# CS415 Module 8 Part B - Files and File Systems

Athens State University

**Outline**

## Contents



## 1 Universal Model of Input and Output

**The Bit Stream Model of Input and Output**


**Fig. 17.1 | Data hierarchy.**

- Data is passed to and from the OS as stream of bits
- Applications apply some form of logical organization to this stream

At the lowest level, all forms of data are managed as a stream of bits. For convenience reasons, this bit stream is managed in terms of 8-bit bytes. Non-binary data is encoded into a binary form (for example, representing characters as ASCII and strings as collections of characters). But this encoding is an application-level process rather than an operating system requirement.

### The Universality of Input and Output

- A major innovation introduced by the UNIX operating system was the concept of *universality of input and output*
  - This means that the same four system calls: `open()`, `read()`, `write()`, and `close()` are used to perform I/O on all types of files
- As result, applications can use a single interface to access many different device types

Consider a command like the Linux `cp` command. As this command uses the Universal I/O system calls, it can work with many different file types:

```
cp test test.old   # Copy a regular file
cp a.txt /dev/tty  # Copy a regular file to a terminal
cp a.txt /dev/lp   # Copy a regular file to the system default terminal
cp /dev/tty b.txt  # Copy from this terminal to a file
cp /dev/pts/16 /dev/tty # Copy input from another terminal
```

This implies that each file server and device driver must implement the same set of I/O calls. The Linux programming interface also provides an API for cases where we need to do something to a device that isn't covered by the system calls outside the Universal I/O model.

## 2 The Universal I/O Model

### The Four System Calls In The Universal I/O Model

`open()` Open an existing file or create and open a new file

`read()` Read a number of bytes from an open file referred to by an integer file descriptor

`write()` Write a number of bytes from an open file referred to by an integer file descriptor

`close()` Close an open file indicated by an integer file descriptor

The Universal I/O model propagates up through the application programming interface. The C++ Streams objects is built on top of these system calls.

### Operations Outside the Universal I/O Model

- The `ioctl()` system call is a general-purpose system call for doing the things fall outside the Universal I/O model
- How one uses the the `ioctl()` system calls varies from device to device
- One should refer the appropriate documentation for details

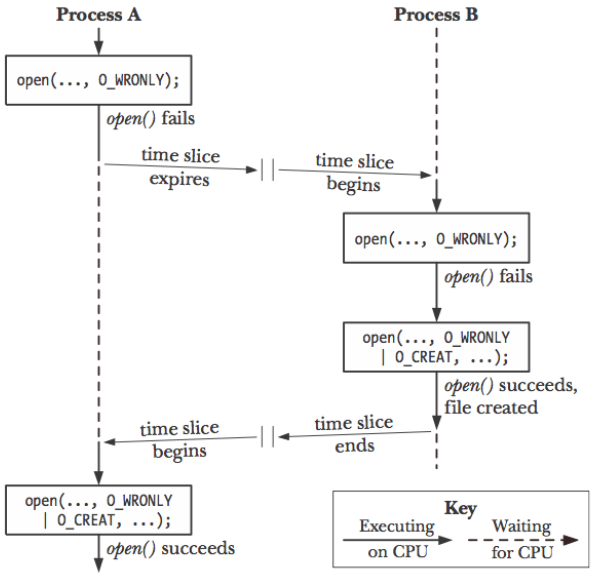
### Open Files

- The operating system must track several pieces of data to manage open files
  - **Open-file table**: tracks open files
  - **File pointer**: Pointer to the last read/write location, maintained per process that has the file open
  - **File-open count**: Counts the number of times a file is open, which is used to determine when to remove a file from the open-file table
  - **Access rights**: Per-process data that indicates what each process is allowed to do to this file

### Locking

- Provided by some operating systems and file systems
  - Similar to reader-writer locks for threads
  - **Shared lock**: like a reader lock - several processes can acquire concurrently
  - **Exclusive lock**: like a writer lock, single process holds the lock
- Controls who has access to a file
- Two types:
  - **Mandatory**: access denied depending on locks held and requested
  - **Advisory**: processes can find status of locks and decide what to do

### The Need For File Locking




- Consider what happens if a file tries to open a non-existent file and is then swapped by the dispatcher
- In this case, both processes would think they have created the file

