Lecture 9, 10/1/13
Scribe notes

Last week for Silvia peaches at the Farmer's Market. ☺

I/O Caching:
File integer descriptors: 0, 1, or 2 for FILE stdin, stdout, stderr – these are standard streams

For example, (and these are pulled from w01-syncbyte, source code in lecture notes)
O_CREAT: create
O_TRUNC: truncate
O_SYNC: synchronous, I/O immediately writes to the underlying media.

w01-syncbyte (2800 B/s), w02-syncblock (1180000):
Difference between syncbyte and syncblock: difference in the size of the array.
Syncbyte: 1 byte writes
Syncblock: 512 byte writes
Requests in syncblock have more units (511 more units).

Basic formula (rule of thumb) for calculating the cost of work: NR + NUK

If we have a series of N requests, each of size U units, then the total cost of the series of requests = NR + NUK, where R is the per request cost and K is the per unit cost.

Cost[syncbyte] = NR_{syncbyte} + N*1*K
Cost[syncblock] = (N/512)R_{syncblock} + (N/512)*512*K

The difference between these costs all comes down to the per request bytes.

(definition) Batching: Sppeds up a series of requests by reducing the per-request cost, by grouping data into fewer requests.
Synchronous I/O: I/O that accesses the underlying media directly

Syncbyte and Syncblock are not that different. Compare the throughput, syncblock is ~420 times greater.

What are some of the per request costs that seem to dominate these programs (syncblock and syncbyte)?
Calling 'write' a variable number of times. Syncbyte: called once per character. Syncblock: called once per 512 characters.

A useful tool: diff –u

File comparison utility that outputs the differences between two files.

Files are either stored in buffer cache or directly on the disk. The former is significantly faster to access, as demonstrated in lecture with comparative testing, whereby each run proved faster than the previous run of byte + block programs. Naturally leads into a discussion of the caching hierarchy.

Caching hierarchy: **//should be represented as a pyramid**

1. registers
2. SRAM
3. DRAM
4. flashmen
5. hard disk

- Expense is highest at 1, lowest at 5
- size is smallest at 1, biggest at 5
- speed is fastest at 1, slowest at 5.

Cache:

Fast storage used for copies of data primarily stored on slower storage used to speed up for future requests.

Buffer cache:

Operating system's memory cache for files.

Prefetching:

Cache policy where we load data into the cache that is predicted to be useful later.

- Is prefetching always a good idea? Quite simply, no. Logically easier if all programs behave exactly the same. Sequential.

We store files on disk and flash because these are non-volatile and durable, though slow.

disk → memory/buffer cache/OS → application (makes read/write requests).

The OS's goal is to make the applications run as quickly as possible. The policies the OS puts in place to handle reads and writes are very different. Simply, the OS can satisfy a write request without knowing the contents of the disk, which is not true of a write request.

As the application writes more data to the buffer cache, the data which was already in the buffer cache is slowly being written to disk, which then frees up portions of the buffer cache for reuse.

OS capable of guessing which bytes of memory will be useful in the future, thereby enabling it (especially in the case of sequential I/O), to predict which memory should be considered.

Program r09-revbyte shows that prefetching is not always a good idea.

- program works by reading a file in reverse order. Not the same as sequential order! Harder to prefetch!

-lseek: repositions read/write file offset

Useful debugging tool: strace

- monitors system calls used by a program and prints out system calls in human-readable format.