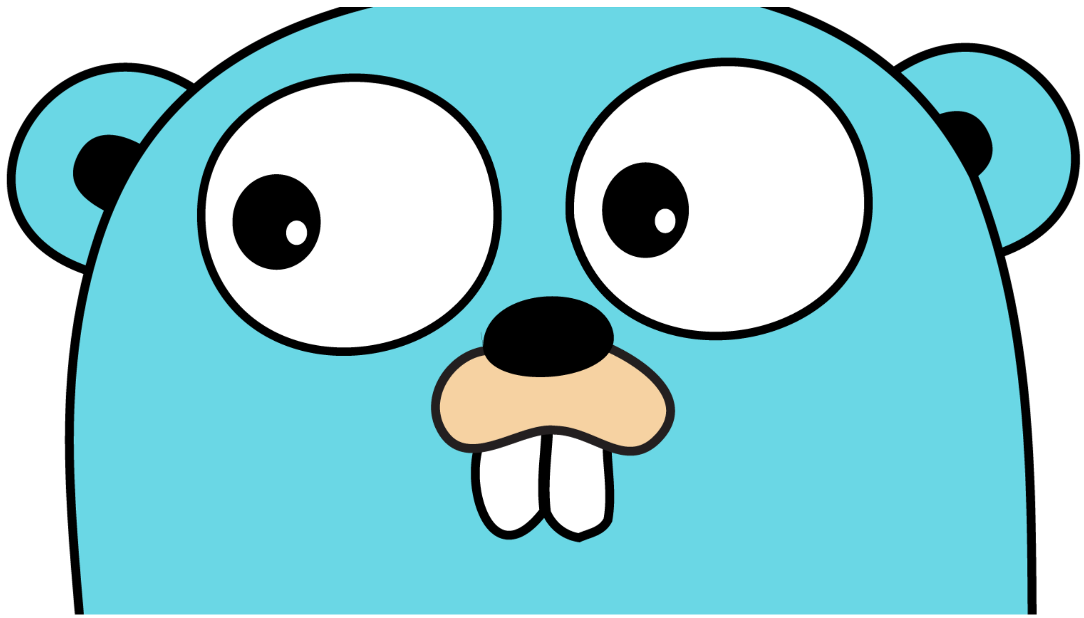


Go`ing, go`ing.. gone!

Some bits on the Go runtime

June 5th, 2014 for the GoTO meetup group

Peter Kieltyka / peter@pressly.com

Personal exploration

- Writing backend software for the last 10 years
- Started with writing small UNIX network apps in C
- Wrote web apps with PHP for a few years, discovered and fell in <3 with Ruby, tried to make it fast for years after that
- Tried Node, Scala..
- Found Go

Go’s goal: to make writing concurrent server software easy, safe and fun

Go’s concurrency model

- Inspired by Tony Hoare’s CSP (Communicating Sequential Processes)
- Message passing vs. shared state
- The concurrency primitives offered by Go:
  - goroutines & channels
- No “thread” api for application developers; thats all under the hood (runtime)

The two important pieces for fast network applications

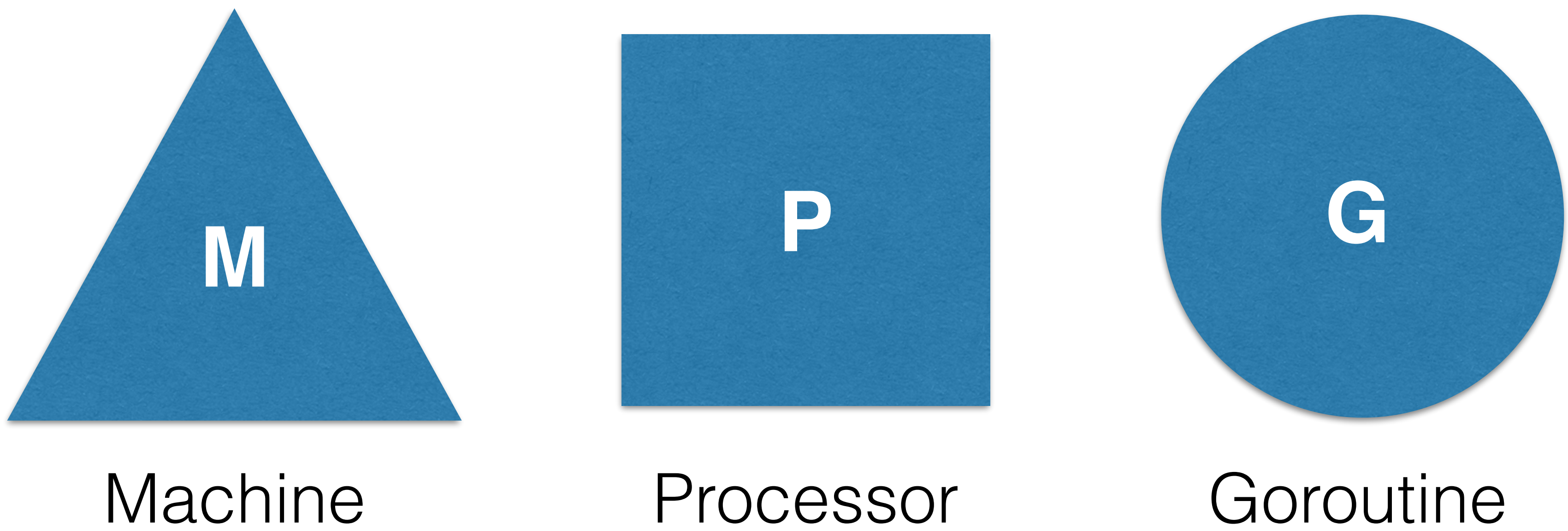
		managed by..
Threading model	→	Scheduler (src/pkg/runtime/proc.c)
Network I/O	→	Netpoller (src/pkg/runtime/netpoll*.goc)