### Metadata: Data About Data

- Maintained by the operating system
  - *UNIX*: no direct support, user conventions
  - *Windows*: Embedded in the file name (extension)
  - *macOS*: Current file systems based on UNIX, with attempt to preserve the classic macOS approach of using resource forks
    - Separate file with the metadata for a file, presented to applications as a bundle

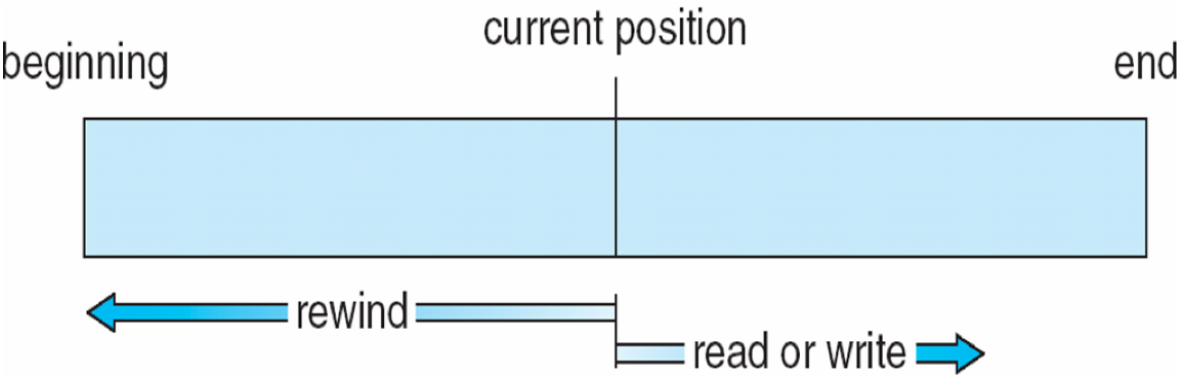
## 2.1 File Structures

### File Structures

- None: sequence of words or bytes
- Simple record structure: Lines, fixed length, variable length
- Complex structures: Formatted document (MS-Office documents), program executable

Note that this is almost always an application decision. Well, application or system support library (example: the .NET Framework).

### Sequential Access




- Three basic access methods: `read next`, `write next`, `reset`

### Direct Access

- **Direct access file**: a file organized into fixed length records indexed by a relative block number
- Access methods:
  - `read record` $n$
  - `write record` $n$
  - `position to record` $n$
  - `read next`
  - `write next`
  - `rewrite record` $n$

### Other Access Methods

- Can be built on top of the base methods
- In most cases, build some form of index for searching the file
  - Remember that lovely data structure: 2-3 trees? Indexes are stored in some form of balanced-tree data structure
  - Index is much smaller, so can be kept in memory
  - Do index of indexes for really large data

- Classic Case Study: IBM's Indexed Sequential-Access Method (ISAM)
  - Small master index, points to disk blocks of secondary index
  - File kept sorted
  - All done by the Operating System (VMS or zOS, in this case)

# 3 File Systems

### Disk Structure

- Disks (regardless of physical medium) are divided into **partitions**
- Redundancy schemes such as **RAID** protect the hardware against failure
- Disks and partitions can be used **raw** - without a file system - or **formatted** with a file system
- An entity containing a file system is known as a **volume**
- Each volume tracks the file system's information in a **device directory** or **Volume Table of Contents** (VTOC)
- As well as **general purpose file systems**, there are **special purpose file systems** that can be found on disk

Examples of a special-purpose file system: database managers such as Oracle or SQL Server define specialized database file systems optimized for database activities while network file systems such as NFS, NetBIOS, and others provide a mapping between the Universal I/O model and the network.

### File Systems Supported by Linux

- The traditional Linux `ext2` file systems
- The native UNIX files systems: System V `ufs`, Minix, and BSD file systems
- MSFT FAT, FAT32, and NTFS
- Apple's HFS and HFS+ file systems
- the ISO 9660 CD-ROM file system
- Journaling file systems such as `ext3`, `ext4`, `zfs`, and `Btfs`

### File-System Structure

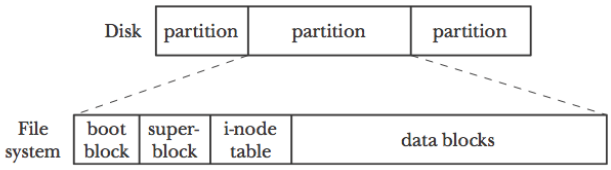



**Figure 14-1:** Layout of disk partitions and a file system

The boot block will always be the first block in a file system. This block contains the program and data required to boot the system. The OS will only use one boot block, but all file systems will have a boot block.

