Erik Pellizzon

Go Developer
Freelancer

https://erikpelli.pp.ua

Download the slides of this presentation

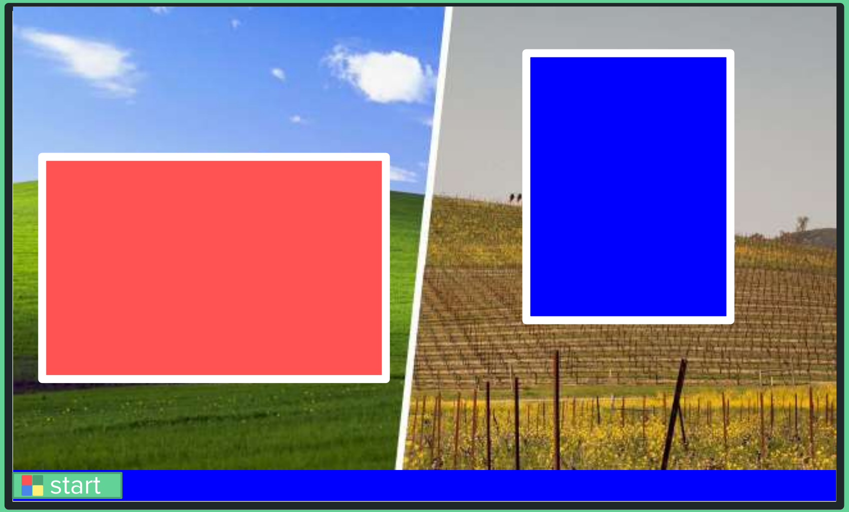
Open Source is like communism, but it works

The rise of green threads

In the presentation, underlined text contains the sources

History

Why did we need concurrency?

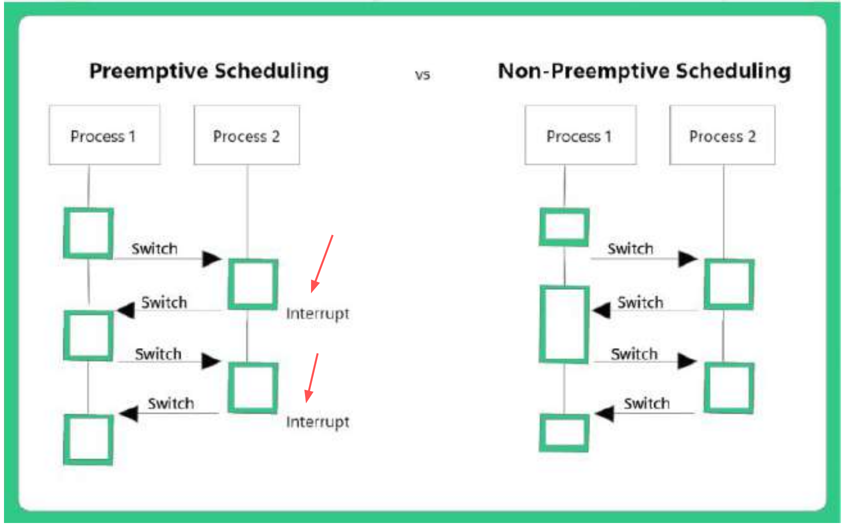


Ancient PCs History

More than 30 years ago!

1991: release of Linux 0.01, with a simple (100 C lines) integrated scheduler (non-preemptive, we have to wait for 2.6 in **2003**)

1995: release of Windows 95, the first Windows with a preemptive scheduler

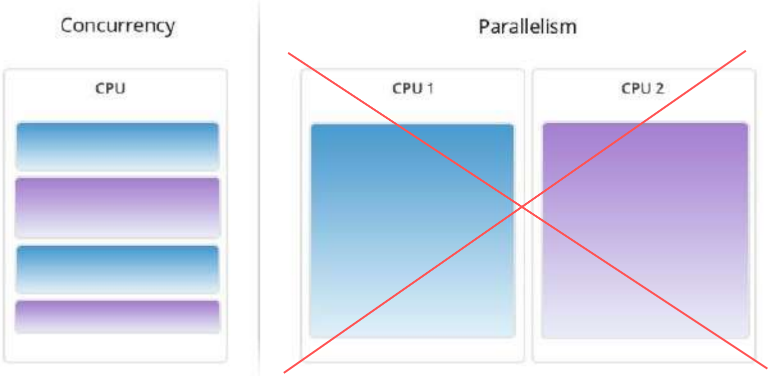


Linux 0.01 -> Intel 80836, 16MHz, 1MB RAM

Windows 95 -> Intel 80486, 16MHz, 4MB RAM, supports application-level threads

---

Only 1 core! Basically, processes and threads were using only “fake” concurrency, handled by the OS Scheduler (part of Kernel)



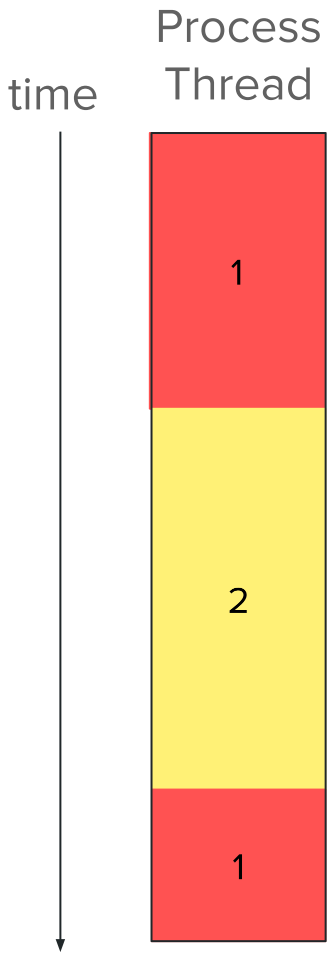
Coroutines

Lightweight concurrency for weak CPUs

Make it clear!

The original purpose of coroutines was to be a lightweight alternative to threads, where the concurrency wasn't handled by the OS scheduler but directly from the process itself.

Since the CPUs had only 1 core, their design was very simple, to reduce overhead, reducing the overhead that OS threads implied.



First notable usages

1967: Simula 67, the first language with support for coroutines (+ the first OOP language). Quite limited, the code explicitly needed to return the control to the runtime (these are the original coroutines).

IBM
System/370
(Mainframe)

Features:
- single core CPUs
- < 10 MB RAM

1997: Java 1.1 introduced Threads API, using "green threads" as the JVM implementation behind it. In **2000,** Java 1.3 replaced them with native threads (+300% with a 4 threads CPU).

