CS61 Lecture 17
November 1st, 2012
Scribe Notes #2

Announcements

- Only two problem sets left, hooray!
- Last time: disk technology
- Today: caching and buffered I/O
- Lecture code checked in on code.seas under l17

Caching

- One of the most important ideas in computer systems
- Performance improvements almost always involve caches somewhere

Taking a look at w01-syncbyte.c

- At the top, we open a file with function call open()
    - open() is the direct system call version of the fopen() call
    - Takes three arguments (read more about them with "man 2 open")
        - The "2" means to look up the *system call* open()
    - Return value is a *file descriptor*, a number that the kernel understands

More on open()

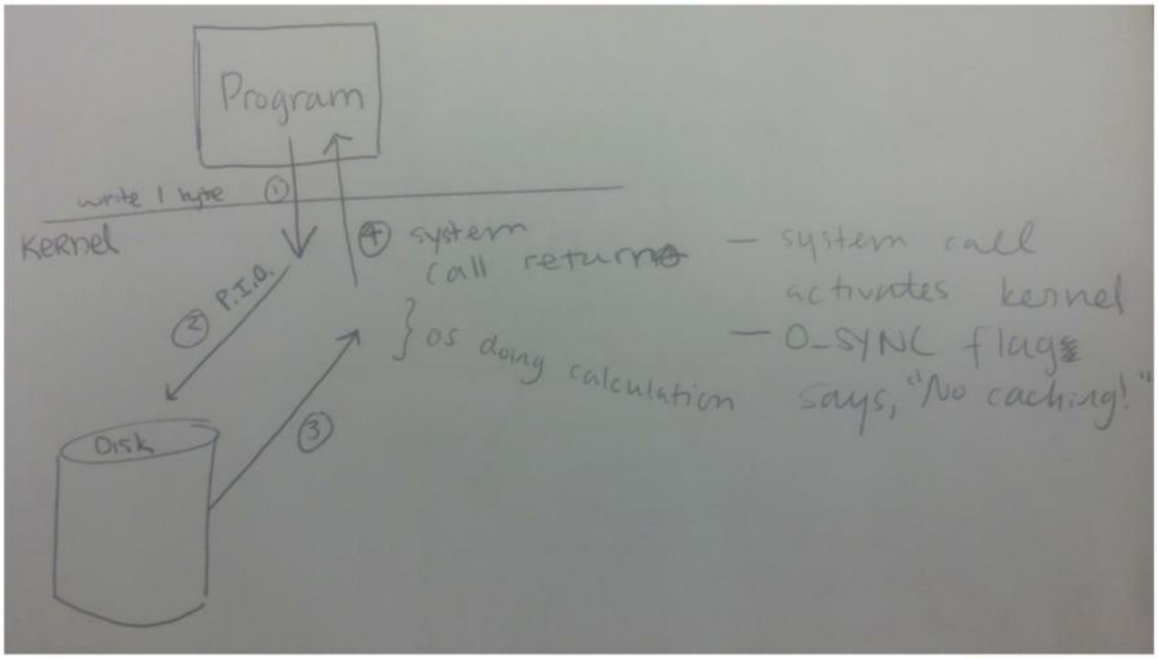
- System calls like write() and open() are interpreted by the kernel — there are wrappers (like fopen(), fread(), fwrite(), etc.) around the more basic system calls
- Flags that can be added to the open() call (not necessary to memorize) include:
    - O_WRONLY: writing to the file, rather than reading
    - O_CREAT: create file if it doesn't exist
    - O_TRUNC: throw away current version of file and start over
    - O_SYNC: write to the disk every time you write (we'll get into this in a few moments)
    - These can be bitwise or'd together to add multiple flags to the instruction

(Question: Does O_SYNC write to the disk every time you write within the loop? Answer: Yes!)

Okay, back to w01-syncbyte.c

- The while loop will loop "size" times (in this case, 10 << 20)
- We also print how much time it takes per byte (n / tstamp() - start)
- Wow, this program is super slow! 26 characters per second?
    - It's silly to write one char, then wait for the disk to finish before writing the next char — terrible performance

    - Disks are very slow mechanically: think of the swinging head, seek time, and rotational latency
    - As a result, the norm is about 100 writes per second (not very fast at all)
        - TIL Eddie's computer is slower than the norm, hee hee



[diagram showing forced interaction between program and disk on each call to write() because of O_SYNC flag]

One optimization? Let's look at syncblock-512.c

- New write() call that writes in blocks of 512 bytes, rather than a char at a time
- Request costs can be broken down into two pieces: per-request overhead (call this R) and per-unit overhead (call this U)
    - We'll consider a unit a byte for read/write
    - So currently, the total cost of this new write() call is R + (512 * U)
    - Which cost is bigger? Presumably the R, since that involves the seek time + rotational latency
- We expect this program to run much faster than the last one, whose total request cost would be 512 * (R + U)
    - And we're right (write)! This program runs MUCH faster.

