Go for your PhD

The language, its tools, best practices and cool toys

6 May 2021

Gaurav Narula

Noémien Kocher

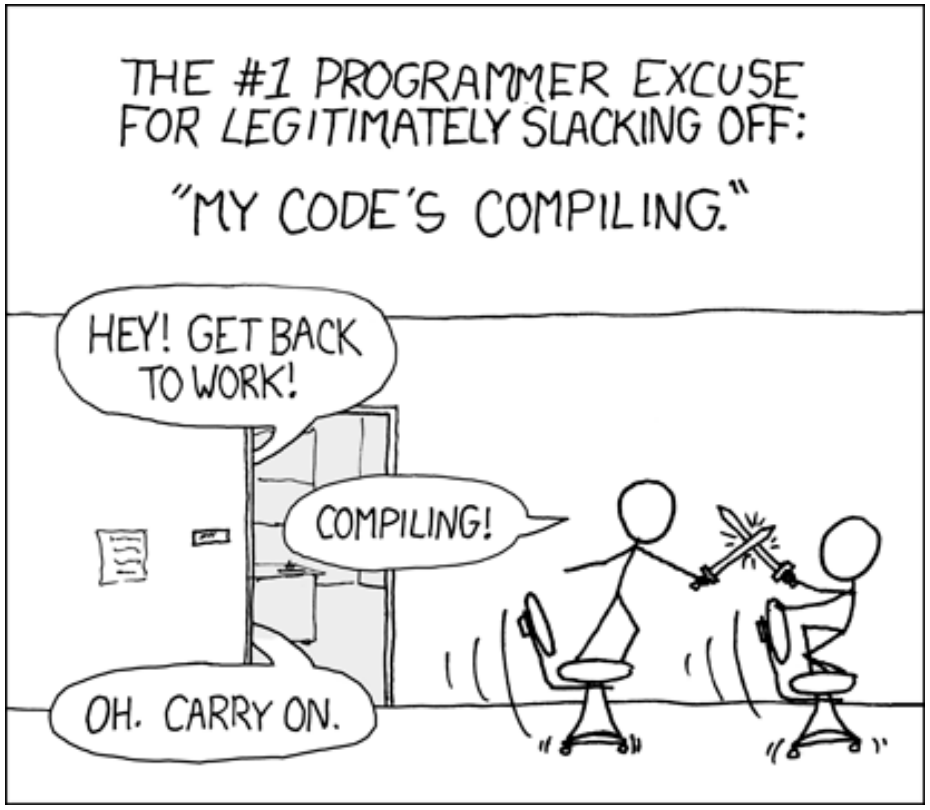
Pierluca Borsò

Talk outline

- Why is Go awesome?
- The tools you're going to love
- How to screw up (or not)
- Understanding and debugging
- Getting that extra performance kick
- Monitoring and measurements

Why is Go awesome? - the origin story

- Imagine you're working at Google, and seeing the company waste a ton of money every day...



- and it's not just compilation !

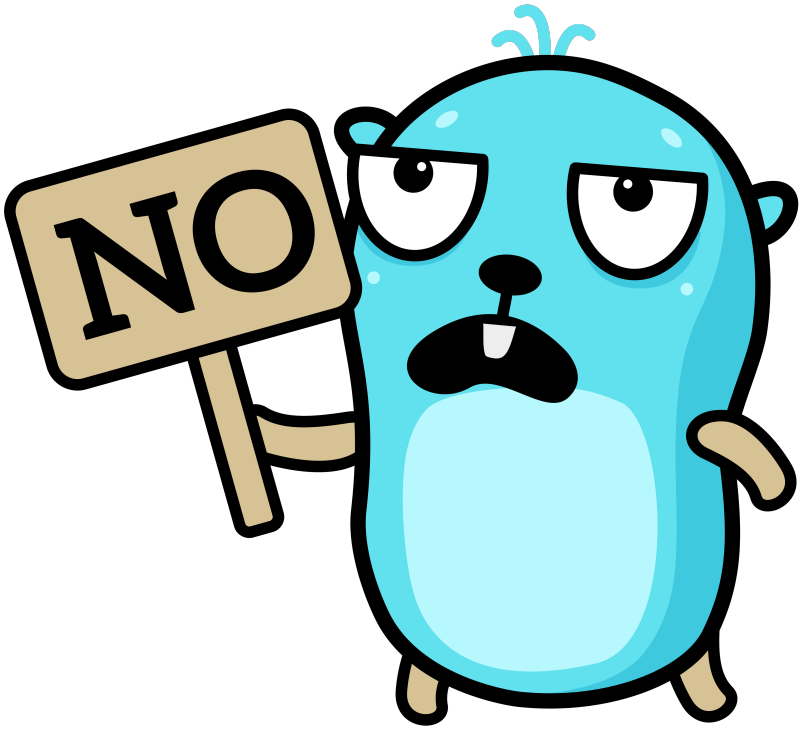
The origin story (cont'd)

