Using unsafe.Pointer to explore Linux system calls

Matt Layher

GoCon Canada, May 31, 2019

Matt Layher

- GitHub, Twitter: **@mdlayher**
- github.com/mdlayher/talks

fastly®

unsafe

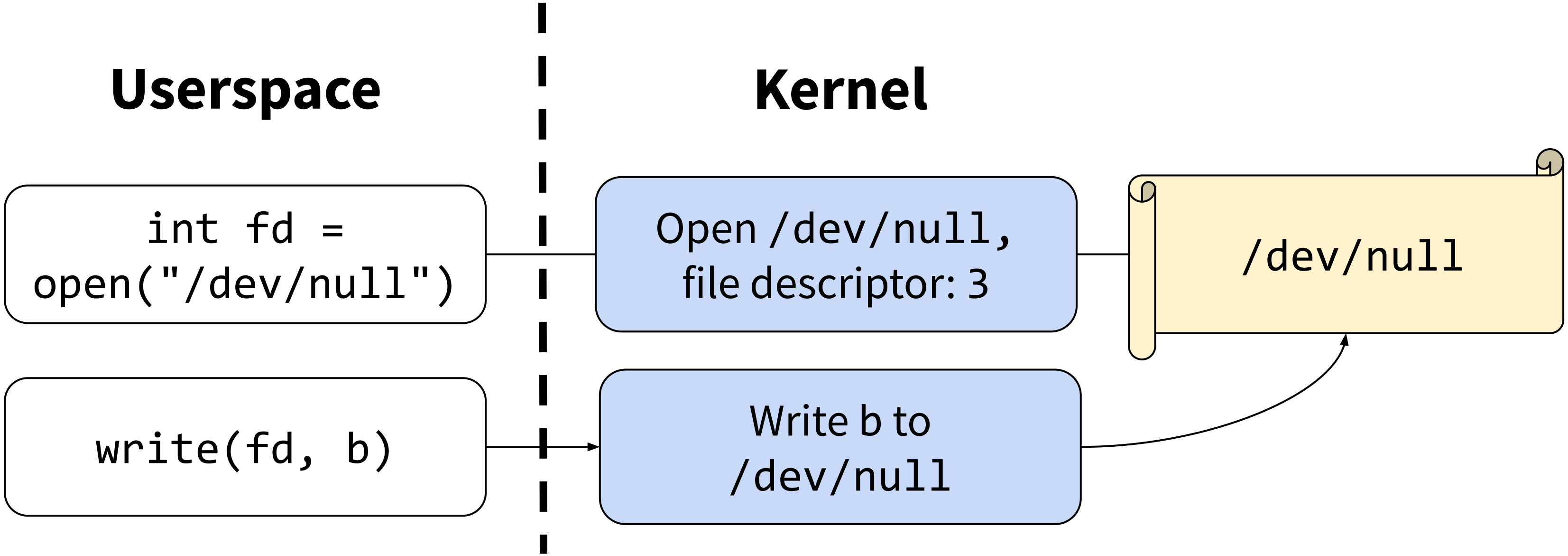
Package `unsafe` contains operations that step around the type safety of Go programs.

`unsafe` and Linux system calls

- Package `unsafe` and Linux **system calls** go hand-in-hand
- Sometimes an escape hatch from Go's type system is necessary
- **Read the rules before you write any `unsafe` code**
  - golang.org/pkg/unsafe/#Pointer
  - `go vet` can catch some mistakes, but don't rely on it

What is a system call?

- A “function call” into the Linux kernel, used to access to files, hardware, etc.



`ioctl(2)` on Linux

- Short for “I/O control”
- “a catch-all for operations that don't cleanly fit the UNIX stream I/O model”
    - UNIX’s “everything is a file” model: `read(2)` and `write(2)`
- Primarily used to pass data structures between the kernel and userspace

Retrieving data with `ioctl(2)` in C

- Suppose we want to retrieve the VM sockets context ID for our system

```c
int fd = open("/dev/vsock", O_RDONLY);
if (fd < 0) {
    perror("failed to open file");
}

uint32_t cid;
if (ioctl(fd, IOCTL_VM_SOCKETS_GET_LOCAL_CID, &cid) < 0) {
    perror("failed to get local CID");
}
printf("CID: %d\n", cid);
```

How can we replicate this program in Go?

