent implementation of program

When executing this program, we got impressive results and a bit different output statements order. When working concurrently, the program firstly displayed result of the anonymous function #2, then elapsed time and lastly, the result of an anonymous function #1. Also, the time of execution was $7.657 * 10^{-6}$ seconds which is almost four times faster than the serial code.

Although the output order was different from the serial when executing the concurrent code, this is not the predefined order and it may vary from run to run, because Go runtime defines and distributes subtasks execution automatically.

Concurrency and parallelism have mostly synonymous meanings, but the difference is that in parallelism the task is split up to smaller subtasks and processed on multiple processor units at the same time. As opposed to the concurrent, parallel applications are executing only one task at a time but dividing it and completing on the multiple threads. Meanwhile the parallelism waits for the task to be finished, concurrency is more about multitasking and the overlapping subtasks lifetimes [3].

One can think that goroutines are the same as the usual operating system threads. But that's not correct. Working with goroutines rather than with threads of your operating system directly has significant advantages. First of all, goroutines are using less memory resources than usual operating system threads. A goroutine has only the 2KB size of stack when initiated, meanwhile OS thread needs approximately 1MB. Furthermore, when more memory is required, the goroutine stack is just copied to another place in memory with doubled size. Secondly, because of their size of stack, the goroutines are much cheaper to run than the OS threads [2]. For example, we have the computer with two processing units and each of them supports eight threads. So, in C++ we can use only sixteen threads (most of the time this number is even less than maximum number of threads due to background usage of another programs that are also need memory to work correctly). But with Go, we can run a hundred goroutines just on one thread, keeping the memory usage rate much lower than with threads. Another one advantage is that Go runtime controls all scheduling and redirecting part. Therefore, if you have a goroutine that has to wait for some input, the new thread will be created and remaining routines will be moved there, so the program execution time will be the most optimal. Moreover, it's making concurrency easier for programmer because he doesn't need to handle all the edge cases manually. And lastly, the main principle of Go concurrency is "Do not communicate by sharing memory; instead, share memory by communicating" [4]. So, there were invented Go channels, which are the perfect way of safe data sharing. And provided way of goroutines communication saves us from race conditions and deadlocks, which are very common problems when working with operating system threads.

So, as we mentioned before, there are also additional tools in Go for concurrent programs design. And their abilities are allowing developers to create

powerful and effective concurrent applications. These tools are the Go channels and select statements.

Firstly, a little bit more information about channels. Channels are the built-in way of the communication between different goroutines. Usually channels are working in both directions, allowing goroutines not only send data, but receive it. By default, goroutines are instantiated and existing in the same address space, so when we're creating two goroutines and address their execution to the shared variable, the output of the program can be unpredictable and incorrect. This happens because we're "communicate by sharing memory" [4], rather than "share memory by communicating" [4], which is contradicts the main Go concurrency concept and makes the code unsafe for execution. Meanwhile, the Go channels are the pipes for effective access to distributed memory. And what about code safety? When receiving or sending data, channel is automatically blocking, so the goroutine won't continue its execution before it receives data from channel. And if the channel is empty, goroutine will be waiting for another goroutines to send needed data to channel and only after receiving will be executed [5]. This is also working vice versa – the goroutine won't go to the next step if the data, that was sent to the channel wasn't received.

To create a channel is as easy as create a goroutine. You just need to use the make() function, as with maps, and in the curly brackets write keyword chan and the type of data, that you want to fetch. So, the definition of the channel mychan with integer data will look like the first statement (see Figure 4):

```
mychan := make(chan int)
var := <-mychan
mychan <- var
```

Fig. 4. Definition of the Go channel

Also the specifics of working with channels is the arrow operator (<–) to organize communication between goroutines. The second statement (see Figure 4) is creating a variable var with the data from channel mychan, while the third statement is sending value of var to the mychan channel.

But it is not all of channel's possibilities. What if your program will get more data, than it can process? The program will stop send this data into the channels because of absence of recipients. Therefore, to deal with it, you may use the data buffer. If there are no handlers available, there is a need to temporarily store the data in queue [7]. Channels have built-in buffering support. Buffered channels can be created using the make() function, the capacity of the pipe is passed as the second argument to the function. If the channel is empty, then the receiver waits until at least one element appears in the channel [6].

Buffering on channels is beneficial for performance reasons. Channels provide a means of passing events between one process and another in case if a program is designed using an event flow or data flow approach.

There are cases, when a program needs its components to remain in sync with the stop step. In this case, unbuffered channels are required.

Otherwise, it is usually useful to add buffering to the channels as it works as an optimization. Deadlock can still be possible though. Buffering must not be added to fix a deadlock. It is much easier to fix a blocking program by starting with zero buffering and thinking through the dependencies [6]. If you are sure there is no deadlock, you can bravely add buffering.

And what if we need to use more than one channel to send and receive data from multiple goroutines? We can create new channels in if-statements, but this is not a good code practice, because Go already has special statement to deal with multichannel situations. It's called select. The select statement lets a goroutine wait on multiple communication operations [5]. Select can be compared to switch

statement but only for Go channels. In each case of select statement you can describe what operations your code should execute when the data is sent to the channel or received by it. The further execution of the code will be blocked by default before at least one channel won't be ready (i.e. data is received or sent). If more than one channels are ready to work, the order of case statements execution will be randomly generated by Go. If you need to handle those parts that are not listed in case statements, the default case can be specified, this approach allows the specific code statements to execute when there's no channels that are ready.

The Go mechanisms for handling concurrency and especially the goroutines are very useful when designing concurrent programs because they provide convenient toolset and handling all the work with OS threads, so the developers don't need to refer to them directly by themselves. As we found out, goroutines are also very time-efficient, so we believe that these practices will be adopted by many other programming languages in the future.

References

1. Myren S. RT capabilities of Google Go. Primo, 2011.
2. Versockas P. Go Scheduler: MS, PS & GS. 2017. URL: https://povilasv.me/go-scheduler/
3. Cox-Buday K. Concurrency in Go: Tools and Techniques for Developers. O'Reilly Media, 2017.
4. Hoars A. Communicating Sequential Processes. Prentice-Hall, 1985.
5. Google. Official Golang Documentation. URL: https://golang.org/doc/effective_go.html
6. Kennedy W., Ketelsen B., St. Martin E. Go in Action. Manning, 2015.
7. Steele T., Kottmann D., Patten C. Black Hat Go: Go Programming For Hackers and Pentesters. No Starch Press, 2020.