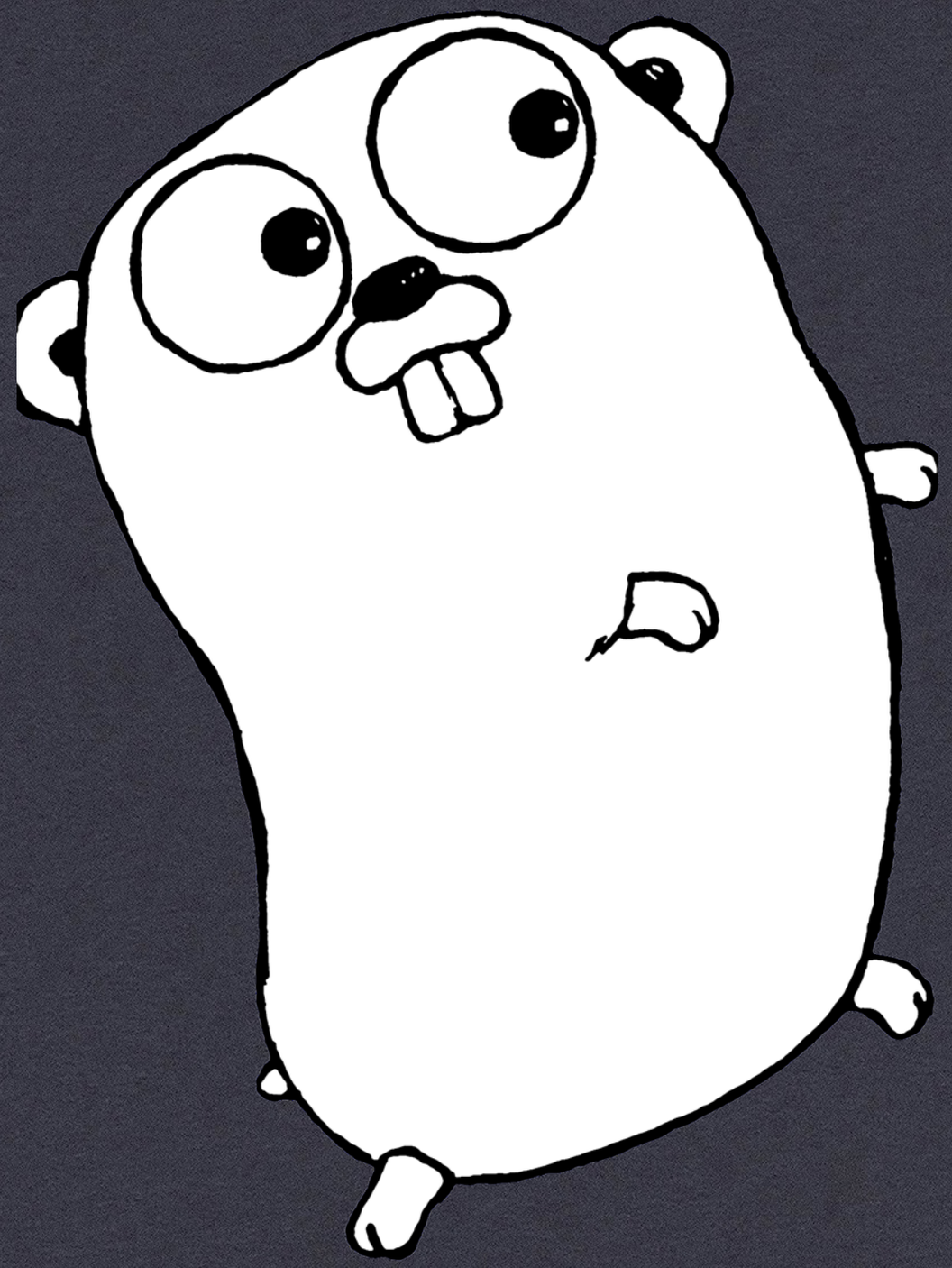
DEMYSTIFYING THE GO SCHEDULER

MATTHEW DALE



Go Concurrency Crash Course

- The atomic unit of concurrent work in Go is the goroutine
  - Started by invoking a function with the go keyword

```
go functionName(param)
```

