Implementing Goroutines

WE'LL COVER THE FOLLOWING

In the current implementation of the runtime, Go does not parallelize code by default. Only a single core or processor is dedicated to a Go-program, regardless of how many goroutines are started in it. Therefore, these goroutines are _running concurrently; they are not running in parallel, which means only one goroutine is running at a time. This will probably change, but until then, in order to let your program execute simultaneously with more cores so that goroutines are really running in parallel; you have to use the variable `GOMAXPROCS`. This tells the run-time how many goroutines can execute in parallel. For example, if you have **8** processors, you can at most run **8** goroutines in parallel.

Using `GOMAXPROCS`

The developer must set `GOMAXPROCS` to more than the default value 1 to allow the run-time support to utilize more than one OS thread. All goroutines share the same thread unless `GOMAXPROCS` is set to a value greater than 1. When `GOMAXPROCS` is greater than 1, they run on a thread pool with that many threads.

Suppose `n` is the number of processors or cores on the machine. If you set the environment variable `GOMAXPROCS >= n`, or call `runtime.GOMAXPROCS(n)`, then the goroutines are divided (distributed) among the `n` processors.

> **Note:** To run in parallel means a goroutine must run on a different thread, and each thread must run on a different processor.

More processors, however, don't necessarily mean a linear improvement in performance, mainly because more communication is needed: the message-passing overhead increases. An experiential rule of thumb seems to be, that for **n** cores, setting `GOMAXPROCS` to **n-1** yields the best performance, and the following rule should also be followed ***number of goroutines*** > ***1*** + ***GOMAXPROCS*** > ***1***

If there is only one goroutine executing at a certain point in time, don't set `GOMAXPROCS`!

Here are some other observations from experiments. On a 32 core machine, the best performance was reached with `GOMAXPROCS=8`; a higher number didn't improve performance in that benchmark. Very large values of `GOMAXPROCS` degraded performance only slightly; using the **"H"** option to **"top"** showed only 7 active threads, for `GOMAXPROCS=100`.

Programs that perform concurrent computation should benefit from an increase in `GOMAXPROCS`. In simple words, `GOMAXPROCS` is equal to the number of (concurrent) threads. On a machine with more than **1** core, as many threads as there are cores can run in parallel, so set: `runtime.GOMAXPROCS(runtime.NumCPU())`, where `NUMCPU` is the number of logical CPUs on the local machine.

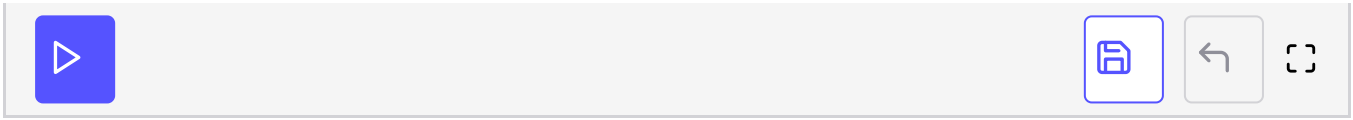
```
package main
import (
"fmt"
"time"
)

func main() {
  fmt.Println("In main()")
  go longWait()
  go shortWait()
  fmt.Println("About to sleep in main()")
  time.Sleep(10 * 1e9) // sleep works with a Duration in nanoseconds (ns) !
  fmt.Println("At the end of main()")
}
```

```
func longWait() {
  fmt.Println("Beginning longWait()")
  time.Sleep(5 * 1e9) // sleep for 5 seconds

  fmt.Println("End of longWait()")
}

func shortWait() {
  fmt.Println("Beginning shortWait()")
  time.Sleep(2 * 1e9) // sleep for 2 seconds
  fmt.Println("End of shortWait()")
}
```



Goroutine

At each step, the above program prints statements to indicate which part of the execution phase the program is in. At **line 9**, a goroutine starts the function `longWait` (defined from **line 16** to **line 20**), which pauses its execution for 5s at **line 18**. Immediately after `longWait` starts, at **line 10**, a goroutine starts the function `shortWait` (defined from **line 22** to **line 26**), which pauses its execution for 2s at **line 24**. Then, the `main()` goroutine pauses itself for 10s at **line 12**.

The 3 functions `main()`, `longWait()` and `shortWait()` are started in this order as independent processing units, and then work concurrently. Each function outputs a message at its beginning and at the end of its processing. To simulate their processing times, we use the `Sleep` function from the `time` package. `Sleep()` pauses the processing of the function or goroutine for the indicated amount of time, which is given in *nanoseconds* (ns, the notation 1e9 represents 1 times 10 to the 9^{th} power, e = exponent). They print their messages in the order we expect, always the same. But, we see clearly that they work simultaneously, in parallel. We let the `main()` pause for 10s, so we are sure that it will terminate after the two goroutines. If not (say if we let `main()` stop for only 4s), `main()` stops the execution earlier and `longWait()` doesn't get the chance to complete. If we do not wait in `main()`, the program stops and the goroutines die with it.

