Starting Systems Programming, Pt 1: Programmers Write Programs

A software article by Efron Amber Licht

MAR 2025

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

This is the first of four articles on the fundamentals of systems programming. It will cover many of the essentials, such as bit manipulation, parsing, filesystems, input/output, syscalls, memory management, and signals. Like many of my article series, this is more of a grab bag than a comprehensive guide - but I hope it will be useful to you.

- 1. Series Introduction
  - 1.1. Programmers Write Programs
    - Note on style & environment
    - note for python programmers
    - example go block
    - example bash block
  - 1.2. Some final caveats:
  - 1.3. Series Overview
- 2. What is systems programming?
- 3. Peeking into the black box: what is a program, anyways?
  - 3.1. hello.go
    - overview
  - 3.2. buildhello.bash
    - Overview

- 4. Investigating the data segment
  - 4.1. finding strings with `findoffset.go`
    - Overview
    - Used in this example:
    - findoffset.py: click here
  - 4.2. writing simple files with `echo.go`
    - Overview
  - 4.3. printing files with `cat.go`
    - overview
    - exercises
  - 4.4. investigating the `hello` program with `findoffset`, `echo`, and `cat`
    - bash script: `catfox.bash`
  - 4.5. Basic Hacking w/ `binpatch.go`
    - Overview
    - Hacking a binary
  - 4.6. reaching into files with `torso.go`
    - Overview
  - 4.7. Investigation: What's with `3814697265625`?
    - IN
    - OUT
- 5. Investigating the code segment.
  - 5.1. reading binary files with `shexdump.go`
    - Exercises
  - 5.2. deserializing hexdumps with `unhexdump.go`
    - overview
    - program: `unhexdump.go`
  - 5.3. hexdump_test.bash
    - overview
  - 5.4. Investigating the ELF header of `hello`
- 6. Conclusion: The Spirit of Systems Programming. - Sidenote: The origin of endianness

1. Series Introduction

1.1. Programmers Write Programs

When's the last time you wrote a program from scratch? For a shocking number of programmers, the answer is 'in school'. This is a pervasive problem in the industry, and it's only getting worse. I interview a lot of candidates, and I've run into people with titles like 'Technical Lead @ Tesla' (or worse yet, Principal Engineer) who can't program their way out of a paper bag. My ordinary interview question is "write me `grep`" - a problem which *should* be appropriate for first or second-year computer science students - and the overwhelming majority of candidates fail it.

I don't think they're *dumb* - they usually aren't - but they don't have the grounding in programming fundamentals - systems programming fundamentals - that they need to "really" program. This is bad for the candidates, bad for the industry, and bad for our increasingly computerized world. Why? Making reliable, intuitive, and efficient software is about minimizing complexity. If all you can do is *add on* to existing programs, you're glued to pre-existing complexity. If you can't **write a program** from scratch, you're stuck in a world of other people's code and other people's mistakes.

The way to get good at something is by doing it. Pitchers pitch, painters paint, and programmers program. So this article will be about **writing programs** - dozens of them. As such, while reading the text of this article might teach you a few things, to really get the most out of it, you'll need to understand the programs. I've provided exercises to help you practice.

Note on style & environment

Where possible, the code in this series will use as few libraries as possible. This is not because you shouldn't use libraries - but because you shouldn't *need* them. I want to show you that you can make practical tools out of simple primitives.

You'll see a number of code blocks throughout this article. These are either go programs or bash shell scripts. If you've had experience with a mainstream programming language like Python, Javascript, or C, you should be able to follow along, but you might want to review pointers a bit.

Go uses `//` for comments, and bash and python use #. I'll start each code block with a comment indicating the language: `// filename.go` or `#!/usr/bin/env bash`.

note for python programmers

> **I have provided python implementations of many programs in this series**. See my `gitlab in the articles/startingsystems/cmd/pythonports directory`.
> I'll try to link the specific files at the head of each go program. The go program should always be treated as the 'canonical' version. I may or may not keep doing this in the next article - it's a lot of work.

---

example go block

Go programs will start with `// filename.go <description>`.

```
1   // minimal.go is an example go program.
```

```
2   // see https://gitlab.com/efronlicht/blog/-/blob/
    58fb4c13f870a73514284617c71027bbe0a76e2a/articles/
    startingsystems/cmd/pythonports/minimal.py for the python
    version.
3   package main
4   import "fmt"
5   func main(){ fmt.Println("this is a go program") }
```

example bash block

Bash scripts will start with `#!/usr/bin/env bash`. They usually contain a series of commands that you might run in a terminal after the `# IN` comment. The `# OUT` comment shows the expected output of the commands.

```
1   #!/usr/bin/env bash
2   # example.bash demonstrates a simple bash script with an #
    IN and # OUT section.
3
4   # IN:
5   echo "this is a bash script"
6
7   # OUT:
8   this is a bash script
```

> #### lemma: sidenotes
>
> Sidenotes will show up in indented boxes, like this. A ‘lemma‘ is a small digression important to clarify a point.

> #### lemma: shebang (`#!`)
>
> The shebang (`#!`) at the beginning of a file tells the operating system what program to use to run it. For example, `/usr/bin/bash` will run the file with the bash shell located at `/usr/bin/bash`. `#!/usr/bin/env bash` tells the OS to use whatever `bash` is in your `PATH` environment variable to run the script. We’ll talk more about all of these things in a later article.

1.2. Some final caveats:

- I’m way more familiar with Linux than Windows, Darwin, or BSD, so this article will be linux-centric. I will occasionally point out differences between Linux and other operating systems - but when I say “the OS”, I might just mean Linux.
- This cannot be a comprehensive guide. Ideally, you’ll have some basic knowledge of computer architecture. If you run into terms you don’t know, like “nibble”, “register” or “file descriptor”, *don’t panic* - in general, you should be able to follow along. I’ll provide a glossary for the things I can think of, but I’m sure I’ll miss

some.
- This series of articles provides the source code for many, many programs. **Reading the code is the core of this series** - the code is usually *more* important than the text. I strongly encourage you to read and modify the code as you go along.