  - Invoked function runs concurrently with the current thread of execution
- Goroutine synchronization is accomplished with channels
  - Message passing construct
  - Buffered or unbuffered

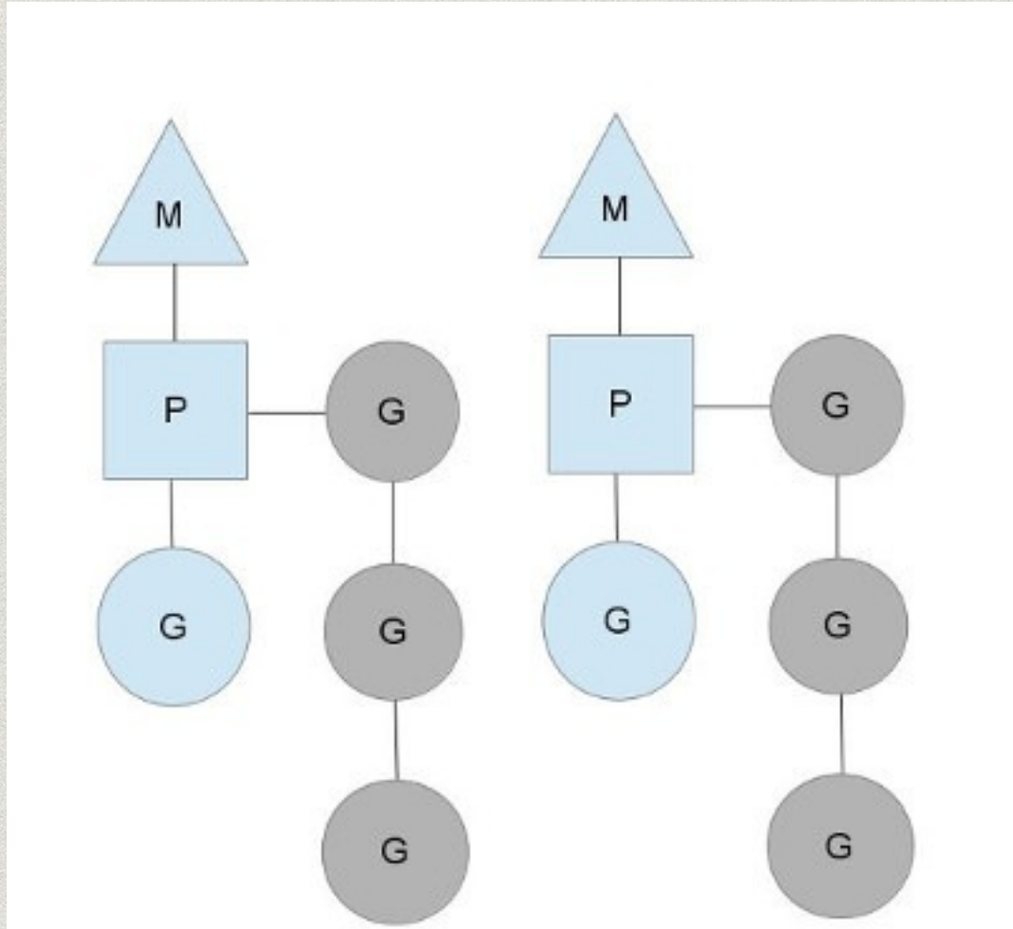
The Go Scheduler

- Attempts to efficiently schedule goroutines on available resources
- The scheduler will pick up a new goroutine when the current goroutine…