Prerequisites

- Discuss type safety, memory layout, and endianness
- Introduce `unsafe` and explain its use cases
- Establish some guidelines on how to make **safe** use of `unsafe`

What is type safety?

- **Type safety** provides safeguards for the programmer, preventing mistakes
- Go is **statically typed**: the compiler checks data types and enforces type safety

```
fmt.Println(1 + "abc")
// BAD: cannot convert "abc" (type untyped string) to type int
// BAD: invalid operation: 1 + "abc" (mismatched types int and
string)

fmt.Println(strconv.Itoa(1) + "abc")
// OK: 1abc
```

Memory layout of Go types

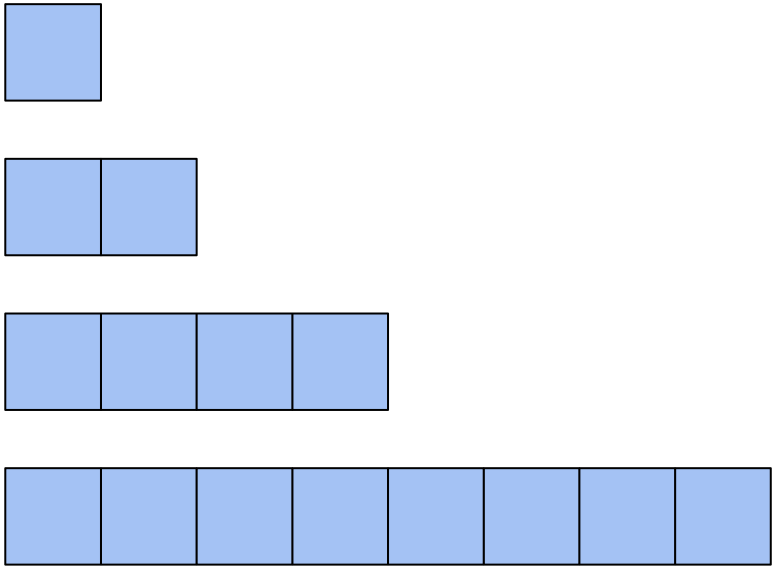
- `intN`/`uintN` family are fixed size integers, 8 bits per byte