- C++ is high performance, but cumbersome
- Usability: let's not go there
- Parallelism: only achieved through specialized libraries
- Memory management: manual
- Deployment: dependencies are a pain (DLLs)
- Code formatting: inconsistent across teams
- Maintenance: changing an API in a library is hell if you're Google

Go as a solution

- High-performance, compiled language inspired by C
- Simple unambiguous syntax, optimized for compilation and human readability
- Statically linked, easy deployment *even across systems*
- Garbage-collected
- Concurrency as a first class-citizen (channels, goroutines)
- Easy to refactor or change APIs: the AST can be manipulated very easily

Go as a solution (cont'd)



Go as a solution (cont'd)

- Go is very opinionated
- There's one right way of doing things, incl. code formatting
- Constrains you as a developer, but it saves everyone's time

Tools you'll love

- go get, go install, go build, go run
- go fmt, goimports
- godoc, go doc
- go vet, staticcheck
- race detection, go test

go fmt and goimports

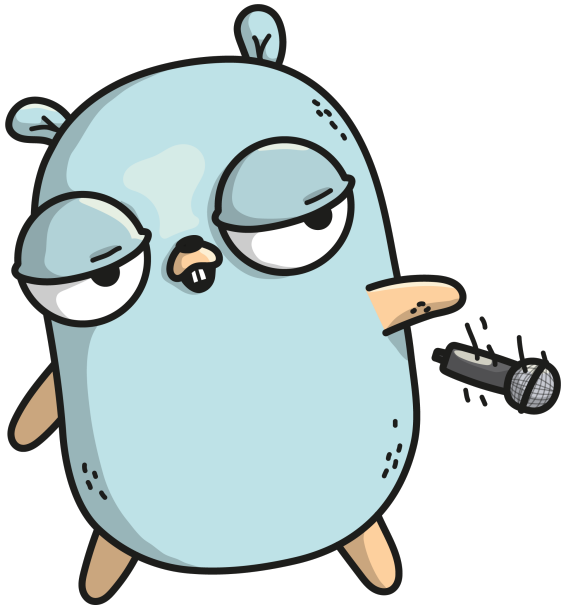
- (demo)

Is my code broken?

- go vet: demo
- staticcheck: demo

Is my code broken? race detection!

- go run **--race** bad_race.go
- You can use --race with all commands (build, install, run, test)
- (demo)



Testing

- Go comes with unit tests, benchmarks and examples - all built in !
- Consider using library `github.com/stretchr/testify/require`

Examples:

- https://github.com/si-co/vpir-code/blob/main/lib/database/merkle_test.go
- https://github.com/dedis/dela/blob/master/crypto/ed25519/example_test.go

Other demos

- godoc, go doc

A word on day-to-day

- You probably don't want to run all of this, all the time
- Yet, you should.
- *Consider* setting up a Continuous Integration pipeline on your Github repo
- **Do** configure your IDE to automate most of it