1. Finishes
2. Makes a blocking system call
   E.g. reading a file
3. Makes a blocking Go runtime call
   E.g. reading from a channel
4. Invokes another function (only happens sometimes)*

*As of Go 1.2 (https://golang.org/doc/go1.2#preemption)

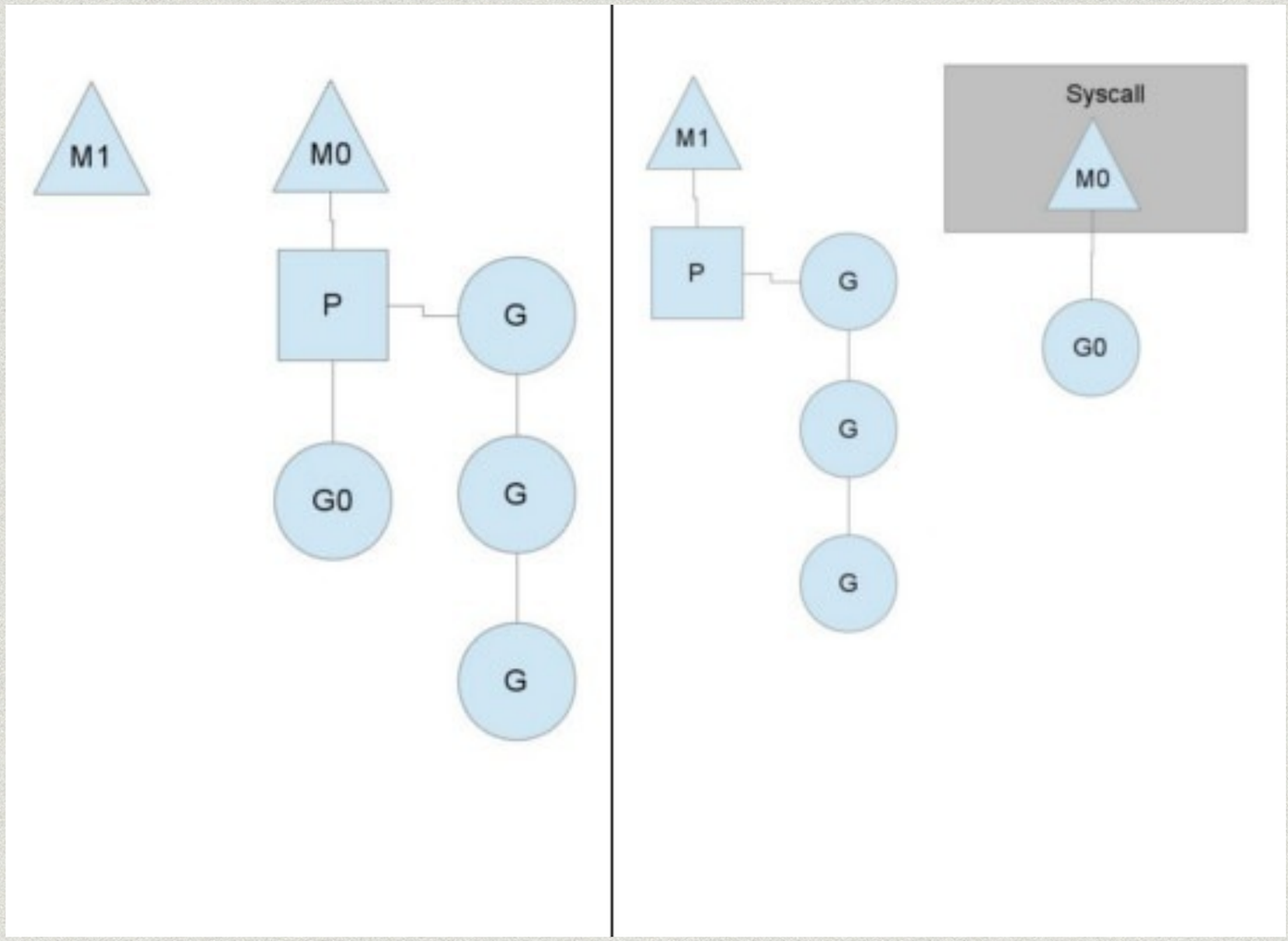
Taken from http://morsmachine.dk/go-scheduler