`int8, uint8`

`int16, uint16`

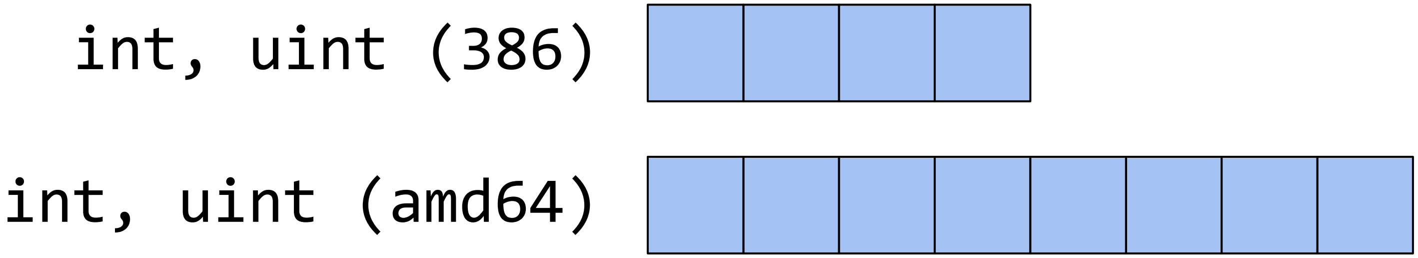
`int32, uint32`

`int64, uint64`



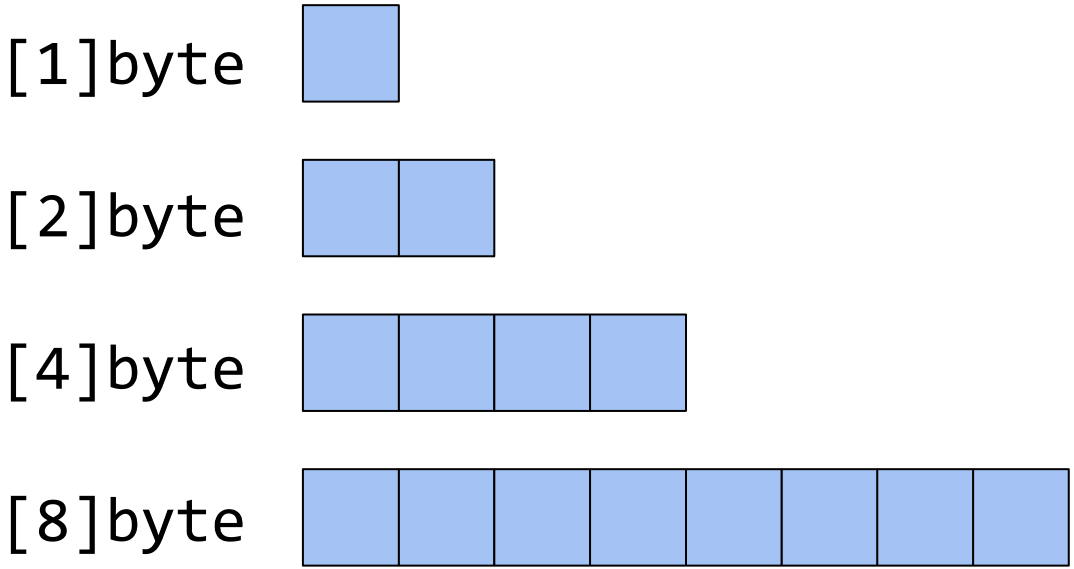
Memory layout of Go types

- `int`/`uint` size varies by CPU architecture; **avoid them** when using `unsafe`



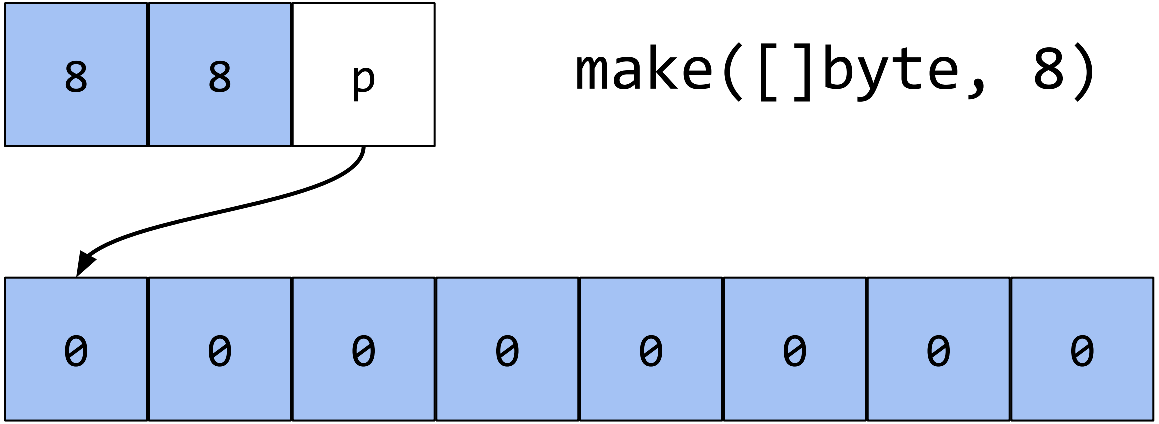
Memory layout of Go types

- The same rules apply for **arrays**, but what about **slices**?



Memory layout of Go types

- Slices must be handled carefully with unsafe code; **remember the slice header**!

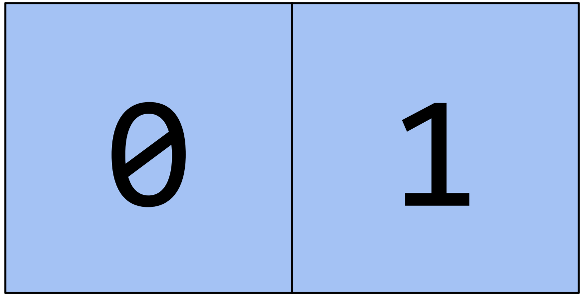


blog.golang.org/go-slices-usage-and-internals

What is endianness?

