Caching & Buffered I/O

Today's lecture code can be found in ~/cs61-lectures/l17
// to pull lecture code, install git first, then:

```
mkdir cs61-lectures
cd cs61-lectures
git init
git clone git://code.seas.harvard.edu/cs61/cs61-lectures.git
```

Today we will be looking mostly at code. To learn about caching and what affects buffered I/O, let's write some bytes from and to files. Look at the code in the following files.

$ w01-syncbyte.c (write 1 byte at a time)

This program opens a file with 'open', (the system call version of 'fopen'). 'open' returns a file descriptor (an int that identifies the file), that is later used to write/read from the file. After the file is opened, the program enters a loop. In the loop, the program repeatedly writes a single character in a buffer into the file using 'write' (the system call version of 'fwrite'). Additionally, the program prints the number of characters per second written to the file.
Note: w01 writes with two important flags, the O_SYNC and O_DIRECT flags. They force every write call to change the file immediately (these flags impact performance negatively, as we will see later).

· `man 2 open // the '2' designates <system call>. Returns a file descriptor`

- What is the size of 10 << 20?
  - 10 << 20 = 10 Megabytes
- Rate: 26 chars/sec, log rate 3.25
- How can we make this faster?
  - Introducing block size (e.g. 512).

$ w02-syncblock.c (block size of 512)

This program is identical to the one above, except instead of writing one character at a time, 512 characters are written at a time.

- Much faster than writing one byte at a time
  - Rate: 14,000 chars/sec, log rate 9.54
- What if we adjust our block size to 2^16?

- Rate: 1,540,000 chars/sec, log rate 14.24

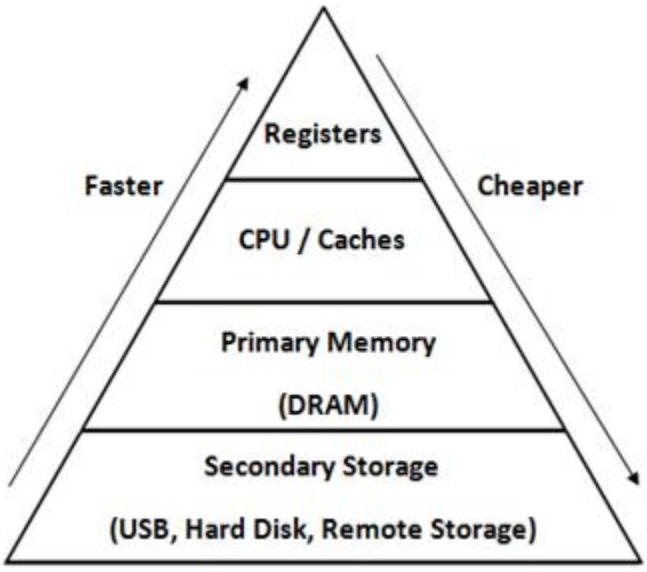
· What if we adjust to 1MB? Or just the entire file size (we get log rate of 16.08)?
- We see that our speed plateaus.

Why is this faster? Calling ‘write’ has two costs: the per-request overhead and the per-unit overhead. Specifically, calling ‘write(fd, buf, sz)’ has the cost of ‘R + sz * U’, where R is the per-request overhead and ‘U’ is the per-unit overhead. In this program, the writing processes is bottlenecked by R. Increasing sz, the size of each block, and decreasing the number of times write is called therefore speeds up the writing process.

$ w03-byte.c (No O_DIRECT flag, one byte at a time)

· Rate: 4,740,000 chars/sec, log rate 13.07
· Theory underneath the hood
- Def: *Batching* - group requests to reduce per-request overhead.
- What is the disadvantage to batching?
  - § Delay: the way we get a batch of requests is by delaying the first request until we have a batch; this means the first request can happen long after the other requests in the batch.
  - § The requests are held in a cache, which is volatile, so if the computer is turned off the data will be lost.
- Ultimately the writing process will be limited not by the rotational latency and seek (per-request overhead); it will be limited by the size of the bus.
- Batching makes use of caching. What is a *cache*?
  - § Fast storage that stores temporary copies of data from slower storage. Speeds up future requests.