IDEs

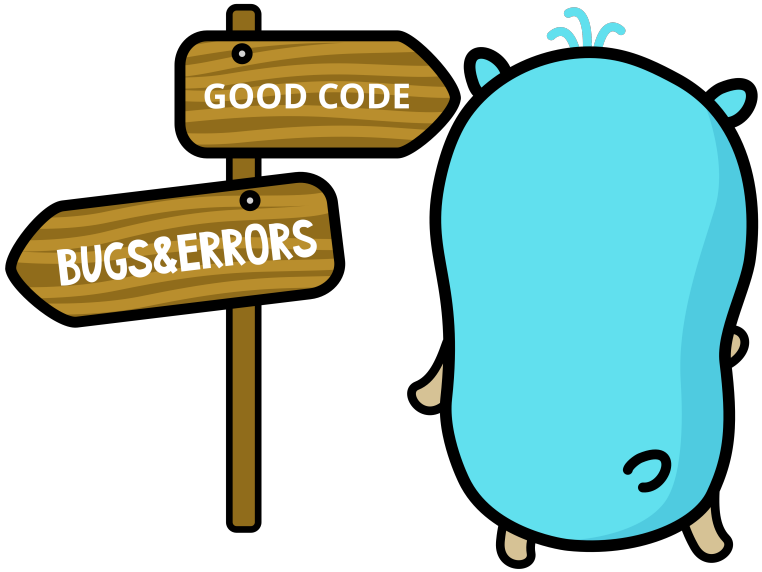
- If you're a Emacs/Vim wizard, keep doing your thing. I'm sure your awesome.
- IDEs (VSCode, GoLand) help you be more productive.
- GoLand is particularly powerful, but it requires configuration.

  Editor/Inspections Tools/File Watcher Tools/External Tools

- We'll come back to IDEs later ;-)

How to screw up (or not)

1. Organize your source
2. Handle errors
3. Use custom types



1: Organize your sources

- (**bad** code + **good** architecture) > (**good** code + **bad** architecture)
- break down your project into independent packages
- iterative process (that's an investment, but worth it)
- Avoid general-purpose packages:
  - ⛔`utils`, ⛔`types`, ⛔`data`, ...
  - Break up by topic or dependency
- Define interface for your packages
- prevents import cycles
- 🏆🏆🏆3 wins: readability / testability / maintainability

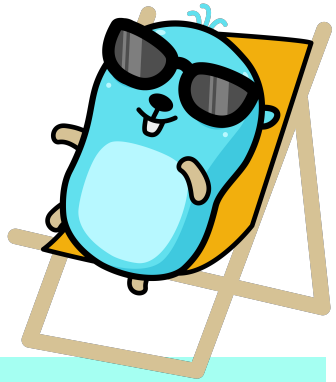
github.com/golang-standards/project-layout (https://github.com/golang-standards/project-layout)

1: Organize your sources (cont'd)

```
server
|   mod.go
|
└──storage          <- standalone functionality
|   |   mod.go      <- interface
|   |
|   └──postgresql   <- an implementation
|       |   mod.go
|
└──otherpackage
    |   ...
```

1: Organize your sources (cont'd)

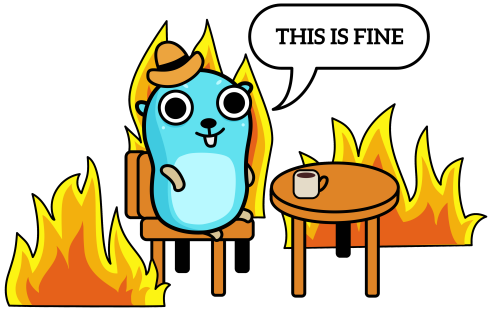
take an interface, delegate responsibility



```
// server/mod.go

func NewServer() server {
    storage := NewMemory()
    return server {
        storage: storage
    }
}
```

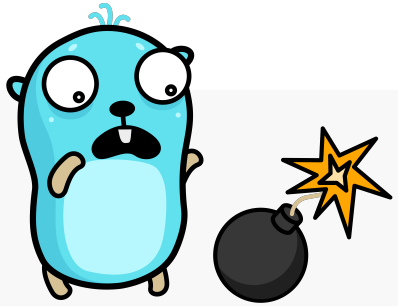
```
// server/mod.go

func NewServer(storage storage.Storage) server {
    return server {
        storage: storage
    }
}

// server/storage/mod.go

type Storage interface {
    ...
}
```

2: Handle errors



Don't **panic()**, `: %v,` **wrap** errors, and `if err != nil {` **fail fast** `}` .

