Achieving concurrency in Go

If I had to choose one great feature of Go, then it has to be in-built concurrency model. Not only it supports concurrency but makes it better. Go Concurrency Model (goroutines) to concurrency is what Docker is to virtualization.

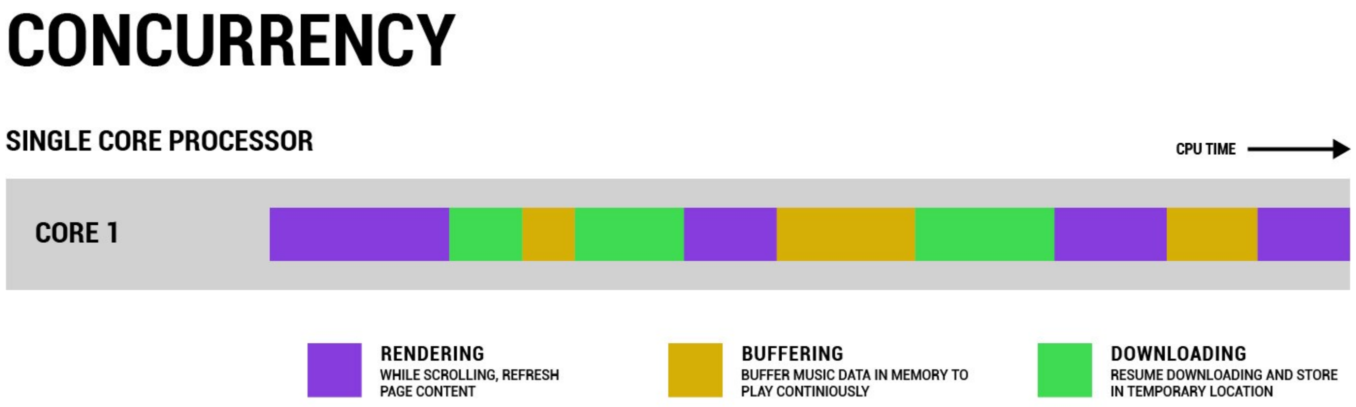
Uday Hiwarale Follow
Nov 3, 2018 · 10 min read

☛ What is concurrency?

In computer programming, **concurrency is ability of a computer to deal with multiple things at once**. For general example, if you are surfing internet in a browser, there might be lot of things happening at once. In a particular situation, you might be downloading some files while listening to some music on the ***page that you are scrolling***, at the same time. Hence a browser needs to deal with lot of things at once. If browser couldn't deal with them at once, you need to wait until all download finishes and then you can start browsing internet again. That would be frustrating.

A general purpose PC might have just one CPU core which does all the processing and computation. A CPU core can handle one thing at a time. When we talk about concurrency, we are doing one thing at a time but we divide CPU Time among things that need to be processed. Hence we get a sensation of multiple things happening at the same time in in reality, only one thing is happening at a time.

Let’s look at the diagram how a CPU manage web browser manages things like in the example we talked about.

