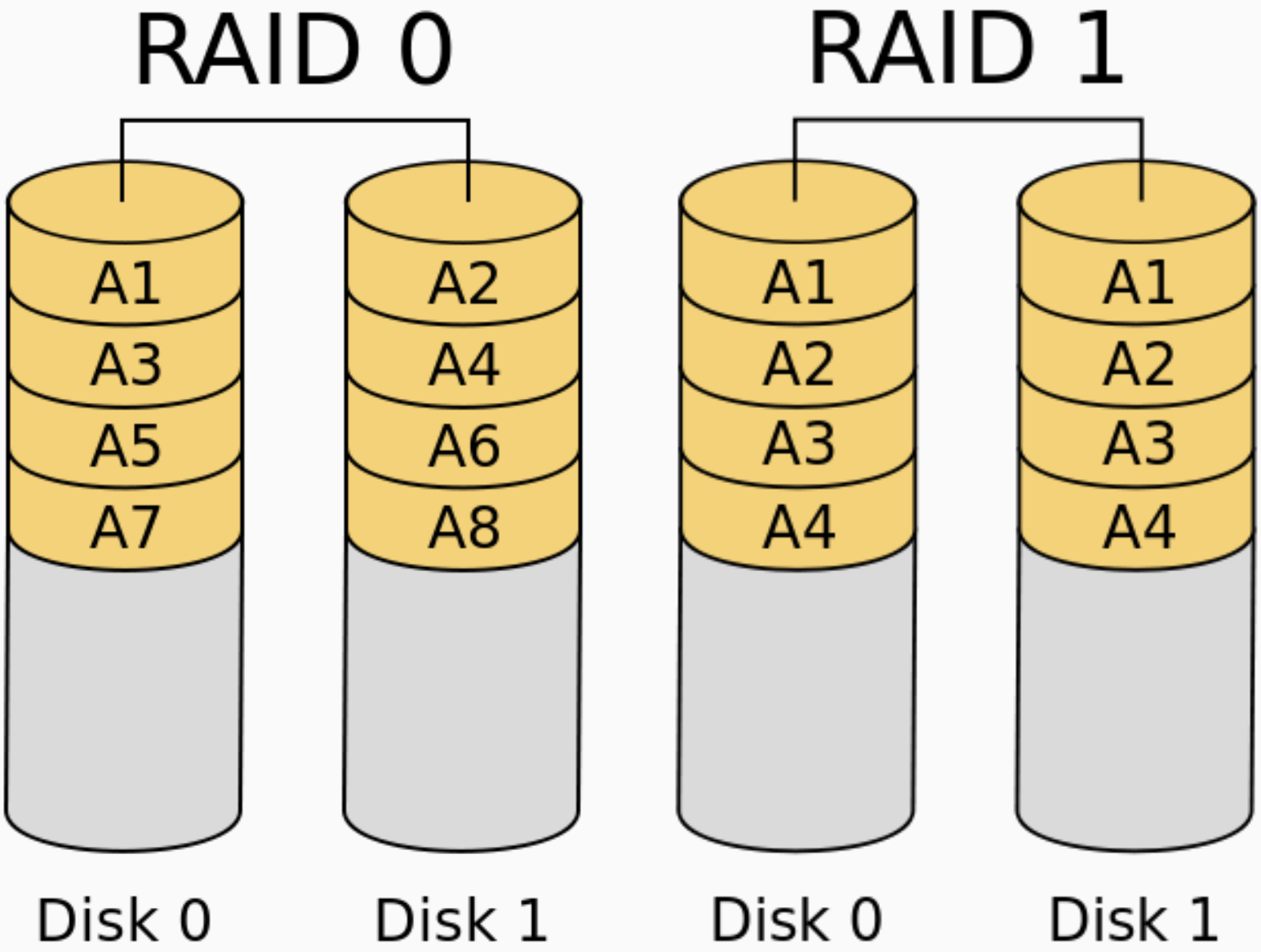
File Systems: The Interface

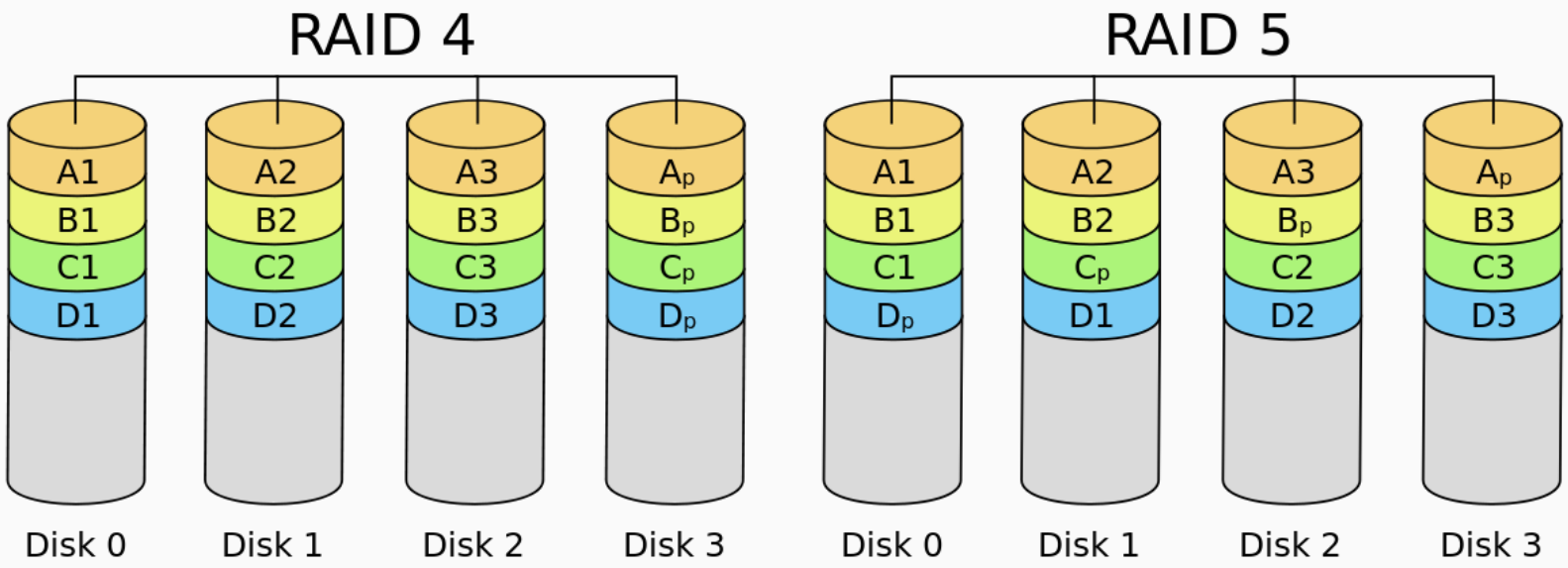
Azza Abouzied

From Last Class: RAID

Redundant Array of Independent Disks



Redundant Array of Independent Disks



Parity computation is

$$\mathsf{P} = \bigoplus_i D_i = \mathsf{D}_0 \oplus \mathsf{D}_1 \oplus \mathsf{D}_2 \oplus \ldots \oplus \mathsf{D}_{n-1}$$

Raid-4 has a dedicated parity disk, while Raid-5 stripes the parity disk across devices. **What benefit do we get from this striping?**

On Parity

Note that $X \oplus X = 0$

So if:

$$\mathsf{P} = \bigoplus_{i} D_i = \mathsf{D}_0 \ \oplus \ \mathsf{D}_1 \ \oplus \ \mathsf{D}_2 \ \oplus \ \ldots \ \oplus \ \mathsf{D}_{n-1}$$

Then to remove the effect of say D_2 from the parity we simply compute:

$$\mathsf{P} \oplus \mathsf{D}_2$$

Now we can change D_2 to $\mathsf{D}_2^{'}$ and recompute the new parity as follows:

$$\mathsf{P}^{'} = \mathsf{P} \oplus \mathsf{D}_2 \oplus \mathsf{D}_2^{'}$$

Performance and Reliability

Consider Raid 0, 1 and 5 and 8 equivalent disks

1. How much usable storage does the system receive?
2. If we only do reads without verification. What is the expected throughput if each disk does 100 requests/second?
3. If we only do writes. What is the throughput now if each disk does 100 requests/second?
4. What is the min number of disks that **may** fail before data is lost?
5. What is the minimum number of disks that **must** fail to guarantee data loss?

File Systems

Why is Memory not enough?

