DSC204A: Scaleable Data Systems Winter 2025

12: File Systems

Instructor: Umesh Bellur *Scribe: Shuyu Wang*

1 File-Based Interprocess Communication (File IPC)

File IPC is a mechanism where processes communicate through the file system by writing and reading from a shared file.

1. The **sender** process writes data to a file.
2. The **receiver** process reads the data from the file.
3. A challenge arises: *when should the receiver read the file?*
4. Therefore, synchronization is essential, and is typically enforced using **file locking** to prevent race conditions or partial reads/writes.

2 Signals for Synchronization

Signals can be used in UNIX systems for simple synchronization between processes. Below is a summary of how signal-based synchronization works:

- Use the `os.kill(pid, signal)` function to send signals such as `SIGTERM`, `SIGUSR1`, or `SIGALRM` to a target process.
- The receiving process should register a **signal handler** using `signal.signal()` to define how it should respond when a signal is received.
- Commonly used signals include:
  - `SIGALRM`: Alarm clock or timer expiration
  - `SIGUSR1`, `SIGUSR2`: User-defined signals

3 File System

3.1 What is a File System?

A file system is software that:

- Organizes and manages data on storage devices such as HDDs, SSDs, and USB drives.

- Maps logical file structures (e.g., file names, directories) to physical locations on disk.
- Provides an interface for basic file operations such as `create`, `read`, `write`, and `delete`.

3.2 Levels of File System Design

1. **Logical Level:**
   - Offers the abstraction of files and directories.
   - Exposes system call APIs such as `open()`, `read()`, and `write()` for file handling.
2. **Physical Level:**
   - Interfaces directly with disk hardware and firmware.
   - Responsible for moving bytes to/from storage devices and main memory (DRAM).

4 Abstractions: File and Directory

4.1 File

A **file** is a persistent sequence of bytes that stores a logically coherent digital object for an application.

- **File Format**: An application-specific standard that defines how to interpret and process a file's bytes.
- Hundreds of file formats exist (e.g., `.txt`, `.doc`, `.gif`, `.mpeg`), varying in data models, types, and domain-specific usage.
- **Metadata**: Summary or organizing information about file content (also called *payload*), stored within the file itself; often format-dependent.

4.2 Directory

A **directory** is a cataloging structure that contains references (links) to files and/or other directories (possibly nested).

- Directories are typically treated as a special kind of file.
- They support hierarchical organization via subdirectories, parent directories, and a root directory.

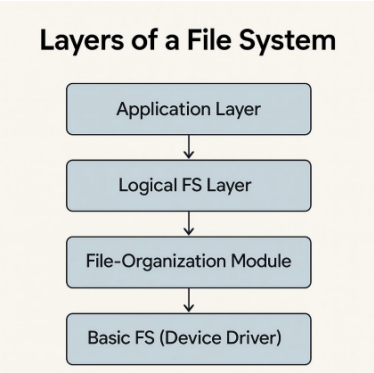
5 Goals of a File System

A well-designed file system must achieve several core goals to ensure usability and robustness:

- **Efficiency**: Enables fast read and write access to data, minimizing latency and maximizing throughput.

- **Reliability**: Ensures data integrity and supports recovery mechanisms in case of crashes, system errors, or power failures.
- **Scalability**: Capable of handling a wide range of file sizes and directory structures, from small files to large-scale datasets.
- **Security**: Enforces access control and permission schemes to protect data from unauthorized access or modification.

6 Hierarchical Organization: Layers of a File System

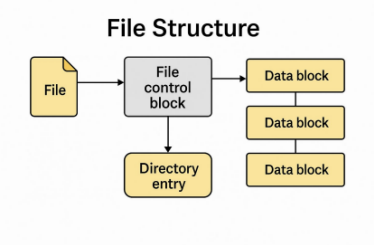


A file system is structured hierarchically into multiple layers to manage complexity and enable modular design. At the top is the Application Layer, where user programs interact with the file system via operations like opening, reading, and writing files. Beneath this is the Logical File System Layer, which handles file and directory metadata, naming, access control, and protection policies. The File-Organization Module maps logical file blocks to physical disk blocks, manages block allocation, and maintains free space information. At the lowest level is the Basic File System, often implemented as a device driver, which communicates directly with the disk hardware to perform raw data transfers between the device and memory.

7 File Elements and Structure

A file in a file system is composed of two primary components: **data** and **metadata**. The data stores the actual content of the file, while the metadata includes organizational and descriptive information such as file size, creation and modification timestamps, file ownership, and access permissions.

Files can be of several types, including *regular files*, *directories*, and *symbolic or hard links*. The operating system provides a standard set of file operations, such as `open`, `read`, `write`, and `delete`, which enable interaction with these file types.