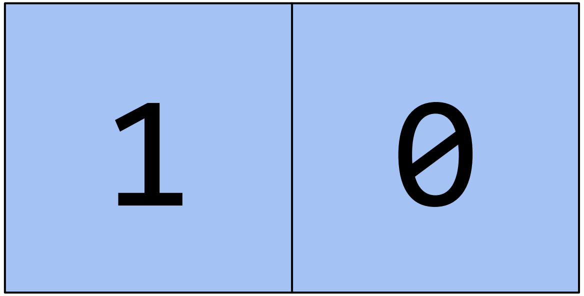
- The way a particular CPU lays out values in memory (also called **byte order**)

`uint16(1)`: **big endian**



Network byte order

`uint16(1)`: **little endian**



x86 CPUs (Intel, AMD)

What is endianness?

- You can store integers as either big or little endian with `encoding/binary`

```
v := uint16(1)
big := make([]byte, 2)
little := make([]byte, 2)

binary.BigEndian.PutUint16(big, v)
binary.LittleEndian.PutUint16(little, v)

fmt.Println(big, little)
// OK: [0 1] [1 0]
```

unsafe

```
$ go doc unsafe
package unsafe // import "unsafe"

Package unsafe contains operations that step around the
type safety of Go programs.

Packages that import unsafe may be non-portable and are
not protected by the Go 1 compatibility guidelines.

func Alignof(x ArbitraryType) uintptr
func Offsetof(x ArbitraryType) uintptr
func Sizeof(x ArbitraryType) uintptr
type ArbitraryType int
type Pointer *ArbitraryType
```

`unsafe.Sizeof()`

`unsafe.Sizeof()` with integers (amd64)

- How much memory does a value actually occupy?

```go
const (
    size8    = unsafe.Sizeof(uint8(0))
    size16   = unsafe.Sizeof(uint16(0))
    size32   = unsafe.Sizeof(uint32(0))
    size64   = unsafe.Sizeof(uint64(0))
    sizeUint = unsafe.Sizeof(uint(0))
)

fmt.Println(size8, size16, size32, size64, sizeUint)
// 1 2 4 8 8
```

unsafe.Sizeof() with a struct (amd64)

- Shouldn’t this struct occupy 14 bytes?

```
var s struct {
    One   uint64 // 8
    Two   uint32 // 4
    Three uint16 // 2
}

fmt.Printf("want: %d, got: %d", 8+4+2, unsafe.Sizeof(s))
// want: 14, got: 16
```

unsafe.Sizeof() with a padded struct (amd64)

- Struct definitions are padded to the next **machine word size (64 bits)**

```go
var s struct {
    One   uint64  // 8
    Two   uint32  // 4
    Three uint16  // 2
    _     [2]byte // 2
}

fmt.Printf("want: %d, got: %d", 8+4+2+2, unsafe.Sizeof(s))
// want: 16, got: 16
```

`unsafe.Pointer`

Provided that T2 is no larger than T1 and that the two share an equivalent memory layout, [`unsafe.Pointer`] conversion allows reinterpreting data of one type as data of another type.

golang.org/pkg/unsafe/#Pointer

Converting `uint16` to `[2]byte`

- Go's type system won't allow this conversion

```go
a := uint16(1)

// BAD: cannot convert a (type uint16) to type [2]byte
b := [2]byte(a)
```

Converting `uint16` to `[2]byte`

- `unsafe.Pointer` conversions defeat Go's type system

```go
// Always check sizes before performing these conversions!
a := uint16(1)
if unsafe.Sizeof(a) != 2 {
    panic("a is not of the expected size")
}

b := *(*[2]byte)(unsafe.Pointer(&a))
fmt.Println(b)
// [1 0]
```

What does this actually do?

```
*(*[2]byte)(unsafe.Pointer(&a))
```

Take the address of a, producing a `*uint16`

```
*(*[2]byte)(unsafe.Pointer(&a))
```

Convert `*uint16` to `unsafe.Pointer`

```
*(*[2]byte)(unsafe.Pointer(&a))
```

Convert `unsafe.Pointer` to `*[2]byte`

```
*(*[2]byte)(unsafe.Pointer(&a))
```

Dereference pointer, producing `[2]byte`

```
*(*[2]byte)(unsafe.Pointer(&a))
```

Breaking down the conversion

- You could write the same operation with intermediate variables if you wanted

```
a := uint16(1)

uint16Ptr := &a
unsafePtr := unsafe.Pointer(uint16Ptr)
arrayPtr := (*[2]byte)(unsafePtr)
b := *arrayPtr

fmt.Println(b)
// [1 0]
```

