9 Tuesday, October 1, 2013

9.1 Announcements

Make sure your code is portable. If it's running on a Mac, but not on Linux, then something's wrong with your code.

9.2 IO Caching

If you remember, we ran several programs that wrote files in different ways, but have the same skeleton.

9.3 Differences between I/O calls

The stuff at the top opens the file. (int fd = ... exit(1))

isatty ("is a tty") always returns false.

STDOUT_FILENO is always 1.

STDIN, STDOUT, STDERR are the standard streams that are open for every program. Their file descriptor numbers are 0,1,2, respectively.

File	File descriptor
stdin	0
stdout	1
stderr	2

If you run w01sync-byte, it will write a bunch of 6's to whatever file you redirect its output to. If you don't redirect, it just prints 6's on your screen. This isn't very useful, so it all gets put in a file called "data".

The loop goes to 512000. Every iteration of the loop tries to write 1 byte, and prints an error if that write fails.

Question: Was file a preexisting file, or will it make one?

Answer: One of the flags was "O_CREAT," which creates a file. It also has the flag "O_TRUNC," which deletes the file if it already exists.

w01-syncbyte runs at about 2900 bytes/sec.

w02-syncblock gets about 1.18 million bytes/sec

What's the difference between these two? The size of the write. The size in syncblock is 512 bytes

syncbyte	1B writes
syncblock	512B writes

Let's come up with a formula/rule of thumb If we have a series of N requests, each of size U units, then the total cost of the series of requests is

$$NR + NUK$$

where R is the per request cost, and K is the per unit cost. If we use this sort of framework, the difference is that requests have more units in them. The unit of rsyncbyte is 1. The unit for syncblock is 512. So in order to do the same number of writes, you need 1/512 as many requests. So the cost for syncbyte $= NR_{\text{syncbyte}} + N \cdot 1 \cdot K$. For syncblock, it's $\frac{N}{512} R_{\text{syncblock}} + \frac{N}{512} \cdot 512K$ We are assuming that the cost to write a byte and syncronize it is constant in both of these programs.

What does this say about R_{syncbyte} and $R_{\text{syncblock}}$? The difference comes from the per request cost. If you divide these throughput numbers (on the spreadsheet), you get 421, which isn't too far from 512. So it looks like the cost to call one write call is about the same no matter what you're writing.

This is called batching – making fewer requests, but each request containing more data. Batching speeds up a series of requests by reducing the total per-request cost by grouping data into fewer requests.

What per-requests costs dominate? Not opening and closing the file since those happen only once.

This is synchronous I/O. This means we go to the disk immediately Synchronous I/O – I/O that accesses underlying media directly. This meanst hat in addition to accessing the operating system, there is the cost of going to the underlying disk (flash memory in this case). If you run it by leaving off the sync (using "diff"),

Aside: You always give diff a "-u" because it is the best THere are 4 types of lines. -,+, ,@

@ gives you context

- only in the left hand file (passed into diff)

+ only in the right hand file

w03-byte gives 900000 Bytes/sec

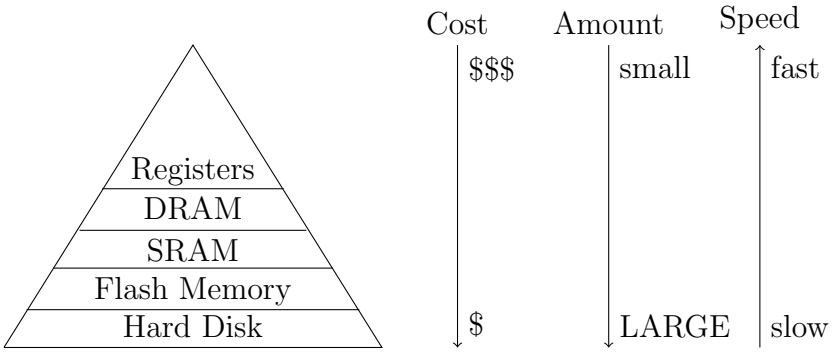
w04-block give 29 millino Bytes/sec

What's the ratio between byte and block? About 32x. It is no longer 512 times faster. This tells us we can no longer assume the per-unit costs are the same. It also tells us that the cost per request is no longer the same. But we have made things radically faster.

