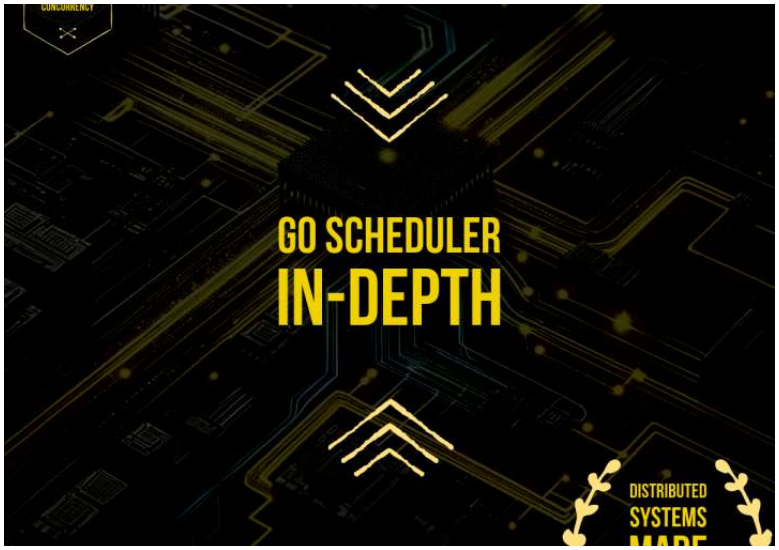


Go Concurrency Series: Deep Dive into Go Scheduler(I)

Pratik Pandey
Senior Software Engineer at Booking.com | AWS Solutions Architect Professional | pratikpandey.substack.com

January 4, 2024

In my last post about Goroutines, we talked about how Goroutines differ from Traditional threads. The Go Runtime manages goroutines, so let's first go over the Go Runtime and then dive deep into the component responsible for all the concurrency magic in Go, i.e. Go Scheduler.

Go Runtime:

The Go runtime is the environment in which the Go code executes. It's a library that gets linked to every Go program(when you build an application executable) and provides critical functionalities needed for program execution. This includes memory allocation, garbage collection, concurrency features (creating goroutines and channels), dynamic types management and other fundamental operations.

The runtime is implemented in Go, with some parts in assembly language for specific architectures. It's bundled as part of the Go language and is not something that developers interact with directly in most cases.

Go Scheduler:

The Go scheduler is a specific part of the Go runtime responsible for managing the execution of goroutines. It's an M:N scheduler, meaning it multiplexes M goroutines onto N OS threads. The primary job of the Go scheduler is to distribute goroutines over available threads and cores efficiently. Its responsibilities include:

- **Goroutine Management**: Deciding which goroutine runs on which thread and when. Go Scheduler acts like an orchestrator, managing the goroutines.
- **Multiplexing:** Go Scheduler is responsible for performing the M:N multiplexing, where it can efficiently run M goroutines on N OS threads. It aims to keep all CPU cores busy and to avoid bottlenecks.
- **Cooperative Scheduling** & **Preemption**: Interrupting goroutines ensures that long-running goroutines don't starve others from CPU time. From Go 1.14, the Go scheduler preempts goroutines, rather than following cooperative scheduling.

M:N Multiplexing:

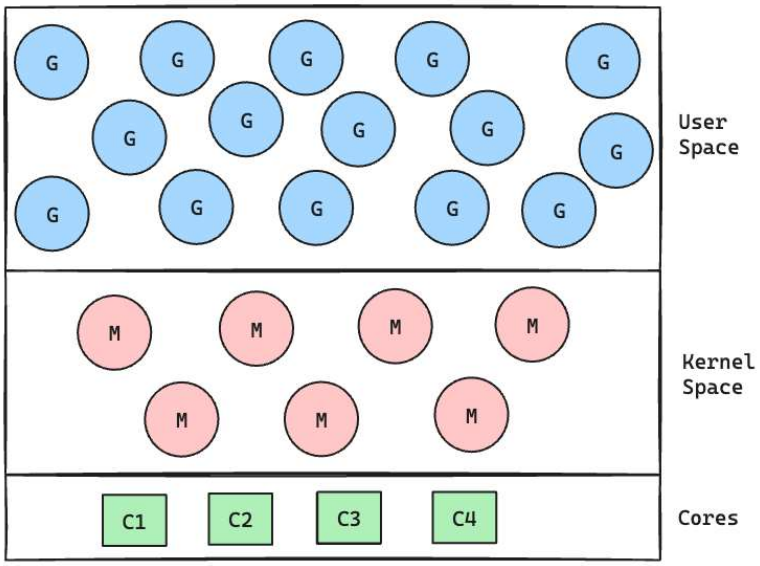
Go Runtime creates the Goroutines, in the user space, while the OS threads are created in the Kernel space. This allows the Go Scheduler complete control over the scheduling of the Goroutines.

Why was this done?

If you think about it, Go applies the same concept that was once applied to create threads. Earlier, we could run one program on one core and we created the concept of threads that allowed us to reuse the core when there were Blocking system calls, by switching the context to a separate thread. This allowed us to have more threads than the number of cores available, however, there can only be as many running threads as Cores(Physical Limitation).

