File Systems

Science

Computer Science

CS 450 : Operating Systems

Michael Saelee <lee@iit.edu>

IIT College of Science

ILLINOIS INSTITUTE OF TECHNOLOGY

What is a file?

- some logical collection of data
- format/interpretation is (typically) of little concern to OS

A filesystem is a *collection of files*

- supports a managed *namespace* of data
- maps & manages file metadata (automatically & explicitly)

Different (overlapping) classes of FS:

- "traditional": hierarchy of on-disk data
- database-backed storage (rich metadata)
- distributed storage (e.g., for MapReduce)
- namespace for *everything* (e.g. Plan 9)

We'll limit most of our discussion to traditional filesystems and regular files

† modern FS implementations are almost all hybrids (of the classes mentioned)

Agenda

- FS goals & requirements
- FS API
- FS implementation
- FS robustness
- Case study: xv6 (Unix)

system call interface (API)
OS-FS interface
FS implementation
FS-device interface
device drivers
devices (HDDs, SSDs)

(reality is not so tidy!)

§FS Goals

I. File *CRUD* API:

- **C**reate
- **R**ead
- **U**pdate
- **D**elete

II. *Protection & Security*

- access control
- ownership & permissions
- encryption

III. *Robustness*

- crashes shouldn't affect FS *validity*
- also try to mitigate data loss (e.g., uncommitted changes)

IV. *Flexibility* & *Scaleability*

- different ways of accessing data
    - e.g., stream vs. memory mapped
- support *exponential growth* in drive capacity

V. *Decoupling* of OS & FS

- FS not tied to OS (or vice versa)

- multiple FSes a single OS (at once)

VI. *Device agnosticism*

- FS shouldn't assume/optimize for a certain type of storage device
  - e.g., HDD vs. SSD vs. RAM disk

VII. Good *throughput* & *responsiveness*

- throughput (in MB/s or IOPS)
- responsiveness ≈ request latency

VIII. Good disk *utilization*

- often least important!
    - usually preferable to trade *spatial inefficiency* for robustness & speed

§FS API

File attributes (file as an ADT):

- name/path (convenient for humans)
- identifier (unique, system-wide)
- type (e.g., executable)
- protection & access control
- creator/owner, size, timestamp
- possibly much more! (e.g., log, tags, …)

Basic operations:

- *Create* @ some location, with specified mode(s), possibly truncating
- *Read*
- *Update*: write content, metadata; adjust position in file (need to track)
- *Delete* = remove from FS

Typical data structures:

- file descriptor
- open file structure
- namespace structure (e.g., directory)
- access control metadata

a) file descriptor

- process-held "pointer" to an open file
- used to identify file to OS/FS for user initiated file operations
  - enables OS encapsulation of file data

b) open file structure

- essentials: position in file & count of referring processes (via FDs)
  - may permit multiple positions
  - flush in-memory struct if count = 0
- also, per open-file access mode(s)

c) namespace structure (e.g., directory)

- tracks position of data "in" FS
- may function as *all-purpose OS namespace* (e.g., even for off-disk data)
- e.g., *full path* from FS "root":

```
/home/lee/.emacs
```

d) access-control metadata

- e.g., “rwx” bits in Unix
  - separate bits for owner/group/all
- or more granular ACLs
  - e.g., read/write/append/readacl/ writeacl/delete/etc., based on user

e.g., Unix file syscalls

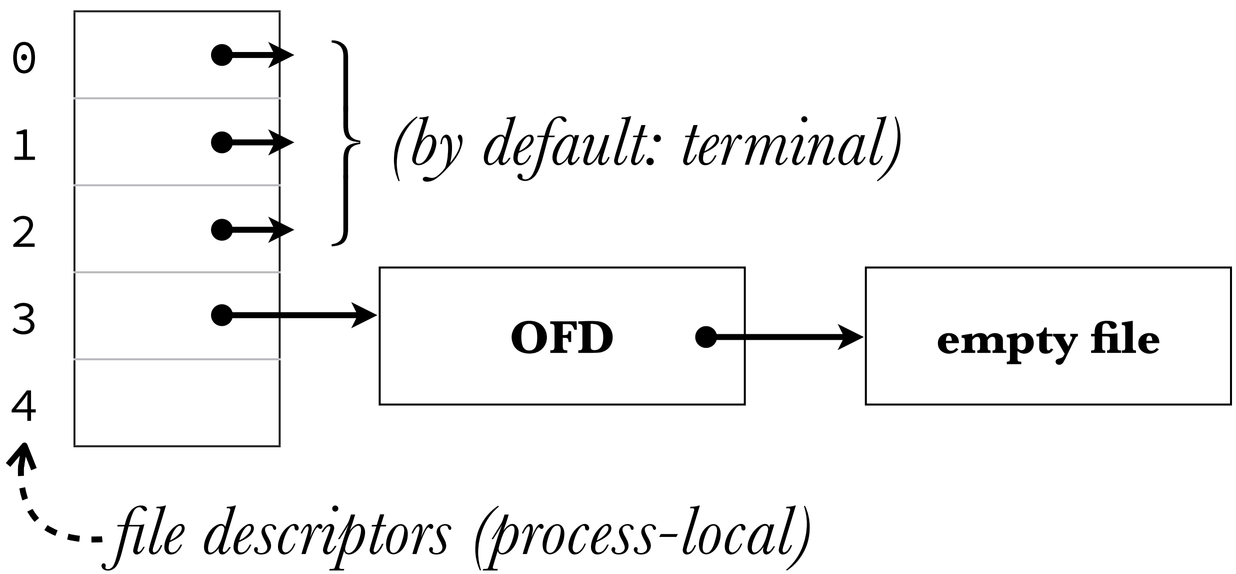
```
     int   open   ( char *path, int oflag, ... );
     int   creat  ( char *path, mode_t mode );
     int   close  ( int fd );

     int   link   ( char *oldpath, char *newpath );
     int   unlink ( char *path );
     int   chdir  ( char *dirpath );

 ssize_t   read   ( int fd, void *buf, size_t nbytes );
 ssize_t   write  ( int fd, void *buf, size_t nbytes );
   off_t   lseek  ( int fd, off_t offset, int whence );

     int   fchmod ( int fd, mode_t mode );
     int   fstat  ( int fd, struct stat *buf );
```

```c
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

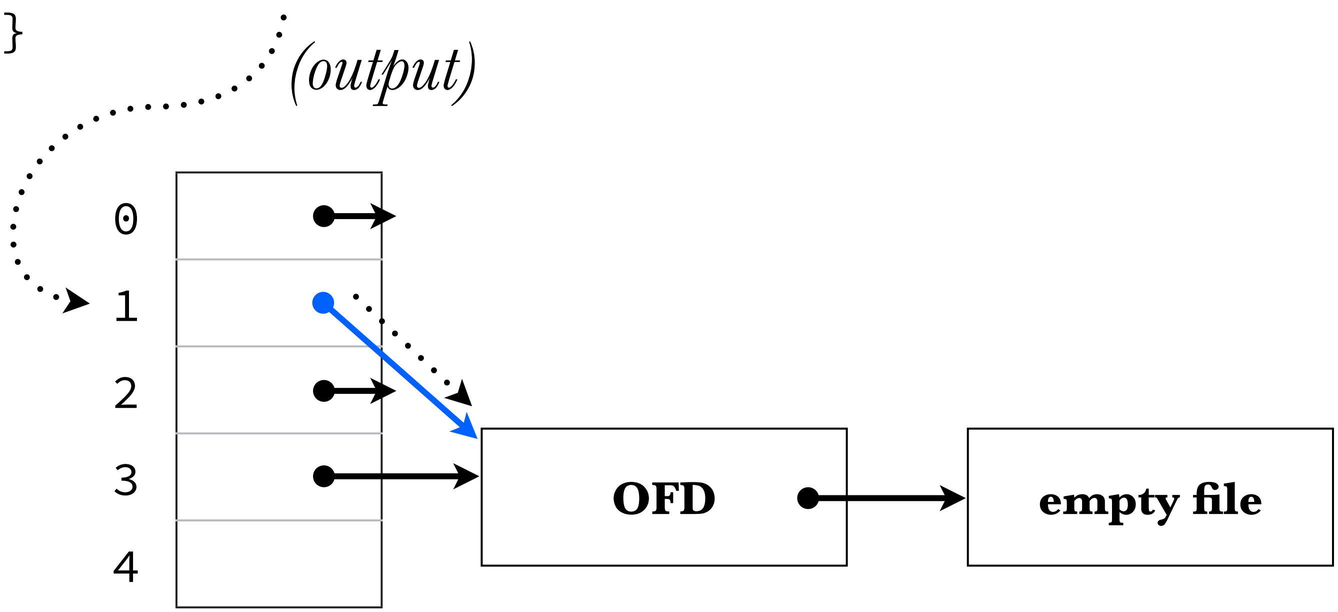
Unix convention of mapping fixed file descriptor values to "standard" in/out is widely copied — allows for *I/O redirection*

```
int main(int argc, char *argv[]) {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    dup2(fd, 1); /* set fd 1 (stdout) to be “foo.txt” */
    printf("Arg: %s\n", argv[1]);
}
```

```
int main(int argc, char *argv[]) {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    dup2(fd, 1); /* set fd 1 (stdout) to be “foo.txt” */
    printf("Arg: %s\n", argv[1]);
}
```



```
int main(int argc, char *argv[]) {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    dup2(fd, 1); /* set fd 1 (stdout) to be "foo.txt" */
    printf("Arg: %s\n", argv[1]); /* printf uses "stdout" */
}
```



```
int main(int argc, char *argv[]) {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    dup2(fd, 1); /* set fd 1 (stdout) to be “foo.txt” */
    printf("Arg: %s\n", argv[1]);
}
```

```
$ ./a.out hello!
$ ls -l foo.txt
-rw-r--r-- 1 lee staff 12 Feb 19 20:36 foo.txt
$ cat foo.txt
Arg: hello!
```

```
int main() {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    if (fork() == 0) {
        dup2(fd, 1);
        execlp("echo", "echo", "hello!", NULL);
    }
    close(fd);
}
```

