JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#31

GOROUTINES: THE M:P:G MODEL

THE MAGIC OF THE GO SCHEDULER

Why not OS Threads?

The Cost of Context Switching

OS threads are expensive. They have a large stack (usually ~2MB) and switching between them requires a "Context Switch" into the kernel, which takes thousands of CPU cycles.

The Go Alternative:
Goroutines start with a tiny 2KB stack that grows and shrinks dynamically. Switching between them happens in "User Space" (inside your app), which is significantly faster.

Defining the Actors

Meet the Trinity - G, M, P

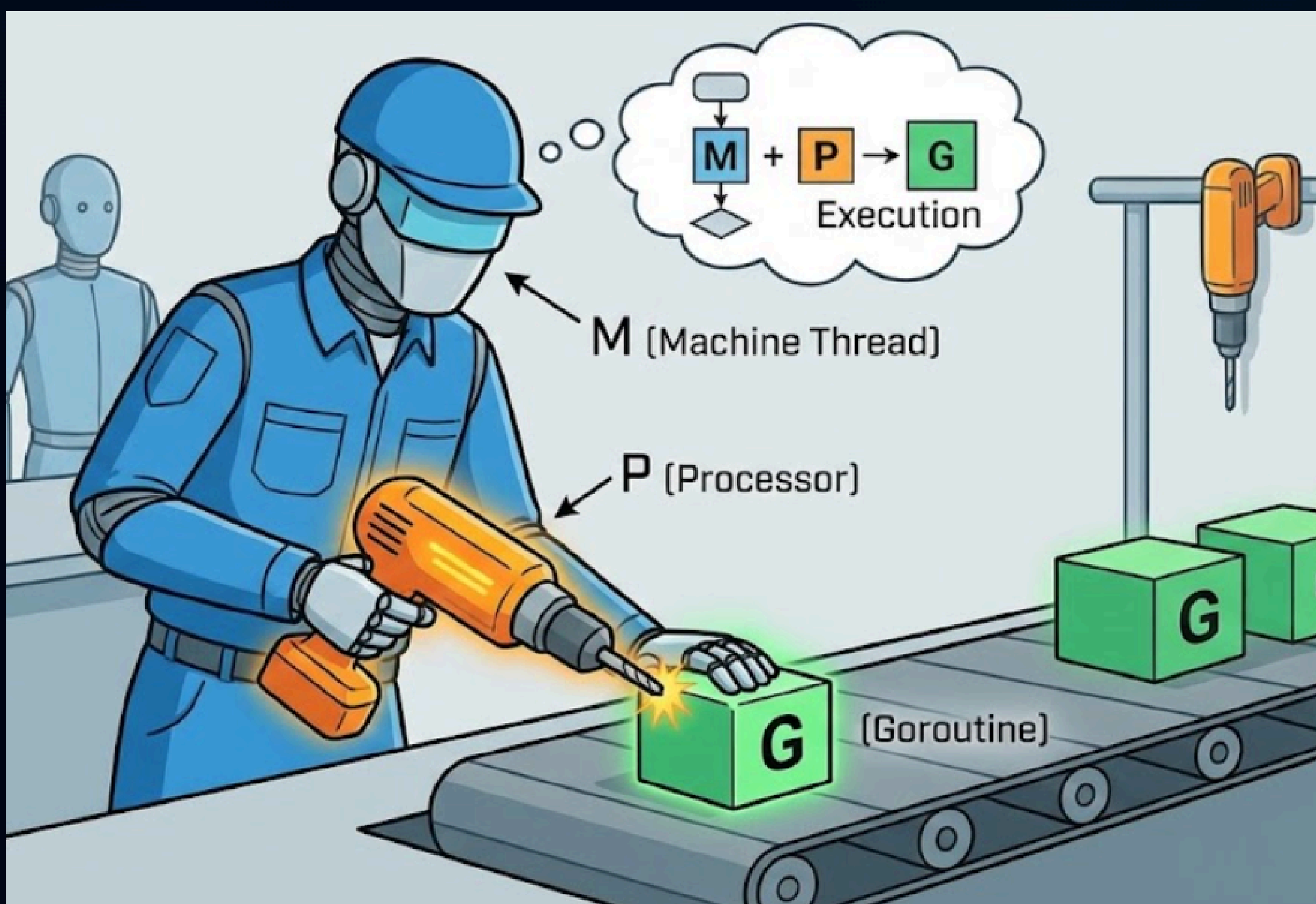
The Go scheduler manages three entities to execute code:

1. **G (Goroutine)**:
Represents the executable code and its stack. It is the "What."

2. **M (Machine)**:
An actual OS Thread. It is the "Where."

3. **P (Processor)**:
A resource that represents a logical CPU. It is the "Context."

The Rule:
An M must hold a P to execute a G.



The Scheduling Loop

How Work is Distributed

Every P has a local "Run Queue" of Goroutines waiting to execute. The M (thread) attached to that P picks a G and runs it.

By default, the number of Ps equals *GOMAXPROCS* (usually the number of CPU cores). This ensures that we don't have more OS threads trying to run than we have actual hardware to support them, reducing unnecessary context switching.

Balancing the Load

Work Stealing

What happens if one P finishes its queue while others are still busy?

The Mechanism: The idle P will look at other Ps and "steal" half of their local run queue.

The Benefit: This ensures all your CPU cores stay busy and prevents "Hot Spots" where one core is overwhelmed while others sit idle.

Dealing with Blocking (Syscalls)

Handing off the Processor

When a Goroutine makes a blocking system call (like reading a file), the M (thread) becomes blocked.

The Handoff:

To keep things moving, the Go runtime detaches the P from that blocked M and moves it to a new (or idle) M.

The Result:

Your other Goroutines keep running on the new thread while the old thread waits for the OS to finish the I/O.

Cooperating with the Scheduler

Preemption and Yielding

Modern Go (since 1.14) uses Asynchronous Preemption. The scheduler can forcibly stop a long-running Goroutine (like a tight loop) to ensure other Gs get a turn.

You don't usually need to call *runtime.Gosched()* anymore. However, be aware that code that doesn't involve function calls or allocations might still be harder for the scheduler to preempt in edge cases.

Tuning for the Environment

The Cost of *GOMAXPROCS*

In containerized environments (Kubernetes/Docker), Go might see the total cores of the physical host, not the "CPU Limit" assigned to the pod.

The Fix:
Use the *uber-go/automaxprocs* library. It automatically sets *GOMAXPROCS* to match your container's CPU quota, preventing significant performance degradation due to CPU throttling.

The Runtime is Your Partner. Recap:

- G is the task, M is the thread, P is the CPU context.
- Work Stealing keeps all cores balanced.
- Syscalls trigger thread handoffs to prevent blocking the whole app.
- Use automaxprocs in Kubernetes.

Sorry for the more theoretic post, but don’t worry - tonight we get hands on! Don’t miss