"In green threads all Java threads execute within one operating system lightweight process (LWP)"

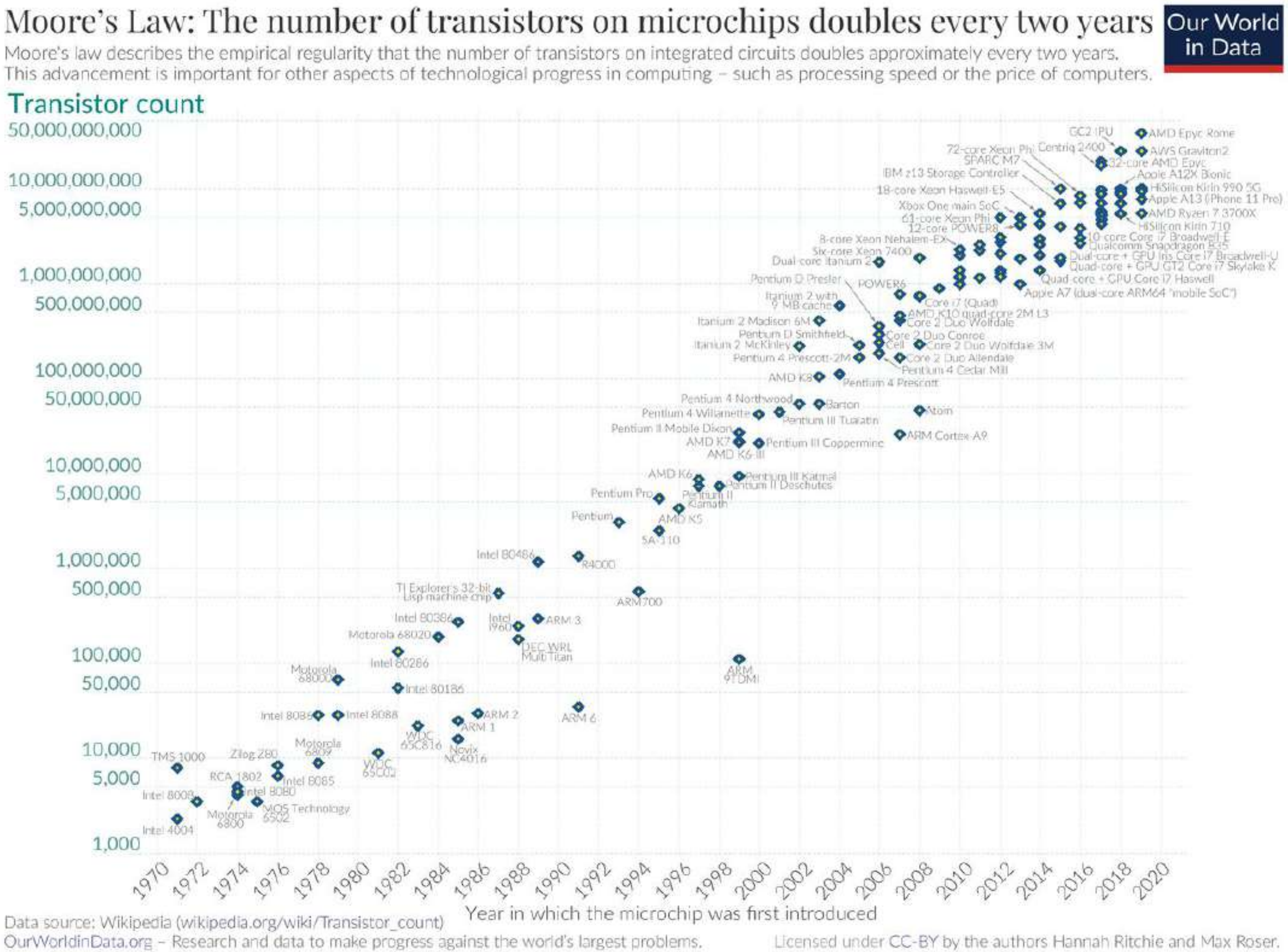
1 : 1

OS Thread ← wraps — Java Thread

exposes API to your code

Still valid nowadays

Modernity



CPU manufacturers introduced multiple techniques to overcome the limits of a single core (including multiple parallel cores)

Homework (for you):

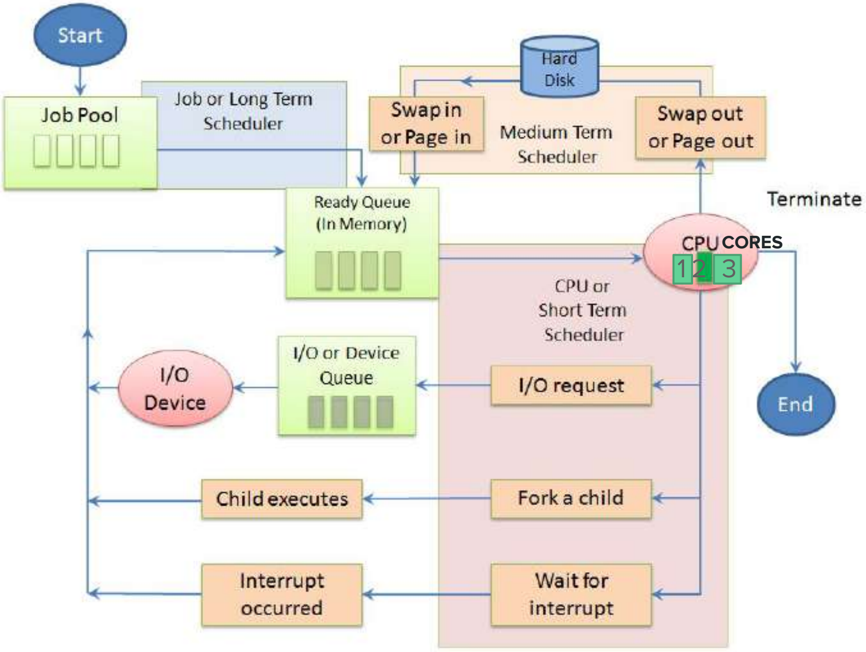
- HyperThreading
- CPU architectures
- CPU cache (L1, L2, L3)
- Power and Energy efficient cores
- CPU Frequency
- Specialized CPU instructions & extensions (e.g. AVX2)
- Internal RISC core in CISC (e.g. x86)

Modern Desktop PC

Typical modern CPU:

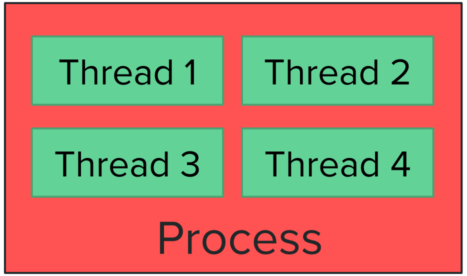
- i5 14600k
- 4GHz (250x on single thread vs 16MHz 80486)
- 20 parallel threads (simplification), 16GB RAM (4096x)

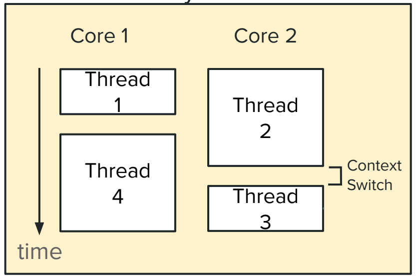
This supports the much more complex scheduler of a Modern OS.

