


# Chapter 11:

# File-System Interface



# Chapter 11: File-System Interface

---

- File Concept
- Access Methods
- Disk and Directory Structure
- File-System Mounting
- File Sharing
- Protection

# Objectives

---

- To explain the function of file systems
- To describe the interfaces to file systems
- To discuss file-system design tradeoffs, including
  - access methods,
  - file sharing,
  - directory structures
- To explore file-system protection

# File Concept

---

- Computers can store information on various storage media, such as:
  - SSDs
  - magnetic disks,
  - magnetic tapes, and
  - optical disks.
- These storage devices are usually **nonvolatile**
  - so the contents are persistent between system reboots
- OS provides a **uniform logical view of stored information**
- OS **abstracts** from the physical properties of its storage devices to define a **logical** storage unit, **the file**.
- Files are mapped by the OS onto physical devices
- A file is a **named collection** of related information that is recorded on secondary storage.
- From a user's perspective, a file is the smallest allotment of logical secondary storage;
  - that is, data cannot be written to secondary storage unless they are within a file.

# File Concept

---

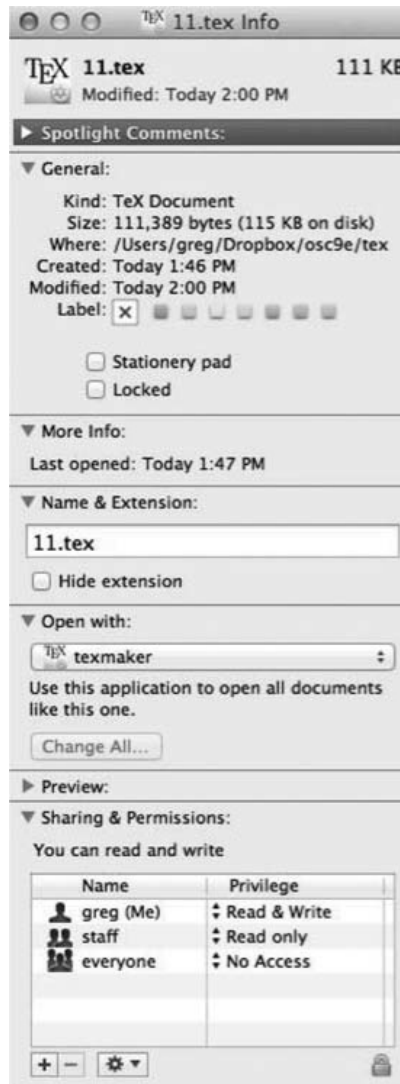
- File types: Commonly, files represent
  - Programs
    - ▶ Both source and object forms
  - Data
    - ▶ Numeric, character, binary
  
- Contents defined by file's creator
  - Many types: **text file, source file, executable file, etc**

# File Attributes

---

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the [directory structure](#), which is maintained on the disk

# File info Window on Mac OS X



# File Operations

---

- File is an **abstract data type**.
  - To define a file properly, we need to consider the operations that can be performed on files.
- **Create**
- **Write** – at **write pointer** location
- **Read** – at **read pointer** location
- **Reposition within file - seek**
- **Delete**
- **Truncate**

# File Operations: Opening a File

---

- Most of the file operations mentioned involve [searching the directory](#) for the entry associated with the named file.
- To [avoid this constant searching](#), many systems require that an [open\(\)](#) system call be made before a file is first used.
  
- ***Open( $F_i$ )***
  - search the directory structure on disk for entry  $F_i$ , and
  - move the content of entry to memory
- ***Close ( $F_i$ )***
  - move the content of entry  $F_i$  in memory to directory structure on disk

# Open Files

---

- Several pieces of data are needed to manage open files:
- **Open-file table:**
  - Tracks open files
  - When a file operation is requested, the file is specified via an index into this table,
    - ▶ so no searching is required
- **File pointer:** pointer to last read/write location,
  - per process that has the file open
- **File-open count:** counter of number of times a file is open
  - to allow removal of data from open-file table when last processes closes it
- **Disk location of the file:** cache of data access information
  - The information (needed to locate the file on disk) is kept in memory so that the system does not have to read it from disk for each operation.
- **Access rights:** per-process access mode information

# File Types – Name, Extension

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

- One can specify an extension to indicated the file type
  - but an extension is not required in general
- An extension may serve as a **hint** to the OS on what to do with the file
  - e.g., on double clicking it

# File Structure

---

- File types also can be used to indicate the **internal structure** of the file.
- Source and object files have structures that **match the expectations of the programs** that read them.
- Further, certain files must conform to a required structure that is understood by the operating system.
  - E.g., OS requires that an executable file have a specific structure
    - ▶ so that it can determine where in memory to load the file and
    - ▶ what the location of the first instruction is.