The main issues:

1. Memory Address Space
    - Limited
    - Large Hadron Collider produces 15PB (10^{15} bytes) a day!
2. Memory is volatile
    - Survive system crashes and process termination.
    - Power failure.
3. Shared data across processes

Why not just use block disk drivers?

A nice simple block device interface:

- Read block *k*
- Write block *k*

But ...

Why not just use block disk drivers?

A nice simple block device interface:

- Read block *k*
- Write block *k*

But ...

How to search for information?
How to protect one user's data from another user?
Free blocks?
How to cache blocks?
Mapping Block size to Page size?

Need An Abstraction

Files are just an abstraction. They ...

1. represent a logical unit of information
2. are persistent
3. support key operations: create/remove/write/read by processes
4. are managed by the OS ... this is the file system
5. are structured
6. have names,
7. have different access modes (sequential/random)
8. have different protection modes
9. have different physical implementations on a block device (e.g. on flash drives or disk drives)

The FS Interface

The Interface vs. The Implementation

Interface (user's / process' view)

How to call a file?

What makes a file?

What operations are allowed?

Implementation (file System view)

How to track free storage: linked list of free blocks or a bitmap?

How to layout a file on disk?

How to ensure reliability

Performance, consistency, caching

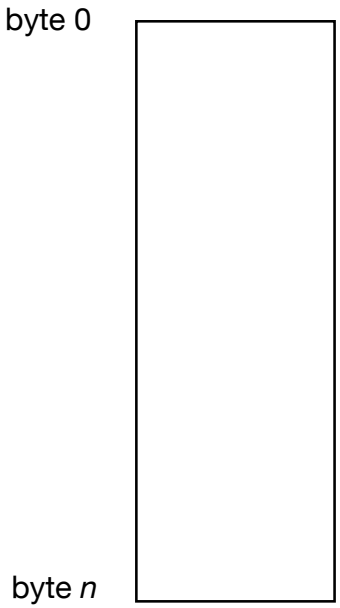
Interface considerations

Design your own interface

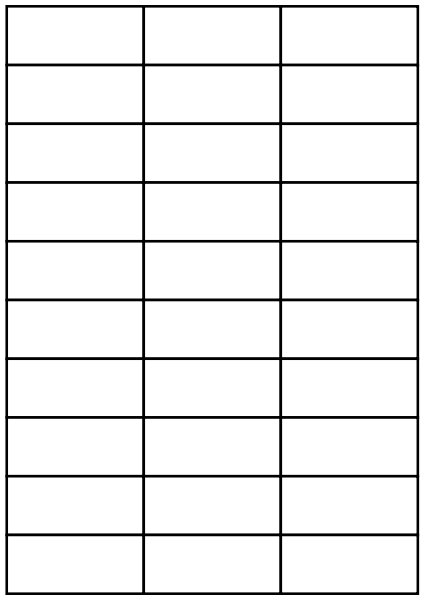
Consider

1. **Structure:** what does the file look like? and in turn what operations would it support?
2. **File Naming:** how do you refer to a file?
3. **File Typing:** how do you encode type? Should you?
4. **Block Access:** How do you expose the bytes of file? Sequentially? Directly?
5. **Organization:** Directories? Links?
6. **Hardware Abstraction:** Device-based mount points?
7. **Special Files:** Would you include more than files?
8. **Access Control:** How do you represent user privileges?

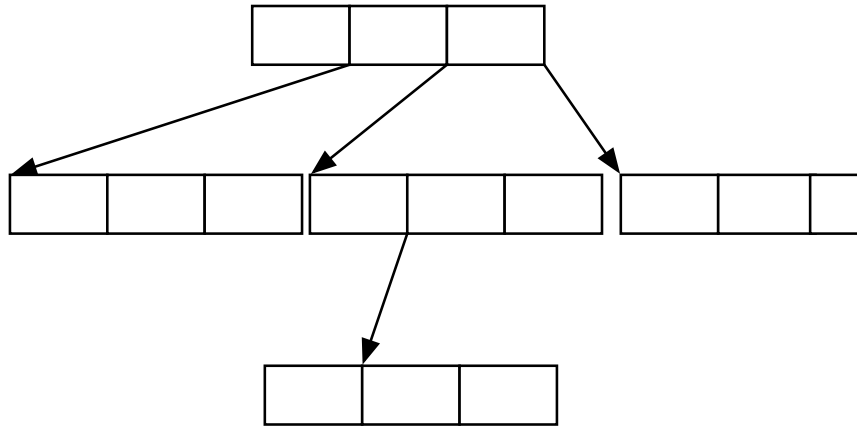
Structure



Bag of bytes
Prevalent file structure



Records
Resurfacing as Key-Value Stores



Trees
Moved to the database (B-trees)
Resurfacing as XML/JSON special systems

Naming

MSDOS: 8 alphanumeric characters, case insensitive, with 3 character extension.
Why? Fits nicely in the 16B directory table entry

