CS 333 :: Topic Notes :: Solid State Drives (SSDs)

Flash-based solid state storage devices (SSDs) are everywhere (phones, laptops, servers, ...), but unlike hard disk drives (HDDs), SSDs are difficult to reason about *externally*. With a hard drive, we can think about the costs of I/Os in terms of physical actions (moving the disk arm, a short rotational delay, and transferring as the platters rotate). With an SSD, our understanding is obscured by the presence of a **flash translation layer** (FTL). To truly know what is going on, we would need to know the internal state of the device as well as the (proprietary and super-secret) FTL's implementation details. For this reason, we mostly treat SSDs as a black box, and we try to give the SSD workloads that we hope are compatible with the things that the FTL is good at.

Learning Objectives

- Be able to describe the internal components of an SSD (banks, blocks, and pages)
- Be able to describe the various responsibility that the FTL performs (logical-to-physical page mappings, garbage collection, wear-leveling, error correction)
- Be able to describe the differences between SLC, MLC, and TLC and how those differences affect the characteristics/performance of the different types of NAND flash.
- Be able to describe the logical steps of writing data inside an SSD

SSD Organization

SSDs masquerade as a random access block device. Internally, they are made up of some RAM, some processing, and NAND flash. However, there is a hierarchy to the flash organization:

- At the lowest level, we can read or write to an individual flash **page**
  - Pages (typically) have a fixed size
  - Once written (also known as "programming" the page), a page cannot be programmed a second time
- Multiple pages are grouped together into **write/erase blocks** (often simply referred to as **blocks**)
  - Since pages can only be programmed once, blocks are, in many ways, treated similarly to LFS *segments*
    - The FTL garbage collects blocks similarly to how LFS garbage collects segments: move all live pages out of the block, then erase the block in its entirety
- Multiple blocks are grouped together into **banks** (sometimes called **planes**).
  - Banks can be accessed in parallel, so we can apply some of the RAID concepts *inside* the SSD
    - FTL designers can exploit the SSD's internal parallelism by striping large writes across multiple banks, increasing SSD performance

NAND Flash

Unlike a hard drive, NAND flash does not have any moving parts. A bit is represented by electric charge stored in a transistor. The level of the charge is probed in order to determine the bit's value.

- In *single* level cells (SLC), there is one threshold, and each cell encodes one bit. If the charge is above (below) that threshold, the value is interpreted as a 1 (0).
- In *multi* level cells (MLC), there are three thresholds, and each cell encodes two bits. If the charge is below the first threshold, the value is interpreted as a 00; if it is between the first and second thresholds, the value is interpreted as 01; between the second and third thresholds, a 10, and past the third threshold a 11.
- In *triple* level cells, three bits are encoded (similarly to MLC)

As the number of bits encoded per cell increases, the precision with which we must be able to charge the cells when programming and detect the charge level when reading, grows. This exacerbates the challenges associated with device wear-out.

Endurance

Unfortunately, SSDs have limited lifetimes. The physical medium that stores the charge becomes less precise the more it is programmed/erased. When the FTL cannot reliably determine the values stored in flash cells, the SSD becomes unusable for persisting data (what use is data that you store, if you can't read it back?).

Ideally, the SSD will degrade slowly and last for a long time. To make this possible, one of the FTL's primary responsibilities is to perform "wear leveling". Wear leveling algorithms attempt to evenly spreads writes over the SSD's pages, smoothing the wear-out. When a page becomes unreliable, the SSD can choose to disable that page and continue to function as long as there are other usable pages. However, as wear-out progresses, the device may suffer a performance decline because the FTL has fewer pages to work with.

Garbage Collection

SSDs cannot erase individual pages; it can only erase at the granularity of a block by draining all of the charge in the block at once. This restriction means that you cannot update pages *in-place* (consider SLC: you can flip a single bit from `0` to `1` by increasing the charge, but you cannot remove charge, so flipping a single bit from `0` to `1` is impossible; hence you can program a page one time).

Block-granularity erases should remind us of the segment cleaning in LFS. If you want to update a page, the FTL can make a new copy of that page and update it's pointer from the old version to the new version. Now, the old version is sitting around in some write/erase block waiting to be reclaimed.

Garbage collection must migrate all the valid pages from the target write/erase block to a safe location, and then update the pointers to those pages. Only then is it safe to erase the target block.