OK, enough ceremony. Let's get started.

1.3. Series Overview

1. #### Programmers Write Programs <— you are here

In this article, we'll talk about what systems programming is, what a program is, and how to interact with the data inside a program. We'll build a program and dig into the data inside it, hack it to change it's behavior, and build a series of software tools to help us understand it that we'll use throughout the series.

1. #### Your program and the outside world: Command-line arguments, environment variables, and syscalls

How do programs interact with the outside world? We'll cover the fundamentals of the UNIX programming environment, including command-line arguments, environment variables, and syscalls, building up to a simple command-line interpreter (aka shell).

1. #### Execution Counts: Hardware, Memory, & Software Performance (COMING SOON)

How do programs interact with hardware? We'll cover the fundamentals of storage and access - registers, memory management, and cache - talk about what *actually happens* when you call a function or system call - and give a crash course on performant programming in general.

1. #### Wait, it's all `gotwo` - the fundamentals of programming, virtual machines, assembly, debugging, and ABIs. (COMING SOON)
   Wait, it's all `goto`? When it comes down to it, programming is some memory, an instruction pointer, and a series of conditional jumps. We'll use our new systems programming skills to build a virtual machine & assembly language that's a valid subset of go. We'll use that to illustrate how debuggers and ABIs work.
   We'll invent a virtual machine and programming language that's a valid subset of go and use it to explore the fundamentals of programming & debugging.

2. What is systems programming?

There's no clear line between "systems programming" and other kinds. A problem might be 'systems programming if it'

- interacts with the operating system or hardware

- has tight performance constraints
- operates at a 'low level', dealing with individual bytes or registers

A **systems programmer** sees a computer as a *physical machine* that can be completely understood, rather than a mathematical or formal abstraction. They understand the *hardware* and *software* of a computer system, and they can **write programs** that interact with both. A systems programmer is unafraid to tear something apart, confident that they can put it back together again.

3. Peeking into the black box: what is a program, anyways?

A *program* is an executable file that your operating system can interpret as a series of machine instructions. That is, it's a combination of code and data that the operating system can load into memory and execute.

Programs come in two main types:

1. the kind that takes input and produces output (the focus of this article)
2. the kind that run indefinitely, waiting for interaction from the outside world (daemons, servers, etc)

When we say 'input' and 'output', we mean bytes. To warm up, let's **write** a program that takes no input but produces output: the nearly 50-year-old classic, "hello, world!".

3.1. hello.go

hello.py: click here

overview

- print a string to standard output

```
1  // hello.go
2  package main
3  import "fmt"
4  func main() {
5      fmt.Println("hello, world!")
6  }
```

3.2. buildhello.bash

Overview

1. call `go build` to compile the program
2. run the program

```
#!/usr/bin/env bash
# buildhello.bash builds and runs the hello program.
# IN
go build -o hello hello.go # 1. call 'go build' to compile the program
./hello # 2. run the program
# OUT
hello, world!
```

Great. What's actually *in* the hello program? We expect

- data - including the string "hello, world!"
- code - the instructions to print that string
- maybe some other stuff?

We'll take a poke around and see what we can find. Let's start with the **data** - that is, any of the bytes in the file that aren't executable instructions.

4. Investigating the data segment

Regardless of your operating system or architecture, there's one thing we can be sure will exist in the file: the string "hello, world!". Let's look for it. Better yet, let's **write a program** to look for it - we'll call it `findoffset`.

4.1. finding strings with `findoffset.go`

Overview

We want to look for a specific string in a file and print the offset of the first occurrence.

When it comes down to it, a string is just a sequence of bytes in some character encoding. We'll do that by comparing the bytes in the file to the bytes in the string, one-by-one.

That is, we'll:

1. parse the command line arguments
2. read the file into memory
3. compare the bytes in the file to the bytes in the string, one-by-one
    1. no match: continue at next offset
    2. match: print and exit 0 (ok)

4. exit 1 (error)

Lemma: standard output streams

All programs are connected to three files by default. This is often called "standard i/o", sometimes just `stdio`. They are:

FILE	Name	R/W?	NOTE	PYTHON	JS	GO	Note
STDIN	standard input	R	what you type in the terminal goes here.	sys.stdin	process.stdin	os.Stdin	Input
STDOUT	standard output	W	what the program writes goes here. intended for other programs.	sys.stdout	process.stdout	os.Stdout	
STDERR	standard error	W	where the program writes errors. intended for humans.	sys.stderr	process.stderr	os.Stderr	Error

Used in this example:

function or variable	type	description	notes
`os.Args`	`[]string`	command line	arguments
`os.Exit(int)`			
`fmt.Fprintf(io.Writer, string, ...interface{})`	`int`	write formatted output to a stream (files, memory buffers, etc)	

findoffset.py: click here

```
1   // findoffset.go is a command line tool that finds the
    offset of the first occurrence of a string in a file and
    prints it to stdout.
2   package main
3
```

```go
import (
    "fmt"
    "os"
)

func main() {
    // 1. parse the command line arguments

    // the operating system provides command line arguments to your program.
    // os.Args[0] is the name of the program, and the rest are the the 'real' arguments.
    if len(os.Args) != 3 {
        fmt.Fprintf(os.Stderr, "Usage: findoffset <filename> <string>")
        os.Exit(1)
    }

    filepath, pattern := os.Args[1], os.Args[2]

    // 2. read the file into memory

    // it's inefficient to read the entire file into memory, but it's simple and works well for small files
    b, err := os.ReadFile(filepath) // we'll talk about how reading files works more later, too!
    if err != nil {
    fmt.Fprintf(os.Stderr, "read %s: %v", filepath, err) // HUMAN-READABLE DEBUG INFO should go to STDERR
        os.Exit(1)
    }

    // 3. compare the bytes in the file to the bytes in the string, one-by-one
    for i := 0; i < len(b)-len(pattern); i++ {
        for j := range pattern {  // byte-by-byte comparison
            // 3.1. no match: continue at next offset
            if b[i+j] != pattern[j] {
                break
            }

            // 3.2. match: print and exit 0 (ok)
            if j == len(pattern)-1 { // found it! print the offset & newline & exit
                fmt.Fprintf(os.Stdout, "%d\n", i)  // MACHINE-READABLE OUTPUT should go to STDOUT
                os.Exit(0)
            }
        }
    }
    // 4. exit 1 (error)
    os.Exit(1)
```

```
47  }
```

Looks good. But how do we test it? It would be easier to test `findoffset` if we had a way to create files with specific contents. Let's **write a program** to do that - following the unix tradition, we'll call it `echo`.

