Linux storage system basics
A computer runs programs that are loaded in memory
  - The program is loaded from a storage device
  - The result of an execution is stored to a storage device

Storage device is
  - Nonvolatile
  - Larger than memory
  - Slower than memory
Punch card / punch tape

- Stores program and data to load
  - Indexed by physical meaning (cabinet, ...)

History of storage devices
**Magnetic tape**

**History of storage devices**

- **UNIVAC / 1951**
- **tar**: sequential I/O

<table>
<thead>
<tr>
<th>header</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>header</td>
<td>data</td>
</tr>
<tr>
<td>header</td>
<td>data</td>
</tr>
</tbody>
</table>

![Image of UNIVAC and magnetic tape]
Magnetic disk

- 3 dimensional addressing (head, cylinder, sector)
- Still used as a main storage device
Magnetic disk characteristics

- Mechanical movement defines I/O latency
  - \( T_{seek} + T_{rot} + \frac{Volume}{Bandwidth} \)
  - \( T_{avg} = \frac{1}{2 \cdot RPM} + \frac{Volume}{Bandwidth} \)

- Smaller the request, lower the throughput
  - \( T_{avg} = \frac{1}{2 \cdot RPM} \) vs. \( T_{avg} = \frac{Volume}{Bandwidth} \)
  - 8~32MB I/O does the magic
Non-volatile memories

- **Flash memory**

![Flash Memory Element Structure](image)

- **Future**: PCM, STTM, ZRAM, …

<table>
<thead>
<tr>
<th>Technology</th>
<th>Density</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLC Flash</td>
<td>$4 F^2$</td>
<td>$4 F^2$</td>
</tr>
<tr>
<td>PCM</td>
<td>10 $F^2$</td>
<td>$4 F^2$</td>
</tr>
<tr>
<td>STTM</td>
<td>64 $F^2$</td>
<td>15 $F^2$</td>
</tr>
<tr>
<td>DRAM</td>
<td>6 $F^2$</td>
<td>$4 F^2$</td>
</tr>
</tbody>
</table>
Flash memory characteristics

• **Pros**
  – Small form factor
  – No mechanical parts: good random I/O throughput
  – Low power consumption

• **Cons**
  – Read, program (write), erase interface
    • Slow page programming, even slower block erase
  – Limited block erase cycle
  – Expensive (price / capacity)

• **Types and timing**
  – SLC / MLC / TLC
Storage device characteristics

- Non-volatile
- Slow and large
- Storage hierarchy

Storage hierarchy:
- CPU
  - Register
  - L1/L2 cache
- L3 cache
- RAM
- HDD / SSD
- Tape

Faster

Larger

Cache for RAM
Cache for HDD/SSD
Primary storage
Backup of HDD/SSD
Storage abstraction

- Flat address
- File system: ext4, VFAT, NFS
- Block I/O layer
- Device driver
The Block I/O Layer
Block device concepts

• **Old HDD access method:** (head, cylinder, sector)
  – Differs for each disk
  – SSDs have different structure

• **Generic method:** contiguous address space

  \[
  \begin{array}{ccccccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
  \end{array}
  \]

  – Let storage devices do the complex managements
    • (Head, cylinder) mapping, flash translation layer, ...
    • Bad block management

• **Access units:** sectors (512 bytes) / blocks (4KB)
I/O subsystem architecture

- read/write (device, addr, offset, nr sectors)
- Hard to manipulate
• **Split storage area**: safe-keeping, multi OS

<table>
<thead>
<tr>
<th>/</th>
<th>/usr/local</th>
<th>/home</th>
<th>swap</th>
</tr>
</thead>
</table>

• **Partition table types**
  – Master Boot Record
  – GUID Partition Table
    • Large device support
    • Partition name support
    • Consistency mechanism
Master boot record

- Location: first sector of a disk

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>440</td>
<td>Boot strap code</td>
</tr>
<tr>
<td>4</td>
<td>Disk signature</td>
</tr>
<tr>
<td>2</td>
<td>0x00</td>
</tr>
<tr>
<td>64</td>
<td>Partition table * 4</td>
</tr>
<tr>
<td>2</td>
<td>MBR signature</td>
</tr>
<tr>
<td>2</td>
<td>0xA0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Partition status</td>
</tr>
<tr>
<td></td>
<td>- 0x80: bootable</td>
</tr>
<tr>
<td></td>
<td>- 0x00: normal</td>
</tr>
<tr>
<td></td>
<td>- etc.: error</td>
</tr>
<tr>
<td>3</td>
<td>CHS of first sector</td>
</tr>
<tr>
<td>1</td>
<td>Partition type</td>
</tr>
<tr>
<td></td>
<td>- 0x0b: FAT</td>
</tr>
<tr>
<td></td>
<td>- 0x83: Linux</td>
</tr>
<tr>
<td>3</td>
<td>CHS of last sector</td>
</tr>
<tr>
<td>4</td>
<td>LBA of first sector</td>
</tr>
<tr>
<td>4</td>
<td>Partition size</td>
</tr>
</tbody>
</table>
## GUID partition table