If your code doesn't look like this ...



```
bob, alice, err := retreive()
if err != nil {
    return xerrors.Errorf("failed to get Bob and Alice: %v", err)
}

..., err = op1()
if err != nil {
    return xerrors.Errorf("failed to do op1: %v", err)
}

..., err = op2()
if err != nil {
    return xerrors.Errorf("failed to do op2: %v", err)
}

..., err = op3()
if err != nil {
    return xerrors.Errorf("failed to do op3: %v", err)
}
```

Fail fast

```
err := op1() {
    if err == {
        dothing1()
        ...
    } else {
        return xerrors.Errorf(...)
    }
}
```

```
err := op1() {
    if err == {
        return xerrors.Errorf(...)
    }

    dothing1()
}
```

3: Embed functionalities with struct

```
type server struct {
    router map[string]string
    routerMux sync.Mutex
    ...
}

func (s server) main() {
    s.router.Lock()
    s.router["a"] = "b"
    s.router.Unlock()
    ...
}
```

```
type server struct {
    router *router
    ...
}

func (s server) main() {
    s.router.Add("a", "b")
    ...
}

type router struct {
    sync.Mutex

    routes map[string]string
}

func (r *router) Add(rte, dest string) {
    r.Lock()
    defer r.Unlock()

    r.routes[rte] = dest
}
```

3: Embed functionalities with struct (cont'd)

- not a problem for simple program, but that escalades quickly 🔥
- first step to a good architecture
- think of small pieces of functionalities:
  - "I want something to store my routing that is thread-safe"
- extract to a package when this becomes a standalone functionalities

Bonus: Channels and routines

- "Do not communicate by sharing memory; instead, share memory by communicating"
  - want to share data between routines ? 👉channel
- Do not spawn a dynamic number of routines
  - use a pool with workers
- Always have a plan to terminate your routines 🔥

Two problems (at least)

```
package main

import (
    "fmt"
    "time"
)

func main() {
    users := []string{"A", "B", "C"}
    for _, user := range users {
        go func() {
            time.Sleep(time.Millisecond * 100)
            fmt.Println(user)
        }()
    }
    time.Sleep(time.Second)
}
```

Run

Better

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    users := []string{"A", "B", "C"}

    wg := sync.WaitGroup{}
    wg.Add(len(users))

    for _, user := range users {
        go func(u string) { // still bad, should use a pool
            defer wg.Done()
            time.Sleep(time.Millisecond * 100)
            fmt.Println(u)
        }(user)
    }