4.2. writing simple files with `echo.go`

Overview

1. iterate over the command line arguments
2. print each argument to standard output, separated by spaces
3. terminate with a newline

```
1   // echo prints its arguments to standard output, separated
    by spaces and terminated by a newline.
2   // usage: echo <args...>
3   // see the python port at https://gitlab.com/efronlicht/
    blog/-/blob/0d2327696c01d6a46551fac21521937ee9f6fbe3/
    articles/startingsystems/cmd/pythonports/echo.py
4   package main
5   func main() {
6       // 1. iterate over the command line arguments
7       for i, arg := range os.Args[1:] {
8           if i > 0 {
9               fmt.Print(" ")
10          }
11          // 2. print each argument to standard output,
    separated by spaces
12          fmt.Print(arg)
13      }
14      // 3. terminate with a newline
15      fmt.Println()
16  }
```

Now we can write a simple file, but how do we know what's in it? We can **write a program** to read it - following the unix tradition, we'll call it `cat`.

4.3. printing files with `cat.go`

> `cat` - short for con**cat**enate - combines files and prints them to standard output. But it's more often used to just read a single file and send it to the terminal or another program.

`cat.py`: click here

overview

We want to

1. read each file specified on the command line
2. read it into memory
3. copy that memory to standard output

```
1   // cat reads each file specified on the command line and
    writes its contents to standard output.
2   // usage: cat <file1> [<file2> ...]
3   package main
4   import (
5       "fmt"
6       "os"
7   )
8   func main() {
9       for _, file := range os.Args[1:]
    { // 1. read each file specified on the command line
10          f, err := os.Open(file)
11          if err != nil {
12              fmt.Fprintf(os.Stderr, "open %s: %v", file,
    err)
13              os.Exit(1)
14          }
15          // performance note: it's better to use `io.Copy`,
    but I want to illustrate the process.
16          defer f.Close()
17          b, err := io.ReadAll(f) // 2. read it into memory
18          if err != nil {
19              fmt.Fprintf(os.Stderr, "read %s: %v", file,
    err)
20              os.Exit(1)
21          }
22          os.Stdout.Write(b) // 3. write its contents to
    standard output
23
24      }
25  }
```

Let's write a pair of files with `echo` and read them with `cat`.

```
1   #!/usr/bin/env bash
2   # IN
3   echo "the quick brown fox" > fox.txt
4   echo "jumps over the lazy dog" > dog.txt
5   cat fox.txt dog.txt
6   # OUT
7   the quick brown fox
8   jumps over the lazy dog
```

exercises

- Write a program to number the lines of a file, `numberlines`.
- Write a program to replace non-printable characters in a file, `escapetext`.

4.4. investigating the `hello` program with `findoffset`, `echo`, and `cat`

Let's write a simple file, `fox.txt` and read it with `cat`.

bash script: `catfox.bash`

```
1  #!/usr/bin/env bash
2  # IN
3  echo "the quick brown fox jumps over the lazy dog" > fox.txt
4  cat fox.txt
5  # OUT
6  the quick brown fox jumps over the lazy dog
```

Looks good. Let's use `findoffset` to find the offset of "brown" in `fox.txt`.

```
1  #!/usr/bin/env bash
2  # findbrown.bash looks for the string "brown" in the file
   "fox.txt" and prints the offset.
3  # IN:
4  echo "the quick brown fox jumps over the lazy dog" > fox.txt
5  findoffset fox.txt "brown"
6  # OUT
7  10
```

Seems like it works. Let's find the offset of "hello, world!" in our `hello` program.

```
1  #!/usr/bin/env bash
2  # IN
3  findoffset hello "hello, world!"
4  # OUT
5  721335
```

exercises

- modify `findoffset` to take a second argument which specifies the occurrence of the string to find. For example, `findoffset hello "hello, world!" 2` should find the second occurrence of "hello, world!" in the `hello` program.
- allow negative offsets in `findoffset` to search from the end of the file. For example, `findoffset hello "hello, world!" -1` should find the last occurrence of "hello, world!" in the `hello`

program.

4.5. Basic Hacking w/ `binpatch.go`

Suppose we want to change the behavior of a *compiled* program and don't have access to the source code.

We know the following facts:

- Our program is just a bunch of bytes.
- We can *read* and *write* those bytes.

This is all we need to know to change the behavior.

Let's change our program to write "hello, efron!" instead of "hello, world!" *without* recompiling it. We can do this by **patching** the binary. Let's **write a program**, `binpatch`, to do so.

Overview

We want to copy everything over *except* a specific chunk of bytes. We need to:

1. Parse the arguments
2. Copy everything before the replacement from `file` to standard output (that is, `offset` bytes of the file)
3. Write `replacement` to standard output
4. Skip over the bytes we're replacing
5. copy the rest of the file to standard output

