Short reminder

Serial input (x8 or x16): 25ns (MAX CLK)

PROGRAM: ≈ 220μs/page

Serial output (x8 or x16): 25ns (MAX CLK)

READ (page load): ≈ 25μs

BLOCK ERASE: ≈ 500μs

2048 blocks (2Gb SLC device)

64 pages per block

NAND Flash Block

Data area: 2048 bytes

Spare area (ECC, etc.) 64 bytes

8-bit byte or 16-bit word
• HDD latency
  – Random O(1ms)
  – Sequential O(50us)

• Single flash chip (more realistic model)
  – Write
    2KB * (25ns bit transfer) + 220us = 271us
  – Read
    2KB * (25ns bit transfer) + 25us = 76us
  – Random or sequential?

Q: Is it really better than HDD? What's the secret
Flash has its advantages

• Shock resistant
  – Remember solid-state? no mechanical arm, no moving parts
• More predictable
  – no mechanical arm, no moving parts
• Low power (versus HDD)
• Size matters - can use multiple chips where a single HDD was
SSD Design Basics
• P1: what was here before there were SSDs (and generally now too)
• (Usually) magnetic Hard-Disk Drives (HDDs)
• Simple API:
  – Read/write(start sector, # of sectors)
  – SATA, SAS
• SSDs had (and still have) to fit into the legacy storage stack
• Consequently, SSDs *emulate* an HDD to the host
• Similar drivers, generally same simple API...
• **Q:** So where’s the problem?
Imagine a future with driverless cars only

But we can't have it all now, there are still some annoying humans on the road...
SSDs emulating HDDs

- Pros: easy adoption
SSDs emulating HDDs

• **Pros:** easy adoption
  – Storage software stack unchanged
    • Can use reliable software, used, debugged, and tested for decades
  – Hardware stack unchanged: same interfaces, same slots, no need for special controller
  – No need to support large set of different flash chip drivers
  – Vendors keep proprietary details of chips and custom optimizations and hardware
SSDs emulating HDDs

• Cons:
SSDs emulating HDDs

• Cons:
  – Legacy software with irrelevant assumptions
    • Really slow random accesses, no endurance limit, write in-place support...
  – Legacy hardware with irrelevant assumptions
    • Disk can handle one concurrent random request, throughput is limited (400 MB/s)
Do all flash-based devices emulate HDDs?
Do all flash-based devices emulate HDDs?

- No!
- Embedded devices small and simple, use embedded OSs, never used HDDs anyway
- Earlier Android device also managed flash directly using a flash-dedicated file system
  - YAFFS, developed for embedded devices originally
  - But since Android 2.3 (2010) replaced with ext4....
- Some custom SSDs also don’t emulate HDDs
  - Why?
  - More on that later in the course
- Vast majority of SSDs still emulate HDDs
SSD basics: an independent computing unit
Erase & program gap

• Host still thinks its using an HDD...
  – Sector read/writes

• Problem:
  – HDD sectors can be re-written in-place
  – flash has page-level read/write, but erases in erases block (64-256 pages)
Naïve solution

SSD

Temporary buffer

Write( )

obsolete
valid
erased
Naïve solution

SSD

Temporary buffer

Write( )

obsolete
valid
erased
Naïve solution

Why naïve?
Naïve solution

Write amplification & endurance

Write ( )

- obsolete
- valid
- erased
"All problems in computer science can be solved by another level of indirection"

David Wheeler
Realistic solution: log-structured

- Use a log-structured design*
- Original idea: random writes on HDDs suck → sequentialize all random writes
- Append all modifications to disk **sequentially** in a log-like structure
- Entire disk is treated as one large “cyclic” log

* The design and implementation of a log-structured file system, Rosenblum & Ousterhout, 1992
• This really is a “log” of writes (בעברית: יומן (כתיבות)
• Data is not overwritten ➔ past writes still visible
• Log-structured sounds great! Why don’t all HDD file systems use it?
• **Problem 1:** need to write multiple sectors together
• Log-structured sounds great! Why don’t all HDD file systems use it?
  
