UNIVERSITY of York

Files, Filesystems, and File Formats

Peter Hill

Overview

- Hardware
- Filesystems
- What is a file?
- File formats

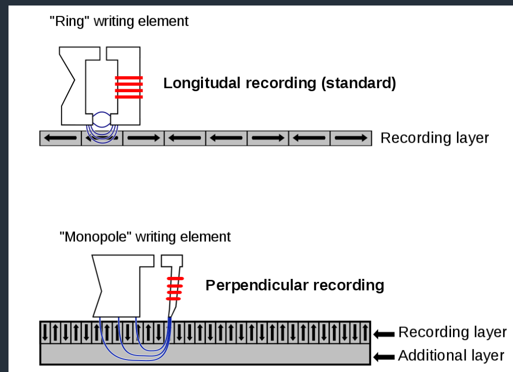
What is a file?

Brett Jordan, CC BY 2.0

Me, CC-BY-SA 4.0

Magnetic recording

- Most common way to store data is via magnetisation:
  - floppy disks
  - hard disks
  - audio or data tapes
- All basically work the same way
- Store information in direction of magnetisation
  - Could be analog like in audio tape
  - or digital like in hard disks
- Hard disk drives (HDDs) great compromise of speed, density, and cost
- Data tapes very slow, but amazingly dense and cheap
  - Great for long-term storage and backups



Types of magnetic writing heads (Wikimedia, Public Domain)

Solid state drives (SSDs)

- SSDs use "floating gate" transistors to store charge
- Insulated "cell" that holds charge and modifies threshold voltage of transistor
- Insulation breaks down after too many writes
- Good density with no moving parts
- Comparatively expensive vs magnetic media
- Much, much faster

Fig. 5. $0.21\mu m$ 2T memory cell content extraction

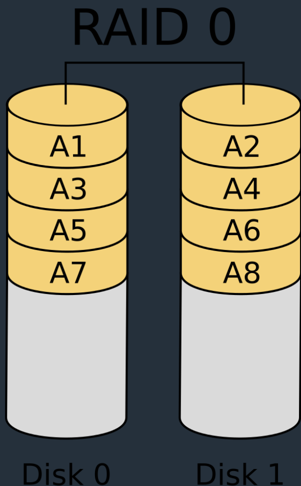
"Reverse Engineering Flash EEPROM Memories Using Scanning Electron Microscopy", F. Courbon et al, CARDIS 2016, https://doi.org/10.1007/978-3-319-54669-8_4

Dealing with failure

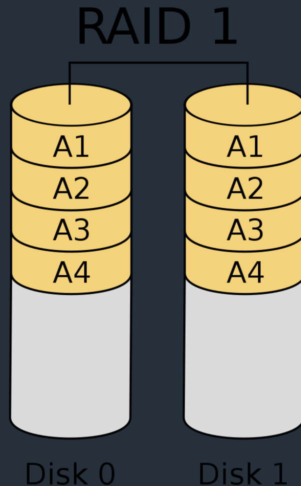
"ShipServ data centre", yurri, CC BY-NC 2.0

- ~millions of hours between failures...
- ...but 100,000 disks in one data centre means tens of disks need replacing each day
- RAID: Redundant Array of Inexpensive Disks
- Writing data to more than one disk at the same time
- Can improve resilience to disk failure
- Can improve IO (input/output) performance
- RAID is not the same thing as backups!

RAID 0 and 1: stripping and mirroring

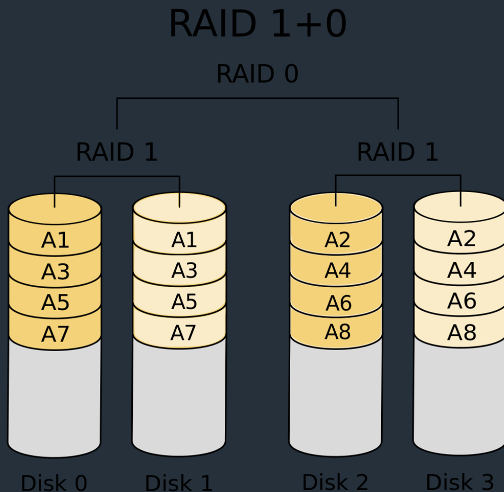


RAID 0, Cburnett, CC BY-SA 3.0



RAID 1, Cburnett, CC BY-SA 3.0

RAID 1+0 (10): stripping + mirroring



RAID 10, Cburnett, CC BY-SA 3.0

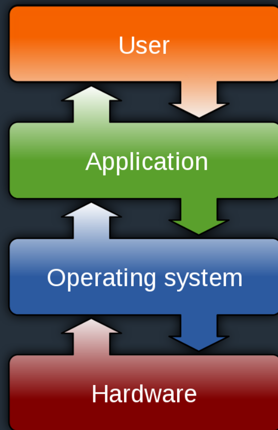
Disk drives

- All* disk drives are based on **sectors**
- Sector is contiguous chunk of data on drive read or written all at once
- OS uses **blocks** (typically 512 bytes)
- Big HDDs may have sectors of 4096 bytes
- Lots of small files can use up more physical disk space than their logical size