OpenVMS distributed file system: NODE“accountname password” :: device : [directory . subdirectory] filename . type ; ver

Unix: 255 arbitrary characters, case sensitive, extensions are just characters and multiple extensions allowed.

NTFS: Windows NT file system allows Unicode characters: $\epsilon\lambda\lambda\eta\nu\iota\kappa\alpha$ letters.

Types in the Metadata

MSDOS: Extensions ".docx": OS opens relevant application

Unix: Magic numbers: (0x7F)('E')('L')('F')
Header, not centrally administered and outside the control of the File System

Windows NTFS, Apple HFS:
Fork/Alternate Data Streams
Each file can have a data fork (actual file) and a resource fork (metadata and how GUI displays the file) and 0+ named forks.
Forks/ADS were the subject of an **interesting security problem**

How do you access the file?

Sequential

- read() //gives you next byte from cursor
- write() //writes at next byte from cursor

Direct

- read(n) //read nth byte
- write(n) //write at byte n

Why is the distinction from an interface perspective weak?

How do you organize/find files?

How do you get the file handle?

1. Hash table?
2. Search through a **directory** structure?
    - flat? tree? graph?
    - where is the metadata located?
    - encode location information in the name?
3. Hybrid hash/directory? What about document locality?

How do you organize/find files?

Links as shortcuts

1. *Hard link:* File reflects information that is contained in multiple directories.
2. *Symbolic/soft link:* A link to the exact path name of the file. We simply follow the path.

Links require some further design considerations:

1. What if a target is deleted?
2. Can a remote symbolic link refer to a local file?

Hardware Abstraction & Mount Points

Unixish perspective

mount //lists all mounted partitions
mount /dev/hda2 /media/movies

1. File system is mounted on a mount point
2. Mount point existing directory in an existing file system
3. Mount initializes a new file system with necessary kernel data structures
4. Operations on mount point redirected to the correct file system
5. The root file system is always special

unmount /media/movies //closes all open files

Windows perspective

C:\blech
No root directory.

Special Files

What are these files?

/dev/hda/

/dev/audio/

/dev/console/

/proc

Access Control Mechanisms

1. **Authentication**

Checking identity

Usually seen as login passwords; credit card numbers with security code and mom's maiden name; driver's license

2. **Authorization**

Is user *x* allowed to do action *y*?

Usually check x's capabilities against a database of rights

3. **Enforcement**

Trusted party has to enforce access controls

Which party would you trust? Kernel!

File Permissions

Most operating systems now support POSIX access control bits
rwx rwx rwx for user, group, all.

You can change them yourself: chmod a+x

Earlier Windows versions did not support POSIX-only file attributes such as hidden, read-only, ...

The interplay of interface and implementation

Access Control Lists

For each resource, indicate which users are allowed to perform which operations.

1. Fits with the POSIX model
2. Checking access requires enumeration
3. With no groups, a generic read access means all users need to be on the list.

Capabilities

For each user, indicate which files they can access.

1. Can only see an object if you are capable of it
2. Selective revocation of rights is hard.
3. What happens when the object is removed?
4. How do you distribute or share capabilities?

Linux VFS

The Linux Virtual File System

A thin interface layer that all processes go through the file system through.

VFS provides a standard interface to all processes with a unified image of files with implementation-specific organization aspects such as the use of **index-nodes** (inodes).

Different file systems still have to conform to the VFS view and create inodes even if they don’t exist.

Read ahead

A Fast File System for UNIX*
by Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, Robert S. Fabry

Analysis and Evolution of Journaling File Systems
by Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
in USENIX 2005

The Design and Implementation of a Log-Structured File System
by Mendel Rosenblum and John K. Ousterhout
in SOSP 1991

Questions?