Multithreading models..

1:1	N:1	M:N
Kernel-level threading	User-level threading	Hybrid threading
OS threads	Green threads	OS threads + green threads
OS Scheduler	App runtime scheduler	OS / App runtime scheduler
Threads are expensive (initial size, 1 to 8 mb per thread)	Fast context switching, but, what about the other cores?	Difficult to implement, balance of language syntax / runtime

Go’s Scheduler

The scheduler's job is to distribute ready-to-run goroutines over worker threads. The main pieces:



G Goroutine

```
struct  G {
    int64       goid;         // Go id
    uintptr     stackbase;    // The stack
    uintptr     stack0;       // Stack pointer
    uintptr     stacksize;    // Stack size
    int16       status;       // Current state in runtime
    …
} // src/pkg/runtime/runtime.h
```

- The `go fn()` we all know and love; spawn a coroutine to later execute ‘fn’. Puts a G on the local run queue.
- 8KB per goroutine on initial size (Go 1.2 and 1.3)
- Stack grows and shrinks as necessary
- main() is also a G
- Status (runtime.h): Idle, Runnable, Running, Syscall, Waiting, Dead
- Executes on an **M**achine in the context of a **P**rocessor

M Machine

```
struct  M {
    int32   id;             // Machine id
    bool    spinning;       // M is out of work and is actively looking for work
    bool    blocked;        // M is blocked
    int32   gcing;          // M is garbage collecting
    G*      curg;           // Current running goroutine
    P*      p;              // Attached P for executing Go code (nil if not executing Go code)
    …
} // src/pkg/runtime/runtime.h
```

- Managed OS Thread for executing Go code
- Start/stop new M's (threads), as necessary, managed by the runtime not the developer
- A thread cache for efficiency
- Special rules for when blocking on a Syscall or executing a CGO function

P

Processor

```
struct  P {
    int32       id;          // Processor id
    uint32      status;      // Current state: Idle, Running, Syscall, Gcstop, Dead
    M*          m;           // Current associated Machine
    G*          runq[256];   // Goroutines queue
    …
} // src/pkg/runtime/runtime.h
```

- Exactly `GOMAXPROCS` P’s
- P schedules Goroutines to run on an attached M in FIFO order
- Context for scheduling G’s on M’s, must attach to an M to execute a new G (code)
- P has local run queue of Goroutines, as well periodically checks a global queue of G’s
- and, when a P’s go run queue is empty, it will steal work from other busy P’s

Peak under the hood

ex.go

```
package main

import (
	"fmt"
	"time"
)

func main() {
	fmt.Println("hey hey, my main G")
	go myFn("hey, wait up")
	time.Sleep(10e6) // give it a sec..
}

func myFn(s string) {
	fmt.Println(s)
}
```

./ex (from proc.c)