`binpatch.py`: click here

```
1   // binpatch replaces a sequence of bytes in file starting
    at offset with a replacement string,
2   // and writes the result to standard output.
3   // Usage: binpatch <file> <offset> <replacement>
4   package main
5
6   import (
7       "fmt"
8       "io"
9       "os"
10      "strconv"
11  )
12
13  func main() {
14      // 1. Parse the arguments
15
```

```
16      // the first argument is the name of the program, so we
    need to check for 4 arguments.
17      // we'll talk more about arguments later.
18      if len(os.Args) != 4 {
19          // having the name of the program is useful for
    error messages, like this one.
20          // error messages are written to stderr, so they
    don't interfere with the output.
21          fmt.Fprintf(os.Stderr, "Usage: %s <file> <offset>
    <replacement>", os.Args[0])
22          os.Exit(1)
23      }
24      var (
25          file        = os.Args[1]
26          offset, err = strconv.ParseInt(os.Args[2], 0, 64)
27          replacement = os.Args[3]
28      )
29      if err != nil || offset < 0 {
30          fatalf("invalid offset: %v\nUsage: %s <file>
    <offset> <replacement>", err, os.Args[0])
31      }
32      // open the file for reading and writing
33      f, err := os.OpenFile(file, os.O_RDWR, 0)
34      if err != nil {
35          fatalf("open %s: %v\n", file, err)
36      }
37      defer f.Close()
38
39
40      // 2. Copy everything before the replacement from
    `file` to standard output (that is, `offset` bytes of the
    file)
41      _, err = io.CopyN(os.Stdout, f, offset)
42      if err != nil {
43          fatalf("copy: %v\n", err)
44      }
45
46      // we're now at the offset where we want to write the
    replacement chunk.
47      // 3. Write `replacement` to standard output
48      _, err = os.Stdout.Write([]byte(replacement))
49      if err != nil {
50          fatalf("write: %v\n", err)
51      }
52
53      // 4. Skip over the bytes we're replacing by throwing
    them away.
54      if _, err := io.CopyN(io.Discard, f,
    int64(len(replacement))); err != nil {
55          fatalf("copy: %v\n", err)
56      }
57
58      // 5. copy the rest of the file to standard output
59      _, err = io.Copy(os.Stdout, f)
```

```
60        if err != nil {
61            fatalf("copy: %v\n", err)
62        }
63    }
64
65    // fatalf prints an error message to stderr with
      fmt.Fprintf, then exits with status 1.
66    func fatalf(format string, args ...interface{}) {
67        fmt.Fprintf(os.Stderr, format, args...)
68        os.Exit(1)~
69    }
```

Let's try changing "brown" to "green" in our `fox.txt` file.

```
1   # IN
2   findoffset fox.txt "brown"
3   # OUT
4   10
```

And use that offset to patch the file...

```
1   # IN
2   binpatch fox.txt 10 "green"
3   # OUT
4   the quick green fox jumps over the lazy dog
```

Hacking a binary

It works! Let's try it on our `hello` program.

> **bash note**: the `$(...)` syntax in the bash shell is called "command substitution" - it runs the command inside the parentheses and replaces the expression with the output of the command. we can use this to feed the output of `findoffset` into `binpatch`.

```
1   # IN
2   binpatch hello $(findoffset hello "hello, world!") "hello,
    efron!" > hackedhello
3   ./hackedhello
4   # OUT
5   bash: ./hackedhello: Permission denied
```

Whoops, forgot to make `hackedhello` executable. Let's fix that.

> We'll cover file permissions later. For now, know that files can be READABLE (`r`), WRITABLE (`w`), and EXECUTABLE (`x`). The `chmod` command changes these permissions. `+x` makes a file executable, `-x` makes it non-executable, and `+r` makes it readable.

```
1   # IN
2   chmod +x hackedhello
```

```
3   ./hackedhello
4   # OUT
5   hello, efron!
```

It works! We've successfully changed the behavior of our program without recompiling it. **This is more hacking than most 'programmers' ever do in their entire careers - and we're just warming up**.

> BTW, anyone can do this. You probably should verify the integrity of your executables against some kind of checksum after you download them... but you probably don't.

You have the power to look inside programs and see what's there - or even to change them! If you take nothing else away from this, remember that - it's *not* a black box, it's *not* magic. They're just bytes.

When it comes down to it, programming is about **transforming data**. It's not about theories, paradigms, patterns - those things are handy, but they miss the point. Programming is about taking data, changing it, and producing new data. You can do this with a text editor, a hex editor, or a program - you can do it with the help of an IDE, a library, github copilot, whatever - but there's no substitute for that fundamental understanding.

Let's keep looking at the data inside our program. What's in the binary *around* what used to be "hello, world!"?

You know the drill - let's **write a program**, `torso` to find out.

> The unix coreutils `head` and `tail` programs read the beginning and end of a file, respectively. Usually you'd use those - torso is a play on those words, reading the 'middle' of a file.
>
> We'll cover arguments and flags in more detail later. For now, know that `-flag` is a common way to specify flags in unix programs.

4.6. reaching into files with `torso.go`

Overview

We need to:

1. define and parse command line flags:
    - choose a `-file` to read from
    - choose an `offset` to read from
    - choose how many bytes to read `-before` and `-after` the offset

2. skip to the first byte we want to read (`offset - before`)
3. copy (`before + after`) bytes to standard output
4. add a newline if requested

