Hard disks, SSDs, and the I/O subsystem

Tyler Bletsch
Duke University

Slides include material from Vince Freeh (NCSU)
Hard Disk Drives 
(HDD)
History

- First: IBM 350 (1956)
  - 50 platters (100 surfaces)
  - 100 tracks per surface (10,000 tracks)
  - 500 characters per track
  - 5 million characters
  - 24” disks, 20” high
Overview

- Record data by magnetizing ferromagnetic material
- Read data by detecting magnetization

Typical design
- 1 or more platters on a spindle
- Platter of non-magnetic material (glass or aluminum), coated with ferromagnetic material
- Platters rotate past read/write heads
- Heads ‘float’ on a cushion of air
- Landing zones for parking heads
Basic schematic

a. side view.
b. top view.
Generic hard drive

[Diagram of a hard drive with labels:
- Platter
- Spindle
- Head
- Actuator Arm
- Actuator Axis
- Power Connector
- Jumper Block
- Data Connector

^ (these aren't common any more)
Types and connectivity (legacy)

- **SCSI (Small Computer System Interface):**
  - Pronounced “Scuzzy”
  - One of the earliest small drive protocols
  - The Standard That Will Not Die: the drives are gone, but most enterprise gear still speaks the SCSI protocol

- **Fibre Channel (FC):**
  - Used in some Fibre Channel SANs
  - Speaks SCSI on the wire
  - Modern Fibre Channel SANs can use any drives: back-end ≠ front-end

- **IDE / ATA:**
  - Older standard for consumer drives
  - Obsoleted by SATA in 2003
Types and connectivity (modern)

- **SATA (Serial ATA):**
  - Current consumer standard
  - Series of backward-compatible revisions
    - SATA 1 = 1.5 Gbit/s, SATA 2 = 3 Gbit/s,
      SATA 3 = 6.0 Gbit/s, SATA 3.2 = 16 Gbit/s
  - Data and power connectors are hot-swap ready
  - Extensions for external drives/enclosures (eSATA),
    small all-flash boards (mSATA, M.2),
    multi-connection cables (SFF-8484), more
  - Usually in 2.5” and 3.5” form factors

- **SAS (Serial-Attached-SCSI)**
  - SCSI protocol over SATA-style wires
  - (Almost) same connector
  - Can use SATA drives on SAS controller, not vice versa
Hard drive capacity

Seeking

- **Steps**
  - Speedup
  - Coast
  - Slowdown
  - Settle
- **Very short seeks (2-4 tracks):** dominated by settle time
- **Short seeks (<200-400 tracks):**
  - Almost all time in constant acceleration phase
  - Time proportional to square root of distance
- **Long seeks:**
  - Most time in constant speed (coast)
  - Time proportional to distance
Average seek time

- What is the “average” seek? If
  1. Seeks are fully independent and
  2. All tracks are populated:
    ➔ average seek = 1/3 full stroke
- But seeks are not independent
- Short seeks are common

- Using an average seek time for all seeks yields a poor model
Zoning

- **Note**
  - More linear distance at edges than at center
  - Bits/track ~ R (circumference = \(2\pi R\))
  - To maximize density, bits/inch should be the same

- **How many bits per track?**
  - Same number for all \(\rightarrow\) simplicity; lowest capacity
  - Different number for each \(\rightarrow\) very complex; greatest capacity

- **Zoning**
  - Group tracks into zones, with same number of bits
  - Outer zones have more bits than inner zones
  - Compromise between simplicity and capacity
Sparing

- Reserve some sectors in case of defects
- Two mechanisms
  - Mapping
  - Slipping
- Mapping
  - Table that maps requested sector → actual sector
- Slipping
  - Skip over bad sector
- Combinations
  - Skip-track sparing at disk “low level” (factory) format
  - Remapping for defects found during operation
Caching and buffering

- Disks have caches
  - Caching (e.g., optimistic read-ahead)
  - Buffering (e.g., accommodate speed differences bus/disk)

- Buffering
  - Accept write from bus into buffer
  - Seek to sector
  - Write buffer

- Read-ahead caching
  - On demand read, fetch requested data and more
  - Upside: subsequent read may hit in cache
  - Downside: may delay next request; complex
Command queuing