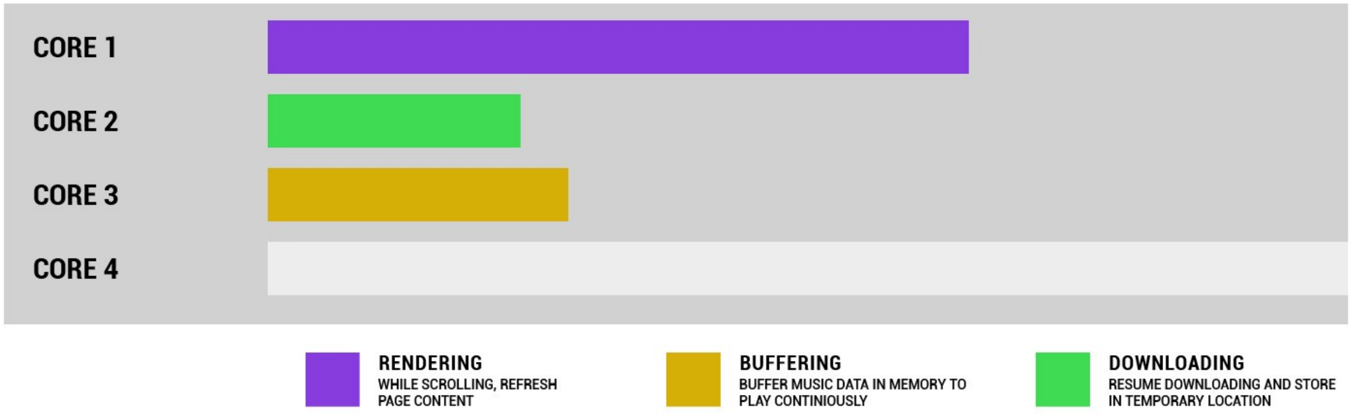

CONCURRENCY
SINGLE CORE PROCESSOR
CPU TIME
CORE 1
RENDERING
WHILE SCROLLING, REFRESH PAGE CONTENT
BUFFERING
BUFFER MUSIC DATA IN MEMORY TO PLAY CONTINIOUSLY
DOWNLOADING
RESUME DOWNLOADING AND STORE IN TEMPORARY LOCATION

So from above diagram, you can see that a single core processor pretty much divide the workload based on the priority of each task, for example, while page scrolling, listening to music may have a low priority, hence sometimes your music stops because of low internet speed but you can still scroll the page.

☛ What is parallelism?

But then the question arises, how about if my CPU has multiple cores? If a processor has multiple processors, then it called as multi-core processor. You might have heard of this term while purchasing a laptop, pc or a smart phone. A multi-core processor has ability to handle multiple things at once.

In previous web browsing example, our single core processor had to divide CPU time among different things. With multi-core processor, we can run separate things at once in different cores. Let’s evaluate that using below diagram.



Concept of running different things in parallel known as `parallelism`. When our CPU have multiple cores, we can use different CPU cores to do multiple things at once. Hence we could say that we might be able to finish a job (consist lot of things) very quickly, but that's not the case. I will come back to this point.

☛ concurrency vs parallelism

Go recommends to use **goroutines** on one core only but we can modify the Go program to run goroutines on different processor cores. For now, think goroutines as Go functions, because they are, but there is more to it.

There are several differences between concurrency and parallelism. While **concurrency is dealing with multiple things at once, parallelism is doing multiple things at once**. Parallelism is not always beneficial over concurrency, we will learn this in upcoming lessons.

At this point, there might be a lot of questions flying in your head and you might have got the idea of concurrency but you might be wondering how Go implements it and how you can use it. To understand Go's architecture of concurrency and how to use it in your code, as well as when to use it in your application architecture, we need to understand what computer processes are.

☛ What is a computer process?

When you write a computer program in languages like `C`, `java` or `Go`, it is just a text file. But as your computer only understands binary instructions which are composed of `0s` and `1s`, you need to compile that code to machine language. This is where compiler comes in. In scripting languages like `python` and `javascript`, the interpreter does the same thing.

When a compiled program is sent to OS to handle, OS allocates different things like memory address space (*where process's heap and stacks will be located*), a program counter, a PID (*process id*) and other very crucial things. A process has at least one thread known as primary thread, while primary thread can create multiple other threads. When the primary thread is done with its execution, process exits.

So we understood that process is a container that has compiled the code, memory, different OS resources and other things which can be provided to threads. **In nutshell, a process is a program in the memory**. But then what are threads, what's their job?

☛ What is a thread?