    wg.Wait()
}
```

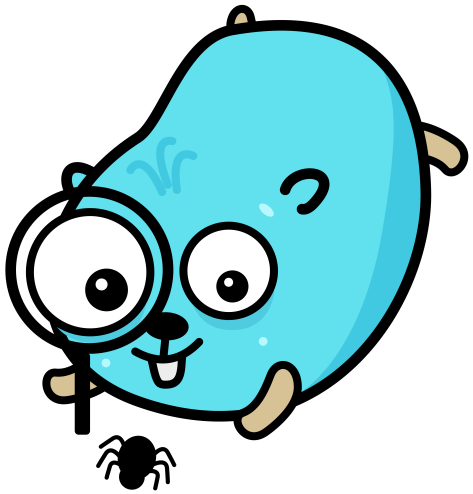
Understanding and debugging

- Printfs everywhere != debugging, let's agree to move past that

- So, what is good debugging?

Understanding and debugging

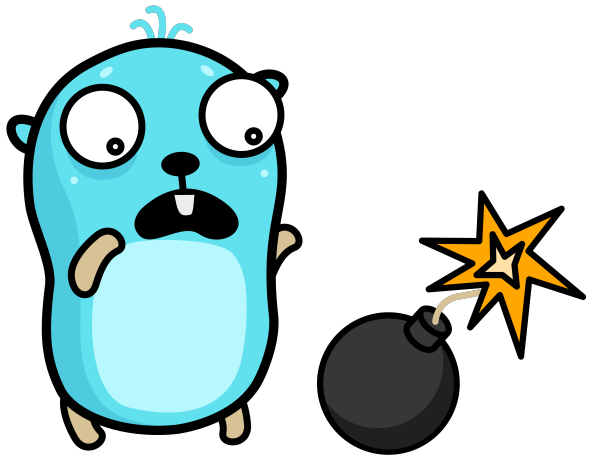
- Debugging is a scientific process of discovery
- Part of it is exploratory
- Most of it is the creation and rejection of hypothesis



- What are the tools of inquiry?

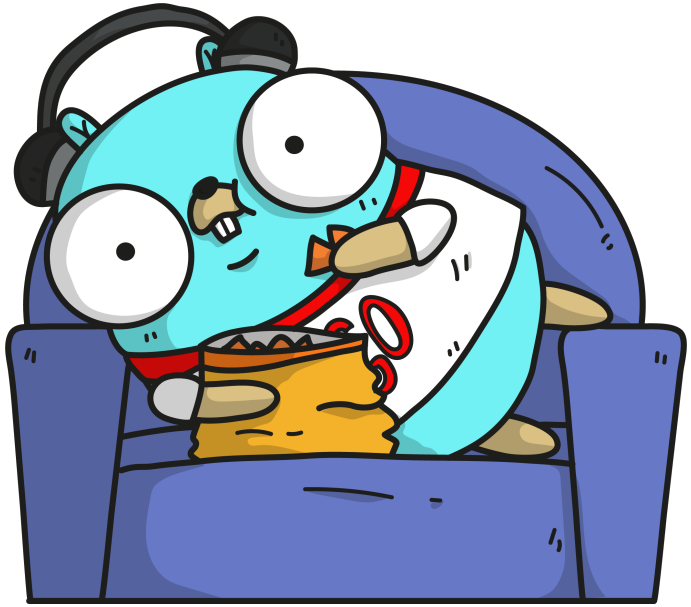
Understanding and debugging: tools

- Logging
  - what is the system doing? sequence and details of operations
- Metrics
  - how is the system performing? # of requests/s, RTT, queue lengths, etc.
- Debugger
  - To look deeper, much much deeper



Let's get down to debugging

- Delve, if you're hardcore
- GoLand or VSCode, if you're lazy like me.



Optimizing

"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; **premature optimization is the root of all evil** (or at least most of it) in programming."

-The Art of Computer Programming

Optimizing

- You can optimize algorithms ahead of implementing them
- You **can't and shouldn't** optimize code ahead of writing it
- Measure first, act later
- (profiling demo)

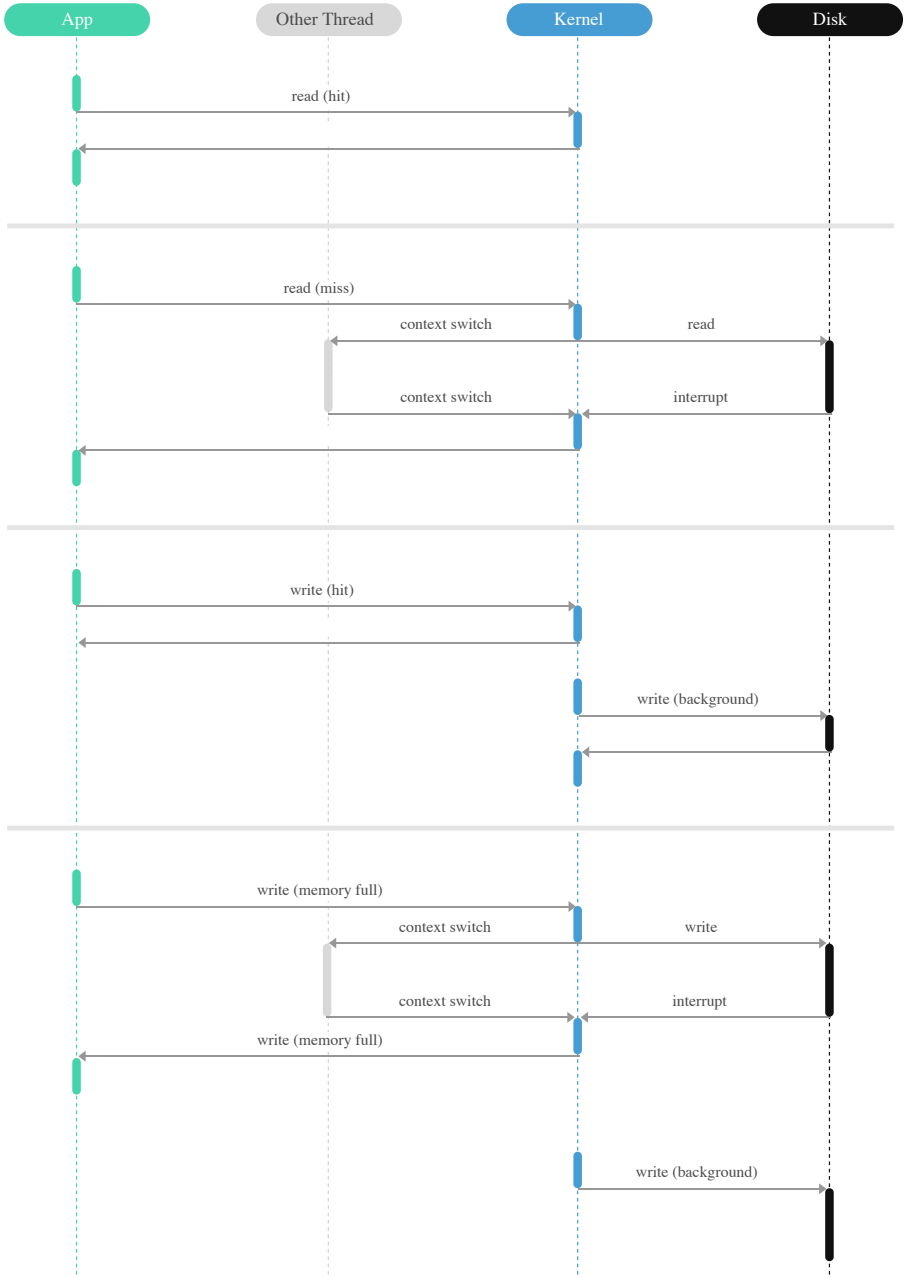
File IO Optimizations

- The suggestions here don't apply for normal scenarios
- As a rule of thumb, prefer to use io.Reader, io.Writer interfaces - *os.File implements them
- However, there might be scenarios where you want to avoid them

Recap: Disk IO and Virtual Memory

- Traditional read/write APIs use the read/write syscalls
- Read syscall:
  - Requests the kernel to read the file at an offset