- Send multiple commands (SCSI)
- Disk schedules commands
- Should be “better” because disk “knows” more

Questions
  - How often are there multiple requests?
  - How does OS maintain priorities with command queuing?
Time line

Read
- Host sends command
- Controller decodes it
- Controller disconnects from bus & starts seek
- Seek
- Rotation latency
- Data transfer off mechanism
- Head switch
- SCSI bus
- Disk mechanism

Write
- Host sends command
- Controller decodes it
- SCSI bus data transfer from host
- Controller starts seek
- Seek
- Rotation latency
- Data transfer to mechanism
- Head switch
- SCSI bus
- Disk mechanism
## Disk Parameters

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Diameter</td>
<td>3.5”</td>
<td>2.5”</td>
<td>1.8”</td>
</tr>
<tr>
<td>Capacity</td>
<td>Improving ☺ 6 TB</td>
<td>73 GB</td>
<td>10 GB</td>
</tr>
<tr>
<td>RPM</td>
<td>7200 RPM</td>
<td>10000 RPM</td>
<td>4200 RPM</td>
</tr>
<tr>
<td>Cache</td>
<td>Improving ☺ 128 MB</td>
<td>8 MB</td>
<td>512 KB</td>
</tr>
<tr>
<td>Platters</td>
<td>~6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Average Seek</td>
<td>About equal ☹ 4.16 ms</td>
<td>4.5 ms</td>
<td>7 ms</td>
</tr>
<tr>
<td>Sustained Data Rate</td>
<td>Improving ☺ 216 MB/s</td>
<td>94 MB/s</td>
<td>16 MB/s</td>
</tr>
<tr>
<td>Interface</td>
<td>SAS/SATA</td>
<td>SCSI</td>
<td>ATA</td>
</tr>
<tr>
<td>Use</td>
<td>Desktop</td>
<td>Laptop</td>
<td>Ancient iPod</td>
</tr>
</tbody>
</table>
Solid State Disks (SSD)
Introduction

• Solid state drive (SSD)
  • Storage drives with no mechanical component
  • Available up to 16TB capacity (as of 2019)
  • Classic: 2.5” form factor (card in a box)

  ![SSD Image]
  Source: wikipedia

• Modern: M.2 or newer NVMe (card out of a box)

  ![Modern SSD Image]
Evolution of SSDs

- PROM – programmed once, non erasable
- EPROM – erased by UV lighting*, then reprogrammed
- EEPROM – electrically erase entire chip, then reprogram
- Flash – electrically erase and rerecord a single memory cell
- SSD - flash with a block interface emulating controller

* Obsolete, but totally awesome looking because they had a little window:
Flash memory primer

• Types: NAND and NOR
  • NOR allows bit level access
  • NAND allows block level access
    • For SSD, NAND is mostly used, NOR going out of favor

• Flash memory is an array of columns and rows
  • Each intersection contains a memory cell
    • Memory cell = floating gate + control gate
    • 1 cell = 1 bit
## Memory cells of NAND flash

<table>
<thead>
<tr>
<th>Single-level cell (SLC)</th>
<th>Multi-level cell (MLC)</th>
<th>Triple-level cell (TLC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single (bit) level cell</td>
<td>Two (bit) level cell</td>
<td>Three (bit) level cell</td>
</tr>
<tr>
<td>Fast: 25us read/100-300 us write</td>
<td>Reasonably fast: 50us read, 600-900us write</td>
<td>Decently fast: 75us read, 900-1350 us write</td>
</tr>
<tr>
<td>Write endurance - 100,000 cycles</td>
<td>Write endurance – 10000 cycles</td>
<td>Write endurance – 5000 cycles</td>
</tr>
<tr>
<td>Expensive</td>
<td>Less expensive</td>
<td>Least expensive</td>
</tr>
</tbody>
</table>
SSD internals

Package contains multiple dies (chips)

Die segmented into multiple planes

A plane with thousands (2048) of blocks + IO buffer pages

A block is around 64 or 128 pages

A page has a 2KB or 4KB data + ECC/additional information
SSD operations

- **Read**
  - Page level granularity
  - 25us (SLC) to 60us (MLC)

- **Write**
  - Page level granularity
  - 250us (SLC) to 900us (MLC)
  - 10 x slower than read