The superblock stores information about the file system. This includes the size of the i-node table, size of logical blocks in the file system, and the size of the file system in terms of logical blocks.

The *i-node table* contains a unique entry for each file or directory on the system. This entry records various information about the file

The data blocks in a file system has the content of the files and directories.

**The I-node Table**

- There is an unique i-node for each file in the file-system
- The i-node stores:
  - File-type, Owner, Group, Access permissions, Timestamps: last access, last modification, and last status change, number of hard links to the file, size of the file in bytes, number of data blocks, and pointers to the data blocks

Maintenance and care of the i-node table is one of the major tasks the OS system performs related to working with files. In early versions of UNIX, this information was cached in memory and was often corrupted. So, a utility called `fsck` was provided to check and fix any corruptions in a file system's i-node table. This problem has become less pronounced with emergence of journaling files systems such as the `ext4` file system. However, you will see `fsck` run on a timed basis in the Linux boot-process.

# 4 Directories

**Folder Or Directory**

- All of this ugly GUI stuff we use today means that we talk a lot about "folders" on our desktop
- In fact, these are "directories" in our operating system
- And build a tree structure whose nodes contain information about files
- Both the directory structure and the files they refer to reside on the disk

**Operations Performed On A Directory**

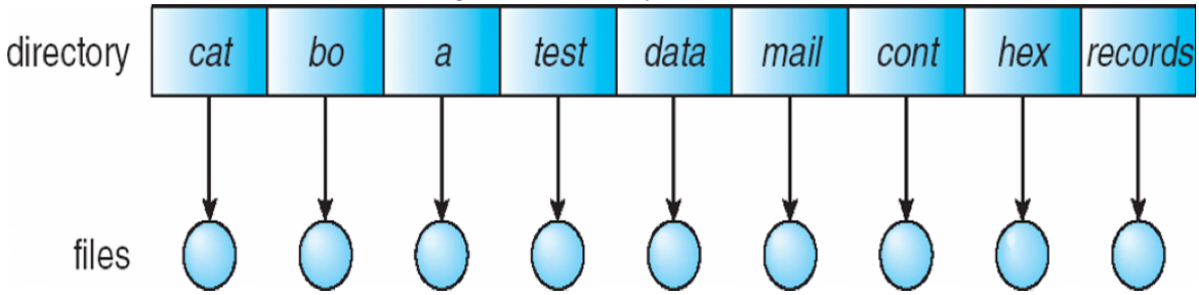
- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

**Design Requirements for a Directory System**

- Efficiency: Need to quickly locate a file
- Naming - convenient to users
  - Two users can have the same name for different files
  - The same file can have several different users
- Grouping - Need to logically group files by properties

**Single-level directory**

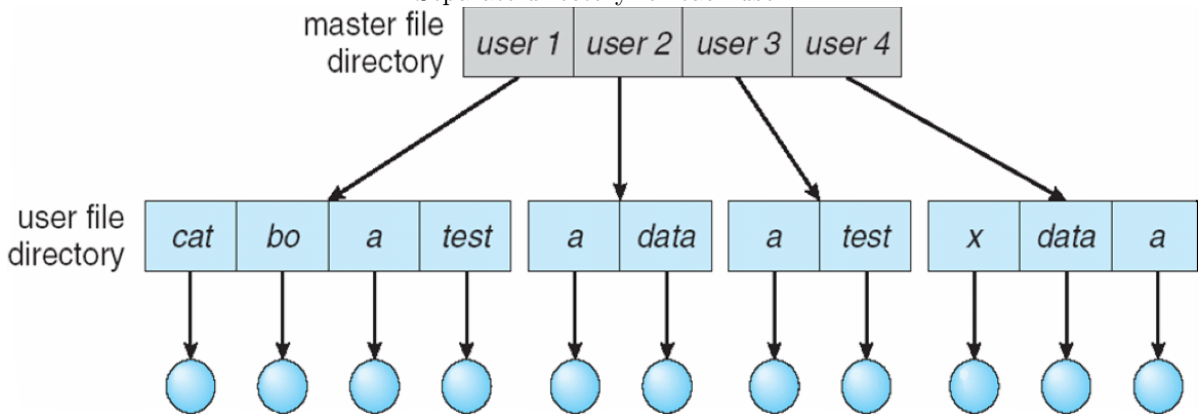
A single level directory for all users:




Two major problems: naming and grouping!