- This "page migration" is one of the most significant sources of *write amplification* in the SSD.
- When performing garbage collection, it seems wise for the FTL to consider "wear leveling" when determining which blocks to select and where to rewrite them.

The Flash Translation Layer (FTL)

Since NAND cannot be overwritten, and since NAND wears out, and since banks can be read/programmed/erased in parallel, the SSD implements what has become known as a "Flash Translation Layer" in firmware. The FTL wears many hats, including:

- Wear leveling
- Error correction
- Logical to physical page translation to mange out-of-place updates

The FTL manages all of these tasks internally so that, from the host computer's perspective, the SSD is just like any other block device: random-access.

Thought Questions

1. Given the write constraints of an SSD, one strategy that seems to make a lot of sense for an FTL is to log-structured writes (like LFS). In the LFS paper, we discussed garbage collection using the simple *greedy* or the *greedy cost-benefit* approach. Suppose you were asked to redesign garbage collection for an SSD. How would device wear-out affect the way you design your GC strategy? (You may want to think about things like initial data placement and victim selection)
2. One thing that you aren't always told about an SSD is its actual physical capacity; instead, the box reports the "usable" capacity. The difference between the actual capacity and the usable capacity is called the over-provisioning percentage. Why might the actual capacity differ from the usable capacity (e.g., what types of things might an SSD designer use the over-provisioned blocks for)?
3. The reason you can't overwrite flash pages is because you can only (naively) program them precisely once --- you must bulk-erase groups of pages at block granularity. Suppose you know the contents of a flash page. Are there *any* modifications that you could make to update that page's contents without fully erasing it (assume that all bits in a fresh page are initially `0`, and programming can flip a bit to `1`)? Can you think of any clever data structure designs that would let you take advantage of the set of legal of overwrites? (This is an open question, with recent top-tier publications exploring clever new data structures and designs)
4. Flash pages have what is called a "page private area" ("out of band area" in the textbook), that is available exclusively to the FTL. I recently read that a particular flash device with 16KiB pages has 2208 additional bytes per page set aside as the page private area. What types of things might the FTL use this page-private area to do? How many bytes to do you think each of those tasks might require? Does 2208 bytes seem like it provides a good ratio of page-private area to data, given that there is 16KiB of data per page?
5. To get peak performance on a high-end SSD, you need to exploit its internal "bank-level" parallelism. Expensive SSDs support very high queue depths (amount of I/O queued up and ready for the device); the system must always have many outstanding I/Os to keep the device saturated so it can maximally utilize the available bandwidth. What types of applications/workloads generate this type of traffic? Under those types of workloads, what other resources may become bottlenecks (e.g., CPU, memory, network)?
6. One thing that is important to consider when evaluating different algorithms is the I/O amplification. The book defines *write amplification* (WA) as "the total write traffic (in bytes) issued to the flash chips by the FTL divided by the total write traffic (in bytes) issued by the client to the SSD." What seem like the main

sources of write amplification given what you know about FTLs? Why is write amplification such a useful measure for evaluating FTL designs (e.g., what are the negative effects of high write amplification)?

7. Suppose you are tasked with writing a new file system because, let's face it, FFS is not the best design out there. You are given two options: (a) you are granted direct access to the SSD internals, and you can design a file system that directly accesses the banks/blocks/pages/cells individually using whatever API you define, or (b) you are given an SSD with an FTL and you can design a file system that accesses that SSD using the standard block interface. What are the trade-offs? When would you choose (a) vs. (b)? What happens if you design your system and then are given a new SSD a year later as new advances are made in SSD hardware?
8. Why does your phone have an SSD instead of a hard disk drive (HDD)?
9. What type of FTL do you think a typical commodity SSD has (page-mapped, block-mapped, or hybrid)?
10. Other commands/interfaces for SSDs are commonly available besides read/write.
    - `TRIM` is one command that lets a user tell the device that a logical block is no longer valid (maybe the file that was using that block was deleted, but the block has not yet had new data written to it). How might an FTL react to a `TRIM` command?
    - "Stream IDs" are a way that an application (or FS) can "tag" a write. Suppose there are a fixed number of streams IDs (e.g., the integers 1-16), and the FS can associate a stream ID with a write request. In this scenario, the SSD is then presented with new data and a Stream ID for that data that it must locate somewhere on its set of available banks/blocks/pages. How might the FTL use this information to make placement decisions? What criteria might an application use when assigning stream IDs to its writes?