```
$ ./a.out
$ cat foo.txt
hello!
```

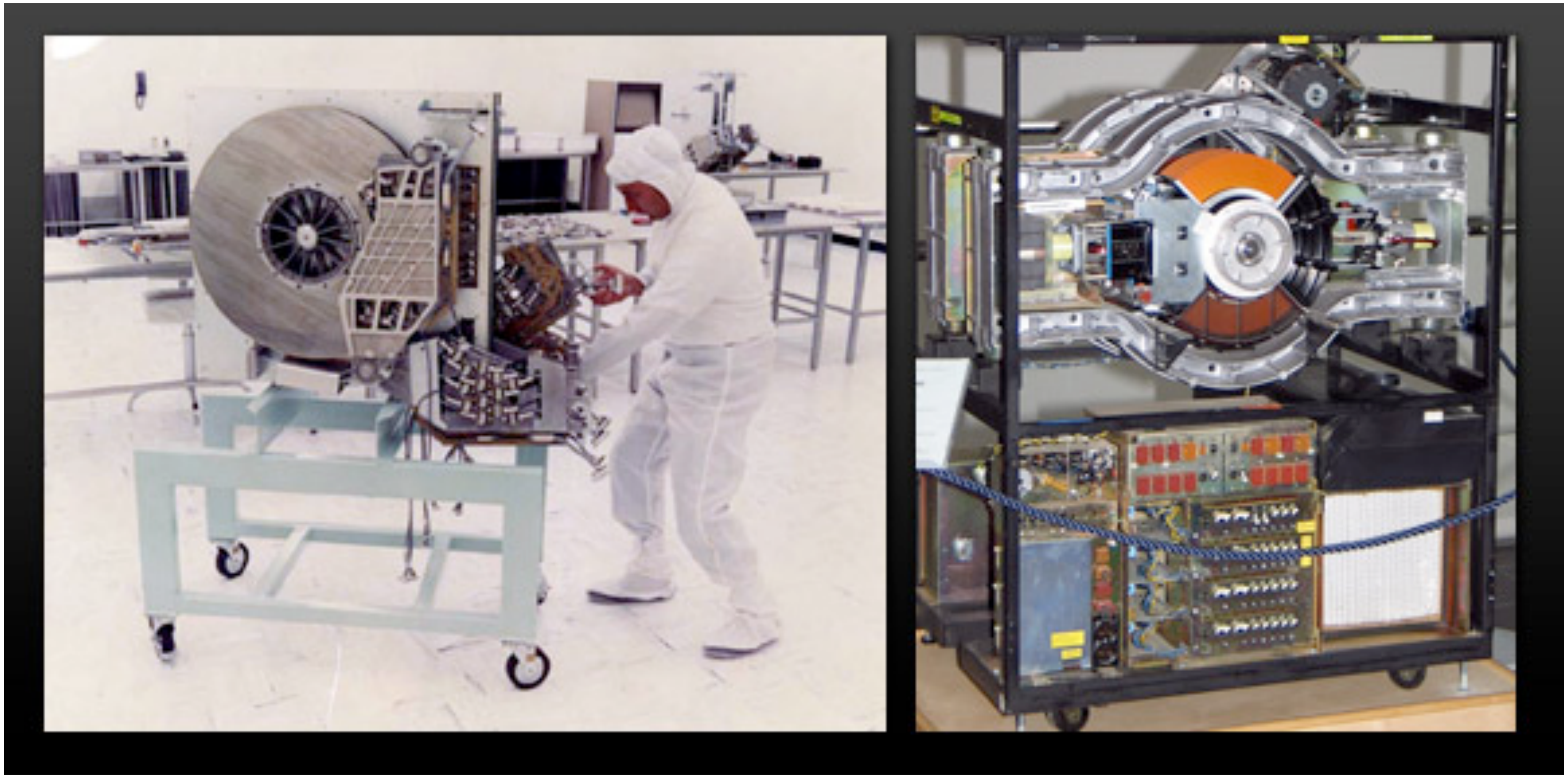
§FS Implementation

1. Mass storage (disk) systems
2. Volumes and Partitions
3. Names and Paths
4. File space allocation
5. Free space tracking

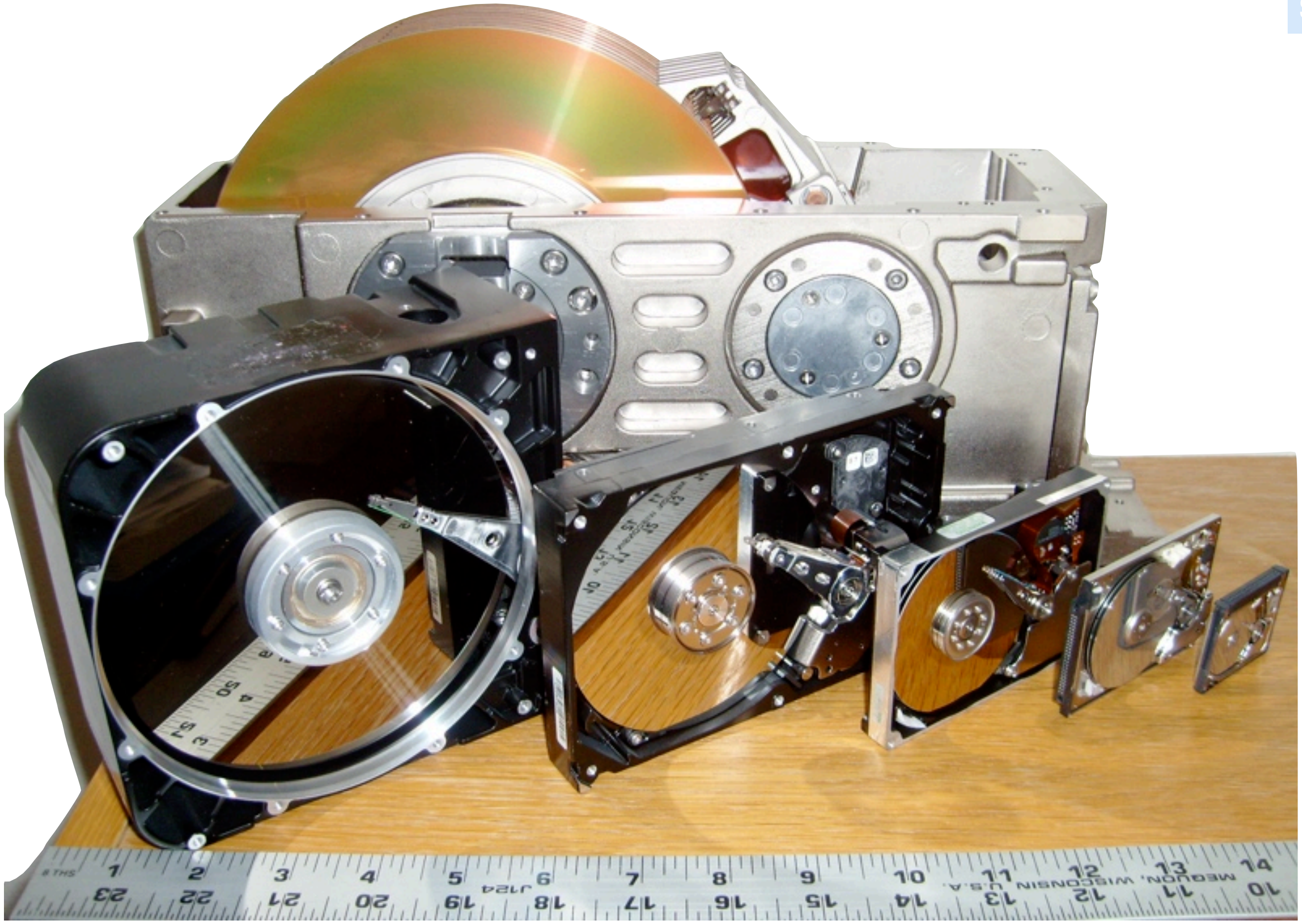
¶ Mass storage systems

magnetic disks (HDDs) provide bulk of secondary storage

- rotating magnetic platters

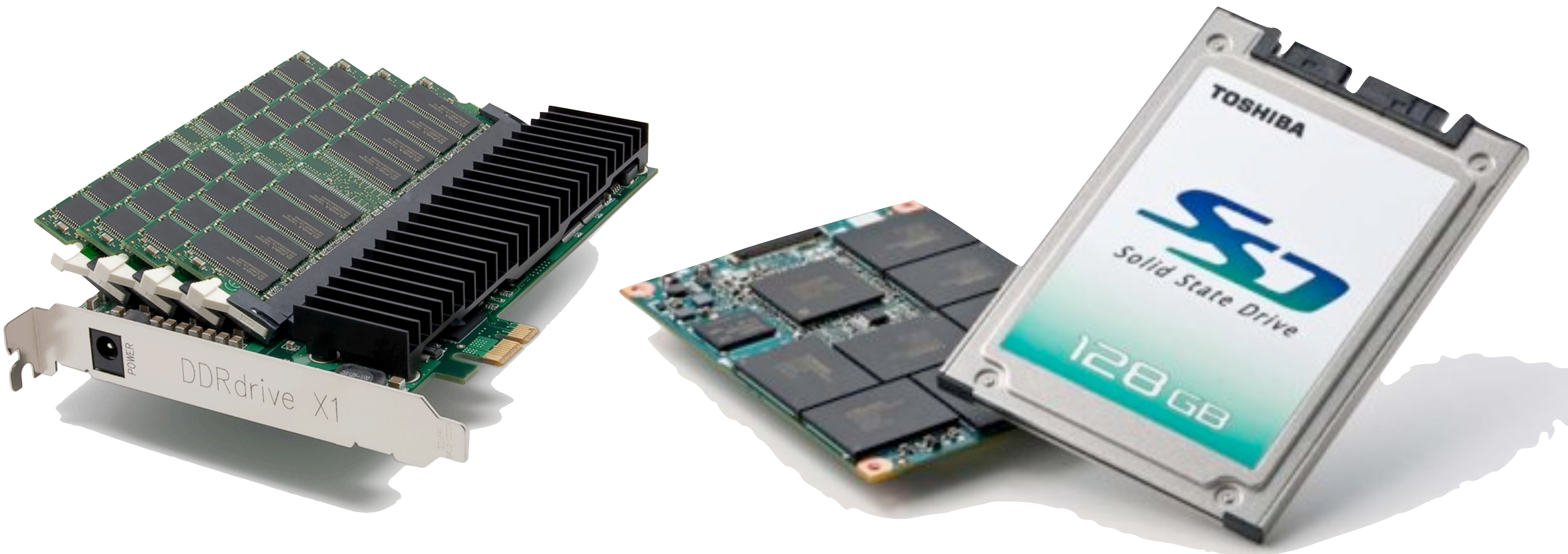


motor & belt driven



smaller & denser, but
still *mechanical*

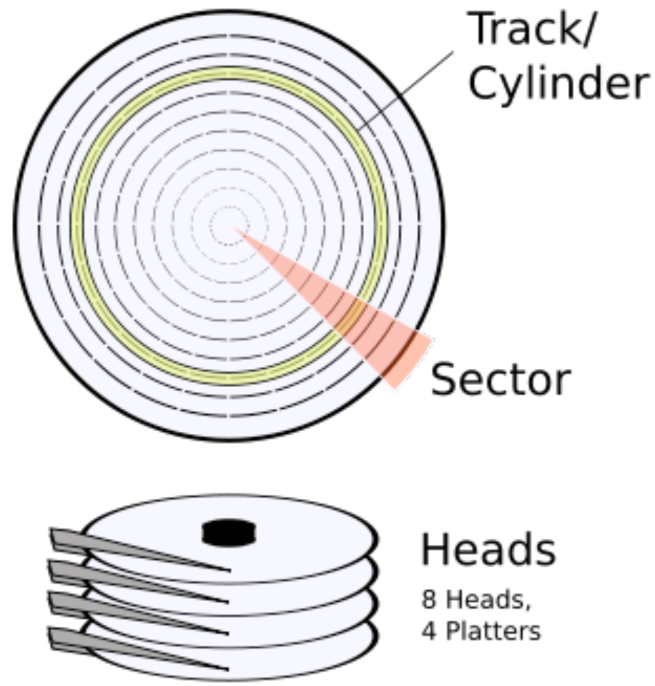
IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY


POWER
DDRdrive X1
TOSHIBA
Solid State Drive
128GB

?!

will focus on traditional HDDs for now …

- still a valuable discussion
- HDDs will remain the mass storage device of choice for some time to come



idealized addressing: Cylinder, Head, Sector a sector, historically, maps to a fixed *512-byte block* of disk space

- minimum disk transfer size
- recently, drives are moving to 4K block sizes (but still support old mapping)

Disk access times = **S** + **R** + **T**

- **S**: *seek time* (head movement)
- **R**: *rotational latency* (depends on angular velocity — usually constant for HDDs)
- **T**: *transfer time* (relatively small)

\+ "spin-up" time (discount for long I/O)

Disk access times = **S** + **R** + **T**

- **S**: move to correct *cylinder*
- **R**: wait for *sector* to rotate under head
- **T**: move head across *adjacent blocks*

Some numbers:

- seek time = 3ms-15ms
- typical RPM = 7200 (range of 5.4-15K)
  - rot. latency = ½ of period
    - e.g., $\frac{1}{2} \times 60/7200 \approx 4.17$ms

Specifications[1]	2 TB	2 TB	1.5 TB	1.5 TB	1 TB	1 TB
Model number	WD2002FAEX	WD2001FASS	WD1502FAEX	WD1501FASS	WD1002FAEX	WD1001FALS
Interface	SATA 6 Gb/s	SATA 3 Gb/s	SATA 6 Gb/s	SATA 3 Gb/s	SATA 6 Gb/s	SATA 3 Gb/s
Formatted capacity	2,000,398 MB	2,000,398 MB	1,500,301 MB	1,500,301 MB	1,000,204 MB	1,000,204 MB
User sectors per drive	3,907,029,168	3,907,029,168	2,930,277,168	2,930,277,168	1,953,525,169	1,953,525,169
SATA latching connector	Yes	Yes	Yes	Yes	Yes	Yes
Form factor	3.5-inch	3.5-inch	3.5-inch	3.5-inch	3.5-inch	3.5-inch
RoHS compliant[2]	Yes	Yes	Yes	Yes	Yes	Yes
Performance						
Data transfer rate (max) Buffer to host Host to/from drive (sustained)	 6 Gb/s 138 MB/s	 3 Gb/s 138 MB/s	 6 Gb/s 138 MB/s	 3 Gb/s 138 MB/s	 6 Gb/s 126 MB/s	 3 Gb/s 126 MB/s
Cache (MB)	64	64	64	64	64	32
Average latency (ms)	4.2	4.2	4.2	4.2	4.2	4.2
Rotational speed (RPM)	7200	7200	7200	7200	7200	7200
Average drive ready time (sec)	21	21	21	21	11	11

by contrast, each channel of DDR3-2133 memory has max theoretical throughput:

$$2133 \text{ MHz} \times 8 \text{ bytes} = 17064 \text{ MB/s}$$

… only ~100× more than disk throughput?

138 MB/s is *sustained* rate

- unlikely when dealing with random, fragmented data on disk
- 6 Gb/s (750MB/s) is *buffer to memory* — not indicative of HDD speed

HDDs are best leveraged by reading *contiguous sectors* — i.e., w/o seeking idea: optimize order of block requests to minimize seeks (most expensive operation)

goals:

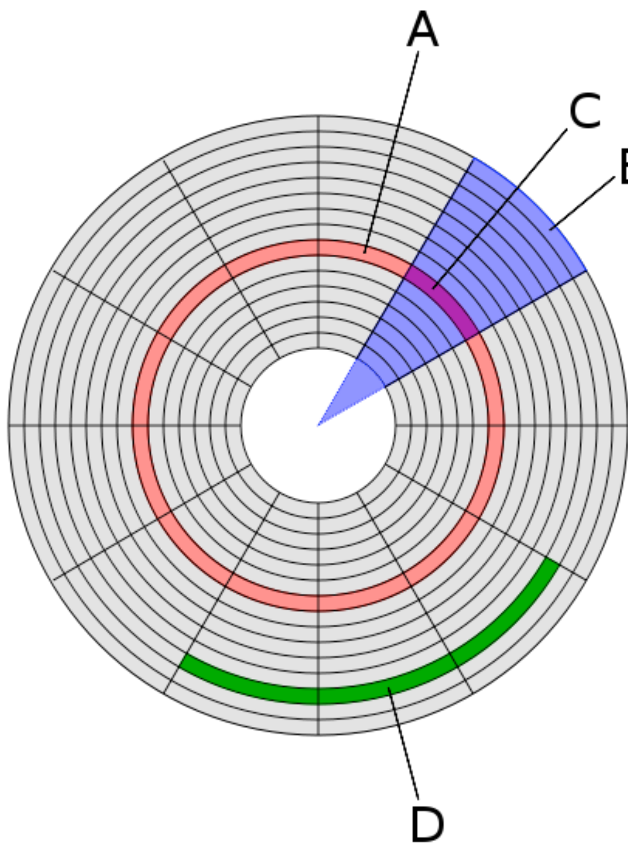
- maximize throughput
- minimize latency per response

province of *disk head scheduler*

CHS is useful for discussion:

- bigger difference in cylinders = larger head movement
- note: heads move as single unit

But CHS is unrealistic in modern drives: low density in outer cylinders!



Modern drives use *logical block addressing* (LBA)

- number blocks starting from 0 (innermost) to outermost, then back in on reverse side
- problem: no disk geometry info!
  - not so bad: LBA_i, LBA_{i+1} are at most 1 cylinder apart

Disk head scheduling problem:

- given requests B_1, B_2, … from processes, what seek order to send to disk controller?

Analogs to scheduling approaches:

- First come, first served (FCFS)
- Shortest Seek Time First (SSTF)
- Nearest Block Number First (NBNF)

as before, SSTF can result in starvation — or at best poor request latency!

how to alleviate starvation problem, and optimize wait time, responsiveness, etc.?

“Elevator” Algorithms

SCAN:

- track from spindle ↔ edge of disk
  - only service requests in the current direction of travel
- keep heading towards spindle/edge even if no requests in that direction

Variants of SCAN:

- C-SCAN: “circular” tracking
- F-SCAN: “freeze” request queue on direction change

LOOK:

- reverse direction when no more requests
- variants: C-LOOK, F-LOOK