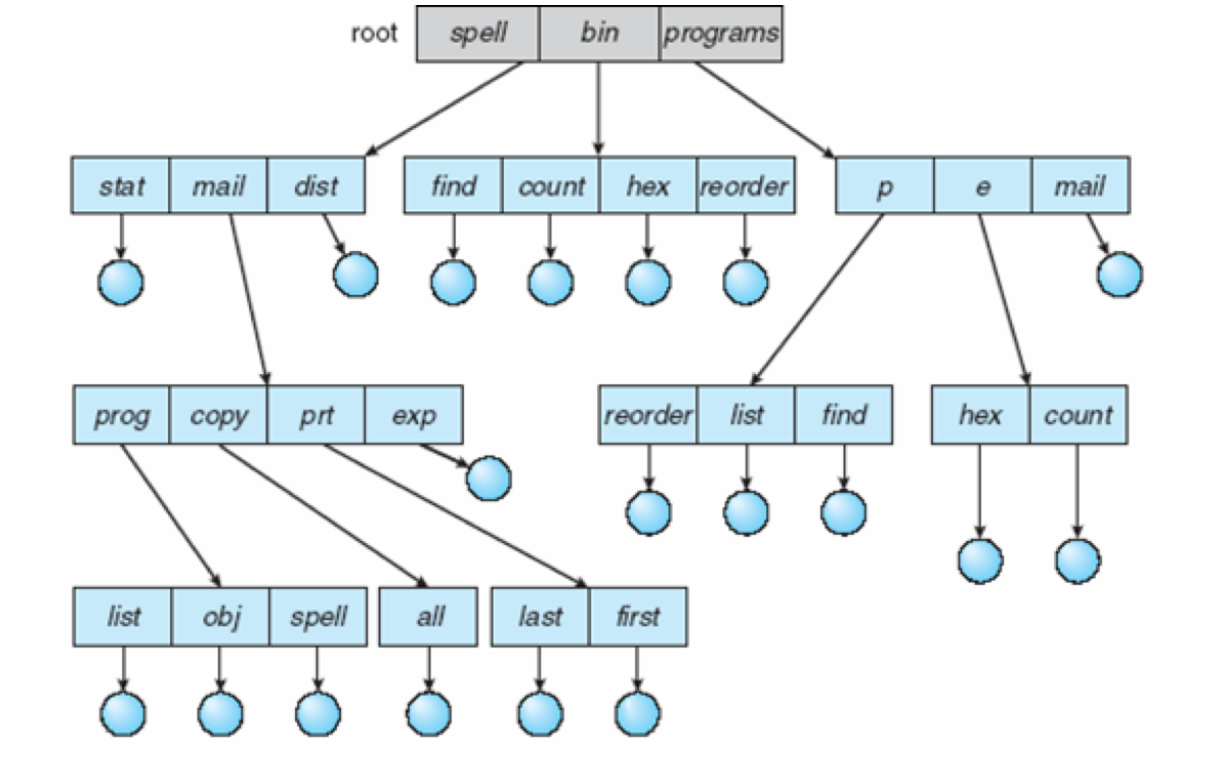
**Two-level directory**

Separate directory for each user




Issues: paths, efficient searching, no ability to group

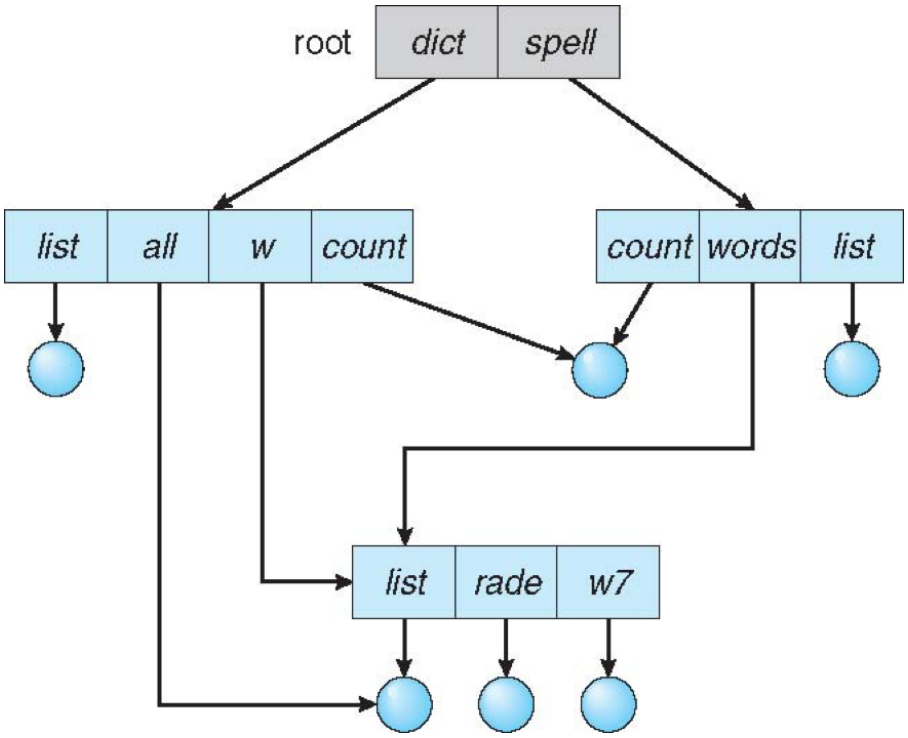
**Tree-Structured Directories**




Tree-structure directories provide efficient searching and grouping capabilities. We do need to introduce the concept of "current directory" or "working directory" to denote where in the tree-structure a user finds files.

This allows to specify file locations in either **absolute path names** or **relative path names**. We also should note that deleting a directory means we that we delete the entire sub-tree rooted by that directory.

**Acyclic-Graph Directories**




This structure introduces the opportunity for aliasing: two different names for the same file. In UNIX-speak, this is called 'linking' and the alias is called a *link*. One must *resolve the link* by following the pointer to locate the file.

One must also avoid dangling pointers in the directory structure. Most often this is implemented using backpointers so we can follow the backpointers to find and remove all references to the physical file.

Note that most UNIX file systems restrict this type of link (known as hard-links) to only within a file system. Later UNIX-en such as Linux introduced the concept of a "soft-link" which is a true alias.