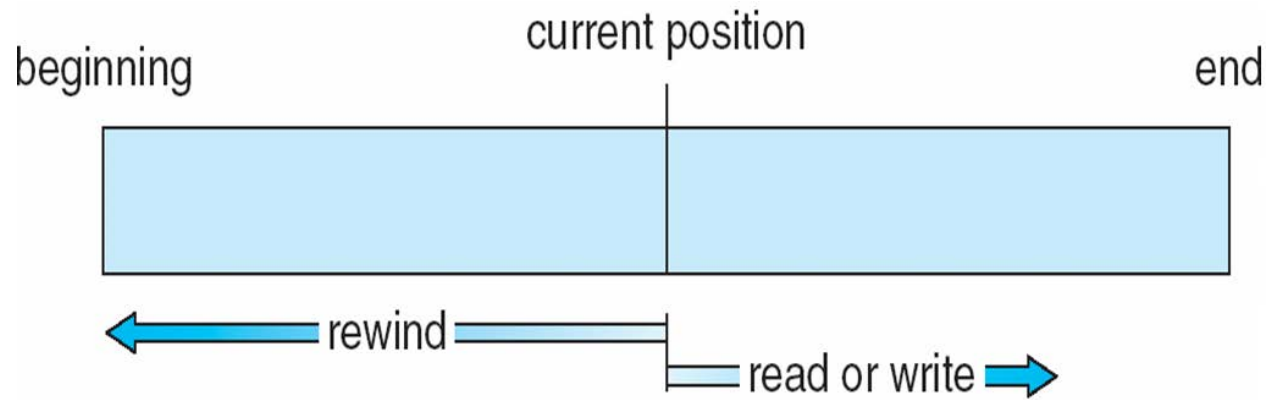
# File Structure

---

- In general, file structure can be:
  - **None**
    - ▶ Sequence of words, bytes
  - **Simple record structure**
    - ▶ Lines
    - ▶ Fixed length
    - ▶ Variable length
  - **Complex Structures**
    - ▶ Formatted document
    - ▶ Relocatable load file
  - Can simulate last two with first method by inserting appropriate control characters
  - Who decides:
    - ▶ Operating system
    - ▶ Program

# Sequential-access File

---



# Access Methods

---

## ■ Sequential Access

- Simplest access method
- Based on a **tape model** of a file
  - ▶ works on both sequential-access devices and random-access ones.
- Information in the file is processed **in order**,
  - ▶ one record after the other.
- Most common method
  - ▶ E.g., editors and compilers usually access files in this fashion.
- **read next**
  - ▶ reads the next portion of the file and
  - ▶ automatically advances a file pointer, which tracks the I/O location
- **write next**
  - ▶ appends to the end of the file and advances to the end of the newly written material (the new end of file)
- **Reset**
  - ▶ a file can be reset to the beginning, and
  - ▶ on some systems, skip forward/backward n records

# Access Methods

---

- **Direct Access (= relative access).**
  - A file is made up of **fixed-length logical records**
  - That allow programs to read and write records rapidly in no particular order
  - Based on a **disk model** of a file,
    - ▶ Since disks allow random access to any file block
  - The file is viewed as a **numbered sequence** of blocks or records.
    - ▶ We may read block 14, then read block 53, and then write block 7.
    - ▶ **No restrictions on the order** of reading or writing for a direct-access file
- $n =$  **relative block number** (relative -- to the beginning of the file)
  - File starts with block 0, then block 1, 2, ...
  - Relative block numbers allow OS to decide where file should be placed
- **Read( $n$ )** – read block number  $n$
- **Write( $n$ )** – read block number  $n$
- **position to  $n$** 
  - **read next**
  - **write next**

