Chapter 39: Files and Directories

1. Introduction

- Files and directories are the fundamental abstractions provided by operating systems for managing persistent data on storage devices.
- **File:** A file is a linear array of bytes, providing a simple and versatile storage abstraction. Each file is identified by a low-level name, typically an **inode number**.
- **Directory:** A directory is a special type of file that contains a collection of mappings between human-readable file names and their corresponding inode numbers. This structure allows for the organization of files in a hierarchical manner.

2. The File Abstraction

- The operating system abstracts the complexities of storage devices (like hard drives or SSDs) into a simple, uniform model of a file as a sequence of bytes.
- This abstraction allows applications to interact with files in a consistent way, regardless of the underlying hardware.
- Each file has associated **metadata**, which includes information such as its size, owner, permissions, and modification times.

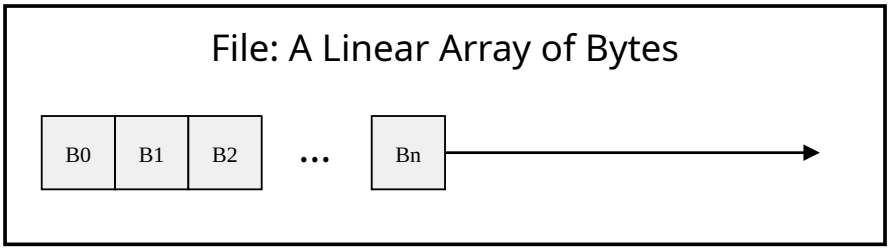
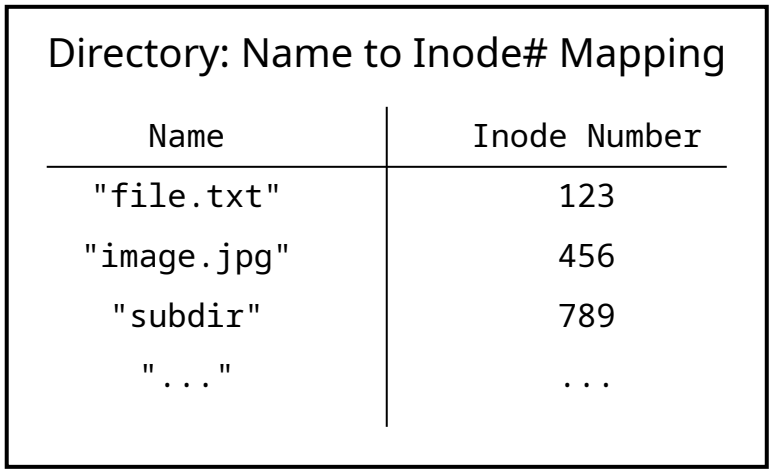


Figure 1: File Abstraction

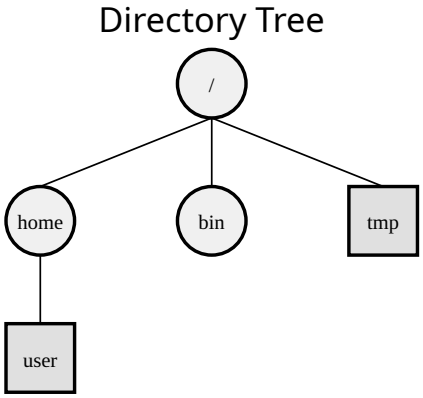
3. The Directory Abstraction

- Directories provide a user-friendly way to organize and locate files.
- They form a **directory tree**, a hierarchical structure that starts from a single **root directory** (denoted by `/`).
- Each directory entry maps a file name to an inode number, which allows the file system to locate the actual data on the disk.



4. The Directory Tree

- The directory tree provides a natural way to organize files and directories.
- It allows for the creation of complex and deeply nested structures, which can be used to model real-world organizational schemes.
- The tree structure also facilitates navigation and searching for files.



5. Core File System Calls

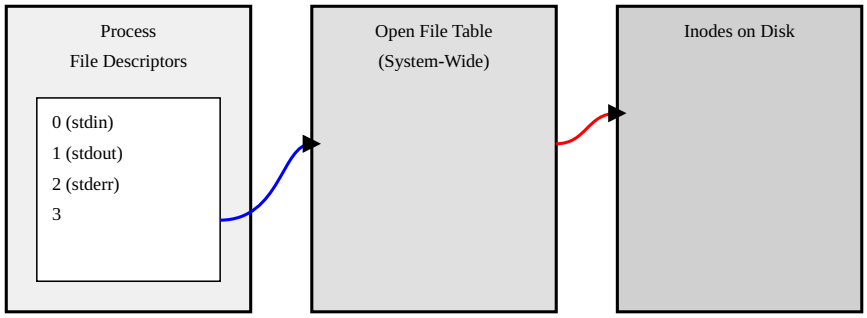
- **`open()`:** This system call is used to open or create a file. It returns a **file descriptor**, which is a small integer that is used to identify the file in subsequent operations.
- **`read()` and `write()`:** These system calls are used to read data from and write data to a file, respectively. They operate on the file sequentially, starting from the current file offset.
- **`lseek()`:** This system call allows a process to change the current file offset, enabling random access to the file's contents.
- **`fsync()`:** This system call ensures that any buffered data for a file is written to the underlying storage device, guaranteeing data persistence.
- **`rename()`:** This system call provides an atomic way to rename a file, which is crucial for maintaining file system consistency.
- **`stat()` and `fstat()`:** These system calls retrieve a file's metadata, such as its size, permissions, and timestamps.
- **`unlink()`:** This system call removes a link to a file. The file's data is only deleted when the number of links to it (the link count) drops to zero.

6. File Descriptors and the Open File Table

- When a process opens a file, the operating system creates an entry in a system-wide **open file table**. This table stores information about all currently open files, such as their current offset and status.
- The process receives a **file descriptor**, which is an index into its own per-process file descriptor table. This table, in turn, points to the corresponding

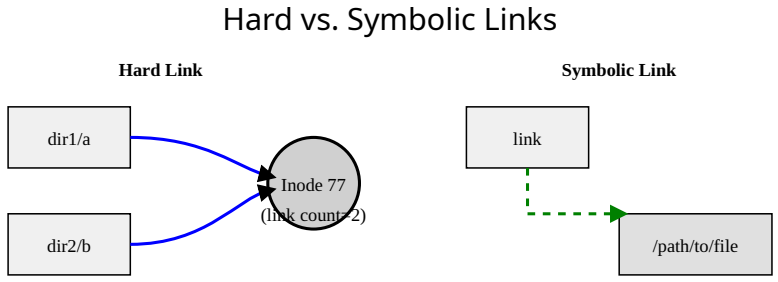
entry in the open file table.
- This mechanism allows for efficient sharing of open files between related processes (e.g., parent and child processes created via `fork()`).

File Descriptors and Open File Table



7. Hard Links and Symbolic Links

- **Hard Link:**
  - A hard link is a directory entry that associates a name with a file. Multiple hard links can exist for the same file.
  - All hard links to a file are indistinguishable from one another.
  - The file's inode stores a **link count**, which is the number of hard links pointing to it.
  - The file is only removed from the file system when its link count becomes zero.
- **Symbolic Link (Soft Link):**
  - A symbolic link is a special type of file that contains a path to another file or directory.
  - It is a pointer to a file name, not an inode.
  - Symbolic links can span across different file systems.
  - If the target file is deleted, the symbolic link becomes a **dangling pointer**.


Hard vs. Symbolic Links
Hard Link
dir1/a
dir2/b
Inode 77
(link count: 2)
Symbolic Link
link
/path/to/file