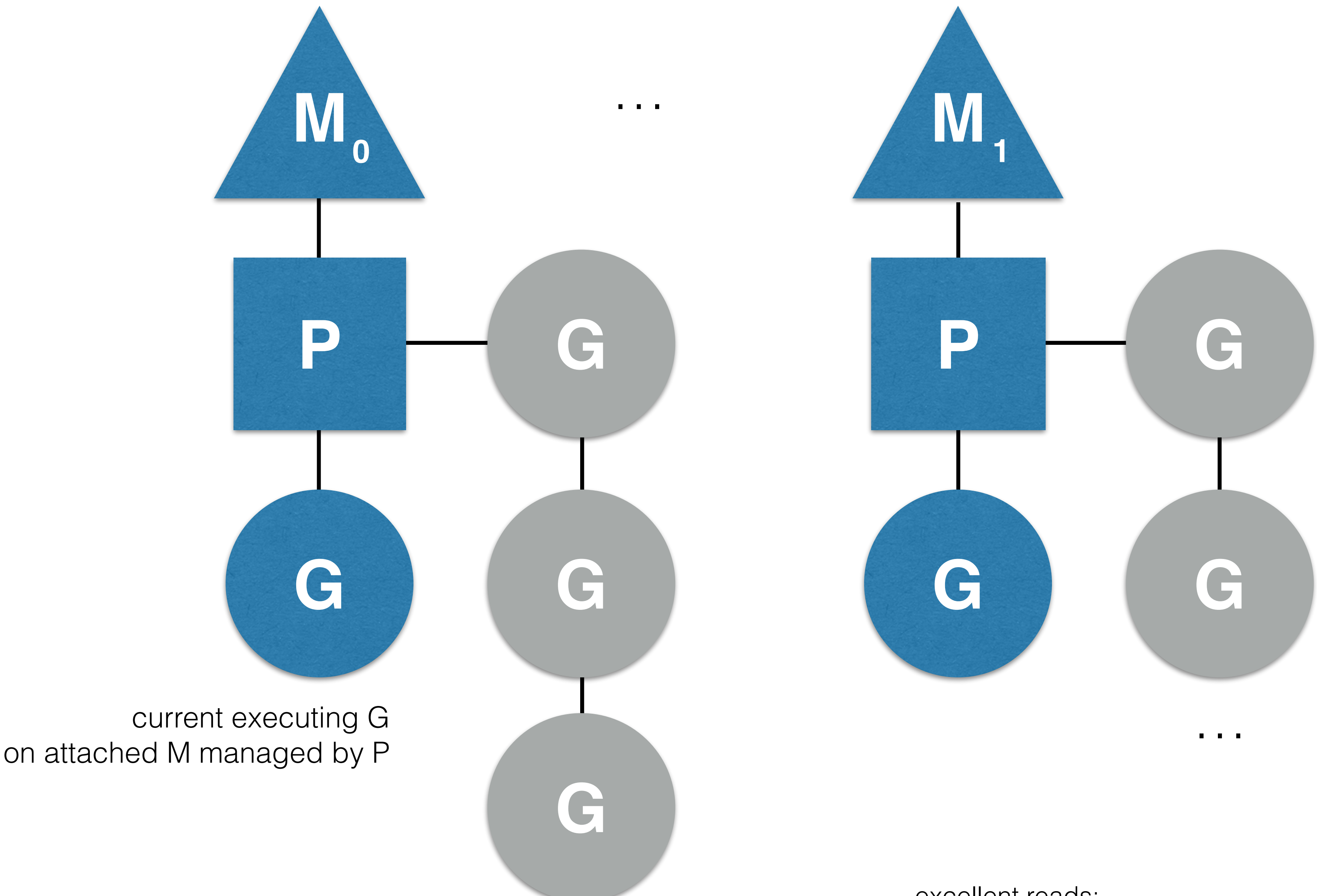
Bootstrap:

- runtime·args()
- runtime·osinit()
    - Set runtime·ncpu
- runtime·schedinit()
    - Set max M count (10k default)
    - Set args / env vars
    - Init first P
- runtime·newproc(&mainFn())
    - Make a new G and queue it
    - Effectively calling `go main()`
- runtime·mstart()
    - Init a new M
    - Find a new G to run
    - Execute the G

Exec main G:

- fmt.Println(..)
- runtime·newproc(&myFn())
    - Make a new G for call to myFn and queue it
- time.Sleep(..)

Scheduler visual; GOMAXPROCS=2



waiting G's
...

excellent reads:
http://morsmachine.dk/go-scheduler
http://golang.org/s/go11sched

Here’s whats really interesting (IMO)..