· Memory Pyramid



  - Things get FAST up the pyramid, things get CHEAP (in terms of expensive to buy) down the pyramid
  - Every layer can be treated as a cache for the layer below
  - CPU can only access registers
  - Dropbox storage is secondary storage, slower than your computer
  - Only secondary storage is permanent memory.
  - What happens if power is suddenly cut from your computer?
    - § Data in temporary memory (caches) lost.
  - There is an asynchronous process that shifts from cache and writing to disk

$ w04-byte.c (Cache involved - no O_SYNC and O_DIRECT flags)

· Rate: 67,700,000 chars/sec, log rate is 18.03

· Why is this so fast?

  - We make no calls to disk. When we write to whole disk at once, we wait for the disk to reply.

$ w05-stdiobyte.c (Library usage)

· Rate: 27,000,000 chars/sec, log rate: 17.11

· We are using fwrite to a standard i/o file

· Using fwrite introduces more overhead (because the library function call leverages more code, and thereby becomes more expensive).

· Despite the extra cost, the program seems to be quite fast. Let's examine what system calls we are making.

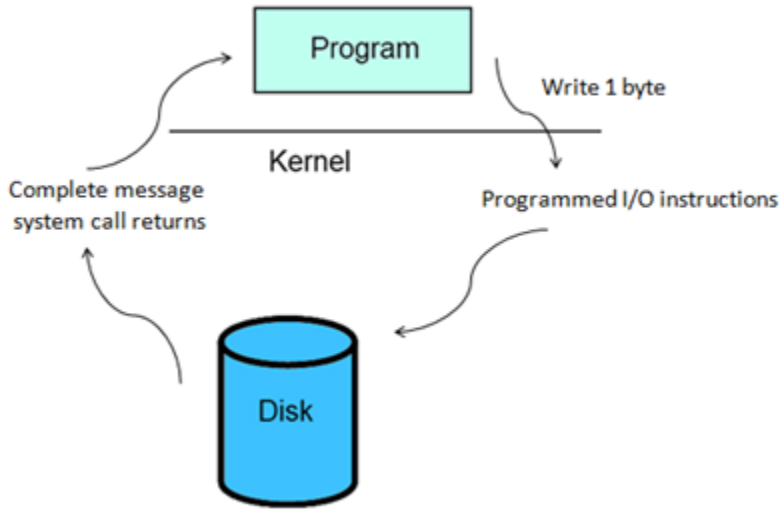
- We will use ‘s-trace’. ‘s-trace’ is a useful debugging tool, shows us exactly what system calls a program makes.

· What is the size of the cache in the stdio library for this file descriptor?
- 1 page = 4096 bytes.

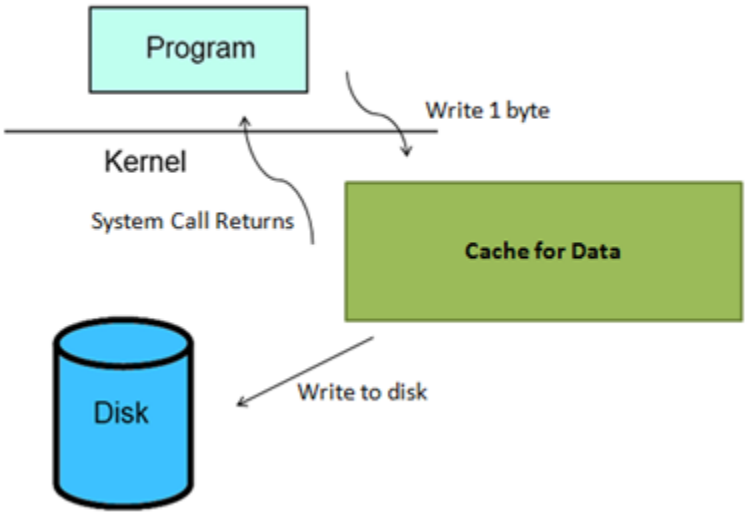
· How can we tell?
- Because that is the unit we are writing to this file in (see output of ‘s-trace’).

· The program is writing to some stdio library cache (located in primary memory) and then making one call to our system using ‘write’. By using fwrite, we get an additional cache, using stdlib, ultimately speeding up the program.