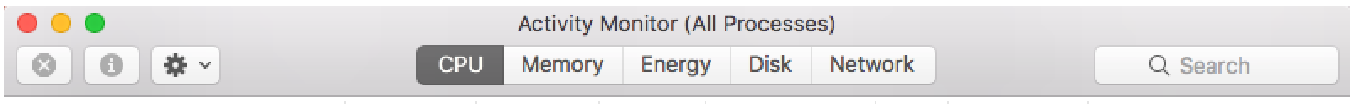
A thread is a light-weight process inside a process. A thread is the actual executor of a piece of code. A thread has access to memory provided by the process, OS resources, and other things.

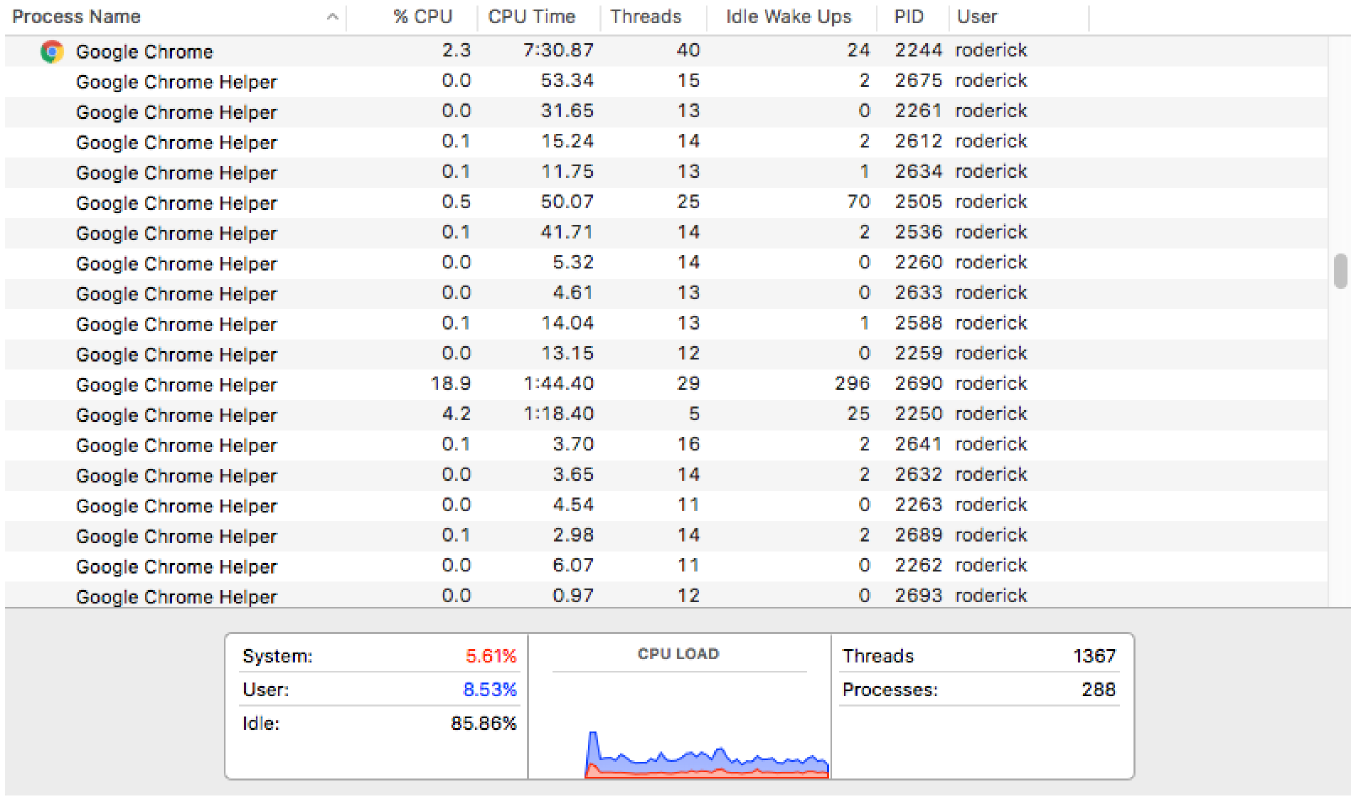
While executing code, thread store variables (*data*) inside the memory region is known as `stack` which `scratch space` where variables hold temporary space. A stack is created at compile time and is normally of a fixed size, preferably 1-2 MB. While the stack of a thread can be used by only that thread and will not be shared with other thread. A heap is a property of a process and it is available to use by any thread. Heap is a shared memory space where data from one thread can be access by other threads as well.