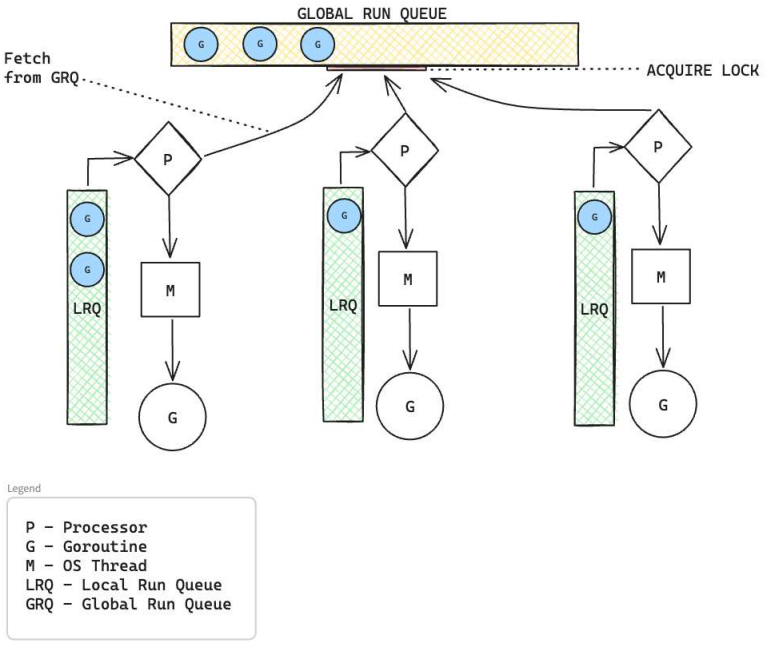
The model of Goroutines is an extension of that, where we create lightweight threads and the number of Goroutines we can have is more than the number of OS threads. Each goroutine starts with a small stack that grows as needed, and the overhead for managing these stacks is much lower compared to traditional threads.

Goroutines are multiplexed onto a smaller number of OS threads by the Go scheduler. This M:N model (many goroutines to few OS threads) means that the number of goroutines can far exceed the number of available cores. The Go scheduler, which operates in user space, can efficiently manage these goroutines, switching between them with less overhead than an OS-level context switch.



User Space Vs Kernel Space

Components of Go Scheduler



Components of Go Scheduler

In the current design of the scheduler, we have the following components:

1. **G(Goroutine):** Goroutine(G) is the basic unit of concurrency in Go. Each Goroutine contains the stack, the instruction pointer, and other necessary state information to execute a given task.
2. **M(OS Thread):** These are actual threads provided by the underlying operating system. The Go runtime schedules Goroutines (G) onto these machines (M) for execution. Each M is responsible for executing Goroutines and interacts directly with the operating system.
3. **P(Processor):** Processor (P) represents a context for executing goroutines. It can be thought of as a local scheduler or a resource that is required to execute goroutines. Each P is bound to a single OS thread (M) at any given time and has its local run queue of goroutines that are ready to be executed. P is responsible for picking up goroutines from the Global Run Queue to populate its local run queue, as well as "steal" goroutines from the local run queue of another P. **The number of processors is the maximum number of GOMAXPROCS.**
4. **Local Run Queue:** Each P has its local run queue of goroutines that are ready to run on that specific processor. These local queues allow for efficient scheduling of goroutines on individual threads without needing constant synchronization across all threads.
5. **Global Run Queue:** This queue holds goroutines that are ready to run but have not yet been assigned to a specific thread's local run queue. Since all P's can access the Global Run Queue to pick up unassigned goroutines, we need to have synchronization mechanisms in place to ensure that no two P's pick up the same goroutine.

How do your Goroutines run?

So, whenever you create a goroutine, the Go Scheduler adds it to the local run queue of the Processor(P) that created it i.e. the processor P which is running the main/child goroutine. This approach is preferred because it keeps the overhead low.

> Adding a goroutine directly to the local run queue of the current P avoids the need for cross-processor communication and synchronization, which can be more expensive in terms of performance. This is because P has been running on a specific OS thread (M), and hence its data (including the LRQ) is likely to be in the CPU cache, leading to faster access and execution.

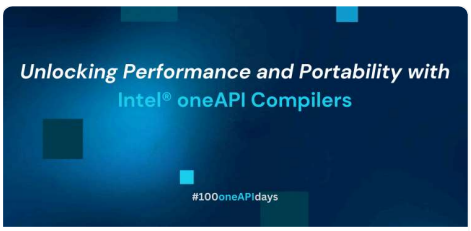
However, if the local run queue of a Processor(P) is full, then the Go Scheduler adds the Goroutines to the Global Run Queue. Goroutines in the GRQ can be picked up by any Processor (P) whose local run queue is empty or needs more work. This allows a great way to load balance between the different Processors.

The Processor (P) in Go's runtime scheduler picks up Goroutines from its Local Run Queue (LRQ) and schedules them to be executed on an OS thread to which the Processor is currently bound. If the local run queue is empty, the Processor can acquire the lock and pick up multiple goroutines(optimization) from the Global Run Queue!

In our next article, we'll continue diving deep into the Go Scheduler where we'll focus on Work Stealing and how Golang ensures fairness! Also, do provide me feedback on what I can do better!

More articles for you

Kotlin Coroutines vs. Threads: Key Differences and When to Use Them
Sumitkumar Dhule
16



Unlocking Performance and Portability with Intel® oneAPI Compilers
Arun GK
26 • 2 reposts

Optimizing Utilization of OpenShift Cluste – A Focus on Resource Allocation and Container Limits
Rupang Mehta

51 • 5 comments