Data Science in Ten Minutes[*]

Spencer Tipping

May 19, 2018

Contents

1 Introduction

Data science is not machine learning.

[*]for extremely large values of ten

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[1], 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
- 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, , 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 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.

[1]Until you can't

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.

2.4 Homework (if that's your thing)

1. Write an ELF Linux executable that prints hello world to stdout, then exits successfully.