[`torso.py`: click here]

```
1
2   // torso reads the 'middle' of a file - the bytes around a
    given offset.
3   // it's not the head of the file, and it's not the tail -
    it's the torso.
4   // usage:
5   //
6   //  torso -offset n -before [b=128] -after [a=128] -from
    file [-newline]
7   //
8   // if no file is given, reads from standard input.
9   package main
10
11  import (
12      "flag"
13      "fmt"
14      "io"
15      "os"
16  )
17
18  func main() {
19      var offset, before, after int
20      var from string
21      var newline bool
22      { // 1. define and parse command line flags
23          flag.IntVar(&offset, "offset", -1, "offset to read
    from: must be specified")               // -offset is
    required
24          flag.IntVar(&before, "before", 128, "bytes to read
    before offset: will be clamped to 0")    // defaults to
    -before 128
25          flag.IntVar(&after, "after", 128, "bytes to read
    after offset: will be clamped to 0")       // defaults to
    -after 128
26          flag.StringVar(&from, "from", "", "file to read
    from: if empty, reads from standard input") // -from is
    required
27          flag.BoolVar(&newline, "newline", false, "append a
    newline to the output")                 // -newline is
    optional
28          flag.Parse()
29      }
30
31      // bounds checking and normalization
32      {
33          before = max(before, 0)      // can't be negative
```

```
34          before = min(before, offset) // can't go past the
    beginning
35          after = max(after, 0)
36          if offset < 0 {
37              fmt.Fprintf(os.Stderr, "missing or invalid
    -offset\n")
38              os.Exit(1)
39          }
40      }
41
42      start := offset - before // where to start?
43      n := before + after      // total number of bytes to
    read
44      if n == 0 {
45          return // nothing to do
46      }
47      buf := make([]byte, n)
48
49      // read from a file
50      f, err := os.Open(from)
51      if err != nil {
52          fmt.Fprintf(os.Stderr, "open: %s: %v\n", from, err)
53          os.Exit(1)
54      }
55      // 2. skip to the first byte we want to read (offset -
    before)
56      _, err = f.Seek(int64(start), io.SeekStart)
57      if err != nil {
58          fmt.Fprintf(os.Stderr, "seek: %s: %v\n", from, err)
59          // make sure to close the file before exiting!
60          // experienced go programmers will use 'defer()',
    but I want this to be accessible to non-go programmers.
61          f.Close()
62          os.Exit(1)
63      }
64
65      // 3. copy (before + after) bytes to standard output
66
67      // first read them into memory...
68      n, err = io.ReadFull(f, buf)
69      if err != nil && err != io.EOF && err !=
    io.ErrUnexpectedEOF {
70          fmt.Fprintf(os.Stderr, "read: %s: %v\n", from, err)
71          os.Exit(1)
72      }
73      buf = buf[:n]
74
75      // then write them to standard output
76      _, err = os.Stdout.Write(buf)
77      if err != nil {
78          fmt.Fprintf(os.Stderr, "write: %v\n", err)
79          f.Close()
80          os.Exit(1)
81      }
```

```
82
83          // 4. add a newline if requested
84          if newline {
85              fmt.Println()
86          }
87          f.Close()
88      }
```

```
1   #!/usr/bin/env bash
2   # IN
3   torso -offset $(findoffset hello "hello, world!") -before
    128 -after 128 -from hello
4   # OUT
5   mismatchwrong timersillegal seekinvalid slothost is downnot
    pollablegotypesaliashttpmuxgo121multipathtcprandautoseedtlsunsafeekmh
    3814697265625wakeableSleepprofMemActiveprofMemFuturetraceStackTabexec
    sweep waitSIGQUIT: qu
```

Looks like a mixture of error messages ("illegal seek", "host is down", "not pollable"), internal go runtime messages ("profMemActive", "profMemFuture"), and some kind of mysterious number, 3814697265625. In the next section, we'll investigate where that number comes from.

exercises

- modify `torso` to work on lines instead of bytes with a `-lines` flag.
- modify `torso` to work on words instead of bytes with a `-words` flag.
- modify `torso` to work on utf-8 codepoints instead of bytes with a `-runes` flag.
- modify `torso` to read from standard input if no `-from` file is given.

4.7. Investigation: What's with `3814697265625`?

If the string appeared in our **source code**, it's got to be in the **binary**. If it's in the binary... it's probably in the **source code**. Maybe in one of our imported packages? Let's look for it the go language source code. Let's **write a program** to find files that contain a string.

The classic unix tool `grep` is perfect for this. We'll **write a program** to do this... but that's for next time, when we've talked a bit more about files.

Let's use it to find the first appearance of "3814697265625" in the go source code for our version of go

```
1   # IN
2   git clone https://github.com/golang/go
3   cd go
4   git checkout v1.23 # or whatever version you're using
```

```
grep -r "3814697265625" .
# OUT
math/big/floatconv.go:  3814697265625,
strconv/decimal.go:     {6,
"3814697265625"},                               // * 262144
```

Which one shows up in our program, though? Let's use `findoffset` to find out.

```
# IN
# is it math/big?
findoffset hello "math/big" || echo "math/big not found"
findoffset hello "strconv" || echo "strconv not found"
# OUT
math/big not found
745064
```

It's **strconv**. Let's use `cat` again to see what's around it.

IN

```
// #!/usr/bin/env bash
cat strconv/decimal.go
```

OUT

```
// Cheat sheet for left shift: table indexed by shift count
giving
// number of new digits that will be introduced by that
shift.
//
// For example, leftcheats[4] = {2, "625"}.  That means
that
// if we are shifting by 4 (multiplying by 16), it will add
2 digits
// when the string prefix is "625" through "999", and one
fewer digit
// if the string prefix is "000" through "624".
//
// Credit for this trick goes to Ken.
type leftCheat struct {
    delta  int    // number of new digits
    cutoff string // minus one digit if original < a.
}

var leftcheats = []leftCheat{
    // Leading digits of 1/2^i = 5^i.
    // 5^23 is not an exact 64-bit floating point number,
    // so have to use bc for the math.
    // Go up to 60 to be large enough for 32bit and 64bit
platforms.
    /*
```

```
21            seq 60 | sed 's/^/5^/' | bc |
22            awk 'BEGIN{ print "\t{ 0, \"\" }," }
23            {
24                log2 = log(2)/log(10)
25                printf("\t{ %d, \"%s\" },\t// * %d\n",
26                    int(log2*NR+1), $0, 2**NR)
27            }'
28        */
29        {0, ""},
30    /* many entries omitted for space ...*/
31        {6, "3814697265625"},                                  //
      * 262144
32    /* many entries omitted for space ...*/
33        {19, "867361737988403547205962240695953369140625"}, //
      * 1152921504606846976
34    }
```

This turns out to be an sophisticated bit of systems programming by the master Ken Thompson himself! We'll take another look at this later... consider it a sneak peak ;).

> **Ken Thompson**
>
> Kenneth Lane Thompson (B: 1943) created Unix, `grep`, `Go`, and a zillion other tools that form as a foundation of every nearly every computer system in the world. He didn't do it himself, but he's legendarily productive - he famously implemented unix pipes over a lunch break. If you've ever used a regular expression, a mac or linux computer, or a smart phone, you've used his work.

That's enough fiddling with the data segment (for now). Let's see if we can get some insight into the rest of the program.