A note on slices versus arrays

- **Arrays** are generally used in unsafe conversions, **not slices**
  - Slice an array after conversion, or take address of first element of slice

```
a := uint16(1)

b := (*(*[2]byte)(unsafe.Pointer(&a)))[:]
fmt.Println(b)
// [1 0]

fmt.Println(*(*uint16)(unsafe.Pointer(&b[0])))
// 1
```

The danger zone

The danger zone

- The unsafe.Pointer documentation covers 6 patterns which are valid
    - **Read and understand these patterns before you make use of `unsafe`!**

Credit: Ashley McNamara, github.com/ashleymcnamara/gophers

The danger zone: reading arbitrary memory

- You **must** be judicious in your use of `unsafe`: **always check type sizes**

```
// All bets are off on what is actually stored in b.
a := uint16(1)
b := *(*[4]byte)(unsafe.Pointer(&a))

fmt.Println(b)
// [1 0 160 93]  <- 1st run in play.golang.org
// [1 0 168 118] <- 2nd
// [1 0 19 119]  <- 3rd
```

The danger zone: pointer arithmetic

- Remember this from C? Say hello to `uintptr`!
    - Take the address of the first element in the array, add (i * 4) each iteration

```
b := []uint32{1, 2, 3, 4}
for i := 0; i < len(b); i++ {
    // Pretty much the worst possible way to print a slice in Go.
    fmt.Printf("%d ", *(*uint32)(unsafe.Pointer(uintptr(
        unsafe.Pointer(&b[0])) + uintptr(i)*unsafe.Sizeof(b[0])),
    ))
}
// 1 2 3 4
```

The danger zone: writing arbitrary memory

- Even worse, you can **write to arbitrary memory**

```
var v uint32

// Overwrite whatever data lives at this address.
*(*[4]byte)(
    unsafe.Pointer(uintptr(unsafe.Pointer(&v)) - 0xffffffff),
) = [4]byte{0xff, 0xff, 0xff, 0xff}
```

```
unexpected fault address 0xbf0004c777
fatal error: fault
[signal SIGSEGV: segmentation violation code=0x1 addr=0xbf0004c777 pc=0x487268]

goroutine 1 [running]:
runtime.throw(0x4b8e08, 0x5)
        /usr/local/go/src/runtime/panic.go:617 +0x72 fp=0xc00004c700
sp=0xc00004c6d0 pc=0x427fc2
runtime.sigpanic()
        /usr/local/go/src/runtime/signal_unix.go:397 +0x401 fp=0xc00004c730
sp=0xc00004c700 pc=0x43a8f1
main.main()
        /home/matt/src/github.com/mdlayher/tmp/main.go:12 +0x38 fp=0xc00004c798
sp=0xc00004c730 pc=0x487268
runtime.main()
        /usr/local/go/src/runtime/proc.go:200 +0x20c fp=0xc00004c7e0
sp=0xc00004c798 pc=0x42992c
runtime.goexit()
        /usr/local/go/src/runtime/asm_amd64.s:1337 +0x1 fp=0xc00004c7e8
sp=0xc00004c7e0 pc=0x4511b1
exit status 2
```

When is unsafe actually appropriate to use?

~~syscall~~

golang.org/x/sys/unix

`unsafe` with Linux system calls

- Working with system calls often involves `unsafe` operations
- `syscall` is deprecated in favor of `golang.org/x/sys`
  - golang.org/s/go1.4-syscall
- `ioctl(2), getsockopt(2), setsockopt(2)`
  - These system calls are too flexible to expose a general-purpose API
  - `unix.IoctlGetInt()`, `unix.SetsockoptLinger()`, etc.

ioctl(2) in x/sys/unix

- We can pass a pointer (integer memory address) or even just a regular integer

```
func ioctl(fd int, req uint, arg uintptr) (err error) {
    _, _, e1 := Syscall(SYS_IOCTL,
        uintptr(fd), uintptr(req), uintptr(arg),
    )
    if e1 != 0 {
        err = errnoErr(e1)
    }
    return
}
```

Retrieving data with `ioctl(2)` in C

```c
int fd = open("/dev/vsock", O_RDONLY);
if (fd < 0) {
    perror("failed to open file");
}

uint32_t cid;
if (ioctl(fd, IOCTL_VM_SOCKETS_GET_LOCAL_CID, &cid) < 0) {
    perror("failed to get local CID");
}
printf("CID: %d\n", cid);
```