  • **Problem 1:** need to write sector batches
    – Otherwise, need to wait for disk to return to start of segment
  
  • **Solution:** buffer written data before committing to disk, write at “segment” units (e.g., 128KB, 1MB)
• Remember inode hierarchy?
• Q: Why is synchronous I/O a problem?
• Re-writing file data results in “cascading updates”
• Also, what happens when we fill all the segments?!
Problem 2: free space occupied by invalidated data

- Re-writing data leaves obsolete data behind, a.k.a garbage
- To avoid running out of space, need to re-use sectors occupied by invalidated data
- Simple over-writing → not log-structured
**Solution**: copy-on-write

- One “spare” segment
- Take segment 2 of the log
- Read still-valid data to memory (on demand, or together)
- Copy to empty segment 3
- Use segment 2 as free space!
- **REPEAT**
• Still not ideal for HDDs
• Need to clean segments all the time
  – Lots of seeks to different locations, lots of background reading and copying from different locations in disk...
  – Significant performance overheads
• Perhaps simpler and just as good to just overwrite in-place...

Fortunately, this is a class on SSDs
SSDs as log-structured systems

- Flash chips already partitioned to large “segments” managed as a unit
  - written sequentially, erased as one
- SSDs already need to copy-and-write
  - No write in-place
- SSDs only manage sectors, the file system manages inodes and directories
  - To the file system every sector re-write appears “in-place”
  - **Can use indirection** so that only the SSD is aware of the dirty details → no cascading updates
Naive FTL

• Leave 1 “spare” EU
• Take EU 2 (already written)
• Read valid pages (on demand, or together)
• Copy to EU 3
• ERASE EU 2 and use as free space
• REPEAT cyclically
Naive FTL

- But what if we need to re-write a sector mapped to a physical page at the end of the “log”? 
- Logical address space is much larger than an EU, cant iterate entire log until we reach EU with modified data
Flash translation layer

- SSD firmware responsible for managing flash
- Flash Translation Layer = FTL maps and translates *logical* block addresses (i.e., sectors) to *physical* flash page addresses
- No more static mapping of sectors to the same physical location as in HDDs

- Basic FTL example
Block-level mapping

SSD

Map

LBAs

0-11

24-36

12-23
Block-level mapping

SSD

Map

0-11

24-36

12-23

Read(16)

obsolete
valid
erased

Block-level mapping

SSD

Map

0-11

24-36

12-23

Read(16)

obsolete
valid
erased
Block-level mapping

SSD

Map

LBAs

0-11

24-36

12-23

obsolete
valid
erased

Write(12, )
Block-level mapping

SSD

LBAs
0-11
24-36
12-23

Temporary buffer

Map

obsolete
valid
erased

Write(12, )
Block-level mapping

SSD

LBAs
0-11
24-36
12-23

Map

Write(12, )

obsolete
valid
erased
Block-level mapping

SSD

Map

LBLAs

0-11

24-36

12-23

Write(12, )

obsolete
valid
erased
Block-level mapping

• No seeks as in HDDs
• Smaller performance overhead for copying
• Mapping overhead is minimal
  – 4-byte mapping entry per erase block
  – 1TB SSD, 4MB erase block

\[
\frac{1024 \times 1024 \text{ MB}}{4\text{ MB}} = 1\text{MB of RAM}
\]
Block-level mapping

- No seeks as in HDDs
- Smaller performance overhead for copying
- Mapping overhead is minimal
  - 4-byte mapping entry per erase block
  - 1TB SSD, 4MB erase block

\[
\frac{1024 \times 1024 \text{ MB}}{4\text{ MB}} = 1\text{MB of RAM}
\]

- Q: Can we all go home now? Why not?
What about endurance?

- Some files accessed and modified more frequently → sectors are not re-written uniformly → some EUs contain “hot” data
- If 50% of EUs are “cold”, they are never re-written
  - “Hot” EUs repeatedly erased and move data amongst themselves
  - Wear imbalance kills “hot” blocks earlier
What about endurance?

• **Simple solution:**
  – Track erase counts to differentiate hot/cold blocks
  – Periodically switch data between worn out blocks from hot set to cold set

• More solutions later in the course...
What about metadata?

Goal: quickly recover metadata on startup
- Know which logical addresses each EU covers

- Occasional checkpointing of mapping metadata
- More "spare" EUs

- Write logical address in spare area
- On startup scan all blocks or use battery-backed RAM
What about performance?

- SSDs are arrays of flash units
What about performance?