- **Location:** first 34, last 33 sectors

### Legacy MBR

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Partition type GUID</td>
</tr>
<tr>
<td>16</td>
<td>Partition GUID</td>
</tr>
<tr>
<td>8</td>
<td>First LBA</td>
</tr>
<tr>
<td>8</td>
<td>Last LBA</td>
</tr>
</tbody>
</table>

### Primary GPT Header

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>(128 bytes per entry)</td>
</tr>
</tbody>
</table>

### Partition Entry * 128

#### Legacy MBR

- **Signature (EFI PART):** 8 bytes
- **Version:** 4 bytes
- **Header size:** 4 bytes
- **CRC32:** 4 bytes
- **Reserved:** 8 bytes
- **Primary header LBA:** 8 bytes
- **Backup header LBA:** 8 bytes
- **First usable LBA:** 8 bytes
- **Last usable LBA:** 8 bytes
- **Attributes:**
  - 0: System
  - 2: Bootable
  - 60: Read-only
  - 62: Hidden
  - 63: No auto-mount
- **Disk GUID:** 16 bytes
- **First LBA of entries:** 8 bytes
- **No. of entries:** 4 bytes
- **Size of an entry:** 4 bytes
- **CRC32 of entries:** 4 bytes
• **Logical Volume Manager**
  – *Dynamically adjustable partitioning scheme*
  – **Base unit**: PE (physical extents)
  – **Physical volume**: physical device in LVM
  – **Volume group**: group of physical volumes composing LVM
  – **Logical volume**: user-visible *virtual* device
**RAID**

- **Redundant Array of Independent Disks**
  - Combine disks to
    - Expand storage *capacity*
    - Enhance I/O *performance*
    - Enhance device *reliability and sustainability*

- Hardware implementation: RAID controller
- Software implementation: softraid / fakeraid
Block I/O layer

- Operate on `struct block_device` `<linux/fs.h>`
- Allows random access to a block device
- Supports partitioning, I/O scheduling, ...

Virtual File System

- Ext4
- FAT
- YAFFS
- NFS
- FUSE

Page Cache

Device mapper

- LVM
- RAID
- Cache

I/O scheduler

open(), read(), write(), mkdir(), ...

read/write(sector, length, buffer)
Buffer head I/O mechanism

• Buffer head and BIO
  – submit_bh() vs. submit_bio()

• Buffer head
  – **Buffer**: less than a page size
  – Uses *virtual address* to map blocks
  – Deprecated API, but still used in some places
  – Uses BIO to execute I/O operations
  – **API**: alloc_buffer_head(), get_bh(), put_bh(), submit_bh()
**BIO structure** (since 2.5)

- An array of segments of pages (bio_vec)
- Can use unmapped pages

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>block_device</td>
<td>bi_bdev</td>
</tr>
<tr>
<td>sector_t</td>
<td>bi_sector</td>
</tr>
<tr>
<td>struct bio *</td>
<td>next</td>
</tr>
<tr>
<td>long</td>
<td>bi_rw</td>
</tr>
<tr>
<td>int</td>
<td>bi_size</td>
</tr>
<tr>
<td>short</td>
<td>bi_vcnt</td>
</tr>
<tr>
<td>struct bio_vec *</td>
<td>bi_io_vec</td>
</tr>
<tr>
<td>bio_end_io_t</td>
<td>bi_end_io</td>
</tr>
<tr>
<td>void *</td>
<td>bi_private</td>
</tr>
</tbody>
</table>
• **Life cycle of BIO**

- `bio_alloc(gfp_mask, nr_vecs)`
- `bio_get(bio)`
- `bio_add_page(bio, page, offset, length)`
- `bio_for_each_segment(bvl, bio, i)`
- `bio_put(bio)`
- `submit_bio(rw, bio)`
- `bio->bi_end_io(bio, errno)`
- `bio_endio(bio, errno)`