5. Investigating the code segment.

Let's use `torso` to take a peek at the first 256 bytes of our program.

```
1   # IN
2   torso -offset 0 -after 256 -from hello
3   # OUT
4   □ELF□□□
```

That seems like a lot fewer bytes than we expected. What's going on? The shell isn't sure how to interpret binary data. **ASCII** - the character encoding used by nearly every shell since the 1960s - covers 128 characters. Some of these are printable, like 'a' or '|' - but many are not. The first 32 characters are **control characters** - they *control* the terminal, rather than printing anything. The first 256 bytes of our program must contain a lot of these - and they're confusing the shell.

To look deeper, we need to **serialize** the bytes into a human-readable format. The most common way to do this is to print the bytes as **hexadecimal**. Let's **write a program** to to do this - a take on the classic unix tool `hexdump`.

5.1. reading binary files with `shexdump.go`

This tool will be our first foray into manipulating binary data, a core systems programming skill. We'll read the file in chunks, convert each byte to a pair of hexadecimal digits, and print them to standard output. We'll call it `shexdump` (**s**imple **hex**adecimal **dump**) to distinguish it from the classic `hexdump`.

It would be nice if we could output the hexadecimal bytes in a more readable format - say, 16 bytes per line, with a space between each byte and a newline after every 16 bytes.

What will we need to do?

1. choose the input source: stdin or a file
2. read a chunk of bytes (up to 16) from the input
3. convert each byte in the chunk to a pair of hexadecimal digits
4. print the hexadecimal digits to standard output, space-separated, terminating with a newline

`hexdump.py`: click here

```
 1  // shexdump.go dumps the input as pairs of space-separated
    hexadecimal bytes, with a newline after every 16 bytes.
 2  //
 3  // # example
 4  //  //  #!usr/bin/env/bash
 5  //  //  echo "now is the time for all good men to come to
    the aid of their country" | shexdump
 6  //  //  6e 6f 77 20 69 73 20 74  68 65 20 74 69 6d 65 20
 7  //  //  6f 66 20 61 6c 6c 20 67  6f 6f 64 20 6d 65 6e 20
 8  //  //  74 6f 20 63 6f 6d 65 20  74 6f 20 74 68 65 20 61
 9  //  //  69 64 20 6f 66 20 74 68  65 69 72 20 63 6f 75 6e
10  //  //  74 72 79 20 0a
11
12  package main
13
14  import (
15      "bufio"
16      "fmt"
17      "io"
18      "os"
19  )
20
21  func main() {
22      // 1. choose the input source: stdin or a file
23      var src io.Reader
24      switch len(os.Args) {
```

```
25      case 1:
26          src = os.Stdin
27      case 2:
28          f, err := os.Open(os.Args[1])
29          if err != nil {
30              fmt.Fprintf(os.Stderr, "open %s: %v",
    os.Args[1], err)
31              os.Exit(1)
32          }
33          defer f.Close()
34          src = f
35      default:
36          fmt.Fprintf(os.Stderr, "Usage: %s [filename]",
    os.Args[0])
37          os.Exit(1)
38      }
39      if err := hexdump(os.Stdout, src); err != nil {
40          fmt.Fprintf(os.Stderr, "hexdump: %v", err)
41          os.Exit(1)
42      }
43  }
44
45  // dump the contents of r to w in a hexdump format.
46  func hexdump(dst io.Writer, src io.Reader) error {
47      // performance: small reads and writes are very
    inefficient. while we could write a byte at a time, it's
    much faster to read and write in chunks.
48      r := bufio.NewReader(src)
49      defer w.Flush()
50      for { // 2. read a chunk of bytes (up to 16) from the
    input
51
52          var raw [16]byte // read 16 bytes at a time
53
54          encoded := make([]byte, 0,
    16*3+1+1) // 16 bytes, 3 characters per byte, 1 space
    between bytes, newline at the end.
55          n, err := io.ReadFull(r, raw[:])
56
57          // 3. convert each byte in the chunk to a pair of
    hexadecimal digits
58          const hex = "0123456789abcdef"
59          if n != 0 {
60              for i := range min(n, 8) {
61                  encoded = append(encoded, hex[raw[i]>>4],
    hex[raw[i]&0x0f], ' ')
62              }
63              encoded = append(encoded, ' ')
64              for i := 8; i < min(n, 16); i++ {
65                  encoded = append(encoded, hex[raw[i]>>4],
    hex[raw[i]&0x0f], ' ')
66              }
67              encoded[len(encoded)-1] = '\n'
68
```

```
69                // 4. print the hexadecimal digits to standard
    output, space-separated, terminating with a newline
70                if _, err := w.Write(encoded); err != nil {
71                    return err
72                }
73            }
74            if err == io.ErrUnexpectedEOF {
75                return nil
76            } else if err != nil {
77                return err
78            }
79        }
80    }
```

Exercises

- Add an `-offset` flag to prefix each line with the offset in the file in hexadecimal, terminating with the ending offset of the line (default false)
- Add a `-columns` flag to specify the number of columns to print per line (default 2)
- Add a `-column-width` flag to specify the number of bytes to print per column. (default 8)
- Add a `-squeeze` flag to compress multiple consecutive identical lines into a single line with a `*` and a count (default false)
- Add an `-ascii` flag to suffix each line with the ASCII representation of the bytes (default false). Replace non-printable or non-ASCII bytes with a `.`.

```
1
2   #!/usr/bin/env bash
3   # IN
4   echo -n "now is the time " | shexdump -ascii
5   # OUT
6   6e 6f 77 20 69 73 20 74  68 65 20 74 69 6d 65 20  |now
    is the time |
```

- Add a `-canon` flag to print the output in canonical hex+ASCII format, like `hexdump -C` (default false). **This combines the -ascii and -offset flags and should be exclusive with them.**

```
1
2   #!/usr/bin/env bash
3   // # IN
4   echo
    "now is the time of all good men to come to the aid of
    their country " | shexdump -canon
```

```
1
```

```
# OUT
00000000  6e 6f 77 20 69 73 20 74  68 65 20 74 69 6d 65 20  |now is the time |
00000010  6f 66 20 61 6c 6c 20 67  6f 6f 64 20 6d 65 6e 20  |of all good men |
00000020  74 6f 20 63 6f 6d 65 20  74 6f 20 74 68 65 20 61  |to come to the a|
00000030  69 64 20 6f 66 20 74 68  65 69 72 20 63 6f 75 6e  |id of their coun|
00000040  74 72 79 20 0a                                    |try .|
00000045
```

How do we test if we've written this correctly? We should be able to losslessly convert the output of `hexdump` back into the original file. Let's **write a function** to do that - and **write a program**, `unhexdump`, to use it.