- **Erase**
  - Block level granularity, not page or word level
  - Erase must be done before writes
  - 3.5ms
  - 15 x slower than write
SSD internals

- Logical pages striped over multiple packages
  - A flash memory package provides 40MB/s
  - SSDs use array of flash memory packages

- Interfacing:
  - Flash memory → Serial IO → SSD Controller → disk interface (SATA)

- SSD Controller implements Flash Translation Layer (FTL)
  - Emulates a hard disk
  - Exposes logical blocks to the upper level components
  - Performs additional functionality
SSD controller

- Differences in SSD is due to controller
  - Performance loss if controller not properly implemented
- Has CPU, RAM cache, and may have battery/supercapacitor
- Dynamic logical block mapping
Wear leveling

- SSDs wear out
  - Each memory cell has finite flips
  - All storage systems have finite flips even HDD
  - SSD finite flips < HDD
  - HDD failure modes are larger than SSD

- General method: over-provision unused blocks
  - Write on the unused block
  - Invalidate previous page
  - Remap new page
Dynamic wear leveling

- Only pool unused blocks
- Only non-static portion is wear leveled
- Controller implementation easy
- Example: SSD lifespan dependent on 25% of SSD

Source: Micron
Static wear leveling

- Pool all blocks
- All blocks are wear leveled
- Controller complicated
  - needs to track cycle # of all blocks
- Static data moved to blocks with higher cycle #
- Example: SSD lifespan dependent on 100% of SSD

Source: micron
Preemptive erasure

- Preemptive movement of cold data
- Recycle invalidated pages
  - Performed by garbage collector
  - Background operation
  - Triggered when close to having no more unused blocks
SSD TRIM! Sent from the OS

- **TRIM**
  - Command to notify SSD controller about deleted blocks
  - Sent by filesystem when a file is deleted
  - Avoids write amplification and improves SSD life
Using SSD (1)

- SSD as main storage device
  - NetApp “All Flash” storage controllers
  - 300,000 read IOPS
  - < 1 ms response time
  - > 6Gbps bandwidth
  - Cost: $big
  - Becoming increasingly common as SSD costs fall

- Hybrid storage (tiering)
  - Server flash
    - Client cache to backend shared storage
    - Accelerates applications
    - Boosts efficiency of backend storage (backend demand decreases by upto 50%)
    - Example: NetApp Flash Accel acts as cache to storage controller
      - Maintains data coherency between the cache and backend storage
      - Supports data persistent for reboots
• Hybrid storage
  • Flash array as cache (PCI-e cards flash arrays)
    • Example: NetApp Flash Cache in storage controller
    • Cache for reads
  • SSDs as cache
    • Example: NetApp Flash Pool in storage controller
    • Hot data tiered between SSDs and HDD backend storage
    • Cache for read and write
NetApp EF540 flash array

- 2U
- Target: transactional apps with high IOPS and low latency
- Equivalent to > 1000 15K RPM HDDs
- 95% reduction in space, power, and cooling
- Capacity: up to 38TB

Source: NetApp
# Differences between SSD and HDD

<table>
<thead>
<tr>
<th></th>
<th>SSD</th>
<th>HDD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Uniform seek time</strong></td>
<td>Fast seek time – random read/writes as fast as sequential read/writes</td>
<td>Different seek time for different sectors</td>
</tr>
<tr>
<td><strong>Fast seek time</strong></td>
<td>Seek time dependent upon the distance</td>
<td></td>
</tr>
<tr>
<td><strong>Cost (Intel 530 Series 240GB – $209)</strong></td>
<td>Cost (Seagate Constellation 1TB 7200rpm - $116)</td>
<td></td>
</tr>
<tr>
<td>• Capacity</td>
<td>$0.87/GB</td>
<td>$0.11/GB</td>
</tr>
<tr>
<td>• Rate</td>
<td>$0.005/IOPS</td>
<td>$0.55/IOPS</td>
</tr>
<tr>
<td>• Bandwidth</td>
<td>$0.38/Mbps</td>
<td>$0.99/Mbps</td>
</tr>
<tr>
<td><strong>Power:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active power</td>
<td>195mW – 2W</td>
<td>Average operating power: 5.4W</td>
</tr>
<tr>
<td>Idle power</td>
<td>125mW – 0.5 W</td>
<td>Higher power consumption, sleep mode zero power, higher wake up cost</td>
</tr>
<tr>
<td>Low power consumption, No sleep mode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Differences between SSD and HDD

