


Why goroutines are not lightweight threads?

 codeburst.io/why-goroutines-are-not-lightweight-threads-7c460c1f155f

March 23, 2018

GoLang is gaining incredible popularity these days.

One of the main reasons for that is the simple and lightweight concurrency in the form of goroutines and channels that it offers to the developers.

Concurrency has existed since long ago in the form of Threads which are used in almost all the applications these days.

To understand why goroutines are not lightweight threads, we should first understand how Thread works in OS.

If you are already familiar with Threads, you can directly skip **here**.

What are Threads?

A thread is just a sequence of instructions that can be executed independently by a processor. Threads are lighter than the process and so you can spawn a lot of them.

A real life application would be a web server.

A webserver typically is designed to handle multiple requests at the same time. And these requests normally don't depend on each other.

So a Thread can be created (or taken from a Thread pool) and requests can be delegated, to achieve concurrency.

Modern processors can executed multiple threads at once (multi-threading) and also switch between threads to achieve parallelism.

Are threads lighter than processes?

Yes and No.

In concept,

1. Threads share memory and don't need to create a new virtual memory space when they are created and thus don't require a MMU (memory management unit) context switch
2. Communication between threads is simpler as they have a shared memory while processes requires various modes of IPC (Inter-Process Communications) like semaphores, message queues, pipes etc.

That being said, this doesn't always guarantee a better performance than processes in this multi-core processor world.

e.g. Linux doesn't distinguish between threads and processes and both are called tasks. Each task can have a minimum to maximum level of sharing when cloned.

When you call *fork()*, a new task is created with no shared file descriptors, PIDs and memory space. When you call *pthread_create()*, a new task is created with all of the above shared.

Also, synchronising data in shared memory as well as in L1 cache of tasks running on multiple cores takes a bigger toll than running different processes on isolated memory.

Linux developers have tried to minimise the cost between task switch and have succeeded at it. Creating a new task is still a bigger overhead than a new thread but switching is not.

What can be improved in Threads?

There are three things which make threads slow:

1. Threads consume a lot of memory due to their large stack size ($\geq 1\text{MB}$). So creating 1000s of thread means you already need 1GB of memory.
2. Threads need to restore a lot of registers some of which include AVX(Advanced vector extension), SSE (Streaming SIMD Ext.), Floating Point registers, Program Counter (PC), Stack Pointer (SP) which hurts the application performance.

3. Threads setup and teardown requires call to OS for resources (such as memory) which is slow.

Goroutines

Goroutines exists only in the virtual space of go runtime and not in the OS.

Hence, a Go Runtime scheduler is needed which manages their lifecycle.

Go Runtime maintains three C structs for this purpose:

1. **The G Struct** : This represents a single go routine with it's properties such as stack pointer, base of stack, it's ID, it's cache and it's status
2. **The M Struct** : This represents an OS thread. It also contains a pointer to the global queue of runnable goroutines, the current running goroutine and the reference to the scheduler
3. **The Sched Struct** : It is a global struct and contains the queues free and waiting goroutines as well as threads.

So, on startup, go runtime starts a number of goroutines for GC, scheduler and user code. An OS Thread is created to handle these goroutines. These threads can be at most equal to GOMAXPROCS.

Start from the bottom!

A goroutine is created with initial only 2KB of stack size. Each function in go already has a check if more stack is needed or not and the stack can be copied to another region in memory with twice the original size. This makes goroutine very light on resources.

Blocking is fine!

If a goroutine blocks on system call, it blocks it's running thread. But another thread is taken from the waiting queue of Scheduler (the Sched struct) and used for other runnable goroutines.

However, **if you communicate using channels in go which exists only in virtual space, the OS doesn't block the thread.** Such goroutines simply go in the waiting state and other runnable goroutine (from the M struct) is scheduled in its place.

Don't interrupt!

The go runtime scheduler does cooperative scheduling, which means another goroutine will only be scheduled if the current one is blocking or done. Some of these cases are:

- Channel send and receive operations, if those operations would block.
- The Go statement, although there is no guarantee that new goroutine will be scheduled immediately.
- Blocking syscalls like file and network operations.
- After being stopped for a garbage collection cycle.

This is better than pre-emptive scheduling which uses timely system interrupts (e.g. every 10 ms) to block and schedule a new thread which may lead a task to take longer than needed to finish when number of threads increases or when a higher priority tasks need to be scheduled while a lower priority task is running.

Another advantage is that, since it is invoked implicitly in the code e.g. during sleep or channel wait, the compiler only needs to save/restore the registers which are alive at these points. In Go, this means **only 3 registers i.e. PC, SP and DX (Data Registers) being updated during context switch** rather than all registers (e.g. AVX, Floating Point, MMX).