Trying even bigger blocks

- Our favorite power of 2 is apparently 2^16, so let's put that as our block_size
- Yet again, an enormous difference in rate!

- But if we keep doing this, we'll eventually get to a point where increasing doesn't improve performance by very much
- Under the environment of Eddie's laptop, that point (aka the maximum transfer rate) is around 1MB - 10MB
- To reiterate: the R (per-request overhead) is much bigger than the U (per-unit overhead)

On to w03-byte.c

- Exact same as w01-byte.c, except we've removed the O_SYNC flag
  - Oh my god it's so fast
  - But we're still writing to disk? Why is this happening?!
- O_SYNC slows things down because it forces the operating system to write data right away

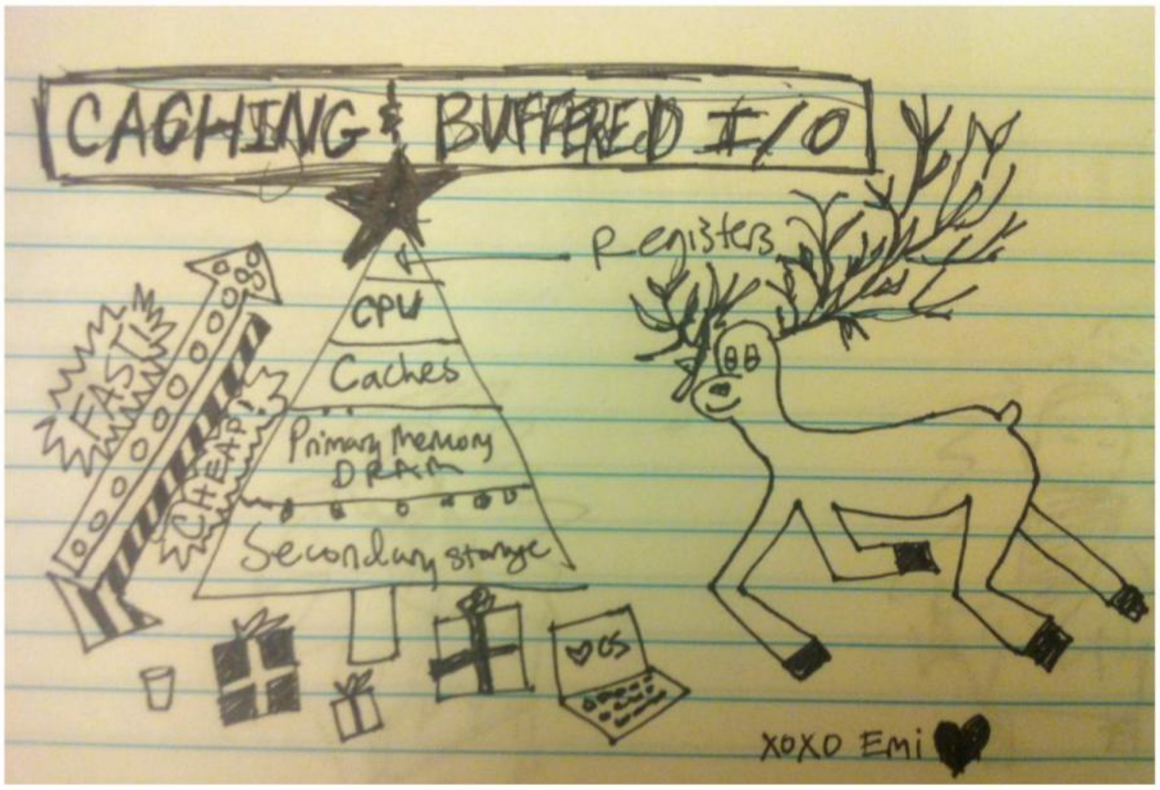
(Operating system designers have observed for many years that writing to the disk is slow, so they've tried to avoid showing that speed bottleneck to designers. What would *you* do as an OS designer? Well, how about ...)