Each file is managed via a data structure known as the **File Control Block**. It contains metadata and pointers to the file's associated **data blocks**, which reside on the disk. The file is also linked to a **directory entry**, which stores its name and provides a reference to the corresponding File Control Block, enabling efficient lookup and management within the file system hierarchy.

8 Directories

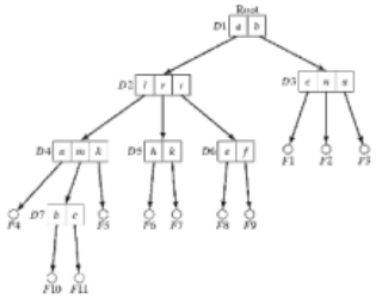


Figure 1: Tree

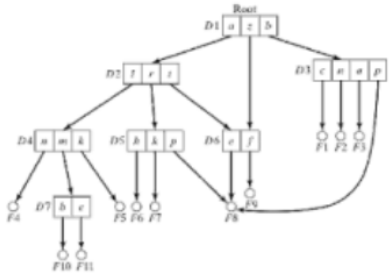


Figure 2: DAG

Directory structures can be **tree-structured** or **DAG-structured**. In a tree-structured directory, operations like search, insert, and delete are simple. However, sharing is asymmetric—each file has only one parent. In contrast, DAG-structured directories raise questions about the semantics of deletion. Any parent can remove a file, but only the last parent can completely remove it, requiring reference counting.

Common directory operations include: `create/delete`, `list` (with support for sorting, wildcards, recursion, and information display), and `change directory` (current, working, default, or home). Additional operations include: `move`, `rename`, `change protection`, `create/delete link` (symbolic), and `find/search routines`.

9 Representation of File on Disk

The operating system abstracts a file on disk as a virtual object for processes.

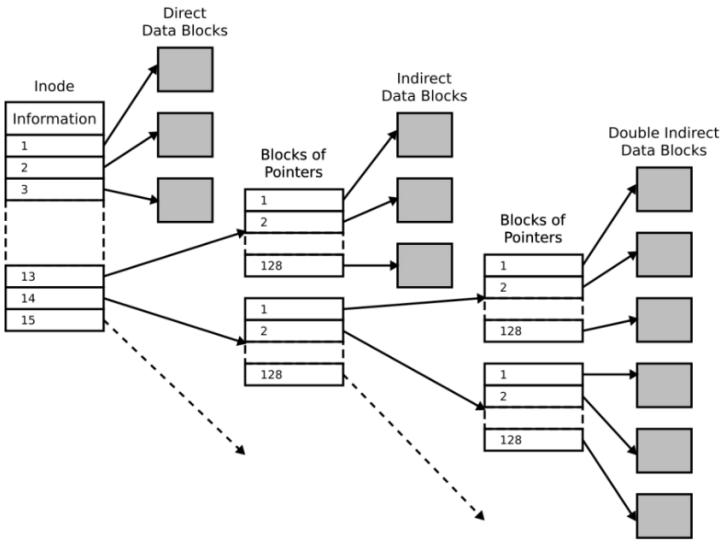
File Descriptor: An OS-assigned positive integer identifier that serves as a reference for a file's virtual object that a process can use.

- Values 0, 1, and 2 are reserved for `STDIN`, `STDOUT`, and `STDERR`.
- The file descriptor is an index into a kernel-maintained table of open files per process, which maps to an `iNode`.

iNode: A persistent data structure that contains:

- Owner ID
- File type
- Protection information
- Mapping to physical disk blocks (affects file size)
- Timestamps: creation time, last use, last modification
- Reference counter

10 Unix/Linux iNode to Disk Block Mapping



In UNIX, the iNode is a data structure that stores metadata about a file, including which disk blocks make up the file. The inode itself is of fixed size and composed of a fixed number of pages — for example, 16 pages. The first page typically stores general metadata, while the remaining pages store pointers to data blocks.

Initially, these pages contain direct disk block numbers. However, this direct mapping limits how large a file can be. To support larger files while keeping the inode size fixed, some pages are used for indirect addressing.

In single-level indirection, the inode page points to a page that contains disk block numbers. These are used only when the file grows large enough to require them. For small files, only the direct pointers are used.

If the file becomes even larger, double indirection is used. This allows the inode to support longer files without changing its fixed structure.

11 Filesystem Calls

Filesystem APIs (FS API) provide system calls to support key file and directory operations. These include:

- **CRUD Operations**: Create, read, update, and delete operations on files and directories.
- **Navigation**: Commands like `chdir` and `cwd` allow processes to change or query the current working directory.
- **Permissions**: Calls such as `chown` and `chmod` enable the modification of file ownership and access rights.
- **Protection**: The `fcntl` system call supports mechanisms like locking and unlocking files to manage concurrent access.