9.4 Cache

How does the OS handle byte and block reading so much faster than byte and block writing r03-byte runs at about 3.4 million. r04-block runs at babout 38 million But if you run r04 on a different file, it gets slower. The first data file was fastest. If you run it again with the new file, it gets faster. If you run it more times, it gets even faster. Why was the data512 file slow at first? The first one was stored in a faster peice of memory called the buffer cache.

Cache Hierarchy



Question: How many registers are there on a 32-bit machine?

Answer: Roughly 8. Some of them are special purpose. You get to use roughly 6. All the work your programs do is being funnelled through those 6 registers. This is why we're thankful for compilers.

A cache is fast storage used for copies of data primarily stored on slow storage used to speed up future requests. The system needs to prepare a cache so that it's useful for future requests.

The buffer cache is the operating system's memory cache for files. It's a cache the operating system uses to speed up access to files. If we look at the storage pyramid, files are stored in disk and flash memory. Why are they stored here? Because they are non-volatile, i.e. data stored there persists after poweroff. The operating system's wants to use all of memory for the buffer cache. If we run one of these files multiple times, the OS is using memory to store a copy of that file. When you run r04 on different files, the operating system kicks out the other file.

The operating system mediates access to other hardware. It doesn't allow an application to directly talk to the disk. It checks the application's requests and turns them into disk requests.

9.5 Analogies for how the buffer cache works:

The operating system uses the rest of the machine's memory for the buffer cache What's the goal of the operating system? To make the applications run as fast as possible. The application makes read and write requests, and the operating system's goal is to make sure the buffer cache has the right stuff to make sure it doesn't have to go to disk to satisfy a request. The policies that the operating system puts in place for reads are different from writes. It can satisfy a write request without knowing what was there before. To satisfy a read request, the os needs to know what was on disk.

How do we satisfy write requests? The operating system saves requests to the buffer cache and waits for free time to write it to disk. It fills up the cache slots $A - L$ with data. Where does M go now that the buffer cache is full? We should force everything to be written to the disk. This makes more room for the new data. As the buffer cache is being filled, the OS is writing to disk in parallel. When the response comes back that A was written, the OS can free that part of the buffer cache and reuse it for M.

9.6 Reading

read(0)

This is a 1 char read. There is nothing the buffer cache. It would be slow if the OS only sends a request for that one byte. The next byte will be at offset 1, and that is in the buffer cache. The disk operates in units of 512 bytes. This is the sector size. So the cache will be useful 511 times. Eventually we get to read(512), which is not in the buffer cache, so we need to pass down another read request. This is still really slow. This is equivalent to syncblock. Regular block is even faster. The OS is guessing about which blocks will be useful in the future. Because our programs read sequentially, they are very predictable. This is called sequential I/O.

What would be a better caching strategy?

Reading the entire size of the buffer cache is rather aggressive and will slow stuff down. Reading some fraction of memory is not a bad policy. When the OS sees read(0), it might ask for a read of 32 KB which fills the buffer cache with info that would be useful in the future. Based on the tests, the OS is reading about 20 blocks.

Question: How big is the buffer cache?

Answer: The size of memory. Everything besides the OS and the applications is the buffer cache.

OS policy: Prefetching

Prefetching is a cache policy where we load data into the cache that is predicted to be useful later. This is also called readahead. Is this always a good idea? No. You might not want to read sequentially or read the entire file.

r09-revbyte uses "lseek"

Repositions the offset of the opened file to "offset." We are setting it to move 1 byte backward. The program is reading this file in reverse order. In forward order gets 3.3 million bytes. In reverse is roughly 2.4 million.

Debugging tool (very useful for problem set): strace

This traces system calls made by the program. Running "strace echo foo" prints a bunch of stuff. Toward the end there is a "write". Strace prints out every system call to stderr. Running strace on r03 shows that it's reading a byte at a time. it shows "read(3,"6",1)". The 3 is the file descriptor for the file, because 0,1,2 are taken already. Running strace on revbyte shows alternating lseeks and reads. This is one of the reasons its slower – it's making twice as many system calls. Getting the OS's attention is expensive. There is overhead in transitioning (context switch).

r11-stdiorevbyte is basically the stdio version of revbyte, it uses fseek and fgets. r05 is the stdio bytes function.

r03 gets 3 million.

r05 gets 28 million.

Running strace on this shows that stdio is reading 4096 bytes at a time regardless of whether or not you ask it to read 1 byte at a time. This is because it has its own cache that its filling up speculatively.

r11-stdiorevbyte gets about 5 million bytes/second.

It is performing 4096 lseeks and one read, rather than alternating.