



Going Go Programming

Golang For The Common Man : <https://github.com/goinggo>

Advertise your products & services here.

Contact: bill@ardanstudios.com

Wednesday, January 29, 2014

Concurrency, Goroutines and GOMAXPROCS

Introduction

When new people join the [Go-Miami](#) group they always write that they want to learn more about Go's concurrency model. Concurrency seems to be the big buzz word around the language. It was for me when I first started hearing about Go. It was Rob Pike's [Go Concurrency Patterns](#) video that finally convinced me I needed to learn this language.

To understand how Go makes writing concurrent programs easier and less prone to errors, we first need to understand what a concurrent program is and the problems that result from such programs. I will not be talking about CSP (Communicating Sequential Processes) in this post, which is the basis for Go's implementation of channels. This post will focus on what a concurrent program is, the role that goroutines play and how the GOMAXPROCS environment variable and runtime function affects the behavior of the Go runtime and the programs we write.

Processes and Threads

When we run an application, like the browser I am using to write this post, a process is created by the operating system for the application. The job of the process is to act like a container for all the resources the application uses and maintains as it runs. These resources include things like a memory address space, handles to files, devices and threads.

A thread is a path of execution that is scheduled by the operating system to execute the applications code against a processor or core. A process starts out with one thread, the main thread, and when that thread terminates the process terminates. This is because the main thread is the origin for the application. The main thread can then in



[Learn More](#)

Using Go to power industrial-strength message queuing and task processing.

Blog Archive

▼ [2014](#) (10)

► [May](#) (1)

► [April](#) (1)

► [March](#) (2)

► [February](#) (2)

▼ [January](#) (4)

[Concurrency, Goroutines and GOMAXPROCS](#)

[Decode JSON Documents In Go](#)

[Be Selected To Attend GopherCon 2014](#)

[Go Package Management For 2014](#)

► [2013](#) (46)

If you like the blog,
you'll love the
book.



turn launch more threads and those threads can launch even more threads. Once we have more than one thread running in our program, we have a concurrent program.

The operating system schedules a thread to run on an available processor or core regardless of which process the thread belongs to. Each operating system has its own algorithms that make these decisions and it is best for us to write concurrent programs that are not specific to one algorithm or the other. Plus these algorithms change with every new release of an operating system, so it is dangerous game to play.

Goroutines and Parallelism

Goroutines are functions that we request the Go runtime goroutine scheduler to execute concurrently. We can consider that the main function is executing on a goroutine, however the Go runtime does not start that goroutine. Goroutines are considered to be lightweight because they use little memory and resources plus their initial stack size is small. Prior to version 1.2 the stack size started at 4K and now it starts at 8K. The stack has the ability to grow and shrink as needed.

The operating system schedules threads to run against available processors and the Go runtime schedules goroutines to run against available threads from the schedulers thread pool. By default the schedulers thread pool is allocated with only one thread. Even with one thread, hundreds of thousands of goroutines can be scheduled to run concurrently. It is not recommended to change the size of the schedulers thread pool, but if you want to run goroutines in parallel, Go provides the ability to change the size of the schedulers thread pool via the GOMAXPROCS environment variable or runtime function.

Parallelism is when two or more threads are executing code simultaneously against different processors or cores. We can achieve running goroutines in parallel as long as we are running on a machine with multiple processors or cores and we add more than one thread to the schedulers thread pool. If we add more threads to the schedulers thread pool but run our program on a single CPU machine, our goroutines will run against multiple threads but will be running concurrently against the single CPU, not in parallel.

Concurrency Example

Let's build a small program that shows Go running goroutines concurrently. In this example we are using the default setting for the schedulers thread pool which is one thread:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("Starting Go Routines")
    go func() {
        for {
            // ...
        }
    }()
}
```

Get it Now

MEAP

Latest Releases

Golang v1.2.1

GDB v7.7

LiteIDE v22

MongoDB v2.4.9

MGO r2014.03.12

Code Downloads

Golang Package

GDB Package

LiteIDE Package

MongoDB

MGO Package

Go Language Docs

Website

Documentation

References

Packages


The Project

Command Tools

Memory Model

Effective Go

About Me




g+ William Kennedy

Follow


229

Ardan Studios
12973 sw 112 st, #153
Miami, FL 33186
bill@ardanstudios.com
ArdanStudios.com
[View my complete profile](#)



Follow Me On Twitter


Sponsor



OutCast

Stay Ahead Of Mother Nature

Sponsor



Ardan Studios

Make the Web and

open in browser

PRO version

Are you a developer? Try out the [HTML to PDF API](#)

pdfcrowd.com