Batching

- Group requests to reduce per-request overhead (i.e. reduce the number of requests by waiting to actually write until they can be grouped into a more optimally-sized block)
- If we choose to write 65000 bytes at once, instead of writing 1 byte 65000 times, we significantly reduce the cost (because of R, the per-request overhead, remember?)
- Strategy used in syncblocks-512
- The cost of this: delay and instability! What if the power cord gets ripped out and your computer turns off? You might lose data you thought you had already written, but which your OS was actually waiting to write to disk while batching because before it is actually written to the disk, data is stored in volatile memory that requires electricity.
- Also, latency: the OS is constantly looking for opportunities to batch since it wants to minimize the number of request

Where do requests go while they're waiting to be batched and sent off? Into a cache!

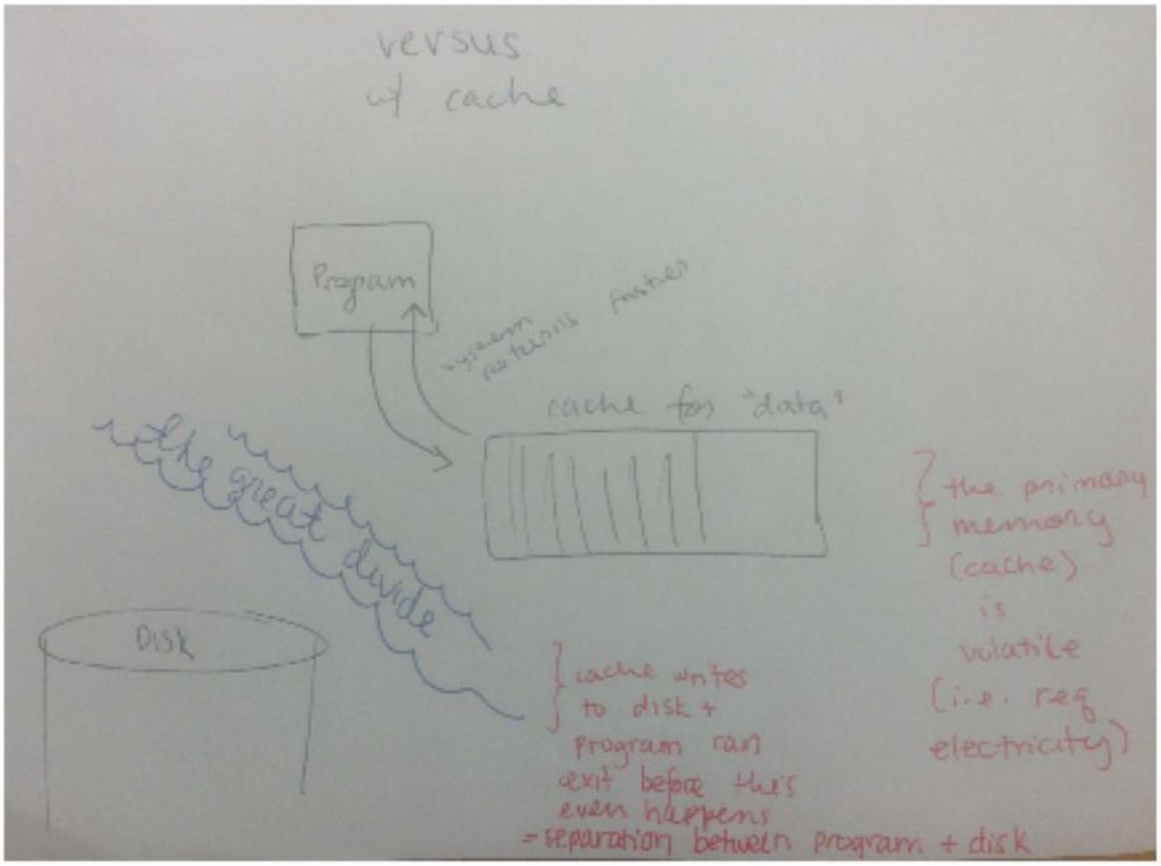
Cache
Fast storage that stores temporary copies of data from slower storage, in order to speed up future requests



[diagram of memory hierarchy]

Memory hierarchy pyramid

- As we move up in the pyramid, we get faster access to memory, but it's quite expensive (like literally $$$ expensive to get more memory in that level) — as we get lower, it's cheaper, but also much slower to access
- The CPU only manipulates the registers at the top
  - If you want to manipulate data stored in secondary storage, you have to move it up to the registers
- Every level can be treated like a cache for the level below!
  - Think of registers as a cache for primary memory, primary memory as a cache for secondary storage, etc.
  - In primary memory, which is more local to the processor, there is an area used as a cache for the file called "data" in our program


versus
w/ cache
Program
system returns faster
cache for "data"
the great divide
Disk
cache writes to disk + program can exit before this even happens
= separation between program + disk
the primary memory (cache) is volatile (i.e. req electricity)