<table>
<thead>
<tr>
<th>SSD</th>
<th>HDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 10,000 to &gt; 1 million IOPS</td>
<td>Hundreds of IOPS</td>
</tr>
<tr>
<td>Read/write in microseconds</td>
<td>Read/write in milliseconds</td>
</tr>
<tr>
<td>No mechanical part – no wear and tear</td>
<td>Moving part – wear and tear</td>
</tr>
<tr>
<td>MTBF ~ 2 million hours</td>
<td>MTBF ~ 1.2 million hours</td>
</tr>
<tr>
<td>Faster wear of a memory cell when it is written multiple times</td>
<td>Slower wear of the magnetic bit recording</td>
</tr>
</tbody>
</table>
Intel X-25E - $345 (older)
SLC
32 GB
SATA II
170-250MB/s
Latency 75-85us

Intel 530 - $209 (new)
MLC
240GB
SATA III
up to 540MB/s
Latency 80-85us

Samsung 840 EVO - $499 (new)
TLC
1TB
SATA III
up to 540MB/s
Which is cheaper?

**HDD?**
Yes!

Cheaper per gigabyte of capacity.

or

**SSD?**
Yes!

Cheaper per IOPS (performance).

Tradeoff!
## Workloads

<table>
<thead>
<tr>
<th>Workloads</th>
<th>SSD</th>
<th>HDD</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>High write</td>
<td>Y</td>
<td>Y</td>
<td>Wear for SSD</td>
</tr>
<tr>
<td>Sequential IO (e.g. media files)</td>
<td>Y</td>
<td>Y</td>
<td>Both SSD and HDD do great on sequential</td>
</tr>
<tr>
<td>Log files (small writes)</td>
<td>Y</td>
<td></td>
<td>Faster seek time</td>
</tr>
<tr>
<td>Database read queries</td>
<td>Y</td>
<td></td>
<td>Faster seek time</td>
</tr>
<tr>
<td>Database write queries</td>
<td>Y</td>
<td></td>
<td>Faster seek time</td>
</tr>
<tr>
<td>Analytics – HDFS</td>
<td>Y</td>
<td>Y</td>
<td>SSD – Append operation faster HDD – higher capacity</td>
</tr>
<tr>
<td>Operating systems</td>
<td>Y</td>
<td></td>
<td>SSD: FAST!!!!</td>
</tr>
</tbody>
</table>
Other Flash technologies - NVDIMMS

- Revisiting NVRAM
- DDR DIMMS + NAND Flash
  - Speed of DIMMS
  - extensive read/write cycles for DIMMS
  - Non volatile nature of NAND Flash
- Support added by BIOS
  - Backup to NAND Flash
  - Triggered by HW SAVE signal
- Stored charge
  - Super capacitors
  - Battery packs

How It Works
If there is a power failure, the supercap module powers NVDIMM while it copies all data from the DDR-3 to on-module flash

When power is restored NVDIMM copies all data from flash to DDR-3 and normal operation resumes

(SNIA - NVDIMM Technical Brief )
In future - persistent memory

- NVM latency closer to DRAM
- Types
  - Battery-backed DRAM, NVM with caching, Next-gen NVM
- Attributes:
  - Bytes-addressable, LOAD/STORE access, memory-like, DMA
  - Data not persistent until flushed

Source: Andy Rudoff, Intel
Basics of IO Performance Measurement
Motivation and basic terminology

• We cover performance measurement in detail later in the semester, but you may need the basics for your project sooner than that...