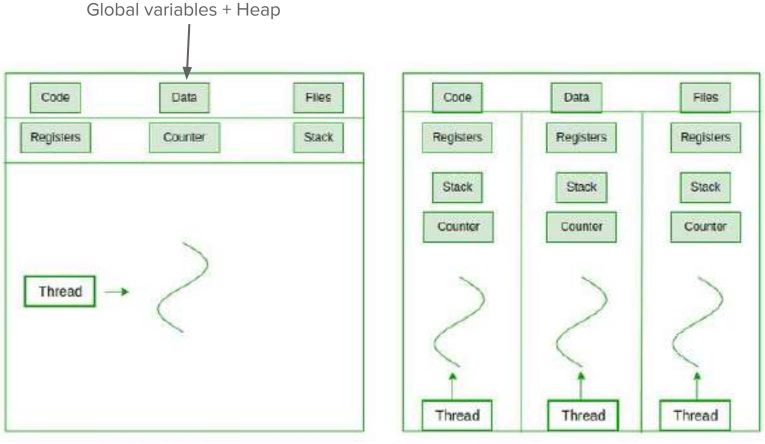


Threads

Modern scheduling =
Concurrency + Parallelism

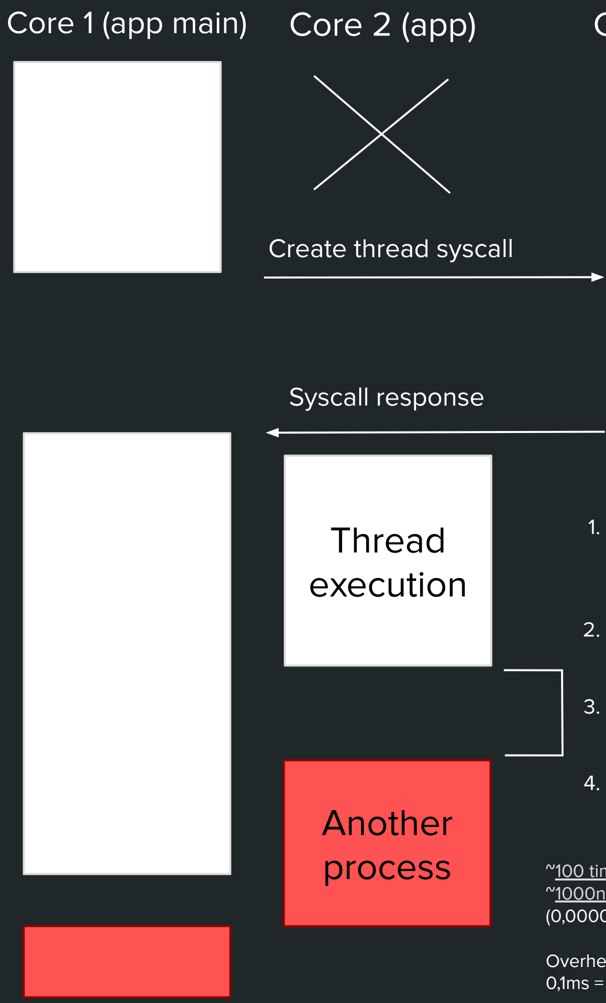






Single Threaded Process
Multi Threaded Process

Simplified diagram using static cores



Thread creation waiting time

1. Thread calls a syscall, performs I/O, preemptive time slot expires
2. Scheduler dumps current registers, stack and counter
3. Scheduler restore R, S and C of another process (or thread)
4. Resume code execution

~100 times per second (every 10/15ms)
~1000ns for the context switch
(0,000001s) = 1 µs

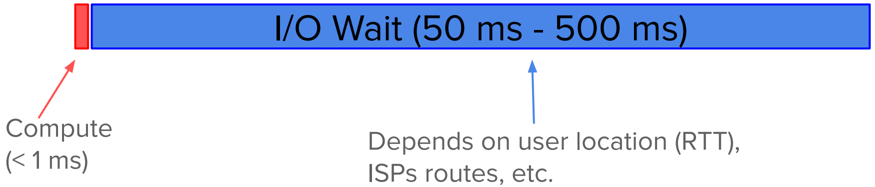
Overhead every 1s: 1000ns*100 = 0,1ms = 0,01%

Threads issues

- A system call is required for each creation and termination of a thread
- High initial stack size (1MB+ in Windows, 10MB in Linux)
- The time the thread is actually started since the request is made is significant (~1ms, up to 10ms if there are a lot of threads to spawn)
- A large number of threads (100+) significantly slows down overall performance, threads are designed to remain few for each process
- Processes have limited control over the scheduling of their threads (apart from choosing the priority)
- Require continuous context switches (only becomes a problem with a really large number of threads, perhaps thousands)

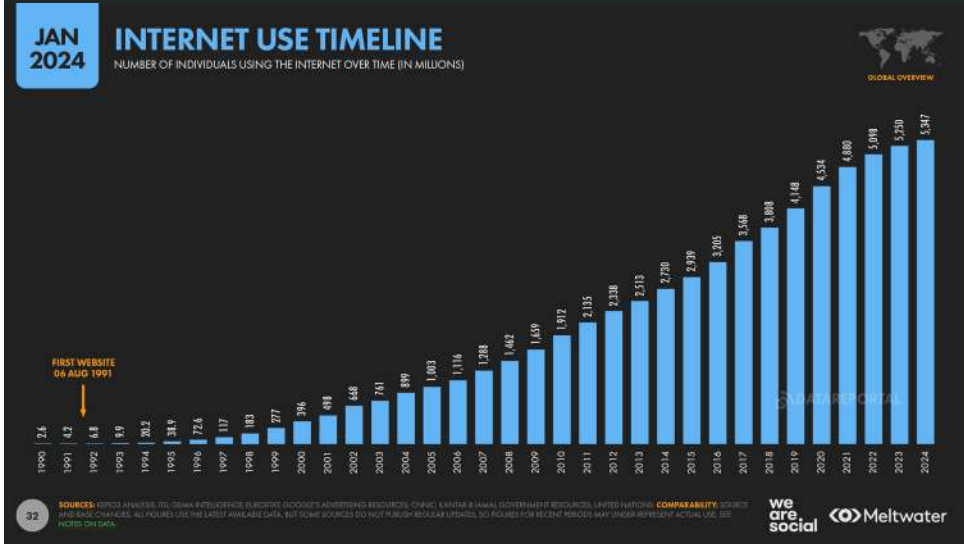
Internet

Typical HTTP Request



- TCP handshake
- TLS exchange (2 RTs)
- HTTP request (to the server) & response (to the client)