[diagram of program + cache + system calls]

Volatile memory

- Data stored in primary memory is volatile — when you pull the plug, all the data goes away and gets corrupted
    - So a disadvantage to leaving something in the cache is that the data might get lost!
- In general, the operating system tries to put data onto the disk soon (within 30s), but the program might exit before any of the data goes to the disk (awkward), such that there is a layer of disconnect between the program, which requests writes to disk, and the actual writing, which the OS does using data stored in the cache
    - This process is kind of asynchronous: there is another process entirely that shifts data from the cache and writes it to disk
    - It's the OS's job to ensure that data written to the cache is used

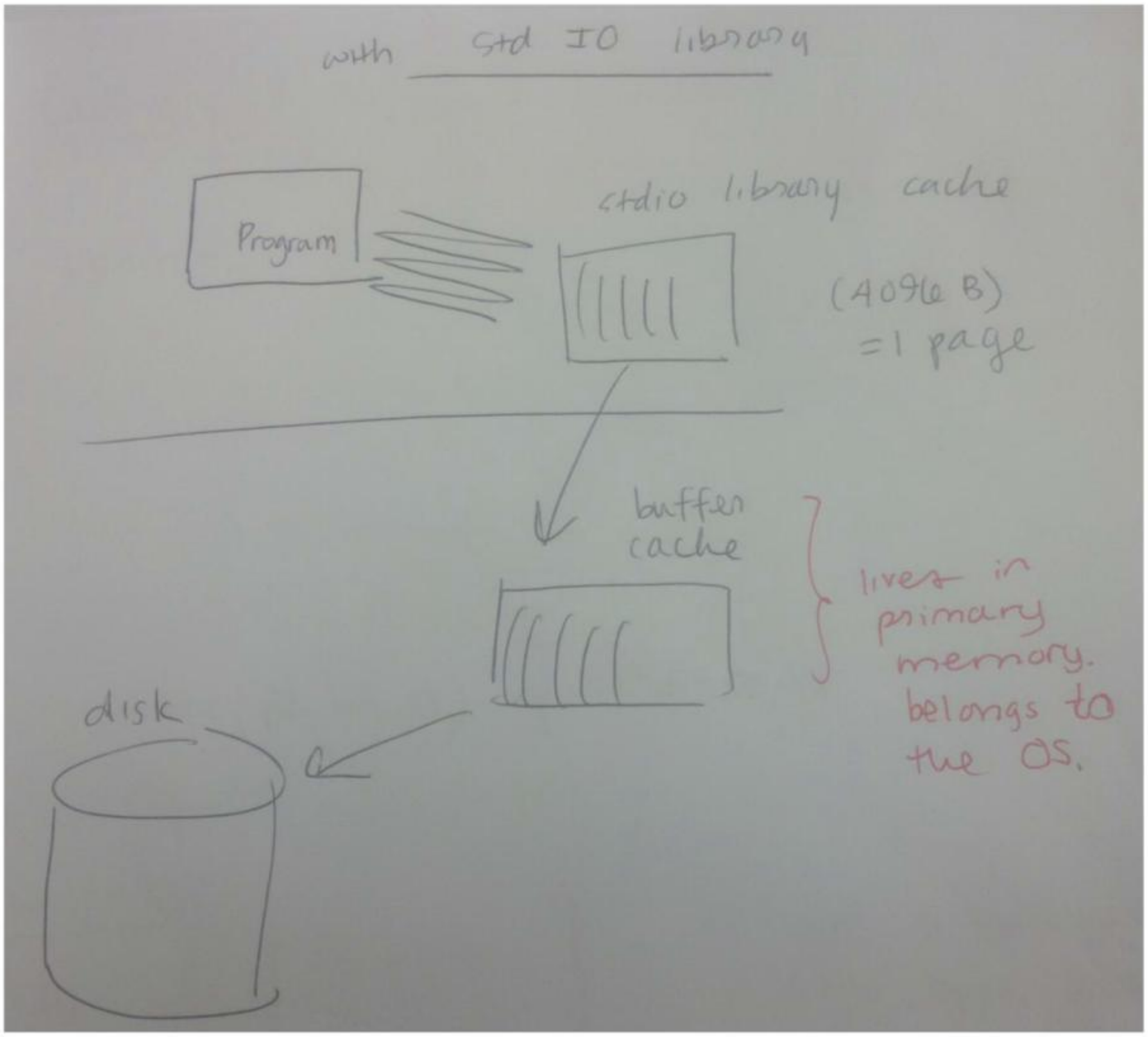
A brief look at w04-block.c

- This program still has no O_SYNC, but we're back to writing 512 bytes at a time, rather than 1 byte
- If this program ran exactly as fast as the per-byte program, it would mean that the R is really low but U is relatively higher

