Peering behind the curtain: Golang, Threads, and You

Casey Callendrello

CoreOS

This presentation:
http://bit.ly/2zr9crF

Who am I?

github.com/squeed
@squeed

CNI maintainer
Rkt developer
K8s contributor

Is golang a “systems programming” language?

What is a “systems programming” language?

Not going to answer either of these questions.

Mostly because I’ve done similar things in PHP.

There are some systems-level things that are easier in other languages.

Systems? Programming?

For this talk, let's define systems programming:

Any task where you call a syscall (or a thin wrapper) directly, rather than a higher-level abstraction.

A Disclaimer

- This is a potentially huge topic.
- I am really, really going to try and be correct.

Some Encouragement?

- POSIX is weird, Linux is weird, glibc is weird
  - “It Just Is”
- We learn these quirks by making mistakes and fixing bugs.
- Don’t feel discouraged that you don’t know everything.
- We all have lots to learn.

The Golang Scheduler

The Golang scheduler

Let’s understand what happens:

```
go func(){
    fmt.Println(“hello”)
  }()
```

The Golang Scheduler - Terminology

Three entities:

- **G**: goroutines
- **P**: “processors” - internal scheduler object
- **M**: “machines” - actual OS threads

The Golang scheduler

Goal: efficiently assign **G**'s to **M**'s

How: introduce **P**'s as abstraction layer

- Assign **G** to **P**, **M** to **P**
- At most **P** running **M**'s

P's are the scheduling "nexus"

- Each **P** has a runqueue

The Golang scheduler

What happens when this code runs?

```
go func(){
    fmt.Println(“hello”)
  }()
```

The Golang scheduler

1. A new **G** is created and added to a runqueue
2. Eventually it is assigned to a **P** and handed to its **M** to execute
3. The **M** executes the code… and promptly blocks on the `write()` syscall!

The Golang scheduler

- An **M** and **G** blocked on a syscall are parked
  - The **P** gets a new **M** and keeps working
  - This **M** is either created or grabbed from a pool
- Eventually, the **M** is unblocked
  - It tries to grab a **P**
  - Otherwise:
    - The **G** is put on the global runqueue
    - The **M** is put in the free pool

The Golang scheduler

Putting it all together:

Goroutines are not tied to threads…

Though they sometimes seem to be.

The Golang scheduler

One more thing:

Setting **GOMAXPROCS** controls the number of **P**'s, not the number of **M**'s. There will always be more **M**'s.

The Golang scheduler

A bonus question: what happens when this code runs?

```
runtime.GOMAXPROCS(1)
go func(){ for {x++} }()
do_work()
```

The Golang scheduler

A bonus question: what happens when this code runs? What's the difference?

```
runtime.GOMAXPROCS(1)
go func(){ for {fmt.Println(“”) }
}()
do_work()
```

Threads in Linux

Threads in Linux

Let’s consider some properties a process has:

- Environment variables
- Working directory
- User ID / Group ID
- Open files
- Priority

Threads in Linux

- Some properties are per-process
- Most properties are per-thread
- You just have to look this up

Threads in Linux: Properties

Per-Process:

- Working dir
- Root (chroot)
- Libraries (dlopen)
- File descriptors
- Memory mappings
- Signal handlers

Per-thread:

- Signal mask
- User and group id
- Namespaces
- Priority (nice)
- Capabilities
- Lots more...

Threads in Linux

- Threads are created from the `clone()` syscall.
- `clone()` duplicates the calling thread's properties exactly.
- We'll examine the implications of this later.

Threads in Linux

What might we want to do?

- Drop capabilities after startup
- `accept()`, switch to the `nobody` user
- Set a task to high / low priority
- Configure a container's network
- Add to a container's mountpoints

On to Golang

A rule of thumb:

If there isn’t a golang standard-library function for it, and you have to call the syscall directly, it’s probably a per-thread property.

Careful!