If [an `unsafe.Pointer`] argument must be converted to `uintptr` for use as an argument, that conversion must appear in the call expression itself.

golang.org/pkg/unsafe/#Pointer

Retrieving data with ioctl(2) in Go

```go
f, err := os.Open("/dev/vsock")
if err != nil {
    log.Fatalf("failed to open file: %v", err)
}

var cid uint32
if err := ioctl(int(f.Fd()), unix.IOCTL_VM_SOCKETS_GET_LOCAL_CID,
    uintptr(unsafe.Pointer(&cid)),
); err != nil {
    log.Fatalf("failed to get local CID: %v", err)
}
fmt.Printf("CID: %d\n", cid)
```

Native endianness

- Integers passed across the userspace/kernel boundary use **native endianness**

```go
func nativeEndian() binary.ByteOrder {
    a := uint16(1)
    switch *(*[2]byte)(unsafe.Pointer(&a)) {
    case [2]byte{0, 1}:
        return binary.BigEndian
    case [2]byte{1, 0}:
        return binary.LittleEndian
    default:
        panic("unknown endianness")
    }
}
```

Linux taskstats interface

- A large type (**328 bytes**); parsing bytes into fields manually could be error prone

```
type Taskstats struct {
	Version                   uint16
	Ac_exitcode               uint32
	Ac_flag                   uint8
	Ac_nice                   uint8
	Cpu_count                 uint64
	Cpu_delay_total           uint64
	Blkio_count               uint64
	Blkio_delay_total         uint64
	Swapin_count              uint64
	Swapin_delay_total        uint64
	Cpu_run_real_total        uint64
	Cpu_run_virtual_total     uint64
	Ac_comm                   [32]int8
	Ac_sched                  uint8
	Ac_pad                    [3]uint8
	_                         [4]byte
	Ac_uid                    uint32
	Ac_gid                    uint32
	Ac_pid                    uint32
	Ac_ppid                   uint32
	Ac_btime                  uint32
	Ac_etime                  uint64
	Ac_utime                  uint64
	Ac_stime                  uint64
	Ac_minflt                 uint64
	Ac_majflt                 uint64
	Coremem                   uint64
	Virtmem                   uint64
	Hiwater_rss               uint64
	Hiwater_vm                uint64
	Read_char                 uint64
	Write_char                uint64
	Read_syscalls             uint64
	Write_syscalls            uint64
	Read_bytes                uint64
	Write_bytes               uint64
	Cancelled_write_bytes     uint64
	Nvcsw                     uint64
	Nivcsw                    uint64
	Ac_utimescaled            uint64
	Ac_stimescaled            uint64
	Cpu_scaled_run_real_total uint64
	Freepages_count           uint64
	Freepages_delay_total     uint64
	Thrashing_count           uint64
	Thrashing_delay_total     uint64
}
```

Linux taskstats interface

- A single `unsafe.Pointer` conversion to `unix.Taskstats` is all we need

```
b := []byte{0x01, /* ... */}
const sizeofTaskstats = int(unsafe.Sizeof(unix.Taskstats{}))

// Always sanity check the structure size before conversion!
if sizeofTaskstats != len(b) {
    return nil, errors.New("unexpected taskstats structure size")
}

stats := *(*unix.Taskstats)(unsafe.Pointer(&b[0]))
```

Other uses for unsafe

- Potential performance improvements in specific situations
- Cgo: passing data between C and Go
- Accessing unexported identifiers with `//go:linkname`

Summary

With great power comes great responsibility

- When you import `unsafe`, **you’re expected to know how to use it safely**
- Do not fear `unsafe`, it’s a vital part of what makes Go work
- When in doubt, seek guidance and ask questions!
    - Gophers Slack: invite.slack.golangbridge.org, #darkarts

Resources and thanks

- Blog: `unsafe.Pointer` and system calls
  - mdlayher.com/blog/unsafe-pointer-and-system-calls
- Source for packages referenced in this talk
  - github.com/mdlayher/taskstats, github.com/mdlayher/vsock
- Thanks to @acln, @jadr2ddude, @kale, @pwaller, @seebs, and @zeebo from #darkarts on Gophers Slack for their review

Thanks!

Matt Layher

mdlayher.com
github.com/mdlayher
twitter.com/mdlayher