Now we got a general idea of the process and a thread. But what is their use?

When you start a web browser, there must be some code which instructs OS to do something. This means we are creating a process. That process may ask OS to create another process for a new tab. When a browser tab opens and you are doing your normal everyday things, that tab process will start creating different threads for different activities (*like page scroll, downloading, listening to music, etc.*) as we seen in previous diagrams before.

Below is the screen-grab of `Chrome Browser` application on macOS platform.





Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
Google Chrome	2.3	7:30.87	40	24	2244	roderick
Google Chrome Helper	0.0	53.34	15	2	2675	roderick
Google Chrome Helper	0.0	31.65	13	0	2261	roderick
Google Chrome Helper	0.1	15.24	14	2	2612	roderick
Google Chrome Helper	0.1	11.75	13	1	2634	roderick
Google Chrome Helper	0.5	50.07	25	70	2505	roderick
Google Chrome Helper	0.1	41.71	14	2	2536	roderick
Google Chrome Helper	0.0	5.32	14	0	2260	roderick
Google Chrome Helper	0.0	4.61	13	0	2633	roderick
Google Chrome Helper	0.1	14.04	13	1	2588	roderick
Google Chrome Helper	0.0	13.15	12	0	2259	roderick
Google Chrome Helper	18.9	1:44.40	29	296	2690	roderick
Google Chrome Helper	4.2	1:18.40	5	25	2250	roderick
Google Chrome Helper	0.1	3.70	16	2	2641	roderick
Google Chrome Helper	0.0	3.65	14	2	2632	roderick
Google Chrome Helper	0.0	4.54	11	0	2263	roderick
Google Chrome Helper	0.1	2.98	14	2	2689	roderick
Google Chrome Helper	0.0	6.07	11	0	2262	roderick
Google Chrome Helper	0.0	0.97	12	0	2693	roderick

System:	5.61%
User:	8.53%
Idle:	85.86%
Threads	1367
Processes:	288

Above screen-grab shows that Google Chrome browser is using different processes for opened tabs and internal services. As each process has least one thread, we can see that a Google Chrome process, in this case, has more than 10 threads.