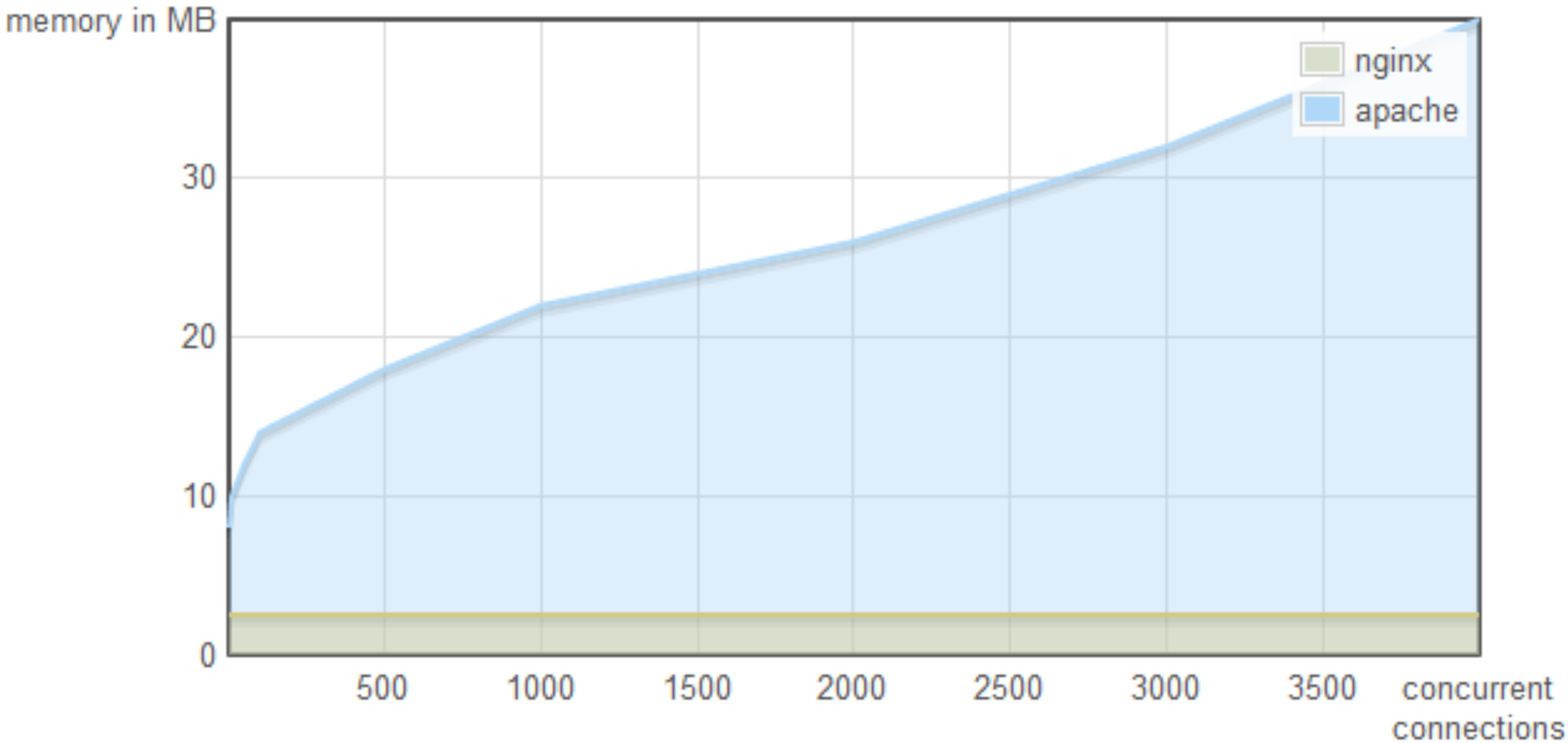
- So, the scheduler abstracts threads with Goroutines, and manages their execution across multiple cores

- On a network operation, a Goroutine will block, and be put back on the run queue until its ready

- Blocking..? how is it so efficient and fast?

- Go actually uses an evented IO under the hood whenever performing network operations.. just like Nodejs, Scala, etc.

A real world example of evented IO kicking butt



*Google it

Netpoller

- Network IO interface to kqueue, epoll, IOCompletionPort
- Kernel-level stateful file descriptor monitoring and event notification of readiness and completion
- As opposed to using select()/poll() which have to check each fd each iteration
  - With thousands of operations, this becomes expensive

Example

```
//…package, imports..
func main() {
	go func() {
		for {
			time.Sleep(10 * time.Millisecond)
			fmt.Println("tick..")
		}
	}()

	var wg sync.WaitGroup
	wg.Add(1)
	go func() {
		resp, _ := http.Get("http://www.google.ca/")
		defer resp.Body.Close()
		fmt.Println(resp.Status)
		wg.Done()
	}()

	wg.Wait()
}
```

Block G on network io operations;
put the G back on the local runq and set
g->status = waiting

The beauty of Go to me is in the design of language (syntax), the threading model and evented network IO

Writing speedy servers has never been easier.

PRESSLY

Thanks!

peter@pressly.com | @peterk