Solid-State Storage Technology

1 Introduction

Welcome to this short lesson on Solid State Storage Technology. Solid State Storage has become quite popular as mass storage technology, and there are many variations, for example NAND flash, NOR Flash, Phase Change Memory, and others.

In this lesson we will focus on NAND Flash. This type of memory does have a number of disadvantages compared to other technologies, such as poor Random READ performance, but it is well suited for file storage, in particular for sequential access data such as multimedia.

Flash and other solid state storage technologies have, similar to hard drives, an interesting performance characteristic, which we will briefly explore.

We will talk about operations that can be done on NAND Flash. The operations are somewhat similar to what we are used to from other block devices, with one main difference that makes NAND Flash quite tricky to deal with.

NAND Flash manufacturers add a so-called Flash Translation Layer, called FTL for short, to hide the differences of NAND Flash from traditional block storage devices. We will explore an example of a basic Flash Translation Layer.

At the end of this lesson you will have a basic performance model of NAND Flash device from an File System Design perspective.

Let's begin

2 NAND-Flash Technology

From a file-system-design perspective NAND Flash memory is a solid-state persistent storage technology that has a "disk-like" user interface. READs and WRITEs are sector based, or, as a result of an unfortunate terminology twist, "page-based" in Flash memory parlance. Nand Flash is ideally suited for storage of sequential data, such as multi-media. It shows poor Random Access performance, but this typically hidden by caching the data in main memory. This is also called "data shadowing". The Flash storage space is structured into **blocks**, which in turn are partitioned into **pages**. Typical sizes are 128kB for blocks and 2kB for pages. In this case a block would have 64 pages.

3 NAND-Flash Operations

Let's look at what operations are supported in flash memory. The blocks are numbered, and the pages are numbered as well, thus supporting block-based and page-based operations.

Flash memory supports the following operations:

- One can READ individual pages, similarly to sectors on a disk.

- One can also PROGRAM individual pages. “Programming” a page means writing data to a page that does not yet contain data. As we will see later, one cannot write data into a page that does already contain data.
- One can also ERASE data from an entire block. There is no erasing data from individual pages.

The performance numbers are from the Micron MT29F NAND Flash family. The read latency is pretty high, but a large part of this is due to random access latency. The first byte to read is 21musec, and each following byte is 25nanoseconds. The time to program a page is 220 musec, and to erase the data from an entire block takes a half a millisecond.

One thing that is clearly missing is the ability to WRITE data to a page if there is already data there. So we cannot OVER-WRITE data. This can be explained by how writing and erasing works together. Let?s have a closer look at a block. When bits are written into a block, one can think of it, as a crude approximation, as punching holes into a paper tape. Once a hole has been punched, we cannot “un-punch” it, so-to-say.

In order to erase the data from this segment of the tape, we have to literally cut the segment of paper from the tape and remove it. Then we have to glue back in a new strip of paper with no holes. Once old segment has replaced by the new strip, we can resume punching holes to write data. Unfortunately, blocks suffer as they are erased. After about 100?000 PROGRAM/ERASE cycles, blocks start wearing out, and they lose their capacity to hold data. It is therefore important to distribute the ERASE operations across the blocks on the device. This is called “wear leveling”.

4 NAND Flash: Naïve Writing

So how would we use a flash memory device? In particular, how do we write to it? Let’s look at a nïve approach, by way of example. Say that the flash device contains no data, and a write request for Page 1 comes in. This is no problem, and we PROGRAM Page 1 with the data to be written. We mark Page 1 as containing data. A write request for Page 7 comes in. And we PROGRAM Page 7 and mark it. Then a write request to Page 9 arrives. and we PROGRAM Page 9 and mark it. When a READ request for Page 1 arrives, we serve it. Same for a READ request for Page 7. A WRITE request for Page 8 is handled as well.

If now a WRITE request for Page 9 comes in, we have a problem because this page contains data already. In order to write to this page, we have to first copy all the other pages with data to memory, in this case Page 8, Then we erase the entire block, so that it is empty. Now we can copy back the pages that we previously had to copy to memory, and now we finally can do the write operation.

Let’s do a brief analysis of what had to be done to write a single page:

- First we had to READ all the pages with data into memory.
- Then we had to erase the block, and then PROGRAM the copied pages back onto the flash device, followed by the actual page that we were writing.