**Need to implement callback fn**

**Type** | **Name**
--- | ---
block_device | `bi_bdev`
sector_t | `bi_sector`
struct bio * | `next`
long | `bi_rw`
int | `bi_size`
short | `bi_vcnt`
struct bio_vec * | `bi_io_vec`
`bi_end_io` | `bio_end_io`
void * | `bi_private`
**Request queues**

- **Request** and request_queue
  - **bios** merged by I/O scheduler

- Request queue **operations**
  - request_fn()
  - make_request_fn()

Used for device mapper implementation

Diagram:
- `bdev`
  - `request_queue`
  - `elevator`
  - `request`
  - `bio`

- `submit_bio()` → `generic_make_request()`
  - `make_request_fn()` → `blk_queue_bio()`

Device mapper → I/O scheduler
Review: Block I/O layer

- **Block device**: abstracting HDDs, SSDs, ...
  - `/dev/sdxn` >= 1, or none
  - `a~z, aa~zz`

- **Accessing block device**
  - Through file system
  - Through device node – allows file operation

- **Linux implementation layers**

  - **Virtual file system**
    - ext4
    - btrfs
    - FAT

  - **Block I/O layer**
    - Device mapper
      - LVM
      - RAID
    - I/O scheduler
      - CFQ
      - NOOP

  - **Device driver**
    - SATA
    - SSD
    - HDD
    - SCSI
    - LVM
    - RAID
Device mapper

http://techgmm.blogspot.kr/2012/07/writing-your-own-device-mapper-target.html
• Generic framework to map block devices onto another
  – e.g. LVM2, software RAIDs, dm-crypt, multipath, ...

• /dev/mapper/*
  – List of virtual block devices managed by device mapper

• dmsetup
  – Tool to manipulate device mapper
Using device mapper

• Installation
  – apt-get install dmsetup

• Example: multipath device
  – dmsetup create mp_test
    0  1024  linear  /dev/sda2  1024
    1024  2048  linear  /dev/sdb2  0
  – Result: /dev/mapper/mp_test

• Mapped device can be nested
  – LVM over RAID
Writing new DM target

• DM target is a **kernel module**
  – Describes mapping behavior

• Include files
  – **Module**: `<linux/module.h>`, `<linux/kernel.h>`, `<linux/init.h>`
  – **Block I/O**: `<linux/bio.h>`
  – **Device mapper**: `<linux/device-mapper.h>`

• **Method functions** to implement
  – `map()`: remapping *bio* to real target device
  – `ctr()`: create a new virtual block device
  – `dtr()`: stop using virtual block device

submit_bio()
dmsetup create
dmsetup remove
Device mapper target

- **Virtual device (/dev/mapper/mp_test)**
  - `struct dm_table`
    - `int num_targets`
    - `struct dm_target * targets`
  - `struct my_cache`
    - `struct dm_dev * cache`
    - `struct dm_dev * backing`
    - `struct my_btree * metadata`

- **Underlying device (/dev/sdb2)**
  - `struct block_device`
    - `->bd_disk->private_data`
  - `struct mapped_device`
    - `<drivers/md/dm.c>`
    - `struct dm_table * map`
    - `struct gendisk * disk`

- **List of targets**
  - `list_head _targets`
  - `struct dm_target`
    - `<linux/device_mapper.h>`
    - `struct dm_table * table`
    - `struct target_type * type`
    - `void * private`
    - `sector_t begin`
    - `sector_t len`

- **User defined data structure**
  - `map()`
  - `ctr()`
  - `dtr()`

- **Additional definitions**
  - `struct block_device`
    - `<bd_disk->private_data>
  - `fmode_t mode`
  - `char [] name`

- **Diagram**
  - Connections between the structures and their functions
Map method

- static int map(struct dm_target *ti, struct bio *bio, union map_info *map_context)

- Return value
  - **DM_MAPIO_SUBMITTED**
    - submitted bio to underlying device: used submit_bio()
  - **DM_MAPIO_REMAPPED**
    - bio request is remapped: changed bi_dev, bi_sector
    - Device mapper should submit bio
  - **DM_MAPIO_REQUEUE**
    - Problem with the bio, *but can finish the request with deferred retry*
    - Map method will be called again
• static int `ctr(struct dm_target *ti, unsigned int argc, char **argv)`
  – Call `dm_get_device()` to lock backing devices
  – Initialize device data structures and worker threads
  – **Return value**
    • 0 on success
    • Negative on error

• static void `dtr(struct dm_target *ti)`
  – Call `dm_put_device()` function
  – Free device data structures
Registering target