Demo: UTSA disk-head simulator

… but FSes may span more than just one storage device!

¶ Volumes and Partitions

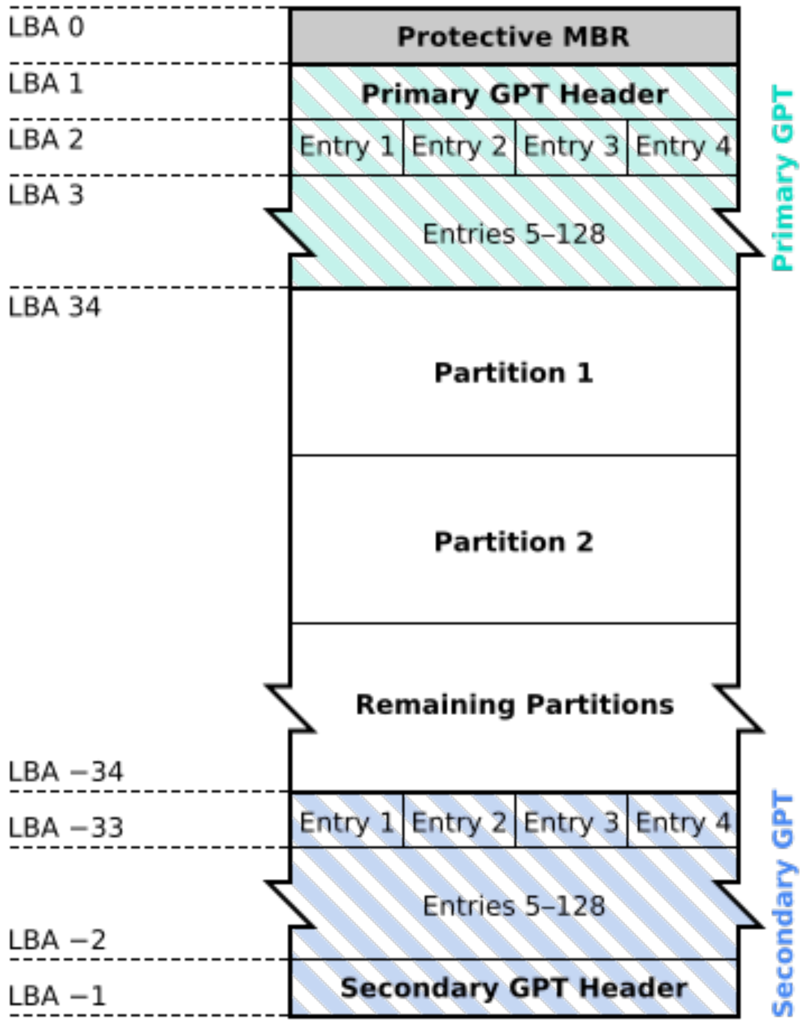
Why volumes & partitions?

- separate logical & physical storage layers
- allow M:N mapping between FSes & disks

A volume is a *logical* storage area.

A partition is a *slice of a physical disk*.

- a *disk* may have *zero or more partitions*
- a *partition* may contain *a volume*
- a *volume* may *span one or more partitions*
- a *volume* may exist *independently of a partition* (e.g., ISO/DMG files)



GUID partition table scheme

courtesy Wikimedia Commons

(typically) partition ≤ volume ≤ FS

- inter-partition / inter-volume FS operations are more expensive!
  - separate metadata structures
  - separate caches

¶ Names and Paths

Requirement: a *fully qualified* filename *uniquely identifies* a set of data blocks on disk

- big filenames & "flat" namespace work, but are hard to reason about
- prefer *hierarchical* namespaces
  - fully qualified filename = name + *path*

`/home/lee/cs450/slides/fs.pdf`

- *absolute* path
- from "`/home/lee/cs450`", *relative* path is "`./slides/fs.pdf`"
  - ("`.`" = current directory)

- one or more *root* namespaces
- typically can *mount* additional filesystems onto global namespace
  - support for multiple filesystems

e.g., Windows:

- C:\foo.txt vs. D:\foo.txt

e.g., Unix

- /home/lee/foo.txt
  vs. /mnt/cdrom/foo.txt

What's in a name?

- path → file must be unique
- file → path??
- consider aliases/shortcuts:
  - `/bin/prog` ↔ `/home/lee/foo_prog`
  - different *paths* may refer to same *file*

Directories provide *linking* structures

- directory maps name → *file identifier*
    - file id is implementation specific
- directories are also files (recursive def)

Link types:

- *hard* link: different names (possibly in different directories) map to same file
    - remove all hard links = removing file
- *soft/symbolic* link: file containing the name of another file
    - independent of whether file exists

note: soft links are possible *across partitions/ volumes*, but hard links aren't (usually)

To "find" a file:

- just need location of *root directory*
  - search recursively for path components
- trickier with multiple FSes
  - each logical *volume* of data contains its own high level metadata

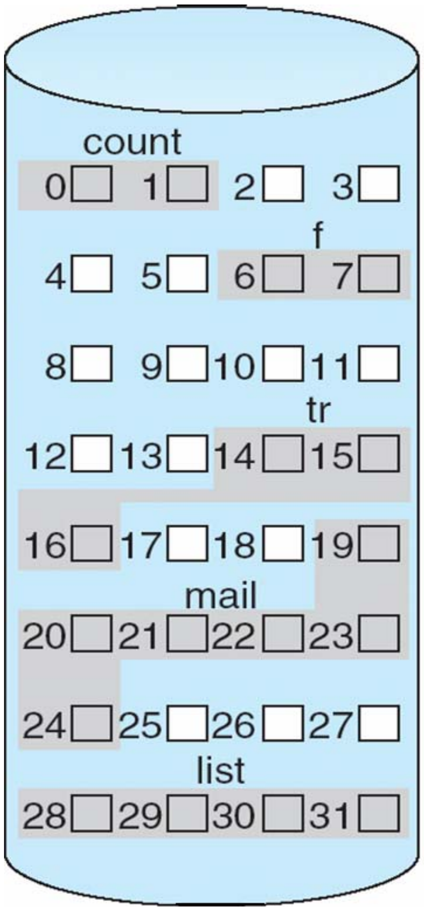
¶ File space allocation

mapping problem: for a given file (by path or id), find (ordered) *list of data blocks* considerations:

- good disk utilization
- efficiency (w.r.t. HDD seeks)
- random access
- scaleability

basic strategies:

- contiguous
- linked (decentralized)
- centralized
    - linked
    - indexed



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

directory may double as metadata store, too (e.g., mode, owner)

contiguous allocation

pros:

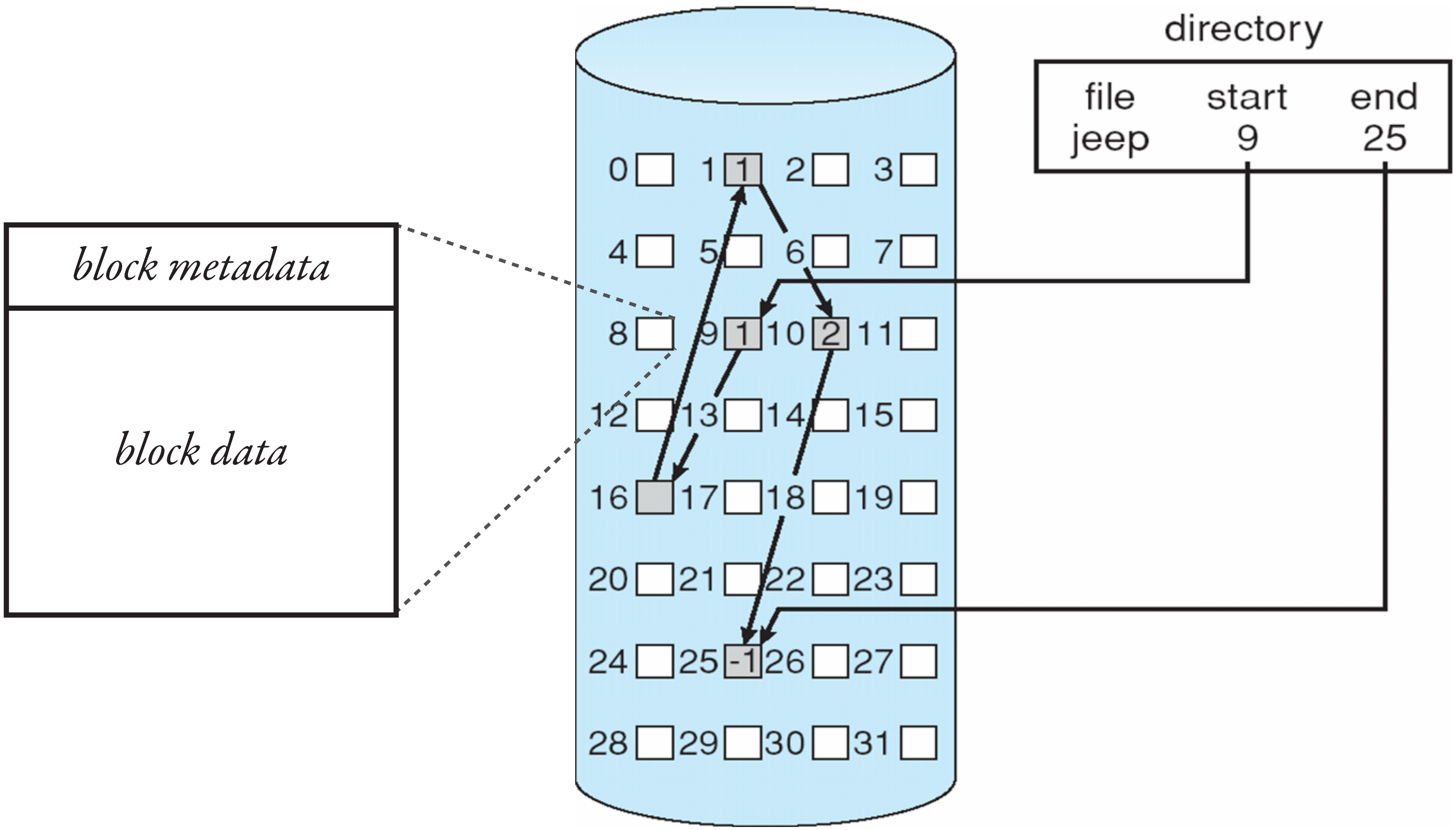
- ideal for sequential HDD reads; reduce seeks → fast!
- random access is trivial

cons:

- clear disadvantage: ***fragmentation***
  - affects utilization, placement ("all or nothing"), resizing

not used on its own, but *contiguous* ***extents*** are used in most modern file systems

- multiple of block size — variable size
  - reserve in advance during allocation
- balance *fragmentation* & *efficiency*



linked allocation (*decentralized*)

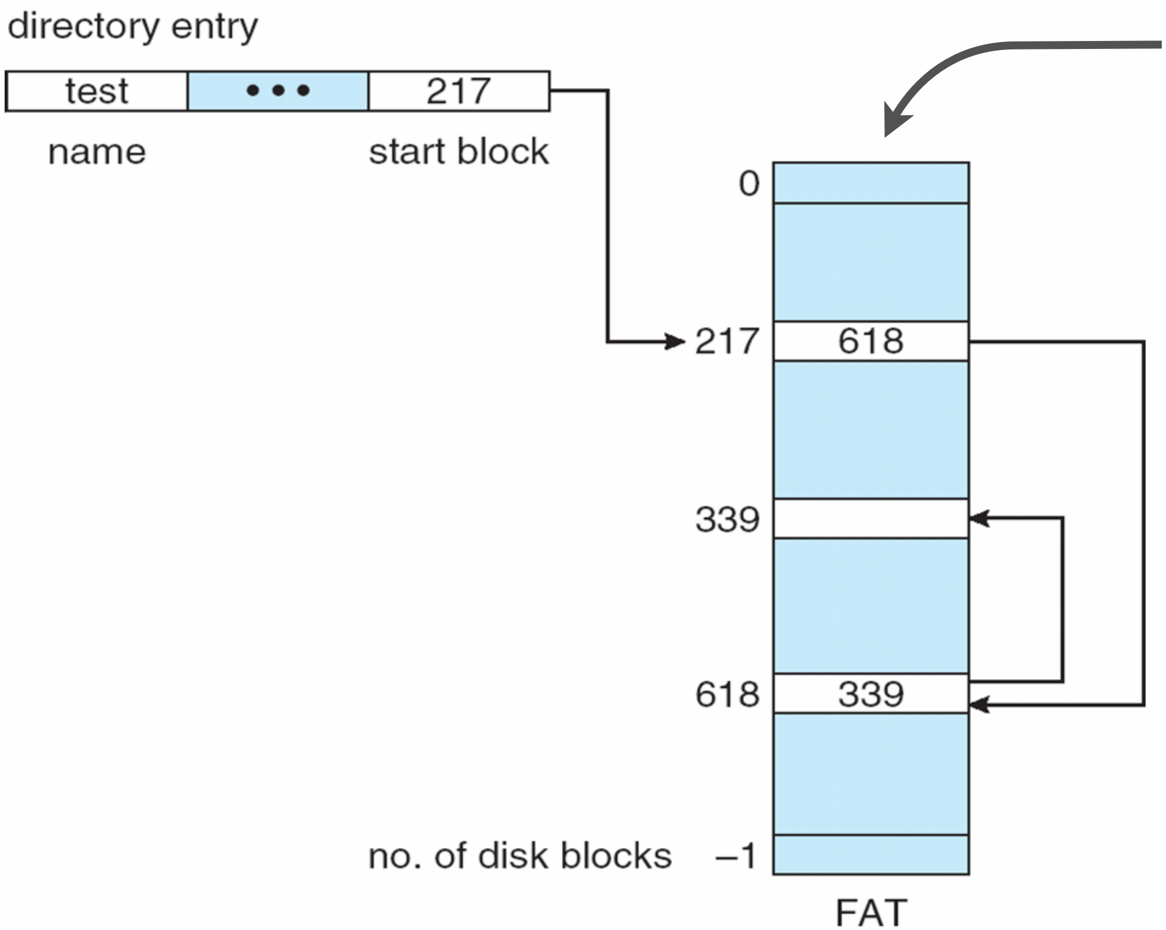
IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

pros:

- good utilization + allows resizing

cons:

- fragmentation → lot of seeks = slow!
- no random access
- hard to protect file metadata!



stored as *per-volume* metadata!

linked allocation (*centralized*)

pros:

- allows for random access
- used with extents, can limit fragmentation

disadvantages:

- centralized file metadata (robustness?)
- overhead incurred by central FAT
- *hard limit* on volume size!

also, unless directories maintain metadata, central structure has very limited space

e.g., where to put mode, ownership, ACL, timestamp, etc.?

e.g., MS-DOS file-allocation table (FAT)

- FAT12, FAT16, FAT32 variants (based on sizes of FAT entry) some MS FAT terminology:

“sector”: physical disk block (512 bytes)

“cluster”: fixed-size extent of 1-256 sectors

(512 bytes - 128KB)

some limits:

FAT12: 4K clusters x 512 = 2MB

FAT16: 64K clusters x 8K = 512MB

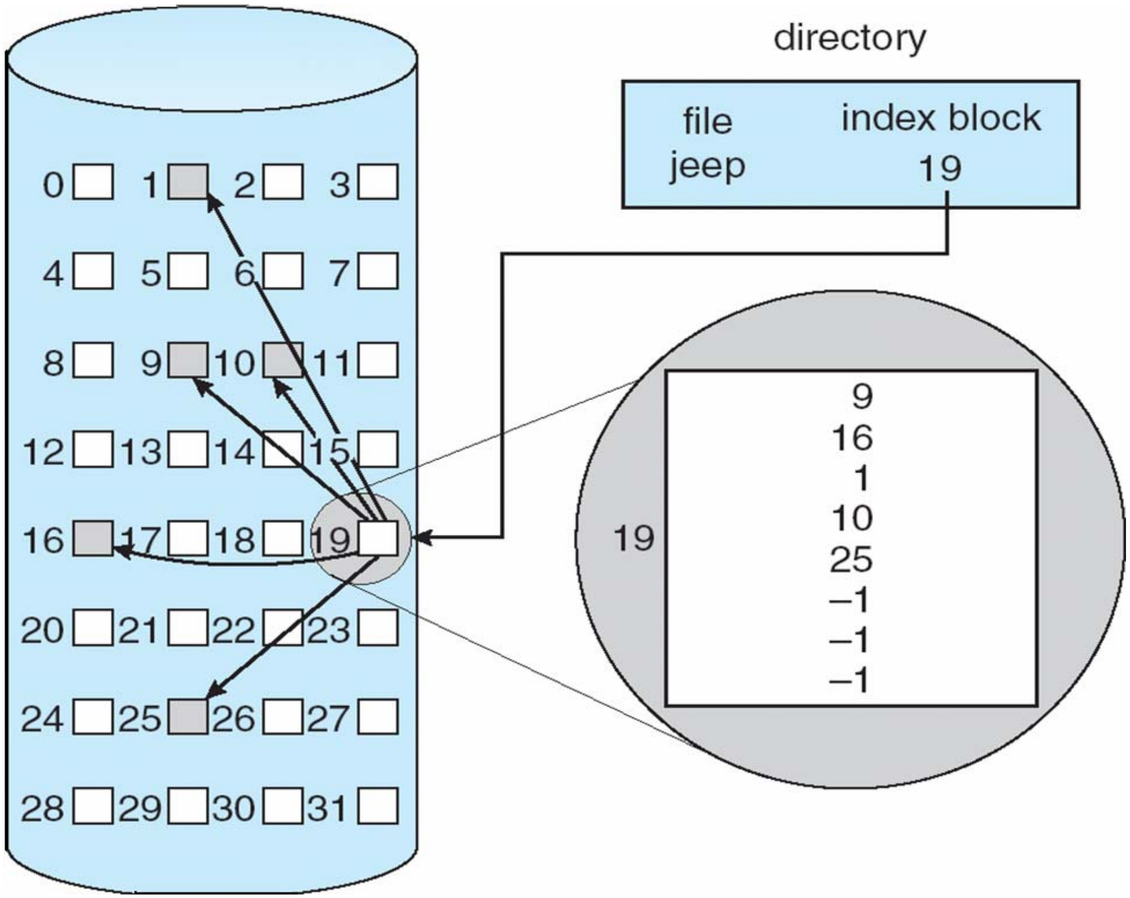
FAT32: only 28-bits of FAT entry useable,

268M clusters x 8K = 2TB

