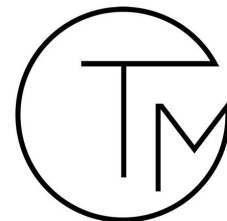


CPU limits & Go

Thomas Gosteli - Bärner Go Talks 2024 no. 3

About me

- Thomas Gosteli, 30 years young
 - interested in 🧐 🏍️ ☕ 🧑🎨 3D 🖨️
- Cloud- and Software Engineer @ TactumMotum (<https://tactummotum.ch/>)
 - 7+ years of experience in cloud native technologies especially with containers & Kubernetes
 - 7+ years of writing Go in professional environments
 - available for hire as a contractor - thomas@tactummotum.ch
- find me on LinkedIn: [Thomas Gosteli](#) or GitHub: [@ghouscht](#)



TACTUMMOTUM.CH



CPU limits?

- are enforced using "cgroups" (control groups)
- cgroups are a Linux kernel feature to isolate/constrain processes
- used by Kubernetes/Docker to enforce resource limits in shared environments
- CFS (Completely Fair Scheduler) on Linux (default from kernel 2.6-6.6) "implements" limits during scheduling

AI GENERATED

Explain CPU limits leveraging cgroups to me like I'm 5 years old.



Sure! Imagine you have a big box of toys, and you and your friends want to play with them. But there's only one of each toy, and sometimes everyone wants to play with the same toy at the same time.

A CPU is like a really cool toy that all the programs on your computer want to play with. But if everyone tries to play with it at the same time, things get chaotic and nobody has fun.

Now, let's say your parents decide to help out. They use some special rules called "cgroups" (short for control groups) to make sure everyone gets a fair turn with the toys.

Here's how they do it:

1. **Organize Players:** They put you and your friends into different groups. Maybe one group likes to play more actively, and another group likes quieter games.
2. **Set Time Limits:** They decide how long each group can play with the toy. For example, Group A can play with the toy for 5 minutes, then it's Group B's turn for 5 minutes, and so on.
3. **Fair Sharing:** If Group A has fewer kids, they might get a bit more time because they don't have to share as much among themselves. If Group B has more kids, their playtime gets divided among more people.

This way, everyone gets a chance to play with the toy without any group hogging it all the time.

In the computer world, cgroups work similarly to manage the CPU. They make sure that each program (like the groups of kids) gets a fair share of the CPU's power (the toy), so the computer runs smoothly and all the programs can do their job without fighting over resources.

There once was a language called Java ...

- Java first appeared 1995 - long before what we now call "containers" or even "cgroups" existed
- Improve docker container detection and resource configuration usage
 - landed in JDK 10
 - released ~ March 2018
 - later on backported to older JDKs including JDK 8
- (see also Allow more flexibility in selecting Heap % of available RAM)