4 Round Trips for each request



>>> More $$$ for servers

CLOUDFLARE NETFLIX

Coroutines++

Becoming popular since 2010s

Optimized for modern CPUs and OS

Various names, similar concepts

Ideal for I/O bound tasks

- insignificant creation time
- up to thousands (or millions) of coroutines with a low memory impact
- CPU is reassigned by language runtime when a coroutine is blocked by I/O

2023 — Virtual Threads (Java, Project Loom)

2021 — AMPHP (+ PHP 8.1 introduced native fibers)

- Fibers

2009 — Goroutines (Go)

- Lightweight threads
- Coroutines

2014 — Python's asyncio uses coroutines internally (3.4+)

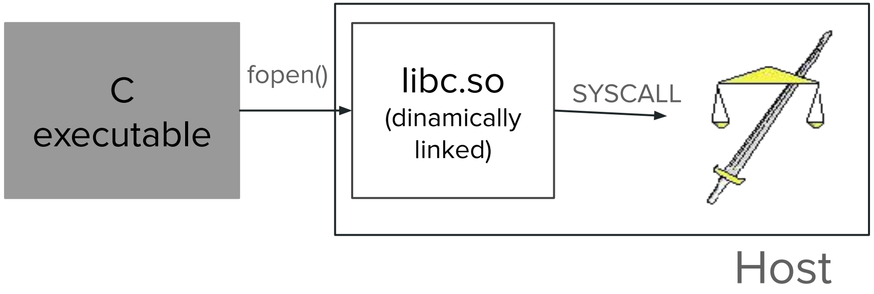
- ...

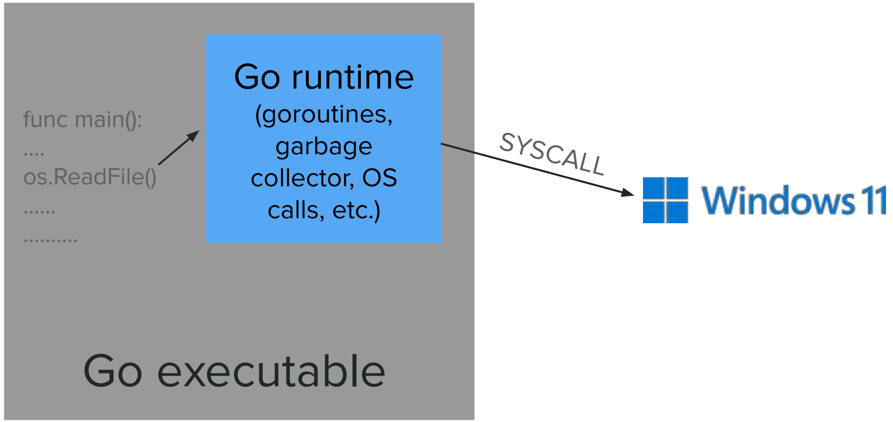
Backend Languages

PART II
Implementation

Go by **Go**ogle

- A real programming language created by real software engineers in a real company, I heard they also organize free developers meetings with alcohol
- Focused on concurrency (goroutines) and simplicity
- 15 years old
- Compiled into a single file (dependencies are statically linked)





Example of optimization for modern CPUs

Maphash

```go
func (h *Hash) Sum64() uint64 {
        h.initSeed()
        return rthash(h.buf[:h.n], h.state.s)
}
```

```go
//go:linkname runtime_memhash runtime.memhash
//go:noescape
func runtime_memhash(p unsafe.Pointer, seed, s uintptr) uintptr

func rthash(buf []byte, seed uint64) uint64 {
        if len(buf) == 0 {
                return seed
        }
        len := len(buf)
        // The runtime hasher only works on uintptr. For 64-bit
        // architectures, we use the hasher directly. Otherwise,
        // we use two parallel hashers on the lower and upper 32 bits.
        if unsafe.Sizeof(uintptr(0)) == 8 {
                return uint64(runtime_memhash(unsafe.Pointer(&buf[0]), uintptr(seed), uintptr(len)))
        }
        lo := runtime_memhash(unsafe.Pointer(&buf[0]), uintptr(seed), uintptr(len))
        hi := runtime_memhash(unsafe.Pointer(&buf[0]), uintptr(seed>>32), uintptr(len))
        return uint64(hi)<<32 | uint64(lo)
}
```

Maphash Runtime

```asm
// func memhash(p unsafe.Pointer, h, s uintptr) uintptr
// hash function using AES hardware instructions
TEXT runtime·memhash<ABIInternal>(SB),NOSPLIT,$0-32
        // AX = ptr to data
        // BX = seed
        // CX = size
        CMPB    runtime·useAeshash(SB), $0
        JEQ     noaes
        JMP     aeshashbody<>(SB)
noaes:
        JMP     runtime·memhashFallback<ABIInternal>(SB)
```

Implemented using a variant of wyhash

asm_<arch>.s (amd64)

AES instructions (& fallback if not supported)

```asm
// AX: data
// BX: hash seed
// CX: length
// At return: AX = return value
TEXT aeshashbody<>(SB),NOSPLIT,$0-0
        // Fill an SSE register with our seeds.
        MOVQ    BX, X0                          // 64 bits of per-table hash
        PINSRW  $4, CX, X0                      // 16 bits of length
        PSHUFHW $0, X0, X0                      // repeat length 4 times tot
        MOVO    X0, X1                          // save unscrambled seed
        PXOR    runtime·aeskeysched(SB), X0     // xor in per-process seed
        AESENC  X0, X0                          // scramble seed

        CMPQ    CX, $16
        JB      aes0to15
```

runtime/ alg.go

Initialized at runtime

```go
// runtime variable to check if the processor we're running on
// actually supports the instructions used by the AES-based
// hash implementation.
var useAeshash bool
```