```
FAT12 requirements : 3 sectors on each copy of FAT for every 1,024 clusters
FAT16 requirements : 1 sector on each copy of FAT for every 256 clusters
FAT32 requirements : 1 sector on each copy of FAT for every 128 clusters

FAT12 range : 1 to 4,084 clusters : 1 to 12 sectors per copy of FAT
FAT16 range : 4,085 to 65,524 clusters : 16 to 256 sectors per copy of FAT
FAT32 range : 65,525 to 268,435,444 clusters : 512 to 2,097,152 sectors per copy of FAT

FAT12 minimum : 1 sector per cluster × 1 clusters = 512 bytes (0.5 KiB)
FAT16 minimum : 1 sector per cluster × 4,085 clusters = 2,091,520 bytes (2,042.5 KiB)
FAT32 minimum : 1 sector per cluster × 65,525 clusters = 33,548,800 bytes (32,762.5 KiB)

FAT12 maximum : 64 sectors per cluster × 4,084 clusters = 133,824,512 bytes (≈ 127 MiB)
[FAT12 maximum : 128 sectors per cluster × 4,084 clusters = 267,694,024 bytes (≈ 255 MiB)]

FAT16 maximum : 64 sectors per cluster × 65,524 clusters = 2,147,090,432 bytes (≈2,047 MiB)
[FAT16 maximum : 128 sectors per cluster × 65,524 clusters = 4,294,180,864 bytes (≈4,095 MiB)]