M - Machine (OS thread)
P - Context (Go scheduler)
G - Goroutine

- If the current goroutine is blocked on a system call…
  - The OS thread processing the goroutine idles waiting for the system call to return
  - The context that was scheduling gorotuines on the blocked thread is moved to a new thread (or a new one is created if none are available)

Taken from http://morsmachine.dk/go-scheduler



HOW DOES GO ASK FOR RESOURCES ABSTRACTED BY THE OPERATING SYSTEM?

The syscall Package

- All calls that can block an OS thread go through the syscall package (except cgo calls)
- Responsible for informing the Go runtime that a potentially blocking system call is about to happen

src/pkg/os/file.go, line 91

```
// Read reads up to len(b) bytes from the File.
// It returns the number of bytes read and an error, if any.
// EOF is signaled by a zero count with err set to io.EOF.
func (f *File) Read(b []byte) (n int, err error) {
    if f == nil {
        return 0, ErrInvalid
    }
    n, e := f.read(b)
    if n < 0 {
        n = 0
    }
    if n == 0 && len(b) > 0 && e == nil {
        return 0, io.EOF
    }
    if e != nil {
        err = &PathError{"read", f.name, e}
    }
    return n, err
}
```

src/pkg/os/file_unix.go, line 186

```
// read reads up to len(b) bytes from the File.
// It returns the number of bytes read and an error, if any.
func (f *File) read(b []byte) (n int, err error) {
    if needsMaxRW && len(b) > maxRW {
        b = b[:maxRW]
    }
    return syscall.Read(f.fd, b)
}
```

Let's follow a call to file.Read…

src/package/zsyscall_linux_amd64.go, line 831 (generated)

```go
func read(fd int, p []byte) (n int, err error) {
    var _p0 unsafe.Pointer
    if len(p) > 0 {
        _p0 = unsafe.Pointer(&p[0])
    } else {
        _p0 = unsafe.Pointer(&_zero)
    }
    r0, _, e1 := Syscall(SYS_READ, uintptr(fd), uintptr(_p0), uintptr(len(p)))
    n = int(r0)
    if e1 != 0 {
        err = e1
    }
    return
}
```

```
TEXT ·Syscall6(SB),NOSPLIT,$0-80
    CALL    runtime·entersyscall(SB)
    MOVQ    16(SP), DI
    MOVQ    24(SP), SI
    MOVQ    32(SP), DX
    MOVQ    40(SP), R10
    MOVQ    48(SP), R8
    MOVQ    56(SP), R9
    MOVQ    8(SP), AX   // syscall entry
    SYSCALL
    CMPQ    AX, $0xfffffffffffff001
    JLS ok6
    MOVQ    $-1, 64(SP) // r1
    MOVQ    $0, 72(SP)  // r2
    NEGQ    AX
    MOVQ    AX, 80(SP)  // errno
    CALL    runtime·exitsyscall(SB)
    RET
ok6:
    MOVQ    AX, 64(SP)  // r1
    MOVQ    DX, 72(SP)  // r2
    MOVQ    $0, 80(SP)  // errno
    CALL    runtime·exitsyscall(SB)
    RET
```