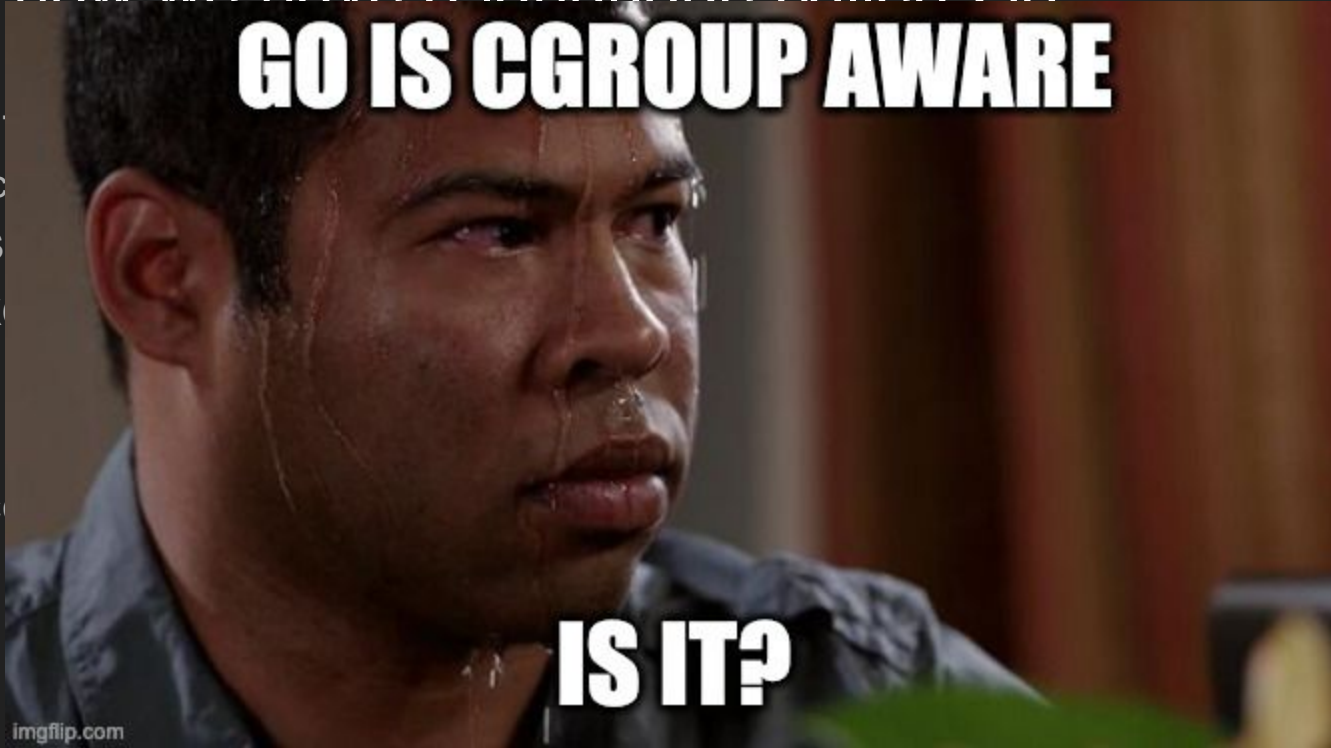
TL;DR: Modern Java is cgroup aware

Disclaimer: I'm not a Java expert, take this with a grain of salt - and yes I know Java still exists nowadays 🙄

... and now we have a language called Go

- Go's
 - c
- Go is
- Dock

Natural c



Go and CPU limits

ChatGPT is a bit wrong!

Go does not honour cgroup limits by default.

GOMEMLIMIT / GOMAXPROCS env. vars or the `runtime` package can be used to "tune" the Go runtime



Yes, Go can automatically detect CPU and memory limits imposed by cgroups (Control Groups) on Linux systems. Here's how it works:

1. **CPU Limits:** Go can detect the CPU limits set by cgroups through the `runtime` package. Specifically, the `runtime.NumCPU` function and related functions in the `runtime` package provide information about the available CPUs as seen by the Go runtime. If your process is restricted to a subset of CPUs by cgroups, Go will reflect this limitation in its CPU-related functions.
2. **Memory Limits:** Similarly, Go applications can detect memory limits set by cgroups. Go's memory management and allocation functions are aware of the memory constraints imposed by the operating system, including those set by cgroups. The `runtime` package provides facilities to query the total system memory and the amount of memory available to the Go process.
3. **cgroup Information:** While Go itself doesn't directly interact with cgroups (as that's typically managed by the operating system), it operates within the constraints set by them. Go programs use the standard system calls and libraries provided by the operating system, which are aware of cgroup limits.

For instance, if a cgroup restricts a process to a maximum of 2 CPU cores and 4 GB of RAM, the Go runtime will work within these limits. If you query the number of CPUs using `runtime.NumCPU()`, it will return 2, reflecting the cgroup configuration.

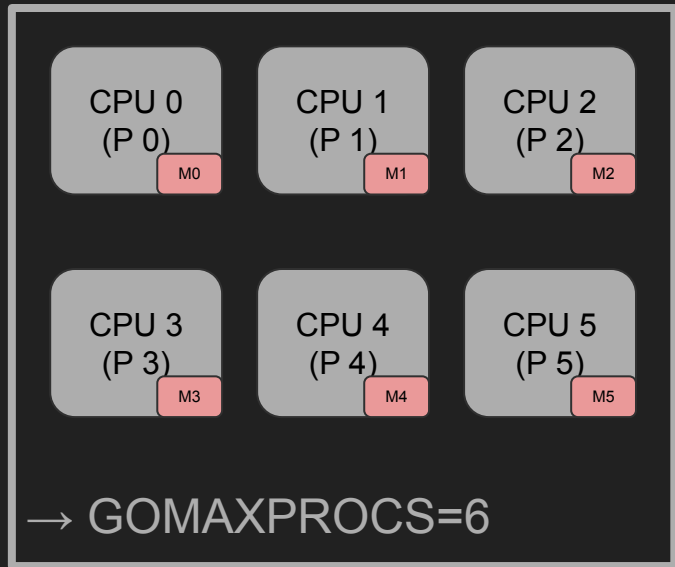
In summary, Go leverages the underlying operating system's facilities and system calls to manage CPU and memory resources, and it respects the constraints imposed by cgroups in a Linux environment. Therefore, it does not need separate or special handling to detect these limits; it inherently adheres to them based on system behavior and configuration.

