Data Science in Ten Minutes*

Spencer Tipping

May 19, 2018

Contents

1 Introduction

Data science is not machine learning. There is no machine learning in this guide, because machine learning is the wrong answer to most problems you're likely to run into. I've never deployed a machine learning system, although I've debugged several misbehaving ones (and will cover how to do that).

A lot of the code examples assume you're running Linux because Linux is the go-to platform for medium/big data processing. It's also ideal for small data due to optimizations in core utilities like `sort`.[1] If you're running a non-Linux OS and want to try stuff, you have a few options:

- Install Docker and create a container from the `ubuntu:18.04` image or similar (I'll assume you have this setup)
- Run a VM like VirtualBox and install Ubuntu Desktop inside it
- Rent a free tier Amazon EC2 instance (you can use the free-tier `t2.nano` or `t2.micro` for data science, and they're ideal for learning because they're resource-constrained)

*for extremely large values of ten

[1] GNU `sort` will compress temporary files, unlike the `sort` that ships with Mac OSX for instance.

- Buy a cheap rack server from eBay and drop Ubuntu Server on it

Realistically, you probably won't want to use OSX or Windows for distributed programming: your binaries won't be portable to cluster machines, and you'll be fighting with things like case-insensitive filesystems, slower core utilities, and general inconsistencies that will increase debugging time.[2]

1.1 Examples

This guide comes with example code in the original repository. Where necessary, I've included package installation commands required to install dependencies on Ubuntu 18.04.

[2] All of this is a non-issue for JVM processes, e.g. Hadoop and Spark, but a lot of data science is best done with native tools that use less memory and have lower iteration time.

2 Linux

Oh yes, we are totally going here. Here's why.

Backend programs and processing pipelines and stuff (basically, "big data" things) operate entirely by talking to the kernel, which, in big-data world, is usually Linux; and this is true regardless of the language, libraries, and framework(s) you're using. You can always throw more hardware at a problem[3], but if you understand system-level programming you'll often have a better/cheaper option.

Some quick background reading if you need it for the homework/curiosity:

- How to write a UNIX shell
- How to write a JIT compiler
- ELF executable binary spec (more readable version on Wikipedia, and this StackOverflow answer may be useful)
- Intel machine code documentation

2.1 Virtual memory

This is one of the two things that comes up a lot in data science infrastructure. Basically, any memory address you can see is virtualized and may not be resident in the RAM chips in your machine. The kernel talks to the MMU hardware to maintain the mapping between software and hardware pages, and when the virtual page set overflows physical memory the kernel swaps them to disk, usually with an LRU strategy.

Here's where things get interesting. Linux (and any other server OS) gives you a `mmap` system call to request that the kernel map pages into your program's address space. `mmap`, however, has some interesting options:

- `MAP_SHARED`: map the region into multiple programs' address spaces (this reuses the same physical page across processes)
- `MAP_FILE`: map the region using data from a file; then the kernel will load file data when a page fault occurs

`MAP_FILE` is sort of like saying "swap this region to a specific file, rather than the shared swapfile you'd normally use." The implication is important, though: *all memory mappings go through the same page allocation cache*, and any page fault has the potential to block your program on disk IO. Clever data structures like Bloom filters, Count-min sketches, and so forth are all designed to give you a way to trade various degrees of accuracy for a much smaller memory footprint.

Sometimes you won't have any good options within the confines of physical RAM, so you'll end up using IO devices to supply data; then the challenge

[3] Until you can't

becomes optimizing for those IO devices (SSDs are different from HDDs, for instance). I'll get to some specifics later on when we talk about sorting, joins, and compression.

2.2 File descriptors

This is the other thing you need to know about.