src/pkg/syscall/asm_linux_amd64.s, line 44

src/pkg/runtime/proc.c, line 1502

```
// The goroutine g is about to enter a system call.
// Record that it's not using the cpu anymore.
// This is called only from the go syscall library and cgocall,
// not from the low-level system calls used by the runtime.
//
// Entersyscall cannot split the stack: the runtime·gosave must
// make g->sched refer to the caller's stack segment, because
// entersyscall is going to return immediately after.
#pragma textflag NOSPLIT
void
runtime·reentersyscall(void *pc, uintptr sp)
{
    // Disable preemption because during this function g is in Gsyscall status,
    // but can have inconsistent g->sched, do not let GC observe it.
    m->locks++;

    // Leave SP around for GC and traceback.
    save(pc, sp);
    g->syscallsp = g->sched.sp;
    g->syscallpc = g->sched.pc;
    g->syscallstack = g->stackbase;
    g->syscallguard = g->stackguard;
    g->status = Gsyscall;
    if(g->syscallsp < g->syscallguard-StackGuard || g->syscallstack < g->syscallsp) {
        // runtime·printf("entersyscall inconsistent %p [%p,%p]\n",
        //  g->syscallsp, g->syscallguard-StackGuard, g->syscallstack);
        runtime·throw("entersyscall");
    }

    if(runtime·atomicload(&runtime·sched.sysmonwait)) {  // TODO: fast atomic
        runtime·lock(&runtime·sched);
        if(runtime·atomicload(&runtime·sched.sysmonwait)) {
            runtime·atomicstore(&runtime·sched.sysmonwait, 0);
            runtime·notewakeup(&runtime·sched.sysmonnote);
        }
        runtime·unlock(&runtime·sched);
        save(pc, sp);
    }
```

Notes on Host Setup

- Modify **/etc/security/limits.conf**
  - Go will quickly use up the default allotment of file descriptors on most Linux distros
  - The default Go HTTP server will panic if listener.Accept() ever returns an error, killing your service
  - Example configuration

```
ec2-user        soft    nofile  100000
ec2-user        hard    nofile  100000
root            soft    nofile  100000
root            hard    nofile  100000
```

FILE I/O DEMO!

BUT HOW DOES THAT WORK WITH NETWORK I/O?

WOULDN’T THAT JUST CREATE A BILLION THREADS?