5.2. deserializing hexdumps with `unhexdump.go`

overview

Our last program, `hexdump` *read* binary and write whitespace-separated pairs of hexadecimal bytes.

This program, `unhexdump` should *read* whitespace-separated pairs of hexadecimal bytes and write the original binary. In other words:

1. choose the input source: stdin or a file
2. read pairs of whitespace-separated hexadecimal bytes from a file
3. convert each pair back to a byte, `unhex`-ing it
4. write that `unhex`-ed byte to standard output

program: `unhexdump.go`

`unhexdump.py`: click here

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
)

```

```
10  // unhexdump.go reverses the process of hexdump, converting
    a hexdump back into a file.
11  // it expects pairs of whitespace-separated hexadecimal
    bytes.
12  func main() {
13      // 1. choose the input source: stdin or a file
14      var src io.Reader
15      switch len(os.Args) {
16      case 1:
17          src = os.Stdin
18      case 2:
19          f, err := os.Open(os.Args[1])
20          if err != nil {
21              fmt.Fprintf(os.Stderr, "open %s: %v",
    os.Args[1], err)
22              os.Exit(1)
23          }
24          defer f.Close()
25          src = f
26      default:
27          fmt.Fprintf(os.Stderr, "Usage: %s [filename]",
    os.Args[0])
28          os.Exit(1)
29      }
30
31      if err := unhexdump(os.Stdout, src); err != nil {
32          fmt.Fprintf(os.Stderr, "unhexdump: %v", err)
33          os.Exit(1)
34      }
35  }
36
37  // unhexdump reads pairs of whitespace-separated
    hexadecimal bytes from r and writes the corresponding bytes
    to w.
38  func unhexdump(w io.Writer, r io.Reader) error {
39      // 2. read pairs of whitespace-separated hexadecimal
    bytes from the input.
40
41      // we're going to start caring a little more about
    performance here. we'll use a buffered reader and writer to
    reduce the number of system calls & allocations (more about
    these topics in later articles).
42      scanner := bufio.NewScanner(r)
43      scanner.Split(bufio.ScanWords)
44      bw := bufio.NewWriter(w)
45      defer bw.Flush()
46      for i := 0; scanner.Scan(); i++ {
47          b := scanner.Bytes()
48          if len(b)&1 == 1 { // odd number of hex digits
49              return fmt.Errorf("odd number of hex digits at
    position %d (%q)", i)
50          }
51          // 3. convert each pair back to a byte, unhex-ing
    it
```

```
52            for i := 0; i < len(b); i += 2 {
53                high, ok := unhex(b[i])
54                if !ok {
55                    return fmt.Errorf("bad hex %x '%c' at
    position %d", b[i], b[i], i)
56                }
57                low, ok := unhex(b[i+1])
58                if !ok {
59                    return fmt.Errorf("bad hex %x '%c' at
    position %d", b[i+1], b[i+1], i+1)
60                }
61
62                // 4. write that unhex-ed byte to standard
    output
63                if err := bw.WriteByte(high<<4 | low); err !=
    nil {
64                    return err
65                }
66            }
67
68        }
69        return scanner.Err()
70    }
71
72    // unhex converts a hexadecimal character to it's value
    (0x0-0xf),
73    // or 0, false if the character is not a valid hexadecimal
    digit.
74    func unhex(b byte) (byte, bool) {
75        switch {
76        case '0' <= b && b <= '9':
77            return b - '0', true
78        case 'a' <= b && b <= 'f':
79            return b - 'a' + 10, true
80        case 'A' <= b && b <= 'F':
81            return b - 'A' + 10, true
82        default:
83            return 0, false
84        }
85    }
86
```

Let's test these programs by hexdumping a file, then unhexdumping it.

5.3. hexdump_test.bash

overview

1. write a file, `moby.txt`
2. hexdump it to `moby.hex`

3. unhexdump it to `moby2.txt`
4. compare the files (we'll use `diff`, a classic unix tool, to do this)

```
#!/usr/bin/env bash
# IN
echo "to the last I grapple with thee; from hell's heart I stab at thee; for hate's sake I spit my last breath at thee" > moby.txt # 1. write a file

shexdump moby.txt > moby.hex # 2. hexdump it
unhexdump moby.hex > moby2.txt # 3. unhexdump it
diff -s moby.txt moby2.txt # 4. compare the files
```

```
# OUT:
Files moby.txt and moby2.txt are identical
```

Looks good. As a final exercise before we close this article out, let's use `torso` and `hexdump` to investigate the first bytes of our 'hello' program and see what we can learn.

5.4. Investigating the ELF header of `hello`

Programs in Linux are stored in the `ELF` (Executable and Linkable Format), which starts with a header that describes the rest of the file. This is a common way to organize file formats - a brief header explaining the rest. Let's use ELF as an example header.

```
#!/usr/bin/env bash
# IN
cat hello | torso -offset 0 -after 16 | shexdump
# OUT
7f  45  4c  46  02  01  01  00  00  00  00  00  00  00  00  00
02  00  3e  00  01  00  00  00  c0  cf  46  00  00  00  00  00
```

`7f` is meaningless to us - it's not ASCII - but `45 4c 46` is `E L F` in ASCII. Beyond that, it's mostly opaque binary - but we can already figure out why `hello` stopped printing with `cat` given our new knowledge of C-strings. The 8th byte of `hello` is a null, which terminates the string and got our shell to stop printing. Let's look at what wikipedia says about the ELF header to learn more.

The **object file type** is identified with a special byte at offset 0x10:

```
# IN
cat hello | torso -offset 0x10 -before 0 -after 1 | shexdump
# OUT
02
```

Value	Type	Meaning
0x00	ET_NONE	Unknown.
0x01	ET_REL	Relocatable file.

Value	Type	Meaning
0x02	**ET_EXEC**	**Executable file.**
0x03	ET_DYN	Shared object.
0x04	ET_CORE	Core file.

Unsurprisingly, our executable is an executable. We can take a peek at our architecture with the **machine type** at offset 0x12.