- Enables the mapping method available to the kernel

- `int dm_register_target(struct target_type *)`
  - struct `target_type`:
    - `name = “target name”`
    - `version = { version }`
    - `module = THIS_MODULE`
    - `ctr(), dtr(), map(), ...`
  - Returns 0 on success

- `void dm_unregister_target(struct target_type *)`
I/O scheduler

• Gathers block I/Os to make large request
  – Minimize seek time

• Scheduling algorithms in OS courses

  FIFO
  Scan
  Elevator
Linux I/O schedulers

• Based on (Linus) elevator algorithm
  – Merges BIOs and executes them

• Available schedulers
  – **noop**: simple elevator
  – **Deadline**: prevent starvation
  – **Anticipatory**: wait for a few msec to merge contiguous I/O
  – **Complete Fair Queuing**: per process queue for fairness

• Choosing and displaying I/O schedulers
  – `/sys/block/{device name}/queue/scheduler`
Block device data structure

- **Block device** (block_device)
  - Physical disk + partition + virtual mapped device
    Ex. /dev/sda, /dev/sda1, /dev/mapper/vol1, ...

- **General disk** (gendisk)
  - Physical/virtual device driver
    Ex. /dev/sda

- **Partitions**
  - gendisk->part_tbl[]
  - bdev->bd_part
Block device and gendisk

struct gendisk

part_tbl

part

struct hd_struct

struct block_device

/dev/sda

bd_contains
bd_part
bd_disk

/dev/sda1

bd_contains
bd_part
bd_disk

/dev/sda3

bd_contains
bd_part
bd_disk

start_sect
nr_sects
Creation / destruction

- **gendisk**
  - struct gendisk *alloc_disk(int minors);
  - int add_disk(gendisk);
  - del_gendisk(gendisk);

- **Block device**
  - struct block_device *bdget_disk(
    struct gendisk *disk, int partno);
  - int __invalidate_device(
    struct block_device *bdev,
    bool kill_dirty);

- **Partition**
  - int rescan_partitions(
    struct gendisk *disk,
    struct block_device *bdev)
gendisk operation

• block_device_operations
  – open(struct block_device *, fmode_t);
  – release(struct gendisk *, fmode_t);
  – ioctl(struct block_device *, fmode_t, unsigned, unsigned long);
  – direct_access(struct block_device *, sector_t, void **, unsigned long *);
  – check_events(struct gendisk *disk, unsigned int clearing);

• request_queue->request_fn()
  – Set by blk_init_queue()
  – Sends I/O requests to device driver (gendisk)
RAID and LVM
Linux software RAIDs

• DM-RAID
  – A device-mapper implementation of fake-RAID
    • Fake-RAID: software RAID supported by SATA chipsets
  – Needs `dmraid` application

• MD-RAID
  – A virtual-block device driver implementation of soft-RAID
  – Needs `mdadm` application
  – Not directly bootable from M/B BIOS
  – Flexible: easy to convert to another RAID level
    • RAID 1  ➔  RAID 5  ➔  RAID 6
MD-RAID example

• Create

```bash
$ mdadm --create --metadata=1.2 --level=1 --raid-devices=2 /dev/md1 /dev/sda /dev/sdb
```

• Add new devices

```bash
$ mdadm --add /dev/md1 /dev/sdc /dev/sdd
```

• Change RAID level

```bash
$ mdadm --grow --level=5 /dev/md1
```

• Repair RAID device

```bash
$ echo repair > /sys/block/md1/md/sync_action
```
Linux LVM2

• Software logical volume manager in Linux kernel
  – Device mapper implementation
  – Needs LVM2 utility package

• Definitions
  – Volume group
    • Storage pool where logical volumes are allocated from
  – Physical volume: block device included in volume group
    • Can be a physical HDD or a group of HDDs combined as a RAID dev.
  – Logical volume: block device that is accessible by users
    • Cannot know which physical volumes are used by a logical volume
LVM2 example

• Creation
  
  $ pvcreate /dev/sda1  
  $ vgcreate vg0 /dev/sda1  
  $ lvcreate -L 100G -n root vg0  
  $ lvcreate -s /dev/vg0/root -n new_root

• Adding physical volume
  
  $ vgextend vg0 /dev/sdb1  
  $ vgmerge vg0 vg1

• Resize logical volume
  
  $ lvresize -r -L +100G /dev/vg0/root

• List LVM status
  
  $ vgdisplay ; pvdisplay ; lvdisplay