• The short version:
  • Sequential workload: **MB/s**
    • Even an SSD does better sequential than random because of caching and other locality optimizations
  • Random workload: **IO/s** (commonly written IOPS)
    • You need to indicate the IO size, but it’s not part of the metric
  • Don’t forget: **latency (ms)**
Measurement methodology

• Basic test: do X amount of IO and divide by time T.
  • Both X and T may be specified or measured
  • Example:
    • Measure time to do 100,000 IOs (X given, T free variable)
    • Write to disk at max rate for 60 seconds, look at file size (T given, X free variable)

• Problem: measurement variance

```bash
$ dd if=/dev/zero of=testfile bs=1k count=1k
1024+0 records in 1024+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.00917473 s 114 MB/s
tkbletsc@LAPIS ~ $ dd if=/dev/zero of=testfile bs=1k count=1k
1024+0 records in 1024+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.0101952 s 103 MB/s
tkbletsc@LAPIS ~ $ dd if=/dev/zero of=testfile bs=1k count=1k
1024+0 records in 1024+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.0108398 s 96.7 MB/s
tkbletsc@LAPIS ~ $ dd if=/dev/zero of=testfile bs=1k count=1k
1024+0 records in 1024+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.0105439 s 99.4 MB/s
tkbletsc@LAPIS ~ $ dd if=/dev/zero of=testfile bs=1k count=1k
1024+0 records in 1024+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.00812217 s 129 MB/s
tkbletsc@LAPIS ~ $ dd if=/dev/zero of=testfile bs=1k count=1k
```
Combating measurement variance (1)

- Measurement varying too much? Make sure your tests are long enough!
  - Otherwise you’re testing tiny random effects instead of the actual phenomenon under study...
Combating measurement variance (2)

• Measurement variance never goes away
  • Need to characterize it when presenting results, or you won’t be trusted!
  • How? Take multiple repetitions show average and standard deviation (or other variance metric)

• ALL data requires variance to be characterized!
  (not just in this course, but in your life)
  • For your projects, failure to characterize variance is likely an automatic request for resubmission!!

• How to present:
  • In tables, show variance next to average (e.g. “251.2 ± 11.6”)
  • In graphs, show variance with error bars, e.g.: 

![Graph showing error bars for test1 and test2](image)
I’m going to **live demo** a lot of command-line tools and concepts: watching live or reviewing a video recording may be of more value than just the slides.
Fundamental concepts in UNIX

- UNIX figured out a **lot** of what is smart in OS design.
- One insight: **Everything is a file**
  - All hardware is represented as special **device files**. Described by “major” and “minor” numbers to tell kernel what device you mean.
  - Devices automatically created in special filesystem “/dev”
  - Includes block devices (e.g., HDDs and SSDs)
    - `/dev/sda, /dev/sdb, /dev/sdc, ... = SCSI Disk A, B, C, ...
  - List block devices with `lsblk`:
Doing basic IO manually

- Can open/read/write/close block devices like any other
  - Requires root access by default (e.g. via `sudo`)
  - Any program can do this – no special interface!
    - Bash commands, python, etc.

- Useful to have a tool for doing basic IO with lots of options
  - Introducing `dd`!

- Basic usage:
  - `dd if=INPUTFILE of=OUTPUTFILE bs=1k count=32`
    - Defaults to stdin if omitted
    - Defaults to stdout if omitted
    - Defaults to 512 if omitted
    - Defaults to all if omitted
  - `dd if=/dev/sdb of=/dev/null bs=1 count=1`
    - Read from disk B
    - Discard result
    - 1 byte in total

- Lots more options, see manpage for details!
Block device tracing

- Kernel can trace the activity to block devices for us
- Install it:
  ```
sudo apt install blktrace
  ```
- Default: blktrace stores trace in binary format in a file; blkparse used to view it in text
  - Can chain the two to get live trace on screen (as root):
    ```
    blktrace -d /dev/sdb -o - | blkparse -i -
    ```

<table>
<thead>
<tr>
<th>Device major,minor</th>
<th>Sequence#</th>
<th>Time (s)</th>
<th>PID</th>
<th>“Action”</th>
<th>“RWBS”</th>
<th>Block#</th>
<th>Blocks</th>
<th>App name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q=Queued</td>
<td>G=Get request</td>
<td>P/U= “Plug”/”Unplug”</td>
<td>I=Insert into device queue</td>
<td>D=Device command issued</td>
<td>C=Completed</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See man blkparse for more.

R=Read
W=Write
N=None (placeholder)
D=Discard (trim)
A=readahead
S=synchronous
more…
Let’s directly use this disk!