```

for char := 'a'; char < 'a'+26; char++ {
    fmt.Printf("%c ", char)
}

}()

go func() {
    for number := 1; number < 27; number++ {
        fmt.Printf("%d ", number)
    }
}()

fmt.Println("Waiting To Finish")
time.Sleep(1 * time.Second)
fmt.Println("\nTerminating Program")
}

```

This program launches two goroutines by using the keyword `go` and declaring two anonymous functions. The first goroutine displays the english alphabet using lowercase letters and the second goroutine displays numbers 1 through 26. When we run this program we get the following output:

```

Starting Go Routines
Waiting To Finish
a b c d e f g h i j k l m n o p q r s t u v w x y z 1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
Terminating Program

```

When we look at the output we can see that the code was run concurrently. Once the two goroutines are launched, the main goroutine is put to sleep for 1 second. We need to do this because once the main goroutine terminates, the program terminates. We want to give enough time for the two goroutines to complete their work.

We can see that the first goroutine completes displaying all 26 letters and then the second goroutine gets a turn to display all 26 numbers. Because it takes less than a microsecond for the first goroutine to complete its work, we don't see the scheduler interrupt the first goroutine before it finishes its work. We can give a reason to the scheduler to swap the goroutines by putting a sleep into the first goroutine:

```

package main

import (
    "fmt"
    "time"
)

```

[FAQ](#)
[How To Write Go](#)

Go References

[Go Wiki](#)
[The Way To Go](#)
[Intro To Go](#)
[Go Talks](#)
[Go Google +](#)
[GDB Reference](#)
[GDB Debugging](#)
[Google App Engine](#)

Go Videos

[Gopher Videos - Gryski](#)

Great Blogs

[Go Blog](#)
[Labix Blog](#)
[MongoDB Blog](#)
[Dave Cheney](#)
[Nathan Youngman](#)
[Honnet Go Tip](#)
[Kyle Isom - Crypto](#)
[Mobile Work For You](#)

Sponsor



Miami Innovation Center

Tickets On Sale Now



April 24th - 26th, 2014

Golang Meetups

[Search Meetups](#)
[Miami](#)
[Tampa](#)
[San Francisco](#)

Go Groups

[Golang Nuts](#)
[Golang Dev](#)
[Go Package Management](#)

```
func main() {
    fmt.Println("Starting Go Routines")
    go func() {
        time.Sleep(1 * time.Microsecond)
        for char := 'a'; char < 'a'+26; char++ {
            fmt.Printf("%c ", char)
        }
    }()

    go func() {
        for number := 1; number < 27; number++ {
            fmt.Printf("%d ", number)
        }
    }()

    fmt.Println("Waiting To Finish")
    time.Sleep(1 * time.Second)
    fmt.Println("\nTerminating Program")
}
```

This time we add a microsecond of sleep in the first goroutine as soon as it starts. This is enough to cause the scheduler to swap the two goroutines:

```
Starting Go Routines
Waiting To Finish
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 a
b c d e f g h i j k l m n o p q r s t u v w x y z
Terminating Program
```

This time the numbers display first and then the letters. A microsecond of sleep is enough to cause the scheduler to stop running the first goroutine and let the second goroutine do its thing.

Parallel Example

In our past two examples the goroutines were running concurrently, but not in parallel. Let's make a change to the code to allow the goroutines to run in parallel. All we need to do is change the default size of the schedulers thread pool to use two threads:

```
package main

import (
    "fmt"
```

MongoDB

[10gen Website](#)

[MongoDB Manual](#)

[MGO Group](#)

[MongoDB Group](#)

[Management Service](#)

[DevOps Best Practices](#)

Versioning Packaging Dependency

[Semantic Versioning](#)

[Go Remote Packages](#)

[Releasing Open Source](#)

[Godep](#)

[Rx Dependency](#)

[New Project Template](#)

Cool Tools

[HTHL Color List](#)

[Color Scheme Designer](#)

LiteIDE Dev
Stackoverflow
Reddit Golang

Events

[GopherCon - Apr 2014](#)

[2014](#)

```

    "runtime"
    "time"
)

func main() {
    runtime.GOMAXPROCS(2)

    fmt.Println("Starting Go Routines")
    go func() {
        for char := 'a'; char < 'a'+26; char++ {
            fmt.Printf("%c ", char)
        }
    }()

    go func() {
        for number := 1; number < 27; number++ {
            fmt.Printf("%d ", number)
        }
    }()

    fmt.Println("Waiting To Finish")
    time.Sleep(1 * time.Second)
    fmt.Println("\nTerminating Program")
}

```

Here is the output for the program:

```

Starting Go Routines
Waiting To Finish
a b 1 2 3 4 c d e f 5 g h 6 i 7 j 8 k 9 10 11 12 l m n o p q 13 r s 14
t 15 u v 16 w 17 x y 18 z 19 20 21 22 23 24 25 26
Terminating Program

```

Every time we run the program we are going to get different results. The scheduler does not behave exactly the same for each and every run. We can see that the goroutines are truly running in parallel. Both goroutines start running immediately and you can see them both competing for standard out to display their results.

Conclusion

Just because we can change the size of the scheduler's thread pool, doesn't mean we should. There is a reason the Go team has set the defaults to the runtime the way they did. Especially the default for the scheduler's thread pool. Just know that arbitrarily adding threads to the scheduler's thread pool and running goroutines in parallel will not necessarily provide better performance for your programs. Always [profile](#) and benchmark your programs and make

sure the Go runtime configuration is only changed if absolutely required.

The problem with building concurrency into our applications is eventually our goroutines are going to attempt to access the same resources, possibly at the same time. Read and write operations against a shared resource must always be atomic. In other words reads and writes must happen by one goroutine at a time or else we create race conditions in our programs. To learn more about [race conditions](#) read my post.

Channels are the way in Go we write safe and elegant concurrent programs that eliminate race conditions and make writing concurrent programs fun again. Now that we know how goroutines work, are scheduled and can be made to run in parallel, channels are the next thing we need to learn.

William Kennedy 9:55 PM



+32 Recommend this on Google

6 comments:



Alejandro Gaviria January 30, 2014 at 4:16 PM

Nice post! Question, the 2nd example used the standard time package to offset the goroutine execution order. Is this the 'status quo,' to manage race conditions in production code? I read further in your post that you should always have read and write operations against an atomic shared resource, -by the same goroutine.

Perhaps, by adhering to the latter, developers would be less contentious to run into the race conditions problems. Reason being, you don't always know the runtime of a given goroutine. This is a fairly new topic for me, which is why I'm openly asking based on your experience and on the consensus of the go community.

Reply

▼ Replies



William Kennedy January 30, 2014 at 5:05 PM

I purposely did not talk about handling race conditions or other channel patterns. I used time.Sleep so the post can focus on concurrency. So NO, do not use time.Sleep the way I did in the post with production code. My next post will talk about unbuffered channels and patterns that you can use in production code.

Reply



Daniele Baroncelli March 10, 2014 at 6:01 PM

Great article! One thing I do not have clear, is whether Go is still able to profit of the multiple cores (and how) when GOMAXPROCS is left to its default value.

Reply

▼ Replies



William Kennedy  March 10, 2014 at 6:10 PM

The concurrency model is designed to take full advantage of the hardware. It is a matter of what your program is doing, and how it is doing it. Go provides you the ability to leverage one thread to its fullest by multi-tasking goroutines against it. This puts less load on the OS and minimizes other OS related activity to provide better performance. If you can get the work done using one thread, it is the best way to go. Adding more threads and more cores never guarantees performance. It can only guarantee potential problems and performance issues. As always, you need to benchmark everything you are doing. If you want to talk more send me an email at bill@ardanstudios.com

Reply



Daniele Baroncelli March 10, 2014 at 8:43 PM

William, thanks for your reply. So, does it mean that if your server has 8 cores but you don't modify the default GOMAXPROCS value, 7 out of 8 cores won't be used at all?

Reply

▼ Replies



William Kennedy  March 10, 2014 at 10:14 PM

That is correct and it is the exact behavior you want. The goal is to maximize performance and that is done by leveraging the resources we have available in a way that is "sympathetic" to how the hardware and operating system works. There are lots of nanoseconds and microseconds that can be lost due to threads being queued and swapped in and out of the CPU. Memory caches that need to be flushed and reloaded, OS kernel mode changes and random I/O access.

Go can take a single thread and reuse it to run many different goroutines. If one goroutine needs to go to sleep waiting for an I/O to finish, Go can reuse the thread to run a different goroutine. Keeping one thread and core busy is much better than trying to leverage two or more cores. Less is more, simple is better.

Go attempts to help us write code that is "sympathetic" to the operating system and hardware,

if we choose to write code the Go way. Slices, goroutines, channels and other aspects of the language work together to help us write code that takes advantage of how our servers work best.

[Reply](#)

Enter your comment...

Comment as:

[Publish](#)

[Preview](#)

[Newer Post](#)

[Home](#)

[Older Post](#)

Subscribe to: [Post Comments \(Atom\)](#)

If you like the blog,
you'll love the
book.

[Get It Now](#)



Powered by [Blogger](#).