Programs don't typically use `mmap` for general-purpose IO. It's more idiomatic, and sometimes faster, to use `read` and `write` on a file descriptor. Internally, these functions ask the kernel to copy memory from an underyling file/socket/pipe/etc into mapped pages in the address space. The advantage is cache locality: you can `read` a small amount of stuff into a buffer, process the buffer, and then reuse that buffer for the next `read`. Cache locality does matter; for example:

```
# small block size: great cache locality, too much system calling overhead
$ dd if=/dev/zero count=262144 bs=32768 of=/dev/null
262144+0 records in
262144+0 records out
8589934592 bytes (8.6 GB, 8.0 GiB) copied, 0.774034 s, 11.1 GB/s

# medium block size: great cache locality, insignificant syscall overhead
$ dd if=/dev/zero count=8192 bs=1048576 of=/dev/null
8192+0 records in
8192+0 records out
8589934592 bytes (8.6 GB, 8.0 GiB) copied, 0.574597 s, 14.9 GB/s

# large block size: cache overflow, insignificant syscall overhead
$ dd if=/dev/zero count=2048 bs=$((1048576 * 4)) of=/dev/null
2048+0 records in
2048+0 records out
8589934592 bytes (8.6 GB, 8.0 GiB) copied, 1.14022 s, 7.5 GB/s
```

2.3 Pulling this together: let's write a program

...in machine language. For simplicity, let's write one that prints `hello world` and then exits successfully (with code 0).

This is also a good opportunity to talk about how we might generate and work with binary data with things like fixed offsets. Two simple functions for this are `pack()` and `unpack()`, variants of which ship with both Perl and Ruby.

The first part of any Linux executable is the ELF header, usually followed directly by a program header; here's what those look like as C structs for 64-bit executables (reformatted slightly, and with docs for readability):

```
typedef struct {
  unsigned char e_ident[16];    // 0x7f, 'E', 'L', 'F', ...
```

```
  uint16_t      e_type;         // ET_EXEC = 2 for executable files
  uint16_t      e_machine;      // EM_X86_64 = 62 for AMD64 architecture
  uint32_t      e_version;      // EV_CURRENT = 1
  uint64_t      e_entry;        // virtual address of first instruction
  uint64_t      e_phoff;        // file offset of first program header
  uint64_t      e_shoff;        // file offset of first section header
  uint32_t      e_flags;        // always zero
  uint16_t      e_ehsize;       // size of the ELF header struct (this one)
  uint16_t      e_phentsize;    // size of a program header struct
  uint16_t      e_phnum;        // number of program header structs
  uint16_t      e_shentsize;    // size of a section header struct
  uint16_t      e_shnum;        // number of section header structs
  uint16_t      e_shstrndx;     // string table linkage
} Elf64_Ehdr;

typedef struct {
  uint32_t p_type;              // the purpose of the mapping
  uint32_t p_flags;             // permissions for the mapped pages (rwx)
  uint64_t p_offset;            // file offset of the first byte of data
  uint64_t p_vaddr;             // virtual memory offset of the data (NB below)
  uint64_t p_paddr;             // physical memory offset (usually zero)
  uint64_t p_filesz;            // number of bytes from the file
  uint64_t p_memsz;             // number of bytes to be mapped into memory
  uint64_t p_align;             // segment alignment
} Elf64_Phdr;
```

If we want a very minimal executable, here's how we might write these headers from Perl:

```
# elf-header.pl: emit an ELF binary and program header to stdout
use strict;
use warnings;

print pack('C16 SSL',
           0x7f, ord 'E', ord 'L', ord 'F',
           2, 1, 1, 0,
           0, 0, 0, 0,
           0, 0, 0, 0,

           2,                           # e_type    = ET_EXEC
           62,                          # e_machine = EM_X86_64
           1)                           # e_version = EV_CURRENT

    . pack('QQQ',
           0x400078,                    # e_entry = 0x400078
           64,                          # e_phoff
```

```
        0)                              # e_shoff

    . pack('LSS SSSS',
        0,                              # e_flags
        64,                             # e_ehsize
        56,                             # e_phentsize
        1,                              # e_phnum

        0,                              # e_shentsize
        0,                              # e_shnum
        0)                              # e_shstrndx

    . pack('LLQQQQQ',
        1,                              # p_type = PT_LOAD (map a region)
        7,                              # p_flags = R|W|X
        0,                              # p_offset (must be page-aligned)
        0x400000,                       # p_vaddr
        0,                              # p_paddr
        0x1000,                         # p_filesz: 4KB
        0x1000,                         # p_memsz: 4KB
        0x1000);                        # p_align: 4KB
```