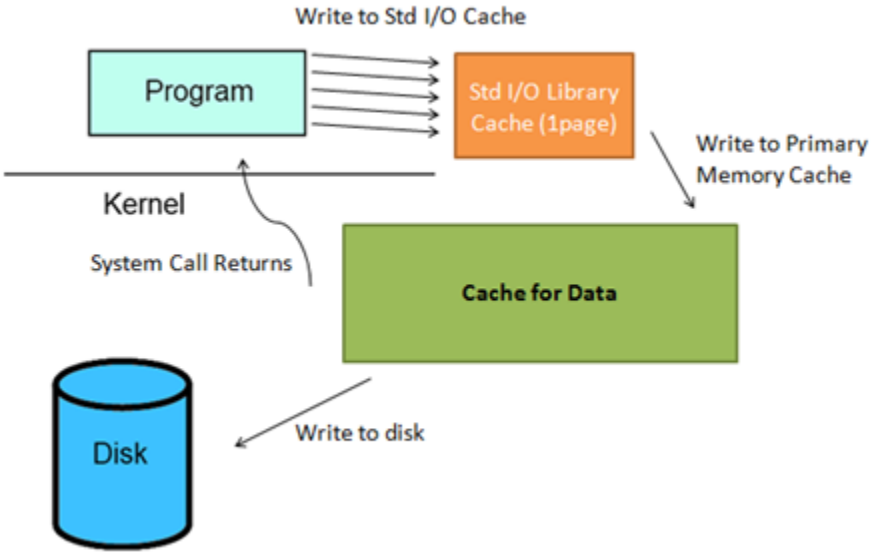
· **Writing with no cache**



· **Writing with a cache**



- **Writing with a cache and stdio (fwrite)**



$ w06-stdioblock (stdio with batching)

- We expect the performance to go up, but not by much.
- The stdio library has some overhead in it
- We still have a winner: writing a blocksize of 65k (2^16), with rate of 409,000,000 chars/sec
- Using primary memory as a cache is faster than system call interrupt process.
- Calling 10^20 stdios calls is much faster than 10^20 system calls. System call overhead is relatively high.

Let's look at reading, which is faster than writing.

$ r01-byte (reads 1 byte at a time)

- In reads, unlike writes, we read more than what is asked in the hope that the data will be used later.
    - This strategy is called *Speculation* - We read more than user is requested and use that data to satisfy future requests. In general, speculation is performing work BEFORE an explicit request to speed up future requests
    - As applied to reading, this is also known as *read-ahead* (or *pre-fetching)*- reading data in advance of explicit requests
- *Read-ahead:* we call read once, then system fills up the buffer.
    - The program pulls bytes out of the buffer. Buffer is filled first. For each read, we can think of this as a ptr that moves forward in the buffer with each read

request. When the ptr gets to the end of the buffer (size 4096 byte = 1 page), buffer erased and filled with new data.

- What does perfect pre-fetching require?
  - What we need is a magical oracle that will tell you everything the program needs in the future, an 'exact predictor'. Such predictors are, in general, impossible (duh). However, there are certain kinds of access patterns that are easier to predict.
  - Such as sequential access. Reading and writing to the end. The reason it is easy to predict is because it has high locality. In other words, it is very, very likely that if you access byte *n*, we will access byte *n+1* next. In fact, sequential access is assumed to be the norm. It's what makes computers tolerable at all. It allows us to optimize for sequential access.

$ r06-stridebyte.c(read using a stride)

- `lseek` - seek is what allows us to go to the last page of a huge pdf without having to read every intermediate page

- Here is a file that is 10 MB big (10*2^20). We will access it in a stride access pattern (e.g. occasionally useful access patterns for blurring images)
  - We read at 0, then read at 1*10^20, then 0+2*10^20, then 0+3*10^20, …
  - We then read at 1, then at 1+1*10^20, then 1+2*10^20, then 1+3*10^20, …
  - We then read at 2, then at 2+1*10^20, then 2+2*10^20, then 2+3*10^20, …
- We observe here that the OS is confused about the stride read pattern--- it observes that the read pattern is not sequential!
- Note that this particular *read-ahead* can eventually adapt to our non-sequential read pattern.

$ r08-stridestdiobyte.c(stride using std i/o caching)

- Adding a cache *reduced* the performance of our system.
- Why?
  - Reads are pre-fetching 4096 bytes for each 1 byte stride read, and this data is being thrown away.
  - Cache needs to reload on each read
  - So data is not being used at every stride. Stdio is not handling this pattern very well, but it can. We will examine how to fix this next time/lecture.