• Write “hello” to the very front of it? Easy:
  • `echo hello > /dev/sdb`

• Read the raw bytes of the disk?
  • Could use ‘cat’, but it will read the whole disk...
  • Can use ‘dd’, but what about non-text content?
  • Need a way to interpret binary bytes so we can see them onscreen
  • We want a **hex dump**
    • Three flavors:
      • `hd`: Gives binary+ascii dump by default (other options available)
      • `hexdump`: Get a binary+ascii dump with `hexdump -C` (other options available)
      • `od`: Gives octal by default (other options available)

```bash
root@esaXX:~ # hd /dev/sdb
00000000 68 65 6c 6c 6f 0a 00 00 00 00 00 00 00 00 00 00 |hello...........|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
```

* means “this row repeats for a while
Living without a filesystem

- So far, no filesystem. Screw it – we don’t need a filesystem!
- I put my taxes at offset 1000
  
  ```
  echo "IRS form 1040 ..." | dd of=/dev/sdb bs=1 seek=1000
  ```
- I put my dog picture at offset 2000
  
  ```
  dd if=dog.jpg of=/dev/sdb bs=1 seek=2000
  ```
- I can retrieve the stuff!
Inventing the filesystem

• Wow, remembering these offsets is hard. I’ll write them down...ON THE DISK!
  • echo “taxes: 1000, dog: 2000, ...” > /dev/sdb

• Wow, manually doing the seeks to read/write areas of the disk is hard. I’ll invent OS functions that do it for me...and update the file locations automatically!!!!!!
  • I’ll call the data containers “files”
  • I’ll organize them into hierarchical “directories”
  • I’ll give them the concept of “size” so I know when they end
  • I’ll keep track of what areas of the disk aren’t used and call that “free”
  • I’ll call that special info that describes files my “meta-data”

• To access data, programs will “open” the file (confirm it exists), then “read” and “write” to it, then “close” it – that’s a great interface!
Life was good, until….  

• “I love that my whole hard drive is now organized!”

• But wait, what’s this? What if you have ANOTHER DRIVE?????
Another UNIX insight: **One global hierarchy**

- A UNIX system has a single root directory with a root file system.
- Other filesystems can be “mounted” in directories under the root.

Also, filesystems don’t have to just hold “real” files on “real” storage devices – there are virtual filesystems:

- `/proc` – info about processes and basic system info (used by `top`)
- `/sys` – info about kernel (used by `blktrace`)
- `/dev` – access to device files themselves (managed by `udev`)
- Ramdisk – files live in memory, wiped on reboot (e.g. `tmpfs`)
See what’s mounted

- Two commands to see what’s mounted:
  - `mount` – shows all filesystems (real and virtual)
  - `df` – shows disk free space on filesystems that have that concept
    - (Side-effect: shows fewer “fake” filesystems, more concise)
Partitioning

- What if I want to put multiple filesystems on one device?
  - Examples:
    - Multiple operating systems (e.g. Windows and Linux)
    - An area for files and an area for virtual memory swap space
    - Keep the OS separate from user home directories (so user data filling up doesn’t affect the OS)

- Solution: **partitioning**
  - Widely supported scheme to divide up a disk; partitions are contiguous and small in number (usually 1-3).
  - Partitions labeled with integer that hints at what type of data is there.
  - Two standards: MBR (deprecated) and GPT (GUID Partition Table).
  - The **partition table** occupies beginning of disk, file systems actually live within partitions. The OS knows about this and gives partitions numbered device files:
    /dev/sdb is partitioned into /dev/sdb1, /dev/sdb2, etc.
Partitioning with `cfdisk`

- Run `cfdisk /dev/sdb`
  - Follow prompts and we can make partitions, set type, etc.
  - Hit “Write” when done. Result in `lsblk`:

```bash
root@esaXX:~# lsblk
NAME      MAJ:MIN RM  SIZE   RO TYPE MOUNTPOINT
sda        8:0    0  120G   0  disk          
sdb        8:16   0  144G   0  disk          
sdc        8:32   0  144G   0  disk          
sdd        8:48   0  144G   0  disk          
sde        8:64   0  144G   0  disk          
sr0        11:0   0    1M   0 ।  blk/driver
```
Filesystem choices