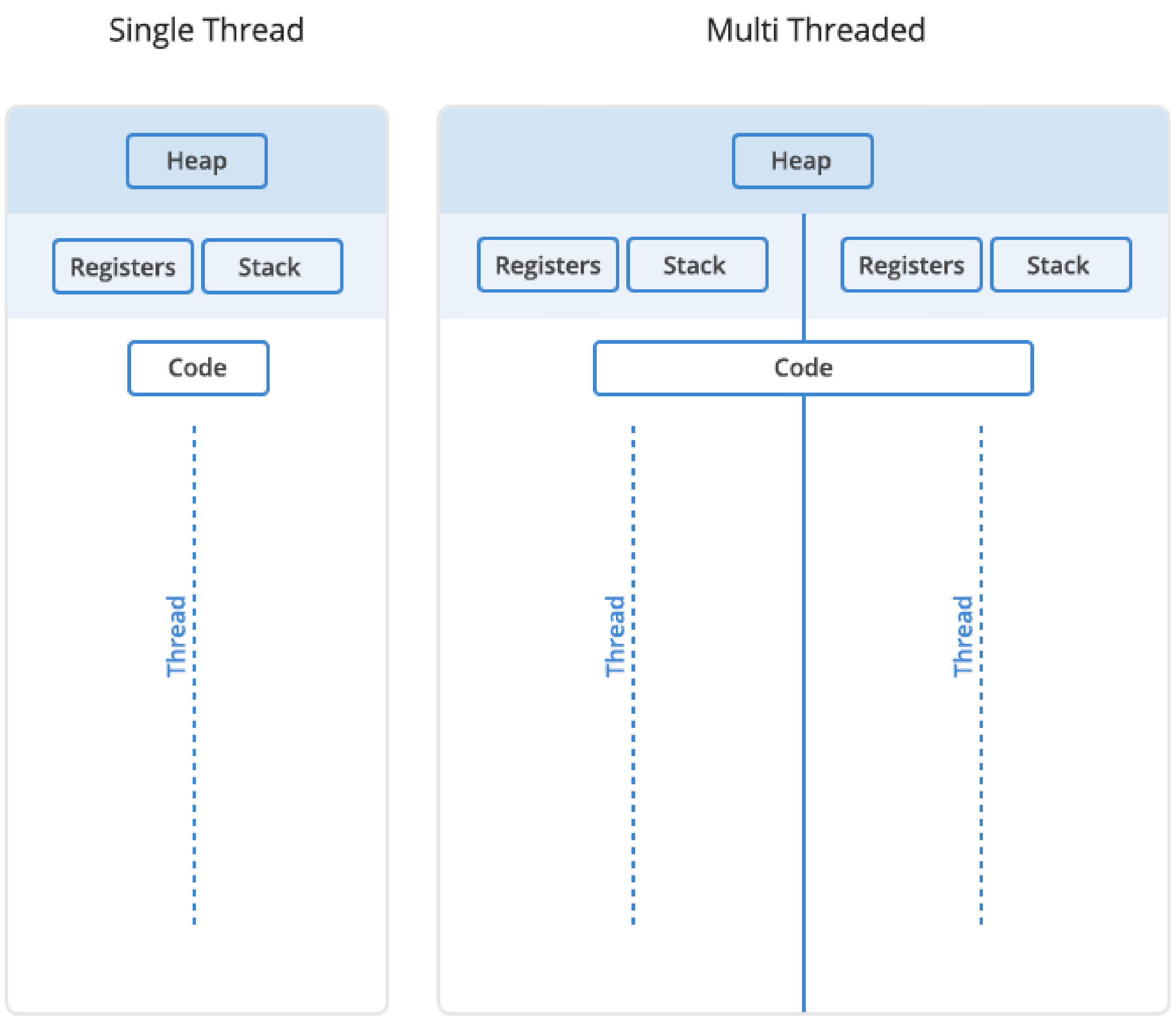
In previous topics, we talked about **dealing with multiple things** or **doing multiple things**. A `thing` here is an activity performed by a thread. Hence when multiple things happen in concurrency or parallelism mode, there are multiple threads running in series or parallel, AKA `multi-theading`.

> *In multi-threading, where multiple threads are spawned in a process, a thread with memory leak can exhaust resources of other thread and make process irresponsive. You might have seen this many times while using browser or any other program. You may have used activity monitor or task manager to see process which is irresponsive and kill it.*

☛ Thread scheduling

When multiple threads are running in series or in parallel, as multiple threads might share some data, threads need to work in coordination so that only one thread can access a particular data at one time. **Execution of multiple threads in some order is called scheduling**. Os threads are scheduled by the kernel and some threads are managed by runtime environment of the programming language, like `JRE`. When multiple threads trying to access the same data at the same time which cause data to be changed or results into unexpected outcome, then `race condition` occurs.

> *While designing concurrent Go programs, we need to look for **race conditions** which we will talk about in upcoming lessons.*



☛ Concurrency in Go

Finally, we reached to a point where we will talk about how Go implements concurrency. Traditional languages like `java` has a thread class which can be used to create multiple threads in the current process. Since Go does not have traditional OOP syntaxes, it provides `go` keyword to create `goroutines`. When `go` keyword is placed before a function call, it becomes `goroutines`.