- But it does run faster! So it looks like the R is pretty high :)

Onwards to w05-stdiobyte.c

- Let's try to make things slow again by using fwrite() instead of a system call
- Wait, it's faster!
    - But why? We're using a library function, which must call write() somewhere
    - How does using standard I/O speed things up compared to system calls?
- The answer, of course, is that the library acts like a cache for the OS
    - Reduces the per-request cost for exceptional control flow by batching the requests
    - The stdio library has one page's (4096 bytes) worth of caching ability, which speeds things up by a factor of five!
        - This is nothing to sneeze at
        - No, seriously, don't sneeze
- The stdio library increases speed, but again, it adds delay (latency) in terms of when data is written to disk: if we were to lose power unexpectedly, or the process suddenly died, the data in the cache would never make it to the operating system, which would in turn never write the data to disk
- So that's a bit dangerous



[diagram of program + cache + standard I/O cache]

The last write() example, w06-stdioblock.c

- This is a version of the standard I/O program that we used, except with a block size instead of a byte
- It's avoiding the overhead of calling fwrite() so many times, so we expect a gain in performance
- But it's not THAT much of a gain, since the cost of fwrite() isn't that high (i.e. the standard I/O library has relatively small overhead)

Reading

Here are ways to read from a file, in order of speed:

- 1 byte at a time (r01-byte.c), using read()
- 1 byte at a time (r04-stdiobyte.c), using fread()
- 512 bytes at a time, using read()
- 512 bytes at a time, using fread()

(Not all that surprising, since we know that fread() probably reads a page — 4096 bytes— at a time!)

The difference: caching for reads

- When we use a cache for writing, we fill it up, and when the cache is full, we send it to the kernel in a single system call and clear out the stdio cache to fill it up again
    - But this method doesn't make much sense for reads, right?
- Instead, the cache will read *more* than was requested in the hope that it will be requested soon
    - Cached data can be used to satisfy future requests
    - (More successful if the program has good locality, i.e. a predictable request pattern)
    - This strategy is called *speculation*

Speculation

- Speculation is performing work *before* an explicit request in order to speed up future requests
- The specific type of speculation that happens with reads and caches is called *readahead*, or *pre-fetching:* reading data in advance of explicit requests!
- How does it work?
    - There is a pointer to somewhere the buffer cache — when there's a requests, the pointer to this buffer moves forward
    - Once the requests catch up to the buffer, the stdio library requests more data from the OS
- To have perfect readahead, you need perfect predictive power about what the file will want to access in the future
    - This is, of course, impossible
    - But sequential access (accessing data in order) has high locality and is easy to predict! It's also the norm
- Speculation is why when we call write() we see 4096 writes (which fill up the buffer) and then a single system call, while when we call read() we see a single system call (which fills up the buffer) and then 4096 reads

When speculation doesn't work: r06-stridebyte.c

- This program has a call to lseek(), which changes the position that we're reading from in a file — we can fast-forward, reverse, etc. without reading every intermediate page
    - Seeking: when the pin head reading off of the disk whooshes to a different specific place on disk
- We call this stride because it has a super-weird stride
    - Reads data at location 0 first, then 2^20, then 2 * 2^20, then 3 * 2^20 ... until the end

    - Then it reads data at location 1, then 1 + 2^20, then 1 + (2 * 2^20), then 1 + (3 * 2^20) ... until the end
    - Then it reads data at location 2, then 2 + 2^20, then 2 + (2 * 2^20) ... you get the idea
- This non-sequential read pattern will slow things down because readahead is less successful, although eventually the readahead will adapt to this unusual pattern (crazy, right?!)

[Insert striding diagram here]

Striding with stdio: r08-stridestdiobyte.c

- This is actually slower than stdio without stride (makes sense, since stride is weird), *and* slower than stride without stdio (doesn't make much sense — didn't stdio was speeding things up before?)
- The reason: the stdio cache reduces performance, since the cache is only 4 kilobytes large, but our stride is an entire megabyte long!
    - Therefore, for every fread(), stdio is fetching and caching data that isn't being used or requested

That's all, folks!

❤,
Jane + Angela