Filesystem drivers

- Already skipped over some layers!
  - like the firmware *inside* the disk
- Filesystems control how operating system interacts with files
- Lots of different types of filesystems, some specialised for different hardware (e.g. optical discs)
- Also virtual filesystems: in UNIX, everything is a file! e.g. `/dev/null`, `/proc/meminfo`

Some common filesystems: - ext4, XFS, btrfs, FAT, exFAT, NTFS, HFS+, Lustre, GPFS



Filesystem hierarchy, Golftheman CC-BY-SA 3.0

What does a filesystem do?

- Manage space:
  - writing blocks/sectors
  - handles fragmentation of disk space
  - disk quotas
- Filenames:
  - case sensitivity: are `MYFILE`, `myfile`, `MyFiLe` the same?
  - allowed characters
  - allowed names (YouTube: Tom Scott, "Why You Can't Name A File CON In Windows")
- Directories or folders
- Metadata:
  - timestamps
  - file size (logical/physical)
  - file location
  - disk metadata
- Permissions

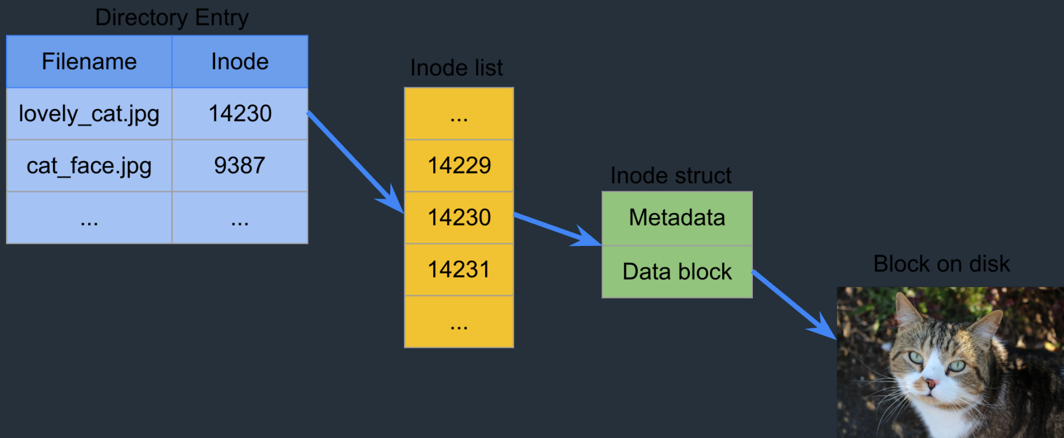
Fancier features

- Journaling:
  - Keep track of changes to be made
  - If there's a crash before completion, can replay changes from last good state
- Encryption:
  - Encrypt the whole disk, rather than individual files
- Copy-on-write:
  - Make zero-sized backups
- Distributed:
  - Beyond RAID, how to have a petabyte filesystem
- Hierarchical storage:
  - Burst buffers

Filesystem data structures

- How does filesystem know where on disk to find `MyFile.txt`?
- Information about files and directories stored in data structures directly on disk
- Directory data structures contain mapping of filename to “inode” number
- Inode table has all the metadata about file – except the filename
- Metadata includes location of file blocks on disk

Filesystem data structures



inode schematic

Filesystem data structures

- Some filesystems have limit on number of inodes $=>$ limit on number of files/directories
- Some filesystems have performance issues with large number of files in a directory
- Can have multiple filenames mapped to same inode!
- All this takes up space on disk

Virtual files

- OS manages all hardware connected to computer through virtual files
  - block devices: hardware that reads and writes in discrete blocks
  - character devices: read and write a character at a time
  - pseudo-files: “files” created by OS
  - regular file: a regular file
- Pseudo-files:
- Can be used for information
  - `/proc/cpuinfo`, `/proc/meminfo`, `/proc/sys/kernel/version`
- or control aspects of OS
  - `/proc/sys/kernel/panic_on_oops`, `/proc/sys/fs/file-max`
- Special types of input
  - `/dev/urandom`, `/dev/zero`, `/dev/full`
- or special types of output
  - `/dev/null`

File formats

- Files are collections of bytes
- Need a program to give them meaning
- Same way computer programs are collections of bytes, and need a CPU to give them meaning
- File format is a standard for specifying the meaning
- Could be collective standard, e.g. JPEG, MP3, LaTeX
- or proprietary, e.g. `.docx`, Photoshop
- or even just internal to a particular program

How does OS know what type a file is?