We will talk about goroutines in next lesson but in nutshell, goroutines behave like threads but technically; it is an abstraction over threads.

When we run a Go program, Go **runtime** will create few threads on a core on which all the goroutines are multiplexed (*spawned*). At any point in time, one thread will be executing one goroutine and if that goroutine is blocked, then it will be swapped out for another goroutine that will execute on that thread instead. This is like **thread scheduling** but handled by **Go runtime** and this is much faster.

It is advised in most of the cases, to run all your goroutines on one core but if you need to divide goroutines among available CPU cores of your system, you can use GOMAXPROCS environment variable or call to runtime using function `runtime.GOMAXPROCS(n)` where `n` is the number of cores to use. But you may sometime feel that setting `GOMAXPROCS > 1` is making your program slower. It truly depends on the nature of your program but you can find a solution or explanation of your problem on the internet. In practical terms, programs that spend more time communicating on channels than doing computation will experience performance degradation when using multiple cores, OS threads, and processes.

Go has an `M:N` scheduler that can also utilize multiple processors. At any time, `M`goroutines need to be scheduled on `N` OS threads that run on at most on `GOMAXPROCS`numbers of processors. At any time, at most only one thread is allowed to run per core. But scheduler can create more threads if required, but that rarely happens. If your program doesn’t start any additional goroutines, it will naturally run in only one thread no matter how many cores you allow it to use.

☛ threads vs goroutines

Since there is an obvious difference between threads and goroutines as we have seen earlier but below differences will shed some light on why threads are more expensive than goroutines and why goroutines is a key solution for achieving the highest level of concurrency in your application.

Above are the few important differences but if you dive deep, you will find out the amazing world of Go's concurrency model. To highlight some of the power points of Go's concurrency strength, imagine you have a web server where you are handling 1000 requests per minute. If you had to run each request concurrently, that means you need to create 1000 threads or divided them under different processes. That's how **Apache server** manages incoming requests (*read here*). If an OS thread consumes 1MB stack size per thread, that means you will exhaust 1GB of RAM for that traffic. Apache provides `ThreadStackSize` directive to manage stack size per thread but still, you have no idea if you run into a problem because of this.

In the case of goroutines, since stack size can grow dynamically, you can spawn 1000 goroutines without a problem. As a goroutine starts with 8KB (***2KB** since Go 1.4*) of stack space, most of them generally don't grow bigger than that. But if there is a recursive operation that demands more memory, Go can increase stack size up to 1GB which I hardly think will ever happen except `for {}` which is obviously a bug.

Also, rapid switching between goroutines is possible and more efficient compared to threads as we saw earlier. Since one goroutine is running on one thread at a time and goroutines are cooperatively scheduled, another goroutine is not scheduled until current goroutine is blocked. If any Goroutine in that thread blocks say waiting for user input, then another goroutine is scheduled in its place. goroutine can block on one of the following conditions

- network input
- sleeping
- channel operation
- blocking on primitives in the sync package

If the goroutine does not block on one of these conditions, it can starve the thread on which it was multiplexed, killing other goroutines in the process. While there are some remedies, but if it does, then it is considered as bad programming.

`Channels` will play a great role while working with goroutines as a medium to share data in between them which we will learn in upcoming lessons. This will prevent race conditions and inappropriate access to shared data between them *as opposed to accessing the shared memory in case of threads.*

More resources

There is a great article on Go scheduler by the name of **Go's work-stealing scheduler** by [https://github.com/rakyll] which you should read to know how Go's runtime manages goroutines.

There is a great talk by **Rob Pike** on concurrency of GoLang with the title "Concurrency Is Not Parallelism".

Since we understood what goroutines are and how they work under the hood, let's dive into the next lesson where we will learn about how to create goroutines and use them in our Go program. goroutines are a great medium to divide the workload among many workers to make the job easy and fast. That's why Go is a perfect programming language to make micro-system architecture for your next application.