Now we can generate the ELF header:

```
$ sudo apt install perl                 # if perl is missing
$ perl elf-header.pl > elf-header
```

You can verify that the header is correct using `file`, which reads magic numbers and tells you about the format of things:

```
$ sudo apt install file
$ file elf-header
elf-header: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically
linked, corrupted section header size
```

Awesome, now let's get into the machine code.

2.4 `hello world` in machine code

The first thing to note is that we're talking directly to the kernel here. Our ELF header above is very minimalistic, with no linker instructions or anything else to complicate things. So we have none of the usual libc functions like `printf` or `exit`; it's up to us to define those in terms of Linux system calls.

There are two calls we'll use for this: `write(2)` and `exit(2)`. You can view the documentation for these using `man`:

```
$ sudo apt install man manpages-dev
$ man 2 write
$ man 2 exit
```

We'll need to pass arguments in registers to match the kernel calling convention. I'll spare you the gory details and cut straight to the machine code:

```
# elf-hello.pl: emit machine code for hello world, then exit successfully
use strict;
use warnings;
print pack('H*', join '',
  '4831c0',                             # xorq %rax, %rax
  'b001',                               # movb $01, %al (1 = write syscall)
  'e80c000000',                         # call %rip+12 (jump over the message)
  unpack('H*', "hello world\n"),        # the message
  '5e',                                 # pop message into %rsi (buf arg)
  'ba0c000000',                         # movl $12, %rdx (len arg)
  '48c7c701000000',                     # movl $1, %rdi (fd arg)
  '0f05',                               # syscall instruction

  '4831c0',                             # xorq %rax, %rax
  'b03c',                               # movb $3c, %rax (3c = exit syscall)
  '4831ff',                             # xorq %rdi, %rdi (exit code arg)
  '0f05');                              # syscall instruction
```

Now we can build the full executable, verify it, and run:

```
$ sudo apt install nasm
$ perl elf-hello.pl | cat elf-header - > elf-hello
$ chmod 755 elf-hello
$ tail -c+121 elf-hello | ndisasm -b 64 -
00000000  4831C0            xor rax,rax
00000003  B001              mov al,0x1
00000005  E80C000000        call 0x16
0000000A  68656C6C6F        push qword 0x6f6c6c65   # corruption from message
0000000F  20776F            and [rdi+0x6f],dh       # (which we jumped over)
00000012  726C              jc 0x80
00000014  640A5EBA          or bl,[fs:rsi-0x46]
00000018  0C00              or al,0x0
0000001A  0000              add [rax],al
0000001C  48C7C701000000    mov rdi,0x1             # now we're back on track
00000023  0F05              syscall
00000025  4831C0            xor rax,rax
00000028  B03C              mov al,0x3c
0000002A  4831FF            xor rdi,rdi
0000002D  0F05              syscall
```

The moment we've been waiting for:

```
$ ./elf-hello
hello world
$ echo $?                                  # check exit status
0
```

Whether through libraries, JIT, or anything else, this is the exact mechanism being used by any program that performs IO of any sort: the program lives in a completely virtual world and interacts with the kernel using the `0f05` syscall instruction, referring to virtual addresses in the process.

2.5 Homework (if that's your thing)

1. Write an ELF Linux executable that consumes data from `stdin` and writes that data to `stdout`, then exits successfully. In other words, `cat` without file support.