• Let’s put a filesystem on, but which one?
  • Common picks:
    • ext4 – common Linux default
    • btrfs – fancy Linux option with lots of special features
    • FAT – classic Windows/DOS filesystem still in use on SD cards; called vfat in Linux
    • NTFS – modern Windows filesystem
    • HFS+ - modern Mac OSX filesystem

• Need to initialize a filesystem: write on-disk metadata structures on that represent empty filesystem. Use mkfs

• Let’s pick a simple filesystem: vfat
  (Why? Because ext4 does fancy background stuff that gets noisy to trace)

• Run mkfs.vfat /dev/sdb1
  • Watch blktrace as it goes – wheeeee!
Let’s mount it

- Make an empty dir as a mountpoint: `mkdir /mnt/blah`
- Mount it: `mount /dev/sdb1 /mnt/blah`
  - Kernel will scan partition and auto-detect type of filesystem
  - Will load correct filesystem driver
  - Now, OS calls to paths under there will get handled by that driver
  - Driver satisfies all OS calls by doing readblock/writeblock requests to the underlying block device
  - That’s how filesystems work!

![Diagram of filesystem structure]

A cache! Let’s experiment and understand this…

Figure adapted from Gotzon Gregor
Test the block cache (1)

```
echo hi > file
• No blktrace output! (OS cache is writeback by default)

cat file
• No blktrace output! (Cache hit)

(Wait about a minute, it posts later to blktrace)
• Yes blktrace output! (Cache being flushed on a timer, see metadata+data changes)

echo hi > file
• No blktrace output! (Writeback cache again)

sync
• Yes blktrace output! (This command forces OS to flush cache)

cat file
• No blktrace output! (Still a hit, just block isn’t dirty in cache)
```
echo 3 > /proc/sys/vm/drop_caches
- Writing to this special file tells kernel to drop caches;
- No blktrace output though, but ramcache was cleared.

cat file
- Blktrace output – we miss because we dropped caches

umount /mnt/blah
mount -o sync /dev/sdb1 /mnt/blah
- Unmount and remount with the ‘sync’ mount option
- Forces writethrough cache mode!

echo hi > file
- Blktrace output immediately! No writeback cache, writethrough instead

cat file
- No blktrace output - it still caches reads
Let’s trace from the other side

- We’ve been tracing the block device
- What about the OS requests?

**strace**

- Shows each OS syscall done by a program.
  - Works on a command by default; can attach to already-running program if desired
  - Have to wade through some “noise” (unrelated calls), not hard with a little experience

- VERY powerful and useful – can determine *behavior* of software without looking at source code or machine instructions!
strace example

root@esaXX:/mnt/blah# strace dd if=/dev/sdb bs=1 count=1
execve("/usr/bin/dd", ["dd", "if=/dev/sdb", "bs=1", "count=1"], 0x7ffec5104518 ...) = 0

{A bunch of openat, pread64, mmap, mprotect, rt_sigaction, brk, etc.: set up dynamic libraries and prep malloc (ignore)}

openat(AT_FDCWD, "/dev/sdb", O_RDONLY) = 3
dup2(3, 0) = 0
close(3) = 0
lseek(0, 0, SEEK_CUR) = 0

{A bunch of openat and read calls relating to “locale” – language translations (ignore)}

read(0, "\0", 1) = 1
write(1, "\0", 1) = 1
close(0) = 0
close(1) = 0
write(2, "1+0 records in\n1+0 records out\n" , 311+0 records in
1+0 records out)
) = 31
write(2, "1 byte copied, 0.000672287 s, 1."..., 381 byte copied, 0.000672287 s, 1.5 kB/s) = 38
write(2, "\n", 1 ) = 1
close(2) = 0
exit_group(0) = ?
+++ exited with 0 +++

Open the input device, rename it to file descriptor 0 (dd likes to pretend its input is always stdin, which is 0)

Read the one requested byte from fd 0 (disk) and write to fd 1 (stdout), then close both.

Report to stderr the statistics. Blue stuff is dd’s actual output to stderr; black is strace telling us about it.
Let’s play

• Let’s try some other strace+dd combos, and let’s watch blktrace as we do!