FAT32 maximum : 8 sectors per cluster × 268,435,444 clusters = 1,099,511,578,624 bytes (≈1,024 GiB)
FAT32 maximum : 16 sectors per cluster × 268,173,557 clusters = 2,196,877,778,944 bytes (≈2,046 GiB)
[FAT32 maximum : 32 sectors per cluster × 134,152,181 clusters = 2,197,949,333,504 bytes (≈2,047 GiB)]
[FAT32 maximum : 64 sectors per cluster × 67,092,469 clusters = 2,198,486,024,192 bytes (≈2,047 GiB)]
[FAT32 maximum : 128 sectors per cluster × 33,550,325 clusters = 2,198,754,099,200 bytes (≈2,047 GiB)]
```

source: https://en.wikipedia.org/wiki/File_Allocation_Table file size limit theoretically = disk limit,

but directory implementation constrains file sizes to 4GB in FAT32



indexed allocation

files identified by index block number

- a.k.a. *inode* number
- directory is an inode "registry"
  - index of file name → inode #
    - each entry is a *hard link*
  - directories are files, too, so they also have inodes

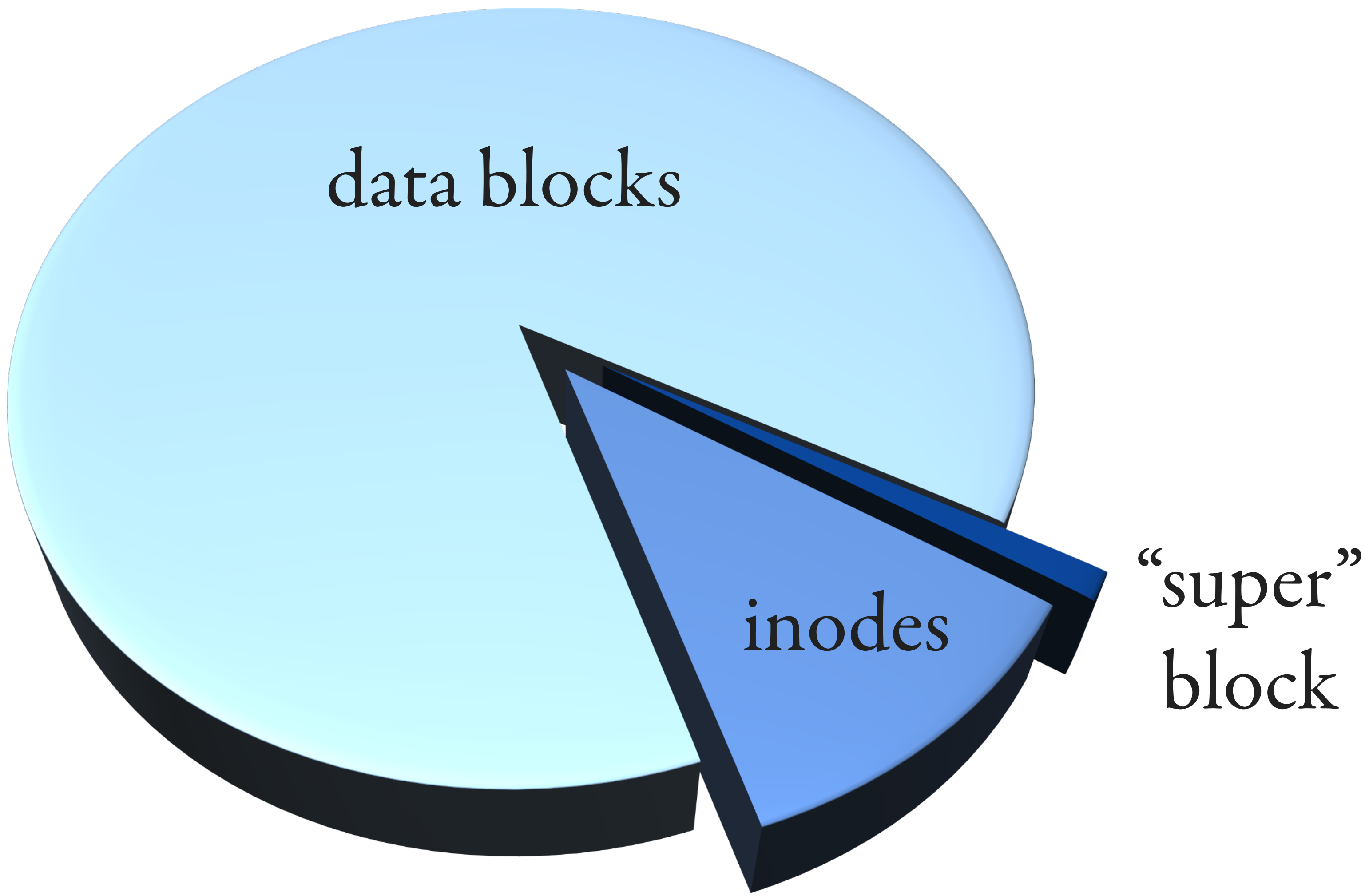
pros:

- allows for random access
- natural metadata store
- used with extents, can limit fragmentation

disadvantages:

- overhead incurred by index nodes
- limit on file size (# block references)

e.g., Unix File System, UFS (and all its descendants)


data blocks
inodes
“super” block

superblock contains FS metadata

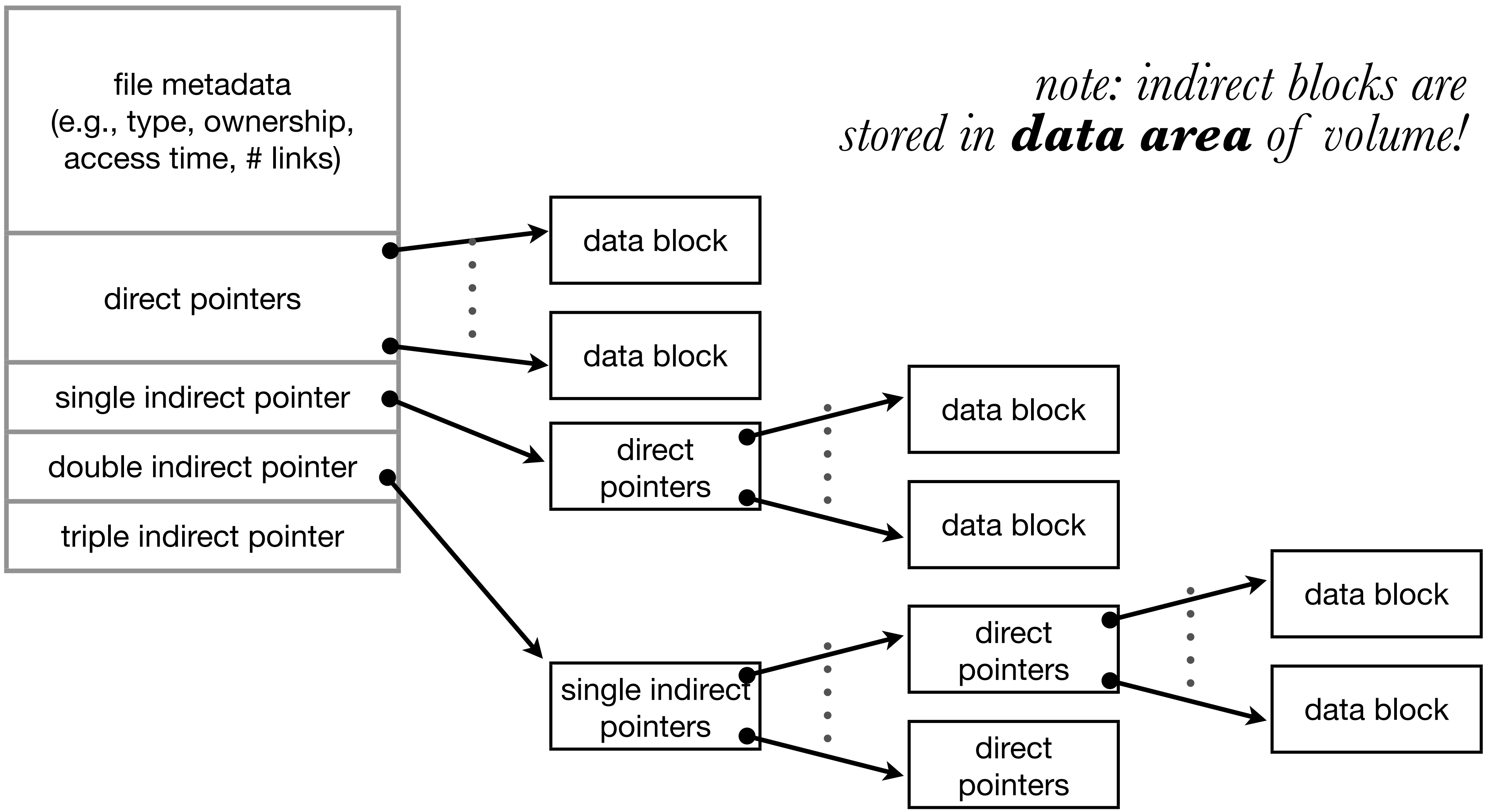
- size of logical blocks
- location & number of inodes

inodes section contains per-file metadata

- # inodes = max # files

"inode" block

note: indirect blocks are stored in ***data area*** *of volume!*



e.g., UFS properties:

- 32-bit i-node pointers
- 4KB i-node/data blocks
- 8 direct, 2 single indirect, 1 double indirect pointer per i-node
- max disk / file size?

- 32-bit i-node pointers
- 4KB i-node/data blocks
- 8 direct, 2 single indirect, 1 double indirect pointer per i-node

max disk size = 4G x 4KB = 16TB

- 32-bit i-node pointers
- 4KB i-node/data blocks
- 8 direct, 2 single indirect, 1 double indirect pointer per i-node

directly addressed: 8 x 4KB = 32KB

- 32-bit i-node pointers
- 4KB i-node/data blocks
- 8 direct, 2 single indirect, 1 double indirect pointer per i-node

each indirect block can hold 4KB / 4 bytes

= 1K pointers

- 32-bit i-node pointers
- 4KB i-node/data blocks
- 8 direct, 2 single indirect, 1 double indirect pointer per i-node

single indirect pointer = 1K x 4KB = 4MB

two single indirect = 8MB

- 32-bit i-node pointers
- 4KB i-node/data blocks
- 8 direct, 2 single indirect, 1 double indirect pointer per i-node

double indirect pointer = 1K x 1K x 4KB

= 4GB

- 32-bit i-node pointers
- 4KB i-node/data blocks
- 8 direct, 2 single indirect, 1 double indirect pointer per i-node

max file size = 32KB + 8MB + 4GB

† variable # block requests per data request (depending on location in file!)

how to keep FS decoupled from OS?

need a *middle layer* — a *mediator* between FS specific constructs & abstract OS file-related operations

VFS: "Virtual File System" layer

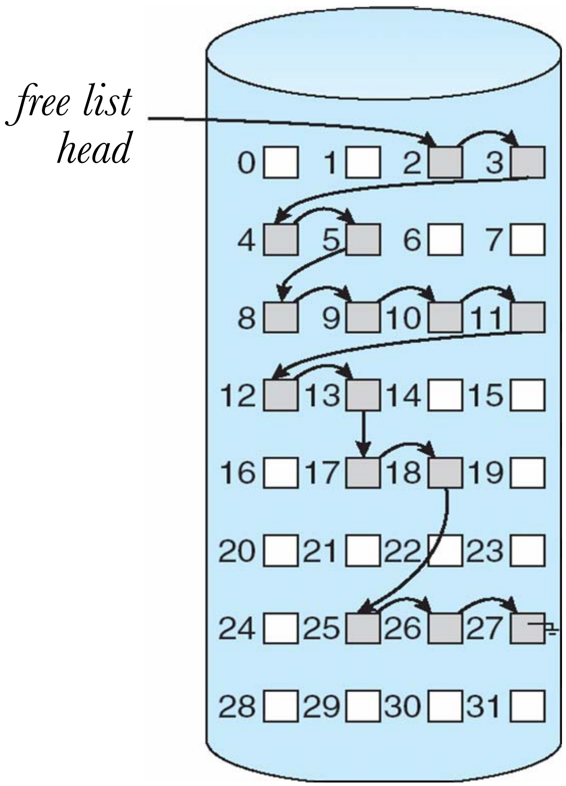