Some pretty bad, nondeterministic things can happen when **M**'s are non-homogenous.

Consider not knowing which user will own created files.

We currently have one tool in our toolbelt:

```
runtime.LockOSThread()
runtime.UnlockOSThread()
```

LockOSThread

- Pins the calling **G** to its **M** (thread)
- Keeps other **G**'s off of this **M**

No more, no less.

LockOSThread

Typical usage:

1. Lock
2. Enter container network namespace
3. Change some container-specific network setting
4. Go back to host network namespace
5. Unlock

LockOSThread

Problem #1: The Cleanup Problem

- The **M** will be reused once the **G** is done
- You need to carefully undo every change you made.
- Call `UnlockOSThread` at your peril!

LockOSThread

Solution #1: The “decorator”

```
fn (n *NS) Do(f func() error) error {
    runtime.LockOSThread()
    n.Enter()
    err := f()
    n.Exit()
    runtime.UnlockOSThread()
    return err
}
```

LockOSThread

Solution #1: The “decorator”

- Ensures the thread will be cleaned up.
  - panic / recover breaks this
  - As long as ns.Exit() doesn’t fail...
- Go 1.10 improves this:
  - Lock / Unlock is now nested.
  - If a locked **G** terminates, kill its **M**.

LockOSThread

Problem #2: The new thread

- Recall that new threads are `cloned`
- If the scheduler decides it needs a new **M**, you don't know which one it will `clone` from
  - GOMAXPROCS doesn't prevent thread creation. (**P** vs **M)**

LockOSThread

Solution #2: Cry, wait for 1.10

- CL 46044 - “Don’t start new threads from locked threads”
  - Keeps a “golden” thread around just to clone from
- Otherwise completely unsolvable!

LockOSThread

Problem #3: The Main Thread?

Some C libraries (e.g. SDL) need to be called from the main (first) thread.

LockOSThread

Solution #3: The Main Thread!

- Call `LockOSThread` in the `init` block
  - This locks the first **G** to the OS's main thread.
- Set up a "command chan"
  - `make(chan func())`
- Only consume it in `main()`

Threads in Golang

What might we want to do?

- Drop capabilities after startup
- `accept()`, switch to the `nobody` user
- Set a task to high / low priority
- Configure a container's network
- Add to a container's mountpoints

Threads in Golang

Let's review:

- Drop capabilities after startup
- `accept()`, switch to the `nobody` user
- ~~Set a task to high / low priority~~
- ~~Configure a container's network~~
- Add to a container's mountpoints

Threads in Golang

Not currently feasible without cgo:

- Drop capabilities after startup
- `accept()`, switch to the `nobody` user
- Add to a container's mountpoints

Bonus: crazy cgo tricks

```
// +build linux,!gccgo

package nsenter

/*
#cgo CFLAGS: -Wall
extern void nsexec();
void __attribute__((constructor)) init(void) {
        nsexec();
}
*/
import "C"
```

References

Scheduler:

- https://rakyll.org/scheduler/
- https://morsmachine.dk/go-scheduler

Nsenter (crazy cgo tricks):

- https://github.com/opencontainers/runc/tree/v1.0.0-rc4/libcontainer/nsenter

Namespaces in operation:

- https://lwn.net/Articles/532748/ and 3 more!

CNI network namespace wrapper:

- https://github.com/containernetworking/plugins/blob/master/pkg/ns/ns_linux.go#L254

CoreOS is running the world's containers

We're hiring (in Berlin!): careers@coreos.com

OPEN SOURCE

90+ Projects on GitHub, 1,000+ Contributors

container linux

rkt etcd

coreos.com

ENTERPRISE

Support plans, training and more

TECTONIC

QUAY

sales@coreos.com

Thanks!

QUESTIONS?

casey.callendrello@coreos.com

@squeed

github.com/squeed

LONGER CHAT?

Slack & IRC

More events: coreos.com/community

We're hiring: coreos.com/careers