When the function `main()` returns, the program exits. It does not wait for other (non-main) goroutines to complete. That is the reason why in server-programs where each request is handled by a response started as a goroutine, the `server()` function, which starts the goroutines, must be kept alive. This is

usually done by starting it as an infinite loop.

Moreover, goroutines are independent units of execution, and when a number of them start one after the other, you cannot depend on when a goroutine will actually be started. The logic of your code must be independent of the order in which goroutines are invoked.

To contrast this with one thread, successive execution, remove the `go` keywords, and let the program run again. Now, the output is:

```
In main()
Beginning longWait()
End of longWait()
Beginning shortWait()
End of shortWait()
About to sleep in main()
At the end of main() // after 17 s
```

A more useful example of using goroutines could be to search for an item in a very large array. Divide the array in a number of non-overlapping slices, and start a goroutine on each slice with the search algorithm. In this way, a number of parallel threads can be used for the search task, and the overall search time will certainly be decreased (divided by the number of goroutines).

If `f(x)` is a function, you can launch `n` goroutines with it through a *for* loop, like this:

```
// launch n goroutines f
for i := 0; i < n; i++ {
  go f(...)
}
```

The `main()` waiting for the goroutines to finish

This can be accomplished with a coordinating channel: the semaphore pattern, which we'll study later in this chapter. Another solution is to use the type `sync.WaitGroup`, which exactly serves this purpose: a `WaitGroup` waits for a collection of goroutines to finish. The main goroutine calls `Add` on the `WaitGroup` object to set the number of goroutines to wait for. Every goroutine takes a pointer to the `WaitGroup` value as a parameter when it is invoked. When each of the goroutines runs, it calls `Done` when finished. The `main()`

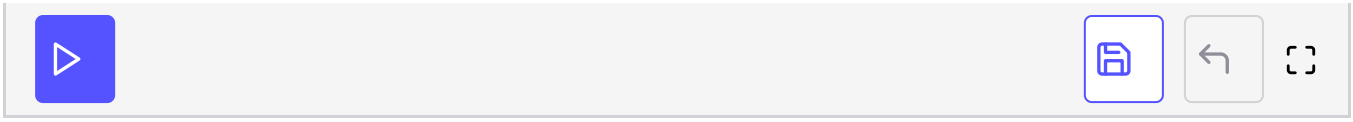
calls the `Wait()` method to block itself until all the goroutines have finished. The following code snippet illustrates this:

```
package main
import (
  "fmt"
  "sync"
)

func HeavyFunction1(wg *sync.WaitGroup) {
  defer wg.Done()
  // Do a lot of stuff
}

func HeavyFunction2(wg *sync.WaitGroup) {
  defer wg.Done()
  // Do a lot of stuff
}

func main() {
  wg := new(sync.WaitGroup)
  wg.Add(2)
  go HeavyFunction1(wg)
  go HeavyFunction2(wg)
  wg.Wait()
  fmt.Printf("All Finished!")
}
```



Waitgroup

To get a synchronized behavior that is not dependent on time, a function launched in a goroutine can register to a `Waitgroup` instance, here `wg`, as one of its parameters. At **line 18**, we construct the `WaitGroup` `wg`, and at **line 19**, we say that it has to wait for *two* goroutines.

Line 20 and **line 21** start our goroutines, passing `wg` as a parameter. Both goroutines will signal `wg` just before their end of execution with `wg.Done()` (see **line 8** and **line 13**). Then, the `WaitGroup` waits in the `main()` function at **line 22**, until it has received as many `Done` signals as there were goroutines registered. Only after that, processing continues with **line 23**.

Specifying the number of cores

Use the `flags` package, as in:

```
var numCores = flag.Int("n", 2, "number of CPU cores to use")
```

In `main()`:

```
flag.Parse()
runtime.GOMAXPROCS(*numCores)
```

Goroutines and coroutines

Other languages like C#, Lua and Python have a concept of coroutines. The name indicates that there is a similarity with goroutines, but there are 2 differences:

- Goroutines imply *parallelism* (they can be deployed in parallel), coroutines, in general, do not.
- Goroutines communicate via *channels*, whereas, coroutines communicate via *yield* and *resume* operations.

In general, goroutines are much more powerful than coroutines, and it is easy to port coroutine logic to goroutines.

You may have noticed that the goroutines in this lesson are not communicating with each other. See the next lesson to see how they can communicate.