- Performance achieved via parallelism
- Waiting for one EU copy-on-write on every write is a performance bottleneck
- Q: Are superblocks the answer?
What about write amplification?

- Modifying 4KB of data → read EU, write new EU, erase old EU...
- Effective write granularity is in MBs (EU size)
- **Write amplification factor (WAF) =**
- Block-level mapping WAF = 100x
- Random write performance degradation
- Q: Sequential write performance is good. Why?

![Diagram](image.png)
Block-level mapping, the verdict

• **Pros:**
  – Low RAM overhead
  – Good read throughput
  – Good sequential performance
  – Simplicity

• **Cons:**
  – Bad random write performance, especially for small requests
  – $WAF = O(100)$

Who uses block-level mapping?
Page-level mapping

- Every write request requires accessing and changing mapping data structure
SSD

Map\_LBAs

Write(1, \textbf{1})

Write(1, \textbf{1})
SSD Map

Write(1, )
What about memory requirements

• Mapping data structure significantly larger
  – Modern file systems use 4KB pages → SSDs typically map in 4KB granularity (even if physical pages >4KB)
  – Page-level mapping (4KB granularity) of 1TB of flash is \( \log(4{\text{KB}} \text{ pages}) \frac{1024^4}{4{\text{KB}}} = 262M \) mapping entries
  – 4B per entry = 1GB of RAM

• More RAM = less profit, more power...
What about performance?

- Performance achieved via parallelism
- Less waiting for large copy-on-write operation
- Can write everything in parallel
What about write amplification?

• Write granularity is page size
• Q: WAF = 1?
For simplicity assume "Greedy" policy, chooses "victim" block with max. obsolete pages.
Memory implications

- Goal: minimize metadata read and update overhead
- Track invalid pages to avoid reading pages with obsolete data
- Page bitmap in RAM, 1 bit per page
• Bitmap of 1TB of flash = 262M entries = 32 MB
• Keep mapping always in RAM
• With mapping table O(1GB)
Space implications

• So far assumed 1 spare EU per flash unit
• 32GB flash chip, 4MB EU
  – 32GB / 4MB = 8192 EUs
  – 1/8192 of pages is free on average
• To reduce cleaning costs SSDs are highly over-provisioned
  – For example, 25% of flash is not exposed to user
  – 750GB SSD = 1TB flash
• Dynamically control over-provisioning → better performance!
  – Use only 50% of capacity
Dynamic over-provisioning (1)

- Using less capacity is expensive (equivalent to buying a larger SSD)
- Can also improve over-provisioning dynamically
  - Users leave free space, delete files
  - File systems rarely utilize ALL logical addresses
- **TRIM/DISCARD/UNMAP** (logical addresses)
  - Nullify mapping entry for logical addresses
<table>
<thead>
<tr>
<th>OS Logical View</th>
<th>SSD Logical View (LBAs)</th>
<th>SSD Physical View</th>
</tr>
</thead>
<tbody>
<tr>
<td>File A</td>
<td>File B</td>
<td>File D</td>
</tr>
<tr>
<td>A1</td>
<td>A2</td>
<td>A3</td>
</tr>
<tr>
<td>B2</td>
<td>B3</td>
<td>B4</td>
</tr>
<tr>
<td>B6</td>
<td>C1</td>
<td>C2</td>
</tr>
</tbody>
</table>

Over Provisioning

SSD writes new data; only SSD knows about OP

Only OS knows location C1 & C2 are no longer valid and SSD keeps rewriting it during GC

OS writes new file to old location; SSD marks old location ready for GC and file E gets written elsewhere

http://www.thessdreview.com
Dynamic over-provisioning (2)

Would you do that?

Fraction of LBA space written: 1/8, 1/4, 1/2, 1/1

random 32K writes, direct I/O, commercial page-mapping SSD (Desnoyers’13)
• Legacy interfaces augmented with relevant SSD-specific command support (circa 2008)
• Discard supported by all major storage interfaces, SSDs, OSs and file systems
• But discard not always enabled in OSs by default...why?
Putting it all together

Simulation, Hu et al.'09
Page-level mapping, the verdict

• **Cons:**
  – Non-volatile memory overhead ($)

• **Pros:**
  – Good performance on all access patterns (random/seq, write/read)
  – Write amplification $O(1)$

• **Result:** high-end SSDs always use page-level mapping

• **Q:** Why didn't all of them?