• Things to observe
  • Note how bs sets the read/write size for OS calls, but a single call could turn into many block IOs
  • Note the effect of read-ahead caching by the OS
  • Note how the cache can be a mix of hits and misses
  • We can use the “-t” option with blkparse to get timing info
    • Observe the correlation between block operations and slower dd results (i.e., cache misses)
Architecture conclusions

- Disks are **block devices**
- All devices in Linux/UNIX are represented by **device files**; can directly interact with
- Disk blocks are cached in RAM by operating system (**buffer cache**)
- This is cumbersome to store data, so we invent **filesystems**
- OS handles filesystems – many filesystems can be mounted at once; the **VFS layer** pivots among them, using the right **filesystem driver**
- Filesystem driver will issue read/write requests to **disk driver**
Tool conclusions

- We learned lots of great tools/commands:
  - `lsblk`: View block devices
  - `df`: View attached “real” filesystems (and free space)
  - `mount`: Without arguments, shows all mounted filesystems
  - `dd`: Simple tool to do sequential IO operations
  - `hd` and `hexdump`: View binary data in human-readable way
  - `mount` and `umount`: Mount and unmount filesystems
  - `cfdisk`: Create and manage disk partitions
  - `mkfs.*`: Create various filesystems on a block device
  - `blktrace` and `blkparse`: Trace IO operations to physical block devices
  - `strace`: Trace system calls being made by a program
  - `sync`: Force OS to flush all dirty blocks in writeback cache to disk
  - `echo 3 > /proc/sys/vm/drop_caches`: Force OS to lose entire block cache content
Questions?
Backup slides
I/O Systems

- Processor
- Cache
- Main Memory
- I/O Controller
- Disk
- I/O Controller
- Graphics
- I/O Controller
- Network

Interrupts
**I/O Interface**

*Independent I/O Bus*

- CPU
- Memory
- Interface
- Peripheral

*Common memory & I/O bus*

- CPU
- Memory
- Interface
- Peripheral

Seperate I/O instructions (in,out)

Lines distinguish between I/O and memory transfers
Memory Mapped I/O

Single Memory & I/O Bus
No Separate I/O Instructions

CPU
Memory
Interface
Peripheral
Interface
Peripheral
CPU
L2
Memory Bus
I/O bus
Memory
Bus Adaptor

ROM
RAM
I/O
Programmed I/O (Polling)

- CPU
- Memory
- IO Controller
- Device

- Is the data ready?
- Read data
- Store data
- done?

busy wait loop not an efficient way to use the CPU unless the device is very fast!

but checks for I/O completion can be dispersed among computationally intensive code
Interrupt Driven Data Transfer

User program progress only halted during actual transfer.
Direct Memory Access (DMA)

- Interrupts remove overhead of polling...
- But still requires OS to transfer data one word at a time
  - OK for low bandwidth I/O devices: mice, microphones, etc.
  - Bad for high bandwidth I/O devices: disks, monitors, etc.

**Direct Memory Access (DMA)**
- Transfer data between I/O and memory without processor control
- Transfers entire blocks (e.g., pages, video frames) at a time
  - Can use bus “burst” transfer mode if available
- Only interrupts processor when done (or if error occurs)
DMA Controllers

- To do DMA, I/O device attached to **DMA controller**
  - Multiple devices can be connected to one DMA controller
  - Controller itself seen as a memory mapped I/O device
    - Processor initializes start memory address, transfer size, etc.
  - DMA controller takes care of bus arbitration and transfer details
    - So that’s why buses support arbitration and multiple masters!
I/O Processors

- A DMA controller is a very simple component
  - May be as simple as a FSM with some local memory
- Some I/O requires complicated sequences of transfers
  - I/O processor: heavier DMA controller that executes instructions
    - Can be programmed to do complex transfers
    - E.g., programmable network card
Top questions to ask about any I/O system:

- **Storage device(s):**
  - What kind of device (SSD, HDD, etc.)?
  - Performance characteristics?

- **Topology:**
  - What’s connected to what (buses, IO controller(s), fan-out, etc.)?
  - What protocols in use (SAS, SATA, etc.)?
  - Where are the bottlenecks (PCI-E bus? SATA protocol limit? IO controller bandwidth limit?)
  - Protocol interaction: polled, interrupt, DMA?