AI GENERATED

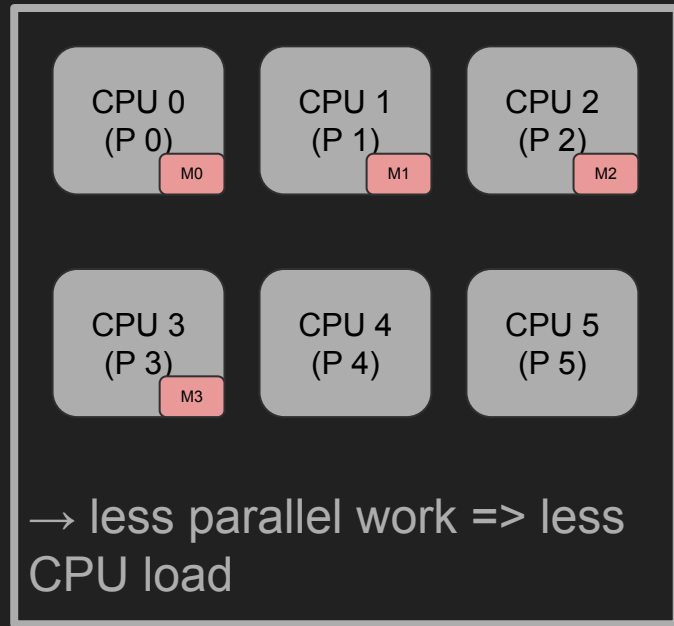
[runtime: make GOMAXPROCS cfs-aware on GOOS=linux](#)

GOMAXPROCS?

default behaviour of Go



GOMAXPROCS=4



G - goroutine

M - worker thread, or machine

P - processor

Goroutine(s) (G) are scheduled on a thread (M) by Go the runtime

GOMAXPROCS?

- Go by default uses all CPU cores - `GOMAXPROCS == #CPU`
- by setting the `GOMAXPROCS` environment variable we can limit/increase the number of threads created by the Go runtime.
- a proper setting (of `GOMAXPROCS`) can help to reduce (tail) latency/runtime in containers with CPU limits
 - CFS by default splits CPU time in slices of 100ms, if you exceed your quota (in the current time slice), CFS will not schedule your process again until the next time slice
 - usually this can be seen in p95/p99 latency spikes



Key takeaways

- if you run any application (no matter the language) in containers and you use CPU (and/or memory) limits make sure your application is cgroup aware.
 - be aware there are two versions of cgroups ([cgroup v2](#)) 🤔
- monitor CPU throttling (e.g. with Prometheus):
`container_cpu_cfs_throttled_seconds_total{}`
- use <https://github.com/uber-go/automaxprocs> in Go applications to automatically set GOMAXPROCS to match your container CPU limits (as good as possible)
 - works with cgroup v1 & v2 by reading the cgroup configuration from `/proc` or `/sys` filesystem
- CPU limits can be everywhere even in your CI/CD pipeline 🙄

IMHO

do not use CPU limits unless you have to and you know what you're doing - use CPU requests instead!

CPU Limits vs Requests		
More details at tinyurl.com/k8s-cpu		
All pods have...	CPU limits	No CPU limits
CPU requests	You are guaranteed CPU between the request and limit Excess cpu is unavailable beyond the limit	You are guaranteed your request. Excess CPU is available and not wasted! **
No CPU requests	You are guaranteed the limit, no more, no less* Excess cpu is unavailable	No one is guaranteed any CPU! Wild west.

* This is because Kubernetes automatically sets the request to the limit
** Excess CPU is given to whoever needs it, prioritized by the size of their request

Note: everything here describes only cpu! Memory behaves totally differently because memory is not compressible.

Image: tinyurl.com/k8s-cpu

Thank you!

<https://github.com/ghouscht/cpu-limits-and-go>

- <https://github.com/uber-go/automaxprocs>
- <https://github.com/golang/go/issues/33803>
- <https://github.com/golang/go/blob/1f0c044d60211e435dc58844127544dd3ecb6a41/src/runtime/proc.go#L22>
- <https://www.kernel.org/doc/html/v5.6/scheduler/sched-bwc.html>
- <https://www.datadoghq.com/blog/kubernetes-cpu-requests-limits/>
- <https://kubernetes.io/docs/concepts/architecture/cgroups/>
- <https://home.robusta.dev/blog/stop-using-cpu-limits>