  - If the page is available in the kernel's page cache, the kernel copies it to user space buffer
  - Otherwise we have a page fault, kernel reads the page into page cache and then copies it into user space buffer
- Write syscall:
  - Requests the kernel to copy the data into page cache.
  - Kernel schedules the sync to disk
  - User can block on sync using fsync

Recap: Disk IO and Virtual Memory


App
Other Thread
Kernel
Disk
read (hit)
read (miss)
context switch
read
context switch
interrupt
write (hit)
write (background)
write (memory full)
context switch
write
context switch
interrupt
write (memory full)
write (background)

Memory Mapping

- mmap allows mapping of an entire file in memory
- Process' virtual address space maps directly to page cache
  - No copying of buffers
- Does not imply you need RAM == size of file
  - Because pages are loaded on demand
- Can also map regions of the file in memory
  - Caveat: offset must be page aligned (usually 4K but varies with the system)

Memory Mapping

- Kernel does have to do more book keeping behind the scenes
- Might be worth it if you're dealing with a large file and want to avoid allocations again and again
- A page fault in mmap doesn't preempt a Go routine
  - you'd underutilise your CPU in that case because an OS thread would be blocked (https://valyala.medium.com/mmap-in-go-considered-harmful-d92a25cb161d)
- Watch out for concurrent write operations to an mmaped file. `ftruncate` for instance doesn't play well with mmap
  - Have a look at Bryan Cantrill's experience with tail (https://youtu.be/vm1GJMp0QN4?t=2463)

Memory Mapping