- Unix centric API between syscall API (open/close/read/write) & FSes
- every FS must implement generic analogues of: *inode*, *file*, *superblock*, *dentry*

each FS object has a table of function pointers (e.g., open/close/read/write) that are used by VFS to map syscalls

¶ Free space tracking

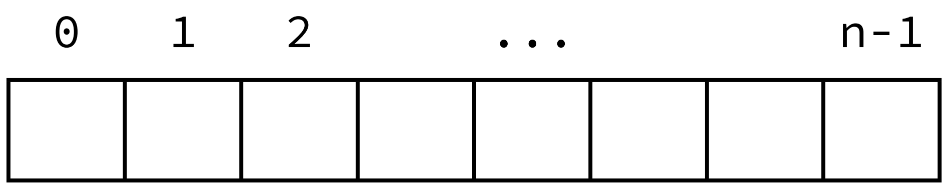
1. linked free blocks
2. free space bitmap
3. general disk-based data structures

1. linked free blocks



- no overhead
- but *expensive* to traverse!
- can optimize as a skip list
    - useful for extent search

2. free space bitmap



$$\text{bit}[i] = \begin{cases} 0 \Rightarrow \text{block}[i] \text{ occupied} \\ 1 \Rightarrow \text{block}[i] \text{ free} \end{cases}$$

- simple to maintain & fast!
  - use machine instr. to locate first '**1**'

- block size = 2^{12} bytes (4KB)
- disk size = 1TB = 2^{40} bytes
    - free space bitmap = 2^{28} bits (32MB)
        - small enough to keep in memory
            - but beware synch issues

optimization:

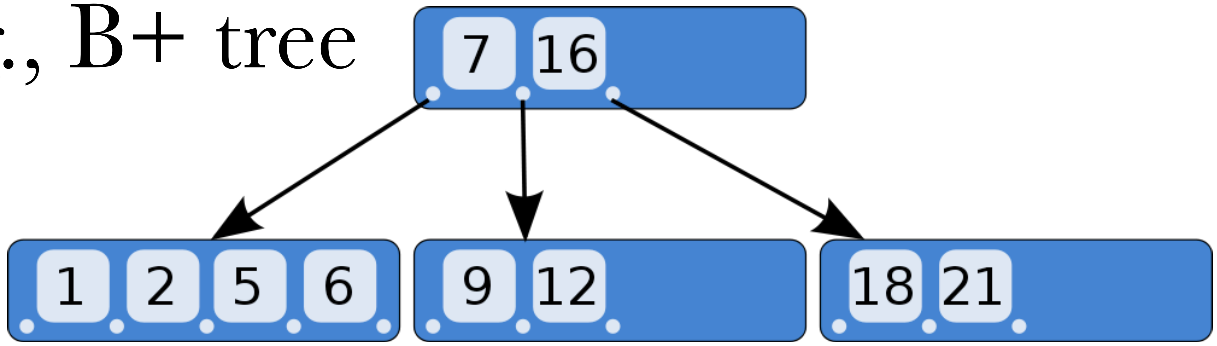
- break bitmap into subsets & build index of # free blocks → subset

  - speed up extent search

  - can lock subsets separately

3. general disk-based data structures

e.g., B+ tree



balanced search tree with very *large branching factor* (# pointers per block)

— worth it?

§FS Robustness

we like to think of the FS (unfortunately) as the "rock" of the OS

— when things go wrong (e.g., BSoD/ panic), hard restart and count on persisted data to save us i.e., FS can't count on OS to play nice!

e.g., unannounced crashes, incomplete operations, unflushed buffers, etc.

cannot ensure durability of in-memory data, but want to preserve *validity* of the file system when possible

e.g., file metadata is accurate, persisted data is not corrupted, etc.

Q: what *might* happen when a crash occurs?

important: differentiate between in-memory (cached) and on-disk (persistent) structures

note: FS aggressively caches data!

e.g., disk block allocation

1. update free bitmap
2. update inode

1. update cached free bitmap
2. update vnode
3. write back inode
4. write back disk bitmap

crash
(durability problem)

user responsibility; e.g., Unix `fsync` syscall

1. update cached free bitmap
2. update vnode
3. write back inode
4. write back disk bitmap

crash

(“free” space in use!)

1. update cached free bitmap
2. update vnode
3. write back disk bitmap
4. write back inode

crash
(lost space)

e.g., file deletion (# links = 0)

1. free inode & data blocks
2. remove directory link

crash
("free" space in use!)

e.g., file deletion (# links = 0)

1. remove directory link
2. free inode & data blocks

crash

("orphaned" inodes)

imminent data corruption vs. storage "leak"

(lesser of two evils)

soft updates: order software updates so that, in worst case, we only ever leak free space

— generally speaking, update free-space structures last leaked space isn't permanent!

can perform manual consistency check of FS e.g., UFS

- manually walk through all i-nodes and directory structures