## 4.1 Tree or Forest: File System Mounting

**File System Mounting**

- File systems must be **mounted** into the directory structure
- This defines where in the "directory-tree" the root of the file system is located
- Exactly how and where varies by operating system

**Tree vs. Forest**

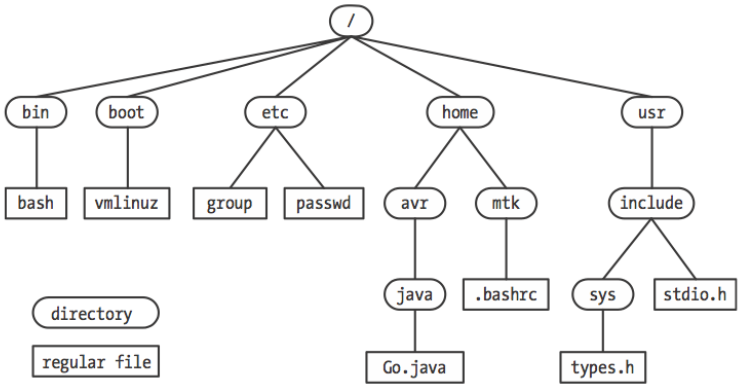



**Figure 2-1:** Subset of the Linux single directory hierarchy

- Linux follows the UNIX convention of a single directory hierarchy, rooted by a special directory named "/" and called "root" or the "root directory

For many years, the Windows operating system took the "forest" approach where each mounted volume (disk drive) had its own complete directory hierarchy. This changed with Windows NT and later and there is a similar hierarchy to what one has with Linux. That hierarchy is mostly hidden from the user by the GUI.

The conventions of where to mount new file systems varies from operating system to operating system. This is complicated by the programs that allow to "automount" volumes. For example, the macOS operating system uses the BSD-variant of the UNIX hierarchy tree and will mount disk volumes on the directory named `/Volumes`.