- Windows just goes by extension
- Others go by magic numbers

```
$ hexdump -Cn8 filesystem_os.png
00000000  89 50 4e 47 0d 0a 1a 0a  |.PNG....|
```

- `file` utility can tell you what type a file is

```
$ file filesystem_os.png
filesystem_os.png: PNG image data, 519 x 768, 8-bit/color RGBA,
non-interlaced
```

- We can open a file with the "wrong" program
  - Will probably get nonsense!

Text files

Plain text

```
- Just the raw text, no formatting
- Pros: good for simple, unstructured text
- Cons: terrible for just about everything else
```

Text files

LaTeX

```latex
\begin{block}{LaTeX}
\begin{itemize}
\item ``Plain text'', but with \emph{formatting} and semantic markup
\item Requires compiling into rendered document
\item Pros:
  \begin{itemize}
  \item Great for maths-heavy text
  \item Great for journals
  \end{itemize}
\item Cons:
  \begin{itemize}
  \item Requires editing-compiling loop to see output
  \item Difficult to collaborate with non-users
  \end{itemize}
\end{itemize}
\end{block}
```

Text files

Word/Open Document Format

- Zipped directory of XML files
- What You See Is What You Get
- Formatting and semantic info
- Pros:
  - Easy to use and share
- Cons:
  - Not simple text file means difficult to integrate into software packages
  - Easy for semantic info to get out of sync with formatting

```
<w:p><w:pPr><w:pStyle w:val="Heading2"
  <w:r><w:t xml:space="preserve">
    Word/Open Document Format
  </w:t></w:r>
</w:p>
<w:p><w:pPr><w:numPr><w:ilvl w:val="0"
  </w:numPr><w:pStyle w:val="Compact"/
  <w:r><w:t xml:space="preserve">
    Zipped directory of XML files
  </w:t></w:r>
</w:p>
```

Text files

Markdown/ReStructuredText

```
## Markdown/ReStructuredText

- "Plain text", but with *formatting* and *semantic* markup
- Requires compiling into rendered document, but still human-readable
- [Github Flavour](https://guides.github.com/features/mastering-markdown/)
- Pros:
    - Great for READMEs and technical documents
    - Most Git web services can render both formats directly
    - ReadTheDocs can automate building as both HTML and PDF
- Cons:
    - Reliant on compiling program for rendering
    - No one standard for Markdown, hodge-podge of extensions
    - ReStructuredText is more complex
```

Configuration files

JSON

- JavaScript Object Notation
- As used in: practically everything
- Pros: Great for data transfer
- Cons: No comments

```
[
  {
    "Ziggy": { "age": 6, "colour": "Tabby" }
  },
  {
    "Lana": { "age": 6, "colour": "Black" }
  }
]
```

Configuration files

YAML

- Yet Another Markup Language
- As used in: Github Actions
- Pros: Less quoting required than JSON
- Cons: More features means more complex

```yaml
# Here are some cats
- Ziggy:
  - age: 6
    colour: Tabby
- Lana:
  - age: 6
    colour: Black
```

Configuration files

TOML

- Tom's Obvious, Minimal Language
- As used in: pyproject.toml
- Pros: Simpler syntax than YAML
- Cons: More opinionated than JSON

```toml
# Here are some cats
[Ziggy]
age = 6
colour = "Tabby"

[Lana]
age = 6
colour = "Black"
```

Output files

Raw binary

- Just dumping the internal state to disk!
  - For example, Fortran unformatted files or Python binary files
- Good for intermediate/checkpoint state
- Pros:
  - Can be very fast to implement and use
  - Guaranteed exact round-tripping (on same machine, version)
- Cons:
  - Risk of not being portable across machines/versions
  - Very low discoverability/not self-documenting
  - Can be difficult to manage complex or growing datasets

Output files

Comma/tab-separated values

- Writing everything out in text
- Good for small to medium sized datasets
- Pros:
  - Near universal, everything from Excel to Haskell can read/write
  - Can be self-documenting to some extent
- Cons:
  - Slow/expensive for lots of data
  - Less good for multi-dimensional or complex data

Output files

HDF5/netCDF

- Hierarchical Data Format
- Structured data files with lots of features
- Good for large or complex datasets
- Pros:
  - High discoverability/self-documentation
  - Great for datasets with growing elements
  - Can attach attributes and coordinates to values
  - Compression means writing can be faster
  - Parallel IO API for HPC filesystems
- Cons:
  - Low-level API is complex
  - Needs special library to read files
  - Strings weirdly complex?

Conclusion

- Disk drives read/write in discrete blocks
  - Implications for writing small files
- Filesystems store file information in inodes
  - Implications for numbers of files
  - Implications for large files
- Choose your file formats carefully