Fun fact

I opened an issue in Go repository (GitHub) while analyzing the source code

Benchmark on my laptop
Average of 5 measures

maphash_runtime (AES)
8.348 ns

Compiled using flag: ***-tags purego*** →

maphash_purego (no AES)
21.692 ns +160%

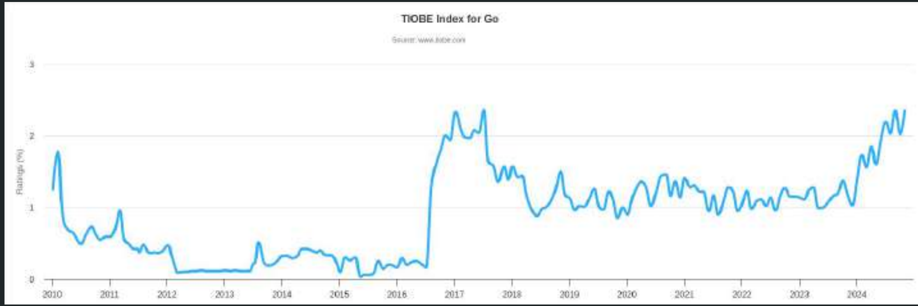
```go
func BenchmarkMapHash(b *testing.B) {
    s := maphash.MakeSeed()

    dataSlice := make([]byte, 128)
    for i := 0; i < len(dataSlice); i++ {
        dataSlice[i] = byte(i)
    }

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        maphash.Bytes(s, dataSlice)
    }
}
```

Initial stage
(2009 - 2014)

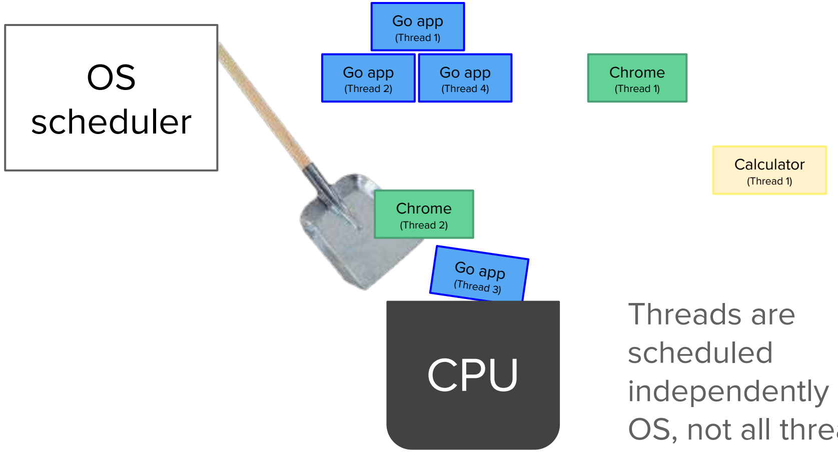
- At that time it was a completely new language, initially the compiler was built using C
- Initially it wasn't extremely optimized, it just needed to work
- It wasn't yet popular or widespread
- Go 1.5 (2015) completely changed the goroutine scheduler and garbage collection, Go 1.14 (2020) added fully preemption (10ms) to the goroutines scheduler (before that, it was cooperative)



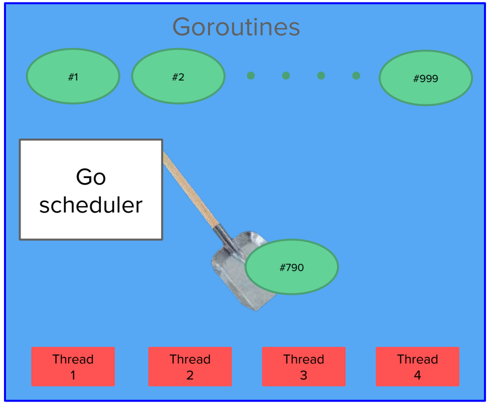
Scheduler

The elephant in the room

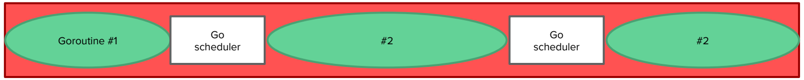
- Go scheduler assigns at a given time a goroutine to an OS thread (and handles the goroutines context switches)
- It limits the number of concurrent active OS threads to the value of environment variable GOMAXPROCS (its default value is the number of CPU threads, e.g. quad core CPU -> its value is 4)
- Part of Go runtime, and it uses CPU itself (overhead). Average goroutine context switch: 50ns (20 times less than OS scheduler context switch).



Threads are scheduled independently by OS, not all threads need to run at the same time.
Both the schedulers (OS and Go runtime) runs on the CPU.
The OS scheduler is part of the kernel (very low level, privileged instructions).



Thread



P-M-G model

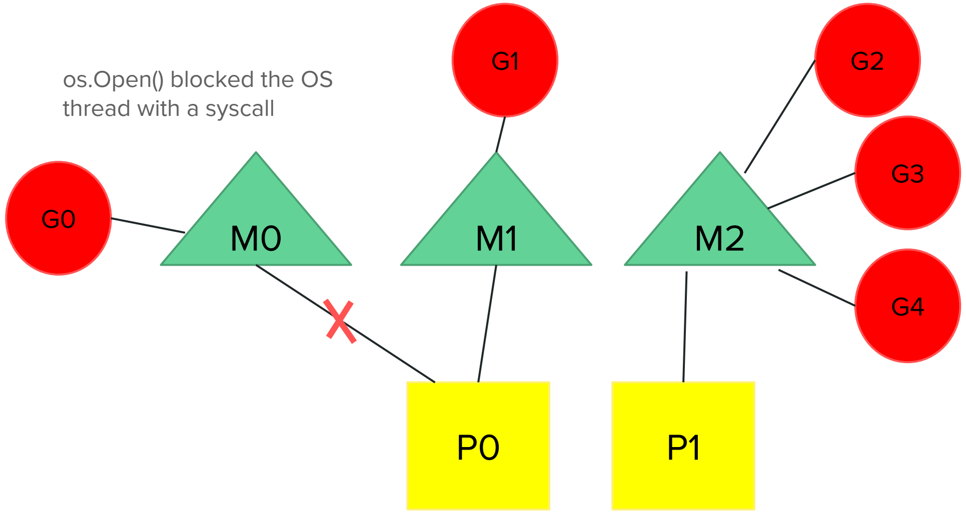
Designed to maximize the performance the CPU can offer.

G: Goroutine

M: Machine, aka OS Thread

P: Processor, aka a logical CPU core (e.g. there are 4 Processors in a quad core CPU)

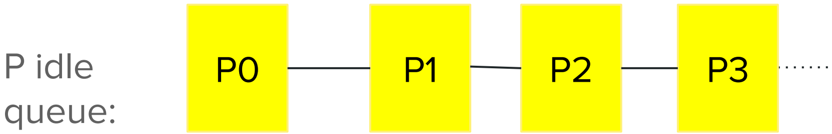
Initial stack size for goroutines (G) is 2KB (5000 times less than the 10MB of a Linux OS thread, virtual memory).
In addition to goroutine stack, the OS Thread (M) has its own 8KB stack to execute the Go runtime (e.g. during goroutines context switches).