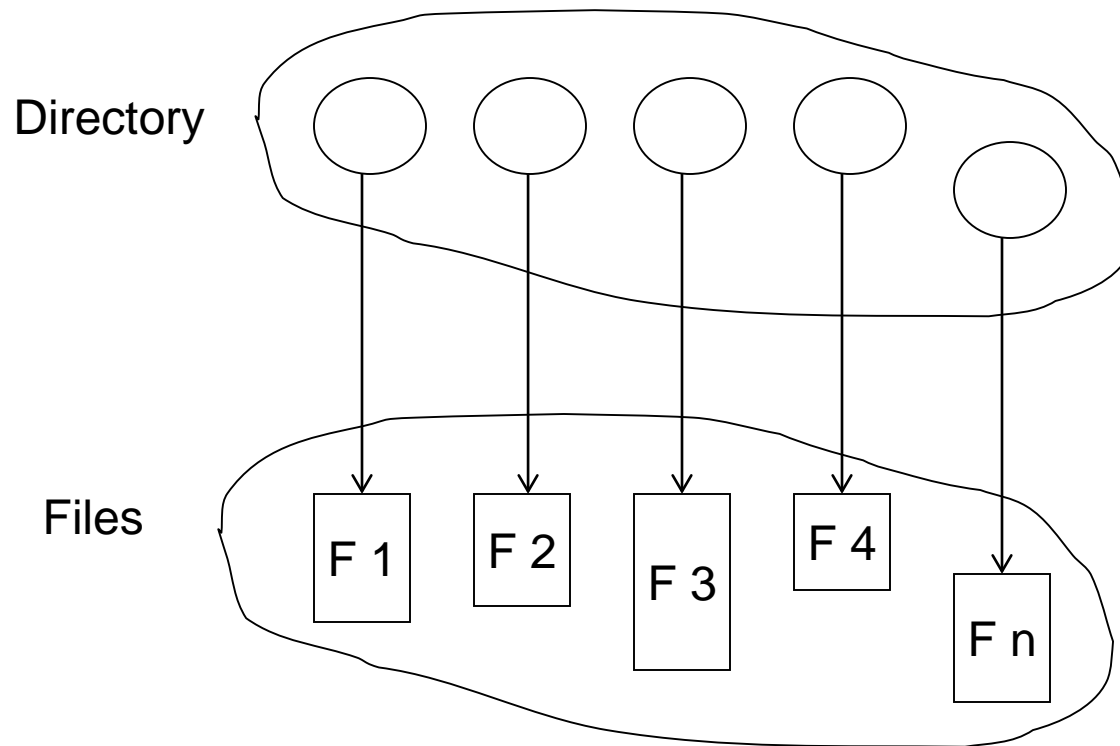
# Directory Structure

---

- Number of files on a system can be extensive
  - ▶ Break file systems into **partitions**
    - treated as a separate storage device
  - ▶ Hold information about files within partitions.
- **Device Directory**
  - ▶ A collection of nodes containing information about all files on a partition.
- Both the directory structure and files reside on disk

