The `runtime` Package

This lesson will provide some useful information on the runtime Package.

WE'LL COVER THE FOLLOWING

Hopefully, you are fully aware by now that concurrency is not the same as parallelism. Now, Go provides us with building blocks of concurrency such as goroutines and channels which allow us to solve the problem concurrently. We can solve the problem through parallelism by running concurrent operations on additional CPUs. This is only possible if the problem to be solved is solvable by adding more CPUs. Thus, we can speed up solving problems by providing the necessary hardware. However, it is not necessary that our program will always speed up by adding CPUs. For example, by adding more CPUs, you are increasing the cost of communication/synchronization between operations as communication between OS threads require a significant cost. Hence, you have to optimize your design and resources.

Let's see how can we control the number of CPUs for our programs to optimize our solution:

`runtime` package

The "runtime" package provides us with functions that allow us to control goroutines by interacting with Go's runtime.

`GOMAXPROCS`

Here is an example of a useful function:

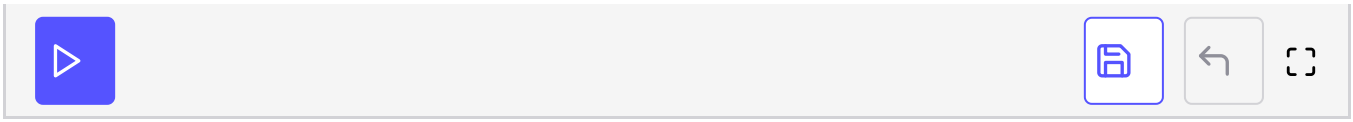
`GOMAXPROCS()` : `func GOMAXPROCS(n int) int`

GOMAXPROCS allows us to set the maximum number of CPUs that can be executed simultaneously. Calling the function will set the number of CPUs to be *n* but will return the previous value set for the number of the CPUs. The default value for this function is the number of CPUs available. If we input n less than 1, the function will return the previous setting. Have a look below:

```
package main

import (
    "runtime"
    "fmt"
)

func main() {
    fmt.Printf("GOMAXPROCS is %d\n", runtime.GOMAXPROCS(3))
    fmt.Printf("GOMAXPROCS is %d\n", runtime.GOMAXPROCS(0))
}
```



`runtime.GOMAXPROCS(3)` sets the number of CPUs as `3` but returns the previous value `2` set for the number of CPUs. Furthermore, `runtime.GOMAXPROCS(0)` just returns the previous setting, which is `3` in this case, because the value of `n` is less than `1`.

You can modify the number of CPUs using the `GOMAXPROCS` to enhance the performance of your program. The number of CPUs needed for a problem depends on the problem. If you set it to 1, you'll eliminate parallelism from your program as now the goroutines will just have one CPU for execution and they will have to execute sequentially despite being parallel in nature.

NumCPU

Here is another function:

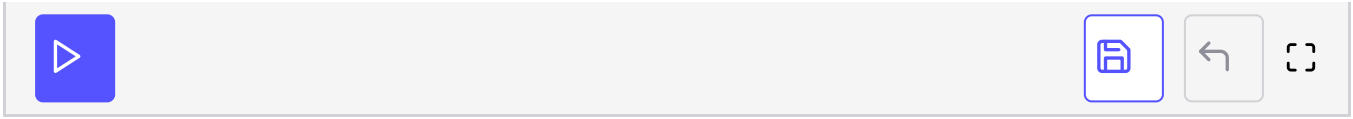
`NumCPU`: `func NumCPU() int`

`NumCPU()` returns the number of logical CPUs that can be used by the current process.

```
package main
```

```
import (
    "runtime"
    "fmt"
)

func main() {
    fmt.Printf("NumCPU is %d\n", runtime.NumCPU())
}
```



As you can see, the number of CPUs used by the above code widget is `2`. The above functions taught in this lesson are used in cases where increasing the number of CPUs will help solve the problem more efficiently. Do explore the documentation for this package as you might come across several other functions that can improve the performance or efficiency of your concurrent programs.