The cost of this is two-fold, in addition to having to deal with the case of us loosing all the data when the system crashes while we are erasing the block. First, the cost of writing a single page to memory using this scheme is a millisecond for this very simple device model, namely 70 microseconds to copy page 8 to memory, then 500 musec to erase the block, 220 musec to write Page 8 back onto the device, and another 220musec to write the new data. For a realistic device with 64 pages under moderate utilization the latency can be much higher, in the multiple-seconds range. Clearly there must be a better way.

An additional cost is **wear-out**. As blocks start becoming unusable after say 100'000 ERASE cycles, doing a cycle per write operation is not acceptable. This is particularly important for hot spots on the device, such as directory blocks or inode blocks, which get modified all the time.

5 Solution: Log-Structured Flash Translation Layer

Flash memory manufacturers therefore add a so-called **Flash Translation Layer** (called FTL for short) between the user and the device. The objective of the FTL is to minimize the number of ERASE cycles and to perform the ERASE operations so that they wear out all blocks about equally. This is called wear-leveling. We describe a very simple flash translation layer, but this should be sufficient to give an idea.

As we will see, the FTL dynamically remaps the page number. For this it uses a translation table. It also keeps track of the state of each page. In this example, pages can be EMPTY, meaning that they don't contain data, they can be VALID, meaning that they contain data that can be read, or they can be INVALID, meaning that they contain data, but it cannot be read or is obsolete. We also keep a pointer to the next empty page.

In this example the first 12 pages contain valid data. To keep the example simple, the pages are not-remapped yet, meaning that Page i in the translation table is located in the physical Page i on the device.

When now a WRITE request for Page 1 comes in, which already contains data, the FTL remaps Page 1 to the next empty page, which is Page 12. Physical Page 1 is marked invalid, and the data is PROGRAMMED into physical Page 12, which is now marked valid. We advance the pointer to the next empty page. When now a WRITE request for Page 7 comes in, we pick the next empty page, namely Page 13, we update the map and invalidate physical Page 7. We write the data into physical page 13 and increment the pointer. We do the same for a WRITE to Pages 9, 8, 3 [...] and 6 [...].

It becomes interesting again when we want to write to a page that has been remapped, Page 9 in this case. This page is remapped already to physical page 14. We remap it to the next empty page, which is Page 18. We invalidate the old physical page, Page 14. write the data to the newly mapped physical page, and increment the pointer. And so on, until we run low on empty pages. In this example, the low watermark is two blocks of empty pages.

6 Garbage Collection in the Flash Translation Layer

If we reach the low-watermark, we need to start reclaiming invalid pages, using a garbage collection process. For this we scan one block at a time. For each block, we copy the valid pages to a new block.

In this case physical Page 0 is copied to the next empty page. At the same time the map for Page 0 is updated. We do the same for Page 2. Once all valid pages of a block are copied, we can erase the block, and reclaim all pages. We continue by copying the valid pages of the next block. and erase the block. We do the same for the next block. and the next.

The next block has only invalid pages, so we can erase the block without copying pages. Finally, we copy the three valid pages in the last block, and erase it.

Now we can handle new requests, in this case a WRITE to Page 6. We invalidate Page 30, remap Page 6 from physical Page 30 to physical Page 0. write the data and update the pointer.

If we have a closer look at the ERASE operations, we see that we need, for N over-write operations, only N over the block size ERASE operations. This number is significantly smaller than in the previous nave approach. In addition, the ERASE operations are naturally spread across all blocks, leading to a more balanced wear-out of the blocks and therefore to a long life-time for the flash device.

7 Solid-State Storage Technology: Summary

We have come to the conclusion of this exploration of NAND flash storage as an example of Solid State Storage Technology. NAND Flash is very common in many applications, ranging from persistent memory in cameras and smart phones, to USB thumb drives, and to high performance Solid State Disks (SSD for short). We described the performance characteristics and the idiosyncrasies of NAND flash memory. And we developed a simple performance model for the device.

In an attempt to make flash memory devices usable as traditional blocks devices such as disks, manufacturers often hide the actual flash memory device behind a so-called Flash Translation Layer, which provides efficient READ and WRITE operations as well as wear leveling. We explored the design of a basic log-based flash translation layer. We hope that you enjoyed this exploration of NAND flash memory.

Similarly to hard disks, flash memory does not have a simple performance model, and just as for hard disks, file system designers must account for the behavior of flash memory and its translation layer.