- allocated i-nodes with 0 links can be reused
- allocated blocks with no referencing i-nodes can be "garbage collected"

the notorious "fsck" can report:

- Unreferenced inodes
- Link counts in inodes too large
- Missing blocks in the free map
- Blocks in the free map also in files
- Counts in the super-block wrong

BUT!

soft updates isn't trivial to implement, and may also conflict with caching needs

no good! FS is already messy to begin with!

another approach to FS robustness:

journaling / logging

a. say what you're about to do
b. do it
c. say that you did it

a. record what you're about to do
b. indicate that you finished (a)
c. do it
d. record that you did it

a. record FS update in journal entry ← **crash**

b. ensure journal entry is persisted

c. perform FS update

d. commit/delete journal entry

no journal entry on reboot;
no possible of FS inconsistency

a. record FS update in journal entry

b. ensure journal entry is persisted ← **crash**

c. perform FS update

d. commit/delete journal entry

on reboot, find partial journal entry;
no FS data corruption possible

a. record FS update in journal entry

b. ensure journal entry is persisted

c. perform FS update

d. commit/delete journal entry

crash

on reboot, journal shows incomplete FS update;
replay entry to ensure FS consistency

a. record FS update in journal entry
b. ensure journal entry is persisted
c. perform FS update
d. commit/delete journal entry ← **crash**

detect completed operation;
commit/delete entry

journal enables FS transactions

crash → replay journal;
skip incomplete entries

drawback?

huge overhead — “write-twice” penalty

† cannot delay persisting journal entries

ease overhead: *physical* vs. *semantic* journals

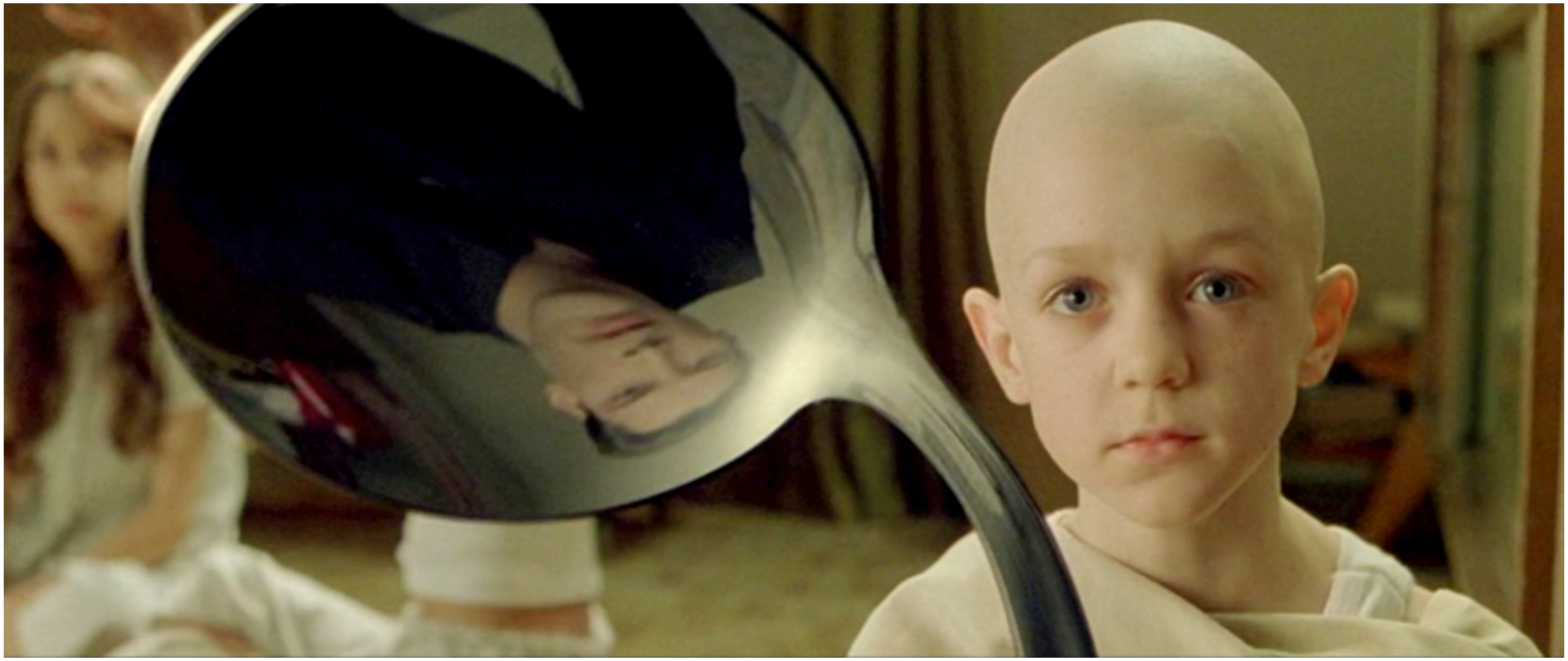
physical = record *block-level data* in journal

semantic = record *logical intent* when possible also, ensuring FS *consistency* arguably more important than short-term *data loss*

complete vs. metadata-only journal

Q: is there a way to eliminate the write-twice penalty and still get transactional behavior?

hint: think back to persistent data structures used to implement MVCC



“there is no spoon”

(the file system *is the journal*)

log-structured FS: all FS updates are persisted to the end of the journal

- file updates are effectively *copy-on-write*
- current FS state = log replay

for efficiency, periodically:

- garbage collect unreachable blocks, deleted files, etc., from log
- write FS checkpoints to avoid full replay

interesting benefit of LFS: *most writes are sequential* (but reads are scattered throughout the log) nifty idea, but horrible fragmentation!

impractical with HDDs, but what about SSDs?

- robustness w/o write-twice penalty. Hmmmmmmmm.

interesting: SSDs already kind of do LFS with TRIM wear leveling — writes occur elsewhere on disk from “replaced” block

- long term performance of SSDs has similar pattern to LFSes
- SSDs are also fast-to-read, slower-to-write

Soft updates, journaling, and LFSes