12 Files vs. Databases

12.1 Q: What is a database? How is it different from just a bunch of files?

A database may seem like just a collection of files, but it differs significantly through abstraction and structure.

12.2 Key Differences

- **Virtualization of Files** and **Binary Representation on Disk Storage** enable databases to manage data more efficiently.
- Databases provide support for:
  - Maintenance (e.g., inserting and deleting records)
  - Performance
  - Usability (querying)
  - Security and privacy

12.3 Database and Data Model

- A **database** is an *organized* collection of interrelated data.
- A **data model** is an abstract, formal framework to define how data is organized.
- Examples of data models include: Relations, XML, Matrices, and DataFrames.

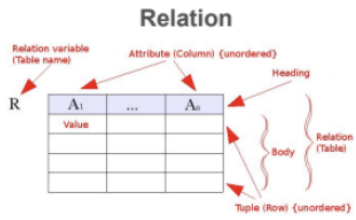
12.4 Files vs. Databases: Data Abstraction

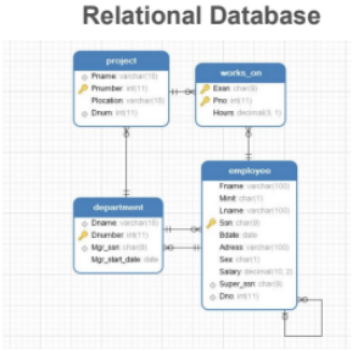
- Every database is an *abstraction* built on top of data files.
- **Logical level**: Data model for higher-level reasoning.
- **Physical level**: Describes how bytes are layered on top of files.
- All data systems (e.g., RDBMSs, Dask, Spark, TensorFlow) are application/platform software that use OS system calls to manage data files.

13 Data as File: Structured

Structured Data is a form of data with regular substructure.

13.1 Relation and Relational Databases





A **Relation** is a table with unordered rows (tuples) and columns (attributes). Each relation has:

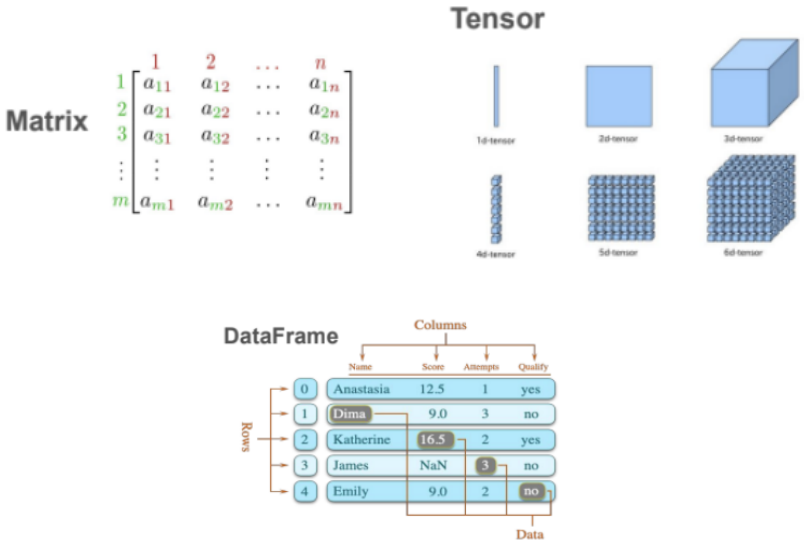
- A table name (relation variable)
- A heading (column names)
- A body (data values)

Relational databases are composed of related tables and store data in relations. Most RDBMSs and Spark serialize relations as **binary file(s)**, often compressed.

13.2 Matrix, Tensor, and DataFrame Representations

Other common structured formats include:

- **Matrix**: 2D numeric grid with fixed dimensions
- **Tensor**: Higher-dimensional generalization of a matrix (e.g., 3D, 4D, etc.)



- **DataFrame**: Labeled rows and columns, supports heterogeneous types and missing data

These formats are typically serialized as restricted ASCII text files (e.g., TSV, CSV) or as binary files.

14 Comparing Structured Data Models

Q: What is the difference between Relation, Matrix, and DataFrame?

- **Ordering**: Matrix and DataFrame have ordered rows and columns; Relation is unordered on both axes.
- **Schema Flexibility**:
  - Matrix: Only numbers, fixed schema
  - Relation: Requires predefined schema
  - DataFrame: No strict schema; mixed types and labels allowed
- **Transpose**: Supported by Matrix and DataFrame; not supported by Relation.

15 How Does a Relational Database Store Data?

Database Files: A relational database uses multiple physical files to store data. These include:

- **Data Files**: Store the actual data, such as tables and indexes.
- **Log Files**: Store transaction logs, which are used for data recovery and ensuring consistency.
- **Control Files**: Contain metadata about the database structure, including the location of data files.