```
import (
        // ..
        "github.com/edsrzf/mmap-go"
)

func fixTypo() error {
        f, err := os.OpenFile("./data", os.O_RDWR, 0644)
        if err != nil {
                return xerrors.Errorf("failed to open file: %v", err)
        }
        defer f.Close()

        buf, err := mmap.Map(f, os.O_RDWR, 0)
        if err != nil {
                return xerrors.Errorf("failed to mmap file: %v", err)
        }
        defer buf.Unmap()

        buf[10] = 'S'

        err = buf.Flush()
        if err != nil {
                return xerrors.Errorf("failed to flush file: %v", err)
        }
        return nil
}

// Before
// Hello DEDIs!

// After
Hello DEDIS!
```

Memory Mapping

```
import (
        // ..
        "github.com/edsrzf/mmap-go"
)

type MyData struct {
        Data *[32]byte // fix size of 32B
}

func parseStruct() error {
        f, err := os.OpenFile("./struct", os.O_RDONLY, 0644)
        if err != nil {
                return xerrors.Errorf("failed to open file: %v", err)
        }
        defer f.Close()

        buf, err := mmap.Map(f, os.O_RDONLY, 0)
        if err != nil {
                return xerrors.Errorf("failed to mmap file: %v", err)
        }
        defer buf.Unmap()

        myData := make([]MyData, 10000)
        for i := 0; i < 10000; i++ {
                // careful here
                myData[i].Data = (*[32]byte)(unsafe.Pointer(&buf[32*i]))
        }

        for i := 0; i < 10; i++ {
                fmt.Printf("%x\n", *myData[i].Data)
        }

        // this should cause a SIGBUS
        // myData[1].Data[0] = 0

        return nil
}
```

Direct IO

- As the name implies, it allows bypassing the page cache completely!
- Data is copied directly to/from the userspace/disk
- Writes/Reads must be block aligned - usually 512B.
- Useful if you want to control caching on the application level
- Allows finer control over scheduling IO - remember writes just mark a page dirty?
    - the Go runtime and the kernel would still be scheduling your thread

Direct IO

```
// import "github.com/ncw/directio"

source, err := directio.OpenFile("./source", os.O_RDONLY, 0644)
if err != nil {
    return xerrors.Errorf("failed to open source: %v", err)
}

dest, err := directio.OpenFile("./dest", os.O_RDWR | os.O_CREATE | os.O_TRUNC, 0644)
if err != nil {
    return xerrors.Errorf("failed to open dest: %v", err)
}

for {
    // read in multiples of block size
    buf := directio.AlignedBlock(directio.BlockSize)
    n, err := io.ReadFull(source, buf)
    if err == io.EOF {
        return nil
    }
    if err == io.ErrUnexpectedEOF {
        _, err = io.Copy(dest, bytes.NewBuffer(buf[:n]))
        if err != nil {
            return xerrors.Errorf("failed to write last block: %v", er
        }
        return nil
    }

    if err != nil {
        return xerrors.Errorf("failed to read from source: %v", err)
    }

    _, err = io.Copy(dest, bytes.NewBuffer(buf[:]))
    if err != nil {
        return xerrors.Errorf("failed to write to dest: %v", err)
    }
```

File IO Optimisations (summarizing)

- read/write go through page cache and involve copy between kernel/user space
- mmap bypasses the kernel as long as your data is in the page cache
- DirectIO bypasses the page cache completely
- Don't consider the last two as a default - your workload and benchmarks should guide which API to use.

CPU Optimizations

- Technically, you can use assembler code within Go
- Ask yourself: can you do better than the compiler?
- If you want to venture down this path, best of luck

Thank you

Gaurav Narula

Noémien Kocher

Pierluca Borsò