= *software* based solutions

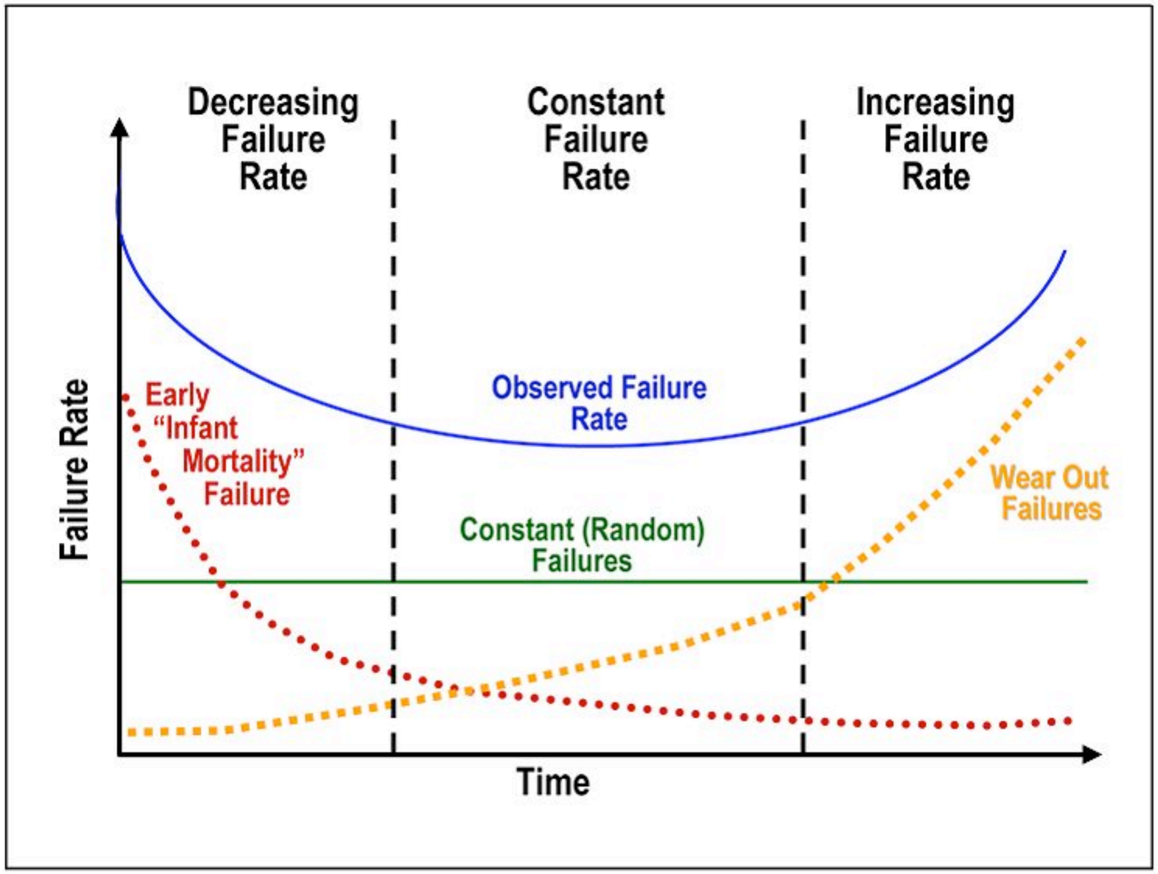
hard drive crash? #$%&#$#!!!!

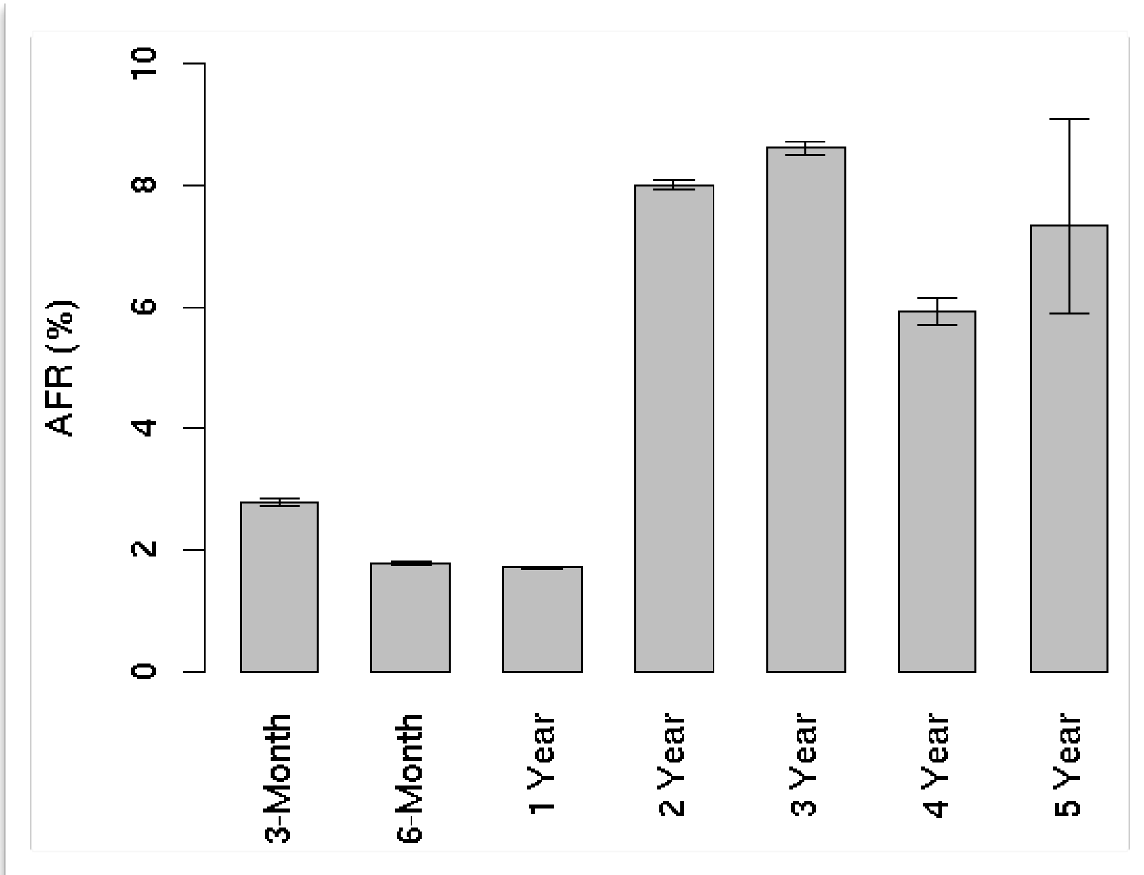
§Hardware level robustness

mean **t**ime **t**o **f**ailure

1,000,000+ hours!

“crap”


Decreasing Failure Rate
Constant Failure Rate
Increasing Failure Rate
Failure Rate
Observed Failure Rate
Early "Infant Mortality" Failure
Constant (Random) Failures
Wear Out Failures
Time



Failure Trends in a Large Disk Drive Population (Google, FAST '07) hard drive failure:

question of **when**, not **if**!

redundancy

preventing downtime

preventing data loss

Redundant **A**rray of **I**ndependent **D**isks

data robustness

secondary objectives:

- increased capacity
- improved performance

RAID array = one logical disk

transparent to OS/FS (ideally)

software vs. hardware RAID

RAID "levels"

combination of techniques

1. mirroring
2. striping
3. parity

Data bits	Odd Parity	Even Parity
0101010	00101010	10101010
0000011	10000011	00000011

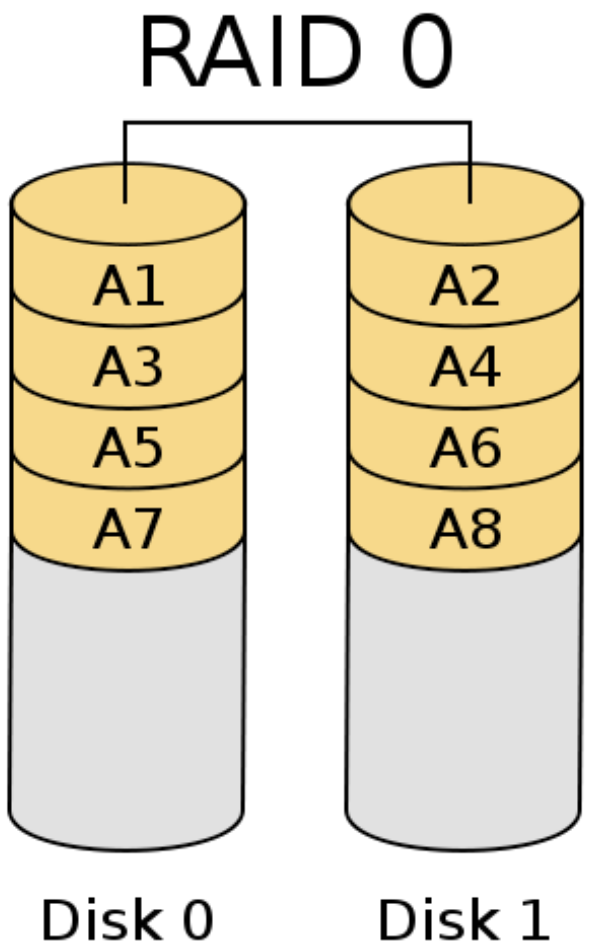
Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	X		X		X		X		X		X		X		X		X		X		
	p2		X	X			X	X			X	X			X	X			X	X		...
	p4				X	X	X	X					X	X	X	X					X	
	p8								X	X	X	X	X	X	X	X						
	p16																X	X	X	X	X	

Diagram courtesy Wikipedia

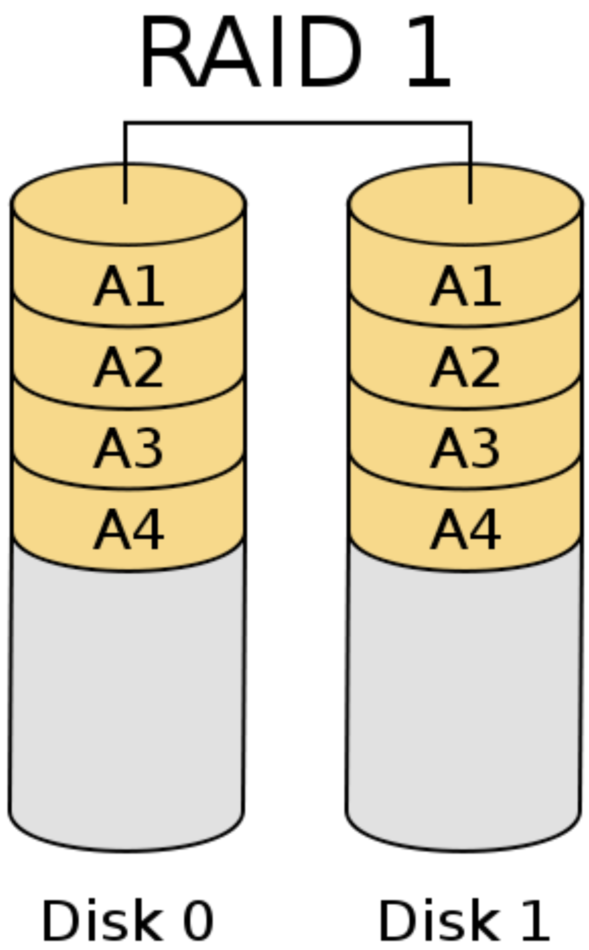
```
// x = A, y = B
x = x ^ y ; // x = A^B
y = x ^ y ; // y = A^B^B = A
x = x ^ y ; // x = A^B^A = B
```

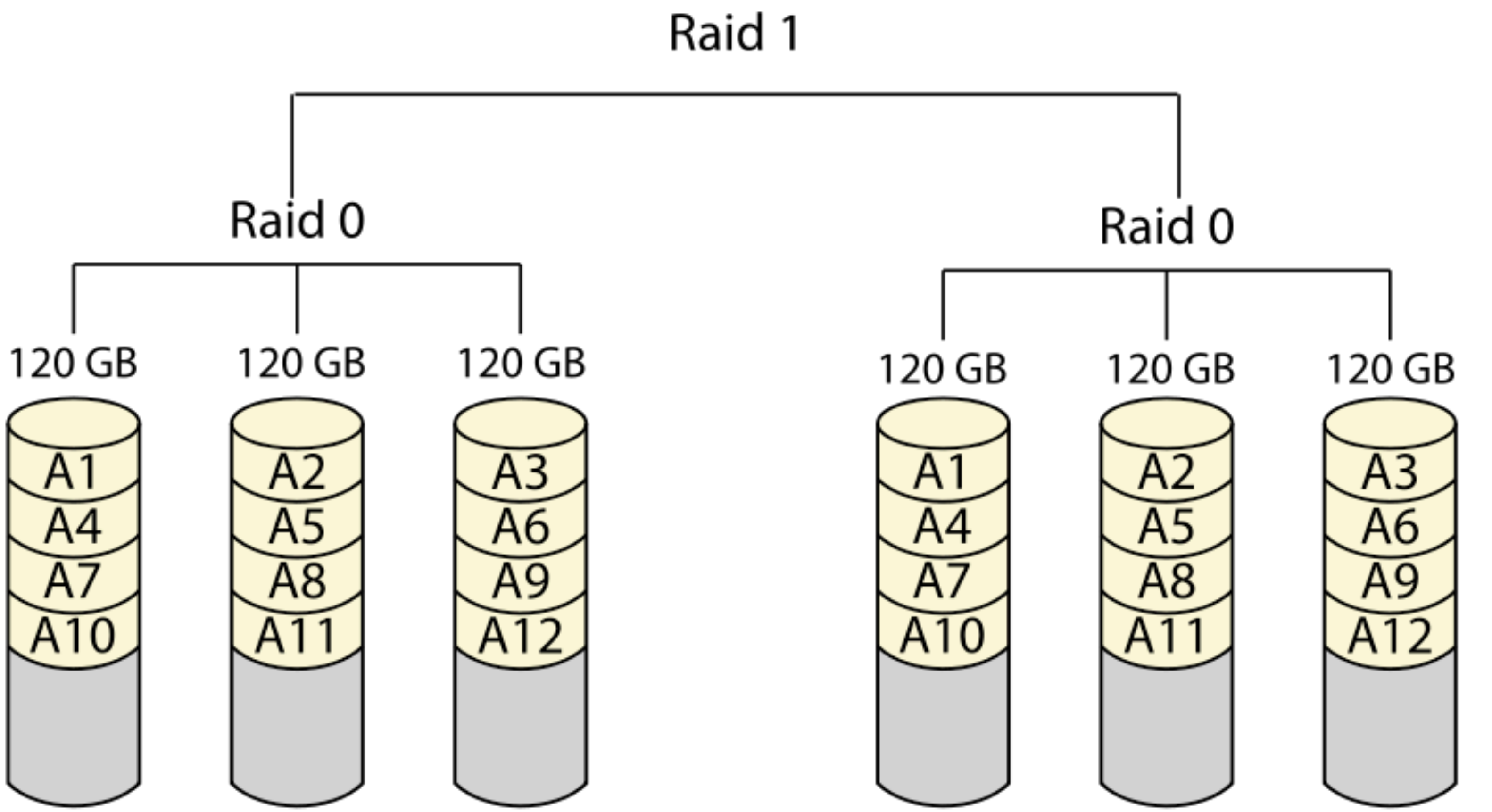
$$B_1 \oplus B_2 \oplus \ldots \oplus B_{N-1} \oplus B_N \Rightarrow B_P$$

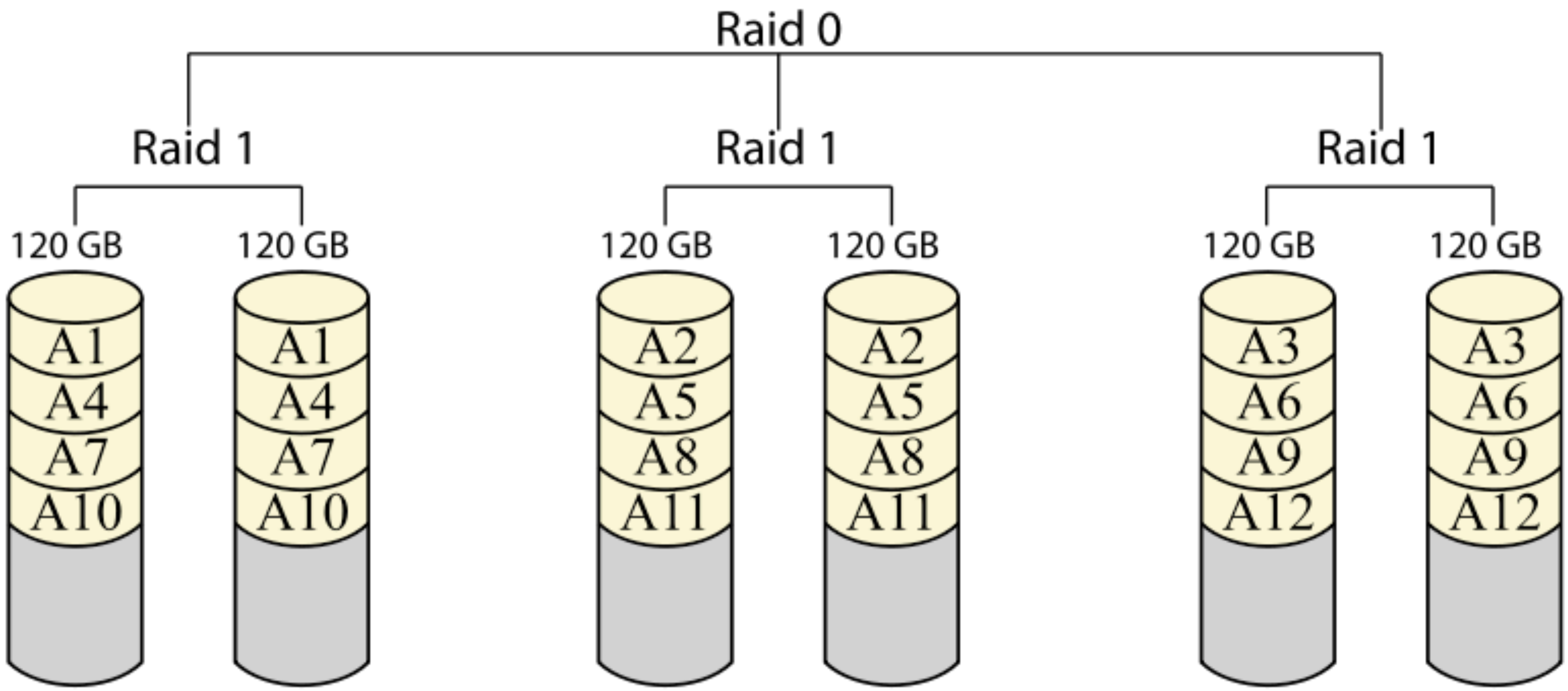
$$B_1 \oplus B_2 \oplus \ldots \oplus B_{N-1} \oplus B_P \Rightarrow B_N$$

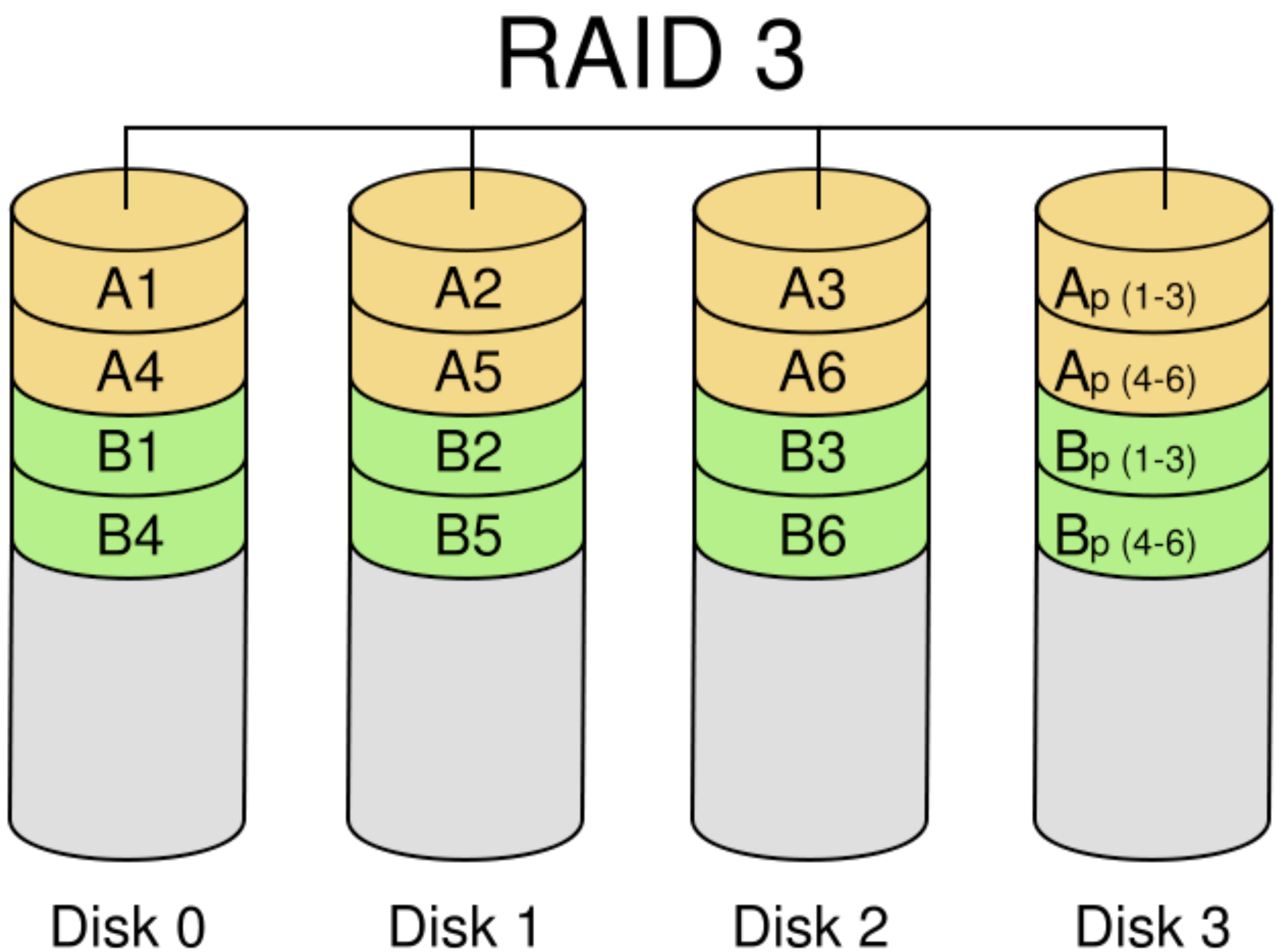


figures courtesy Wikimedia Commons

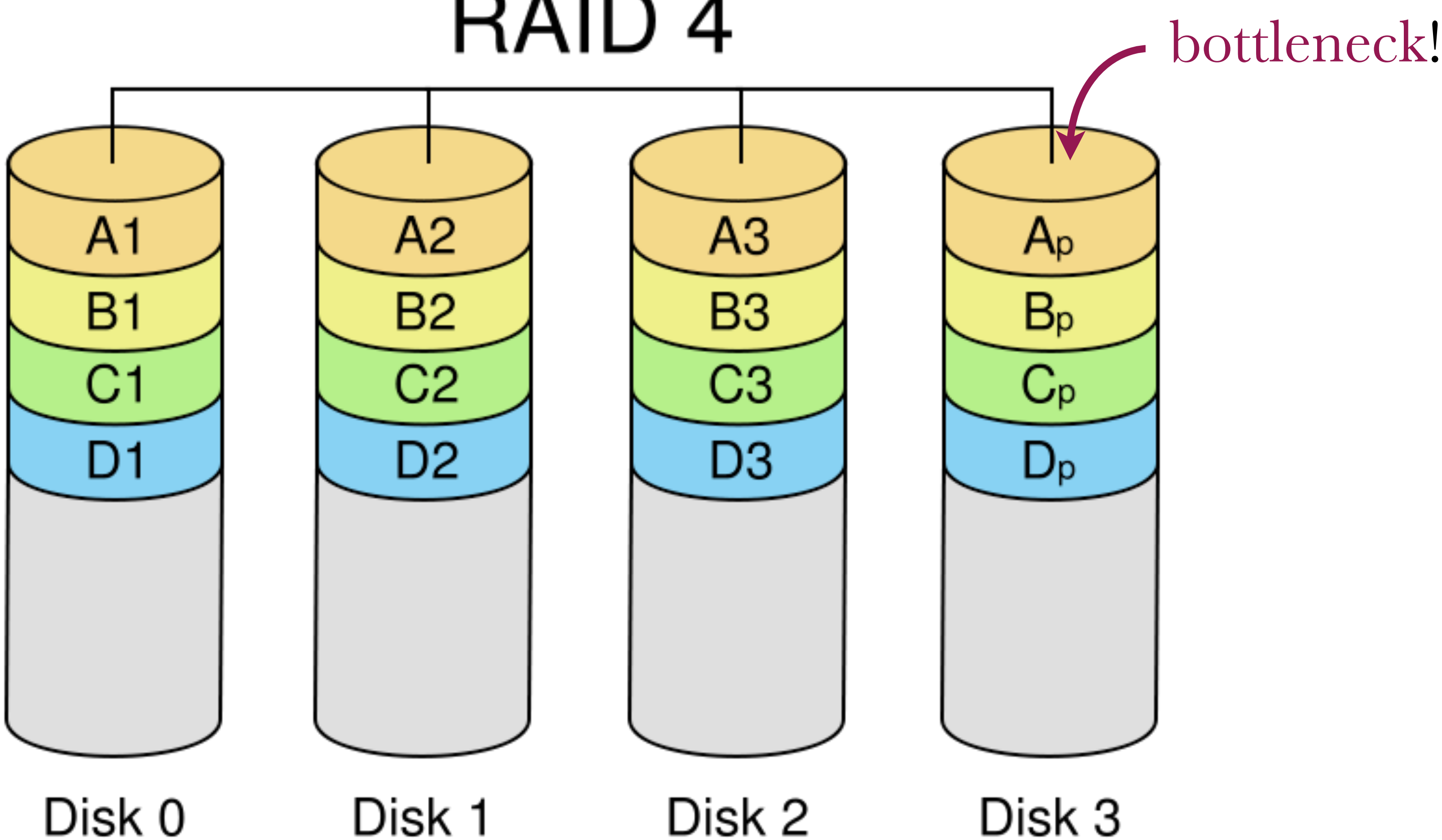

RAID 1
A1
A2
A3
A4
A1
A2
A3
A4
Disk 0
Disk 1


Raid 1
Raid 0
Raid 0
120 GB
120 GB
120 GB
120 GB
120 GB
120 GB
A1
A4
A7
A10
A2
A5
A8
A11
A3
A6
A9
A12
A1
A4
A7
A10
A2
A5
A8
A11
A3
A6
A9
A12


Raid 0
Raid 1
Raid 1
Raid 1
120 GB
120 GB
120 GB
120 GB
120 GB
120 GB
A1
A4
A7
A10
A1
A4
A7
A10
A2
A5
A8
A11
A2
A5
A8
A11
A3
A6
A9
A12
A3
A6
A9
A12

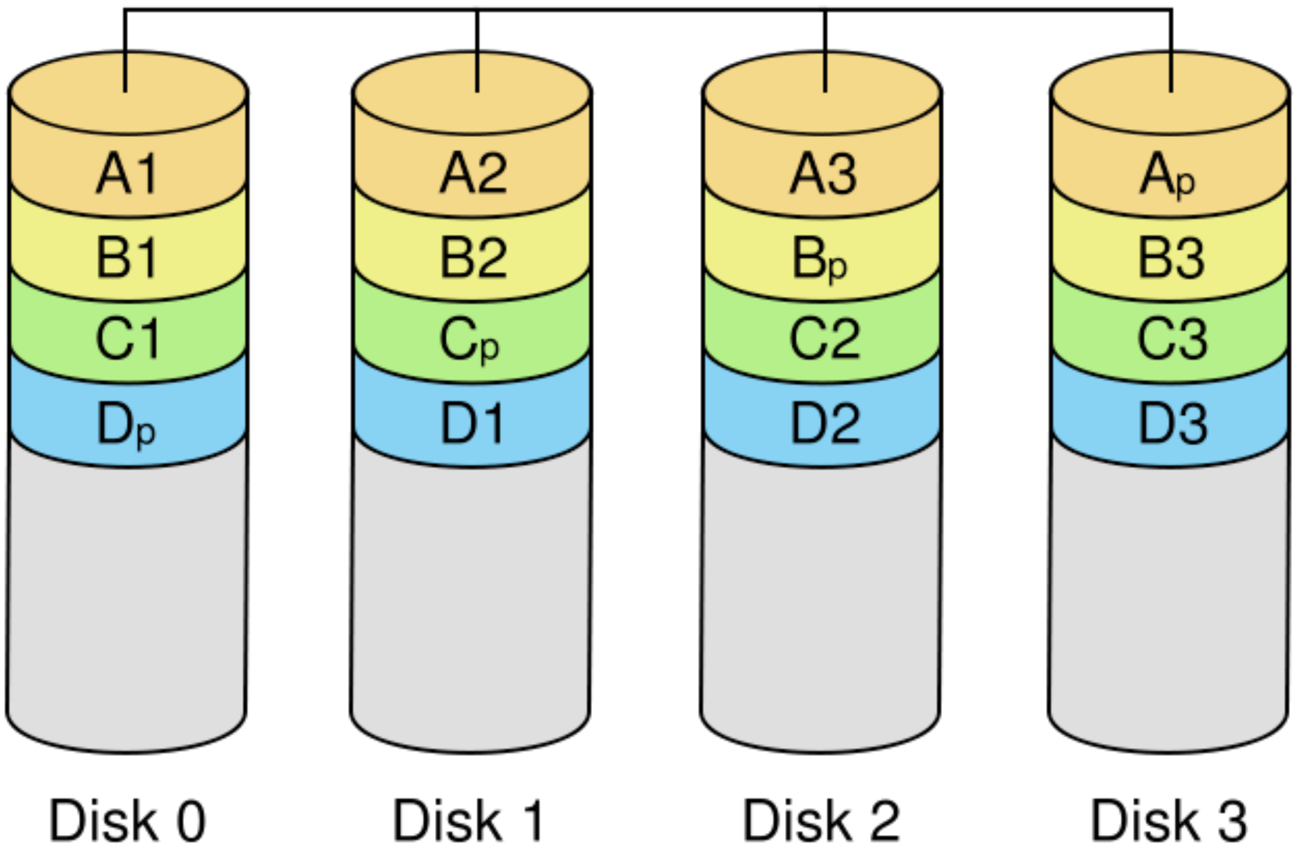

RAID 3
A1
A4
B1
B4
A2
A5
B2
B5
A3
A6
B3
B6
Ap (1-3)
Ap (4-6)
Bp (1-3)
Bp (4-6)
Disk 0
Disk 1
Disk 2
Disk 3

RAID 4



Update: $A_1 \oplus A_2 \oplus A_3 \oplus A_3 \oplus A_3' \Rightarrow A_P'$

RAID 5


A1
B1
C1
Dp
A2
B2
Cp
D1
A3
Bp
C2
D2
Ap
B3
C3
D3
Disk 0
Disk 1
Disk 2
Disk 3

write penalty

battle **a**gainst **a**ny **r**aid **f**ive

http://www.baarf.com/

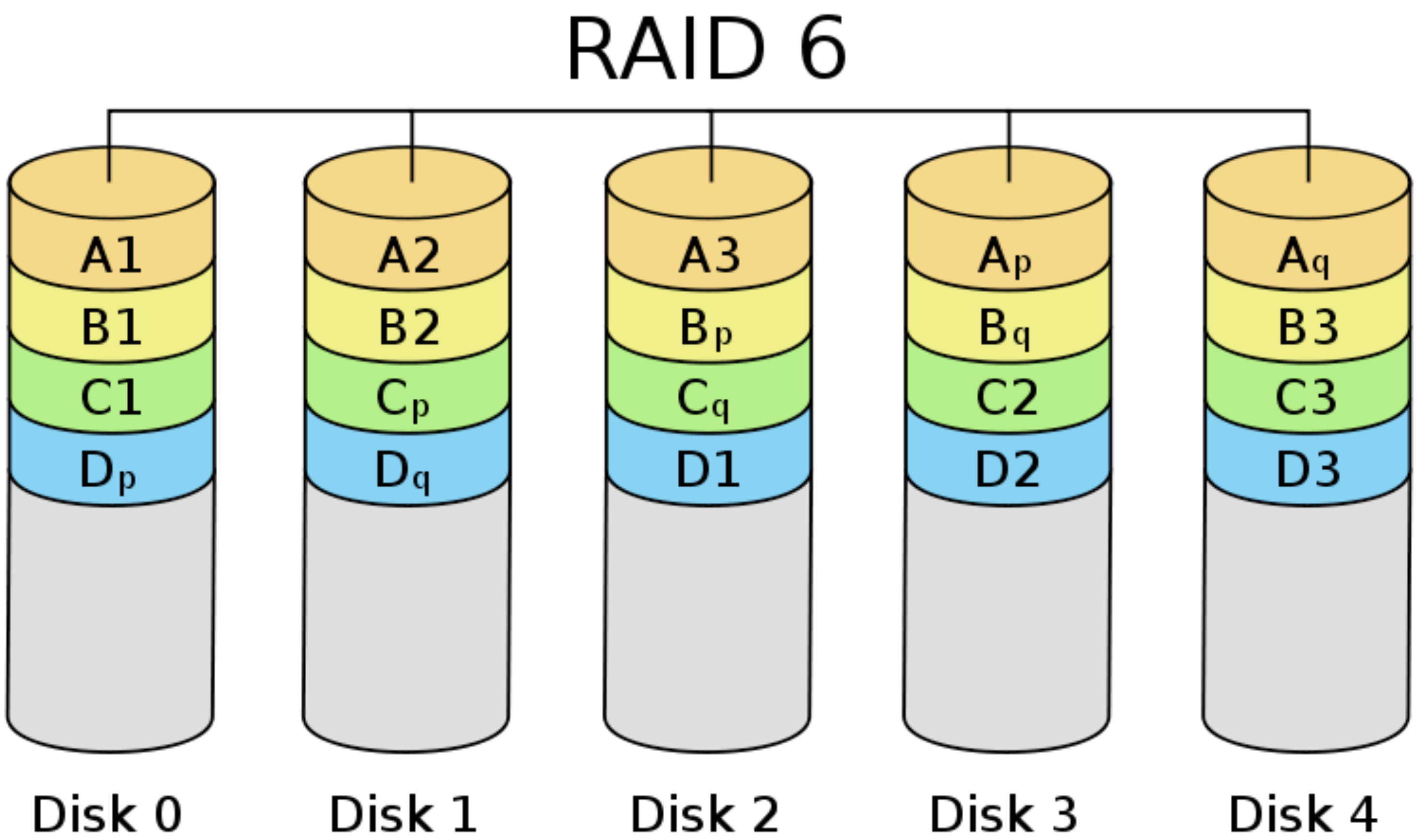
data & parity updates separate

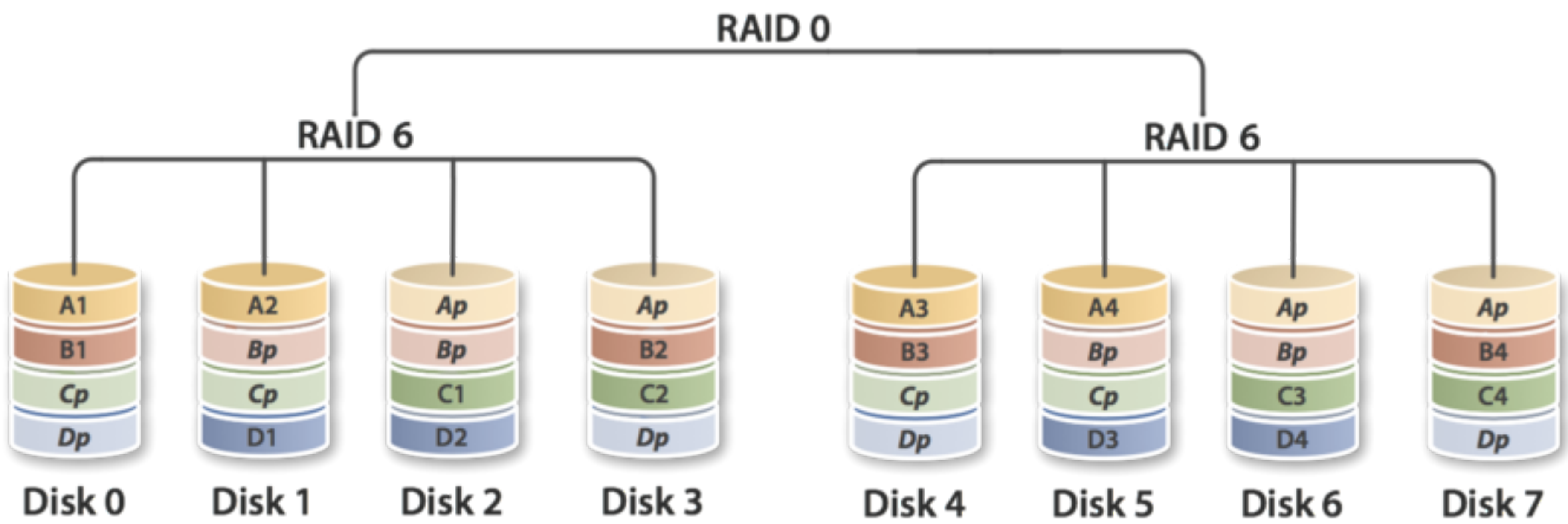
failure in between?

write hole

caching / non-volatile storage

vs. RAID 10


RAID 6
A1
A2
A3
Ap
Aq
B1
B2
Bp
Bq
B3
C1
Cp
Cq
C2
C3
Dp
Dq
D1
D2
D3
Disk 0
Disk 1
Disk 2
Disk 3
Disk 4


RAID 0
RAID 6
RAID 6
A1
B1
Cp
Dp
A2
Bp
Cp
D1
Ap
Bp
C1
D2
Ap
B2
C2
Dp
A3
B3
Cp
Dp
A4
Bp
Cp
D3
Ap
Bp
C3
D4
Ap
B4
C4
Dp
Disk 0
Disk 1
Disk 2
Disk 3
Disk 4
Disk 5
Disk 6
Disk 7

§Case study: xv6 (Unix)