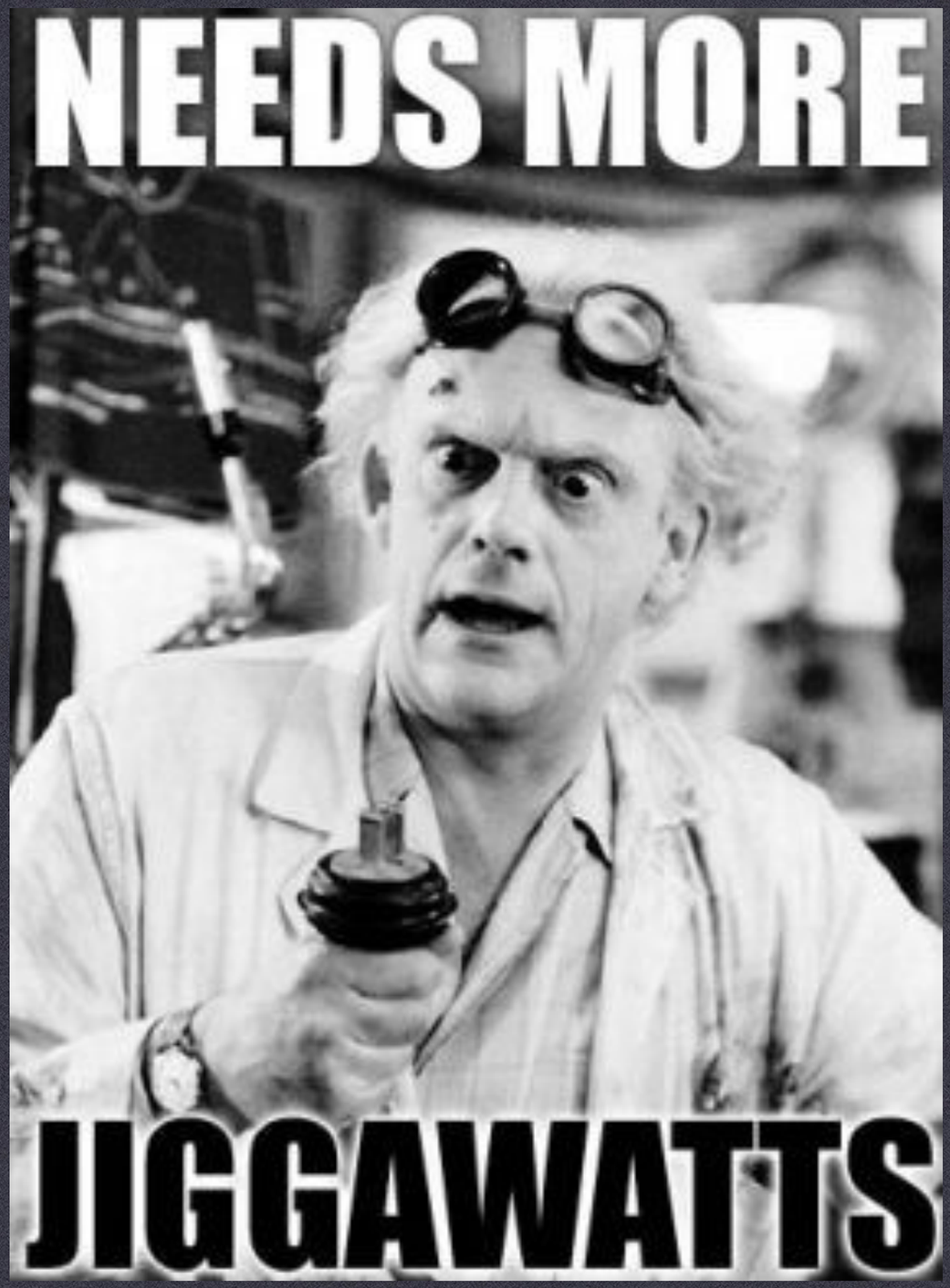
The netpoller

- Runs in its own thread and polls the OS's asynchronous I/O network interface for data
- Goroutines asking for network I/O block waiting for the netpoller to give them data, not for a system event
- Go treats Unix sockets and network connection file descriptors the same, so Unix socket I/O will not block OS threads
- See http://morsmachine.dk/netpoller for a great, concise description of the netpoller

RESOURCE STARVATION OR WHY DID MY GO SERVICE STOP RESPONDING?