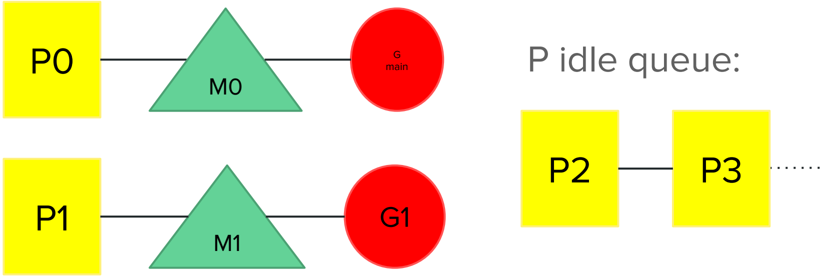
Dual core CPU

"The GOMAXPROCS variable limits the number of operating system threads that can execute user-level Go code simultaneously (M). There is no limit to the number of threads that can be blocked in system calls on behalf of Go code; those do not count against the GOMAXPROCS limit."

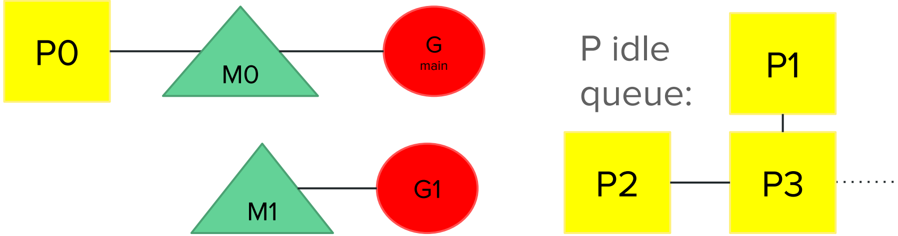
1. Go runtime creates PROCS using the GOMAXPROCS variable, default value is number of cores



2. Runtime assigns P_0 to main goroutine. Runtime also creates a thread M because there is no M in the idle threads list.
G_{main} asks to create another goroutine G_1 (and the runtime creates another thread, M_1)



3. G1 reads a file. The syscall blocks the entire OS thread until it finishes, so Go detaches its M from P, freeing up space for another potential M to be executed in P.



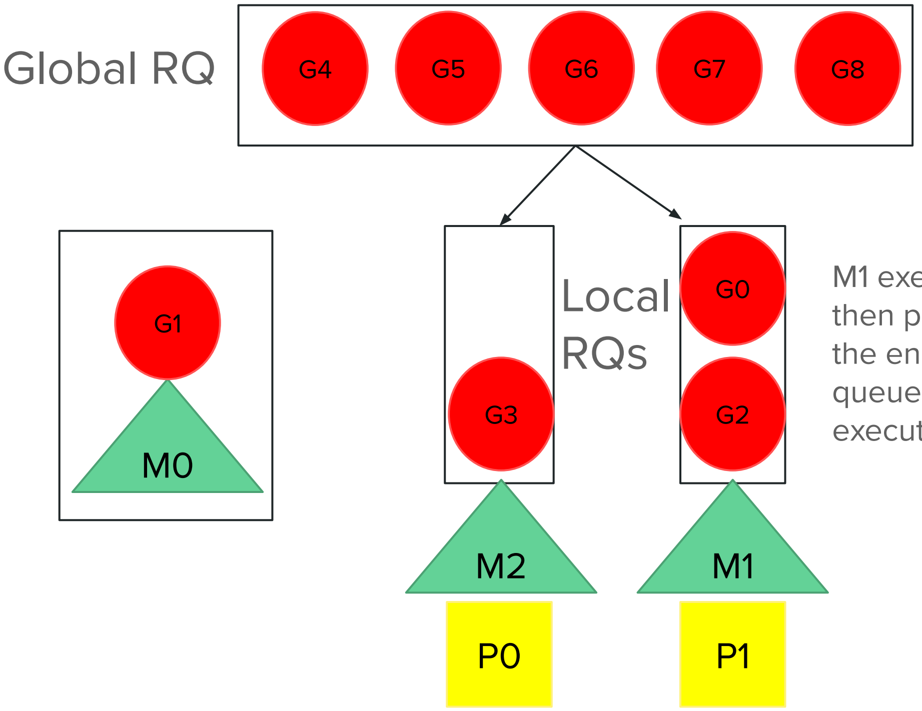
4. Read file syscall finishes. Go tries to do, in order:

1. acquire the exact same P (P1 in our example), if it's still in the idle queue, and resume the execution
2. acquire any P in the idle list and resume the execution
3. put the goroutine back into the Global Run Queue and put the M associated back into the idle threads list (a queue with idle M threads that will be used to recycle existing OS threads when a new thread is needed)

Run queues (FIFO)

- 1 global (shared) <u>run queue</u>
  Go runtime puts the new goroutines in this queue (e.g. after *go func {...}*)

- 1 local run queue for each processor P
  This reduces thread contention to access the queue and takes advantage of CPU caches



M1 executes G2, then put G2 at the end of the queue and executes G0

G1/M0 was running in P0. Goroutine called a syscall to open a file, which locked the whole thread. Go runtime moved the blocked thread/goroutine to idle state and created M2 to run the remaining goroutines in P0 local run queue

If we force the value of P (with GOMAXPROCS) to a number higher than the number of actual CPU cores, there will be more OS threads (M) running at the same time than the number of cores, and the OS scheduler will take care of them, since they're OS threads

Netpoller & Work Stealing

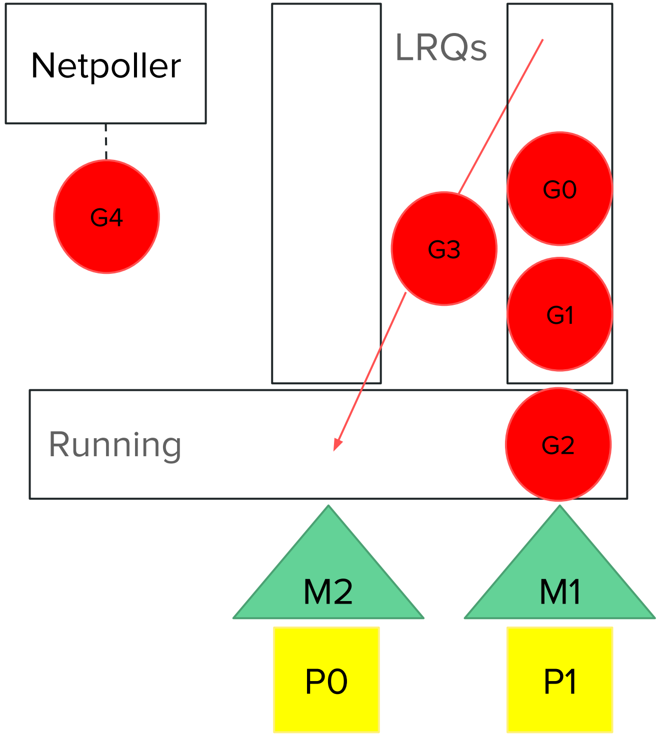
Implemented in runtime.findRunnable()