```
1  # IN
2  cat hello | torso -offset 0x12 -before 0 -after 1 | shexdump
3  # OUT
4  3e
```

This is the **x86-64** architecture. The **endianness** of the file is stored at offset 0x05.

That's great, but where does our program *actually start*? The **entrypoint** is the place in memory where the program starts executing - the initial value of the **instruction pointer**. This is stored in the ELF header, **little-endian**, at offset 0x18.

```
1  # IN
2  cat hello | torso -offset 0x18 -before 0 -after 8 | shexdump
3  # OUT
4  c0  cf  46  00  00  00  00  00
```

Looks like we'll start executing instructions at memory address `0x46cfc0`. More about that later.

That's all the digging we'll do for now - this article's already over twenty pages long! If you made it through - and especially if you did the exercises - congratulations and thank you for sticking with me.

A couple of final notes before we close out:

> **Lemma: Instruction Pointer**
>
> At the most fundamental level, a program works like this:
>
> - read an instruction at the memory address pointed to by the **instruction pointer**
> - do what the instruction says
> - increment the instruction pointer by the size of the instruction
>
> The **entrypoint** is where the **instruction pointer** starts.

Lemma: Endianness

When we want to convert a list of bytes to an integer, there's two ways to do it: either the most significant byte comes first (big-endian), or the least significant byte comes first (little-endian). The x86-64 architecture is little-endian, so the bytes `c0 cf 46 00 00 00 00 00` are 'reversed' when converted to an integer; they're the number `0x0046cfc0` (4640704) in little-endian, but `0xc0cf46000000000` (13893400341275213824) in big-endian. Since we know that this represents a **memory address**, we can figure out that this is a little-endian machine just by looking at the bytes - no machine ever made has 12635974 terabytes of memory, so you're not going to see that in a memory address. Knowing a bit about how computers *actually work* can help you understand what you're looking at.

Exercise: **Write a program** to read a 64-bit little-endian integer from a file starting at a given-offset and print it "big-endian" (most significant byte first).

———————

- 1. Series Introduction
  - 1.1. Programmers Write Programs
    - Note on style & environment
    - note for python programmers
    - example go block
    - example bash block
  - 1.2. Some final caveats:
  - 1.3. Series Overview
- 2. What is systems programming?
- 3. Peeking into the black box: what is a program, anyways?
  - 3.1. hello.go
    - overview
  - 3.2. buildhello.bash
    - Overview
- 4. Investigating the data segment
  - 4.1. finding strings with `findoffset.go`
    - Overview
    - Used in this example:
    - findoffset.py: click here
  - 4.2. writing simple files with `echo.go`
    - Overview
  - 4.3. printing files with `cat.go`
    - overview
    - exercises
  - 4.4. investigating the `hello` program with `findoffset`, `echo`, and `cat`
    - bash script: `catfox.bash`

    - 4.5. Basic Hacking w/ `binpatch.go`
        - Overview
        - Hacking a binary
    - 4.6. reaching into files with `torso.go`
        - Overview
    - 4.7. Investigation: What's with `3814697265625`?
        - IN
        - OUT
- 5. Investigating the code segment.
    - 5.1. reading binary files with `shexdump.go`
        - Exercises
    - 5.2. deserializing hexdumps with `unhexdump.go`
        - overview
        - program: `unhexdump.go`
    - 5.3. hexdump_test.bash
        - overview
    - 5.4. Investigating the ELF header of `hello`
- 6. Conclusion: The Spirit of Systems Programming. - Sidenote: The origin of endianness

6. Conclusion: The Spirit of Systems Programming.

Hopefully I've given you a 'taste' of systems programming. A summary of the "mindset" or "spirit" behind this article would be:

- Build your own tools.
- Look at the data with your own eyes.
- Understand the system rather than relying on abstractions.
- It's all just bytes.
- **Programmers write programs**.

That's all for now! In the next article, Starting Systems 2: Your Program and the Outside World, we'll look at how programs *actually do things* - how they interact with the outside world via files and system calls, how they manage memory, environment variables and command-line arguments, standard input and output, and more.

MORE ARTICLES

Sidenote: The origin of endianness

> The term "endianness" is a reference to Gulliver's Travels. It comes from "ON HOLY WARS AND A PLEA FOR PEACE" by Danny Cohen on April 1, 1980: the internet archive has a copy of the original post. I reproduce the relevant section of gulliver's travels here:

It began upon the following occasion.

It is allowed on all hands, that the primitive way of breaking eggs before we eat them, was upon the larger end: but his present Majesty's grandfather, while he was a boy, going to eat an egg, and breaking it according to the ancient practice, happened to cut one of his fingers. Whereupon the Emperor his father published an edict, commanding all his subjects, upon great penalties, to break the smaller end of their eggs.
The people so highly resented this law, that our Histories tell us there have been six rebellions raised on that account, wherein one Emperor lost his life, and another his crown. These civil commotions were constantly formented by the monarchs of Blefuscu, and when they were quelled, the exiles always fled for refuge to that Empire.

It is computed, that eleven thousand persons have, at several times, suffered death, rather than submit to break their eggs at the smaller end. Many hundred large volumes have been published upon this controversy: but the books of the Big-Endians have been long forbidden, and the whole party rendered incapable by law of holding employments.

During the course of these troubles, the emperors of Blefuscu did frequently expostulate by their ambassadors, accusing us of making a schism in religion, by offending against a fundamental doctrine of our great prophet Lustrog, in the fifty-fourth chapter of the Brundecral (which is their Alcoran). This, however, is thought to be a mere strain upon the text: for their words are these; That all true believers shall break their eggs at the convenient end: and which is the convenient end, seems, in my humble opinion, to be left to every man's conscience, or at least in the power of the chief magistrate to determine.