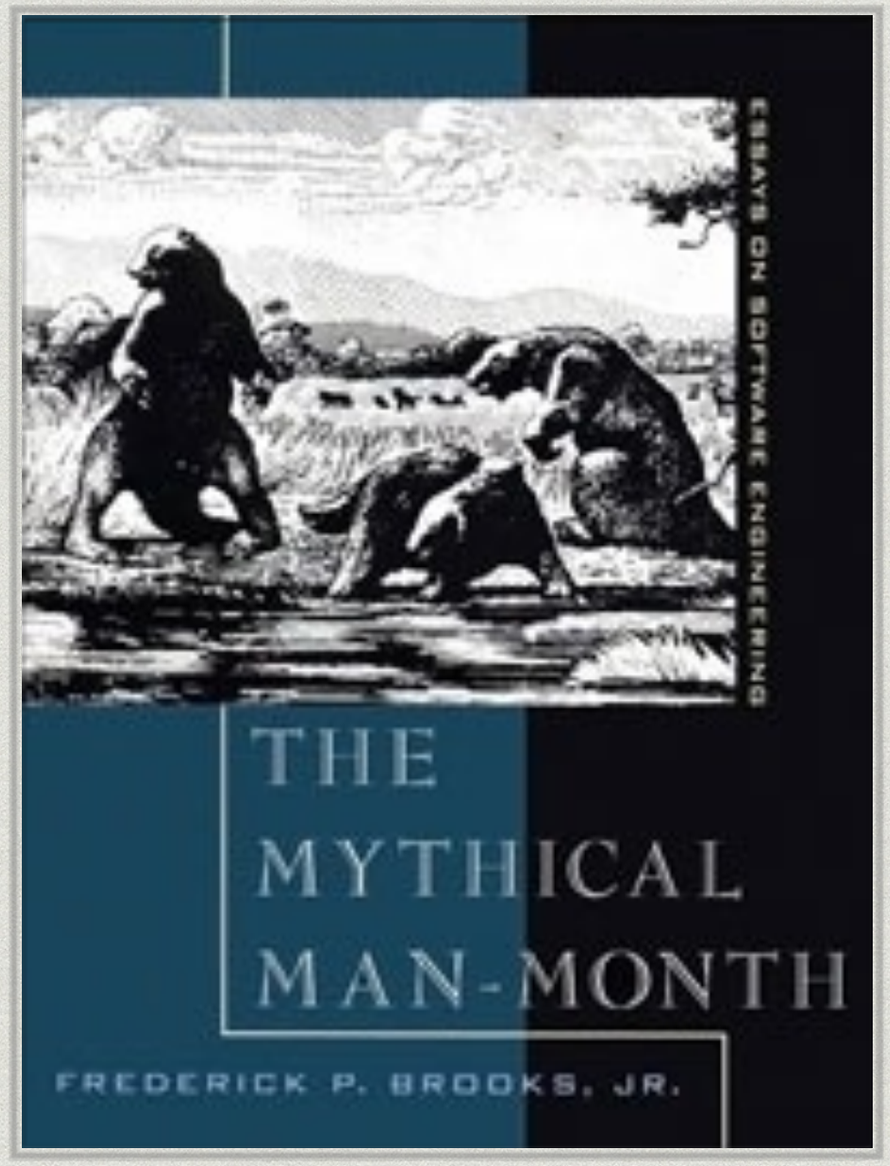
WEB SERVICE PERFORMANCE DEMO!

NOT SO GOOD…



Which Problems Are Essential vs Accidental*?

- Essential
  - A computer can do a finite amount of work per unit time
  - If there is more work than resources, then work must be delegated, queued or dropped
- Accidental
  - Sometimes you and the Go scheduler disagree about what is the most important work to do



**Essential vs accidental complexity is a problem decomposition strategy proposed in Frederick Brooks Jr.'s book* The Mythical Man-Month

Simplify vs Complexify

- Simplify - synchronous instead of concurrent
  - Still requires determining how to do CPU load balancing
    - Pushes performance problems to the front, letting the caller tune on their side
- Complexify - refactor into micro services and add queues
  - Lots of additional code just to solve a “simple” problem

Think bite-sized

- Refactor long-running, non-blocking workloads to add more function calls
  - The Go scheduler will sometimes preempt the running goroutine when it makes a non-inlineable function call
- Explicitly call runtime.Gosched
  - Causes the current running goroutine to yield to the scheduler

Think micro services

- Keep heterogeneous workloads in different Go processes
  - One service handles quick or I/O heavy tasks
  - One service handles long-running
  - OS handles resource scheduling

runtime.LockOSThread

- Locks the current goroutine to the current thread and doesn't allow any other goroutines to run on that thread
- Call **runtime.UnlockOSThread** to unlock the thread.

Extra Go Scheduler Topics

- There are a lot of topics relevant to the Go scheduler but beyond the scope of this presentation, including…
  - Tailoring GOMAXPROCS for your workload
  - How cgo calls interact with the scheduler

Sources and Suggested Reading

- http://morsmachine.dk/go-scheduler
- http://morsmachine.dk/netpoller
- http://dave.cheney.net/2014/06/07/five-things-that-make-go-fast
- http://golang.org/pkg/runtime/
- Code examples used in this presentation can be found at https://github.com/matthewdale/GoSchedulerDemo

QUESTIONS?