What happens when the scheduler has to choose another goroutine to execute:

- Check if it's necessary to perform Garbage Collection
- No GC Needed? *1 time every 61* check if the Global RQ contains any goroutine and, if there is, choose it for the execution (to avoid GRQ starvation if LRQ goroutines never finish)
- Check the Local Run Queue for the next goroutine
- Local Run Queue is empty? Check the Global Run Queue
- Poll Network (e.g. Linux's epoll syscall).
  When executing a network operation (e.g. TCP socket), instead of blocking the entire thread, the goroutine is moved to the netpoller's queue (part of Go runtime).
  In this step, the netpoller is run to see if any of the goroutines associated with it have received data, and if so, the target goroutine is moved back into its LRQ.
- Try to steal a goroutine from the local queue of another Processor (work stealing)

```
}

// Check the global runnable queue once in a while to ensure fairness.
// Otherwise two goroutines can completely occupy the local runqueue
// by constantly respawning each other.
if pp.schedtick%61 == 0 && sched.runqsize > 0 {
        lock(&sched.lock)
        gp := globrunqget(pp, 1)
        unlock(&sched.lock)
        if gp != nil {
                return gp, false, false
        }
}
```



Preemptive scheduling

The execution of a goroutine can stop anytime due to:

- I/O / syscall / call to runtime pkg function
- assigned preemption time slot finishes (10ms)

There is a separated OS thread (M), called sysmon (system monitor).
It's part of Go runtime, but it doesn't have an associated P, so it doesn't limit performance.

If the sysmon thread detects that a goroutine is still running after the time slot end, it sends a SIGURG signal to its thread, forcing the thread to pass control to the scheduler (before Go 1.14, released in 2020, the scheduler wasn't fully preemptive).
The scheduler dumps the program counter, registers and stack (like a context switch) and then executes another goroutine.

Benchmarks

Threads vs Goroutines in Go

GitHub Repository

runtime.LockOSThread()

“LockOSThread wires the calling goroutine to its current operating system thread. The calling goroutine will always execute in that thread, and no other goroutine will execute in it, until the calling goroutine has made as many calls to UnlockOSThread as to LockOSThread. If the calling goroutine exits without unlocking the thread, the thread will be terminated.”

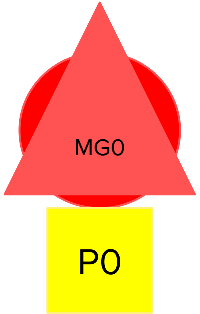
….

```
go func() {
    i++
}
```

vs

```
go func() {
    runtime.LockOSThread()
    i++
}
```

From now on, I will call it a Go “thread”.
This is a 1 : 1 mapping between a goroutine and an OS thread.



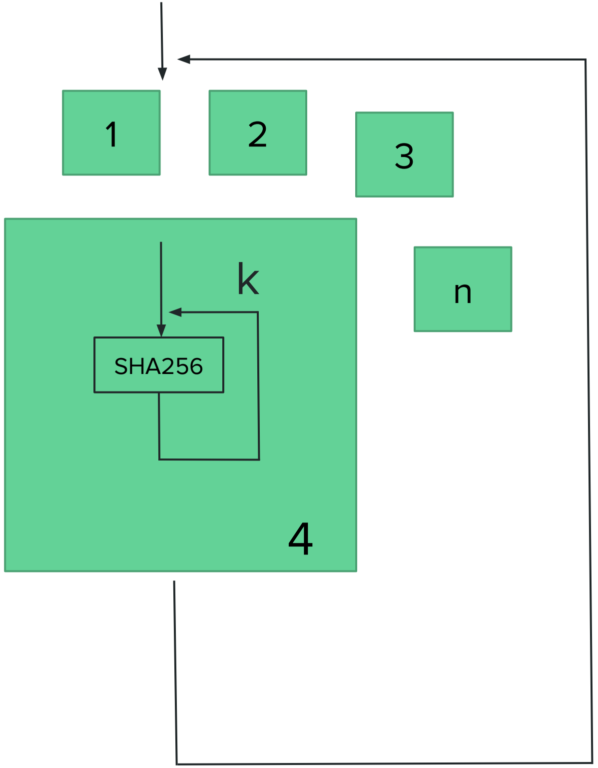
CPU Bound task

Goroutines perform better than I expected for CPU bound tasks, and in fact there isn't much overhead.

I tried different values of *n* (*8*, *32*, *100*) and derived some empirical relations. Here I haven't reported individual data, but you can run the benchmarks yourself using the GitHub repo.

- For lower values of k, goroutines beat threads by ~10-15%
- For the maximum tested value of k (100,000), the difference between the two is insignificant

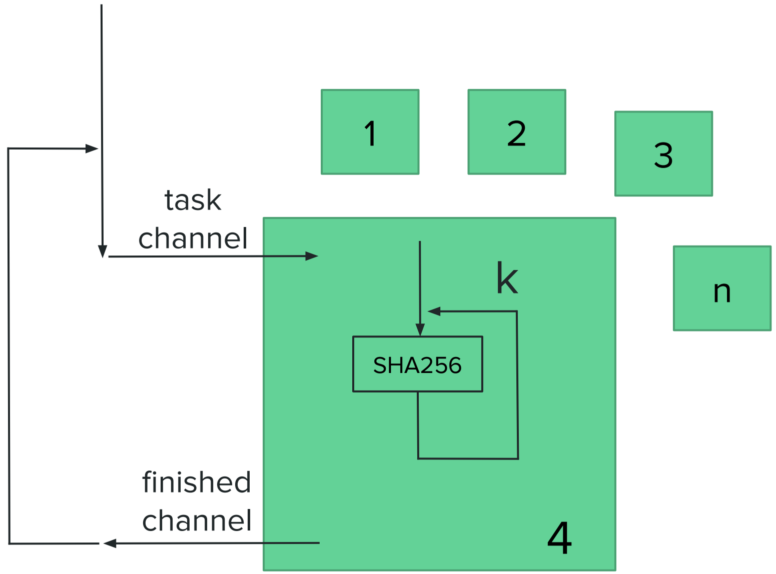
Threads aren't the right choice if we have to continuously spawn them, like we do with goroutines.



$0 < n \leq 100$

$10 < k \leq 100{,}000$

Recycle the threads



- Here we ignore the startup time by starting threads/goroutines before the start of the benchmark and keeping them open. We send the tasks using channels, and then wait for the response.
- After some empirical benchmarks on my 8-core laptop, I saw that as long as n is less than the number of CPU cores, threads beat goroutines (up to ~10%, the lower the n, the higher the advantage of threads)
- After the threshold, goroutines perform better: schedulers, I’m watching at you :D

Should I use threads in Go? (instead of goroutines)

NO

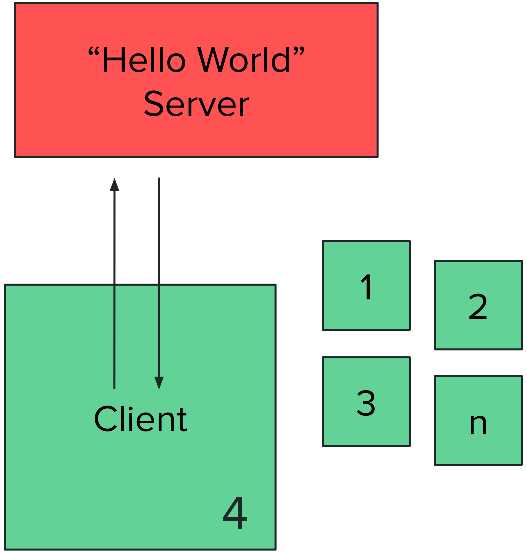
BUT

There are some exceptions where Go “threads” are needed (e.g. high performance networking using XDP)

- Go encourages the usage of goroutines, and you would lose of the most important features of the language if you lock the thread
- To gain some advantage, you need to keep the same threads open all the time
- In some (very) limited circumstances, threads may offer slightly better CPU performances, but memory usage would be generally higher

I/O Bound task

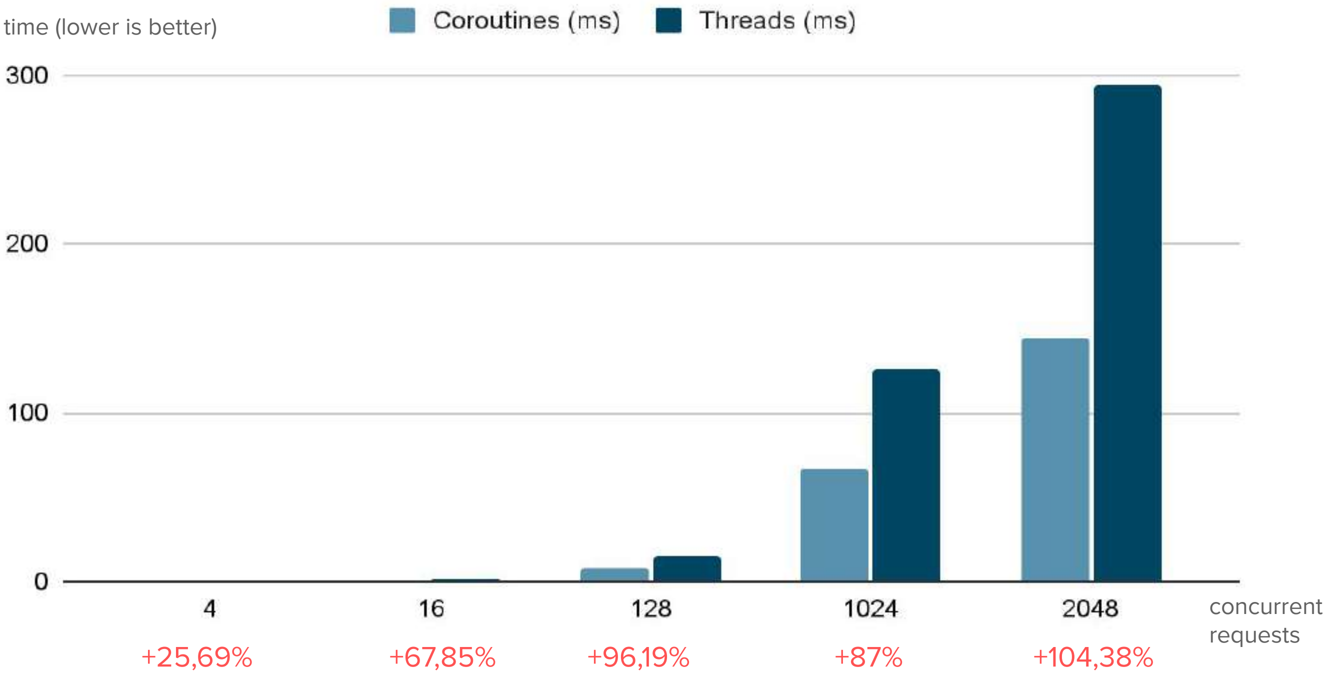
- For each request, we create a new HTTP client, independent of the others, to avoid HTTP caches and keep-alives, which could invalidate measurements
- The server runs in another process on the same local machine, to avoid sharing the coroutines scheduler between clients and server (which could invalidate measurements)
- Each goroutine/thread will run exactly one request, and then wait for the server response, and we keep measuring until all clients have received a response
- GOMAXPROCS is set to a value large enough to execute all threads concurrently (when we use LockOSThread)



	Coroutines (ms)	Threads (ms)	
4	0,3827824	0,4811004	25,69%
16	1,10243	1,8503964	67,85%
128	7,8042888	15,3110384	96,19%
1024	67,4873122	126,2007282	87,00%
2048	143,9963056	294,3022064	104,38%

Each value is the average of 5 measurements

Benchmark



Why not a thread pool in the I/O benchmark?

NO

BUT

There are some exceptions where goroutines pools are the standard (e.g. with persistent DB connections)

- Usually, when working with threads, you spawn a fixed amount of them and put them in a pool to reuse them later and reduce their overhead
- However, if you do this with I/O, you will have a limitation on the number of concurrent requests, since you must know in advance how many threads you want to have (e.g. if the pool contains 16 threads and you have to make 32 HTTP requests, it will take about twice the time compared to directly spawning 32 goroutines)
- It wouldn’t be a realistic benchmark if the pool contains all the necessary threads, because this isn’t replicable in real applications

Download the slides
of this presentation

Q&A time

Erik Pellizzon
https://erikpelli.pp.ua

Go Developer
Freelancer

LinkedIn

Looking for
collaborations