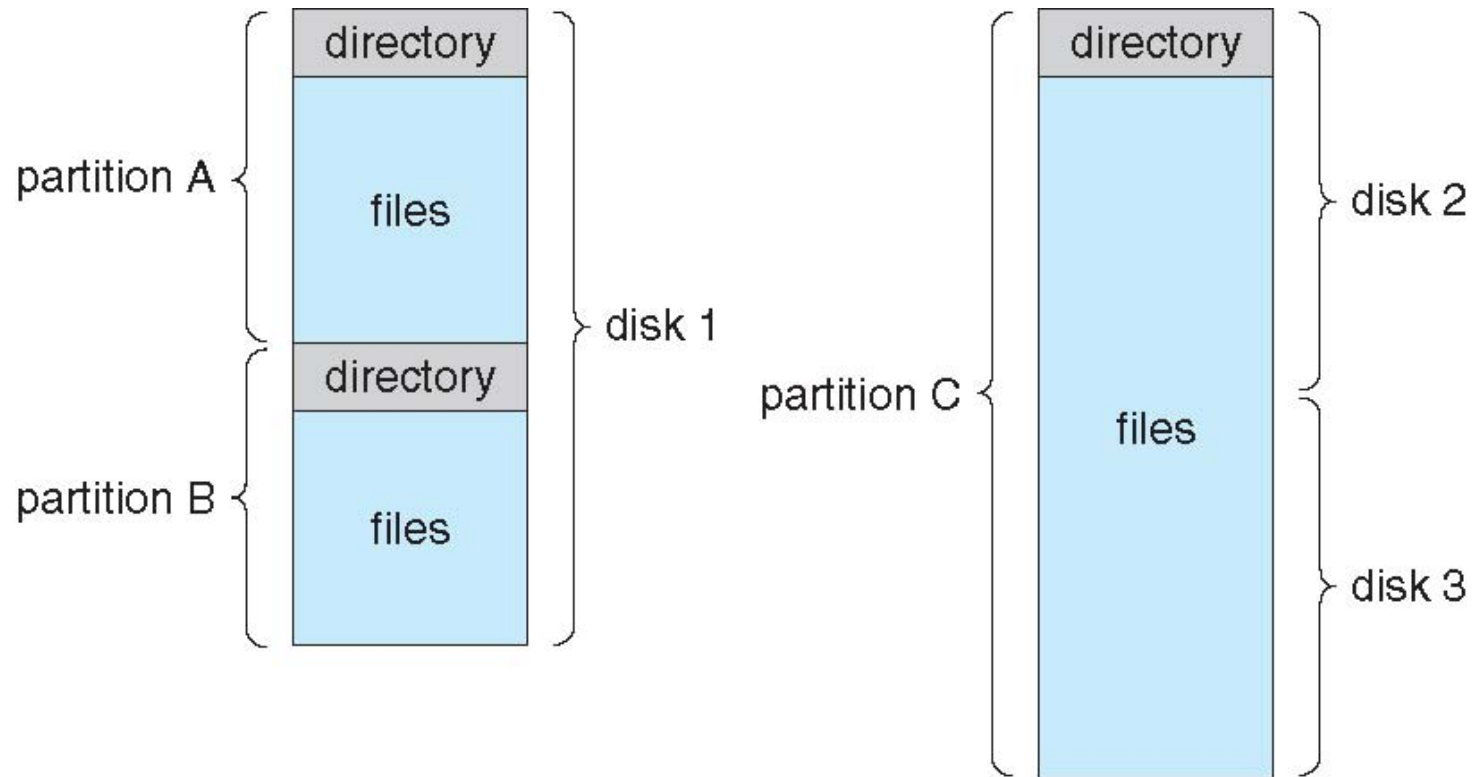
# Directory Structure

- A collection of nodes containing information about all files



# A Typical File-system Organization

---



# Information in a Device Directory

---

- File Name
- File Type
- Address or Location
- Current Length
- Maximum Length
- Date
  - ▶ created,
  - ▶ last accessed (for archival),
  - ▶ last updated (for dump)
- Owner ID,
- Protection information

# Operations Performed on Directory

---

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

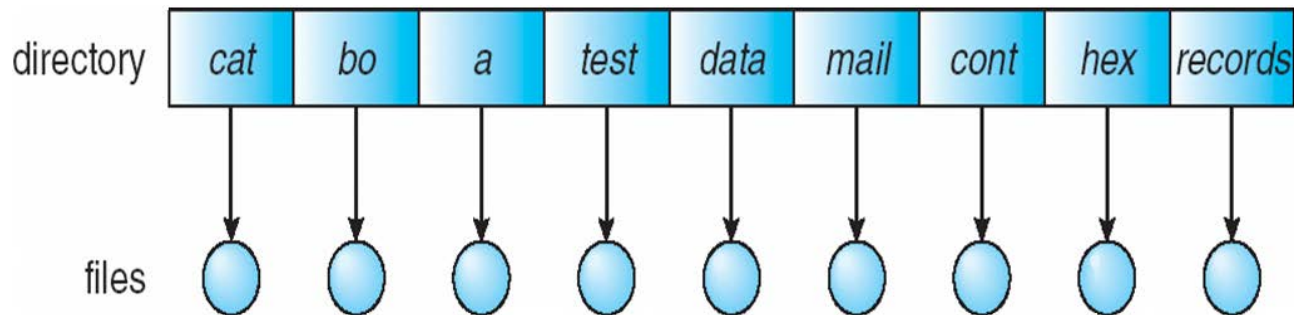
# Goals of Logical Directory Organization

---

- Which factors to consider in deciding the logical directory organization?
- **Efficiency** – locating a file quickly
- **Naming** – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- **Grouping**
  - Logical grouping of files by properties, (e.g., all Java programs, all games, ...)
- ... Let us now consider the most common schemes for defining the logical structure of a directory

# Single-Level Directory

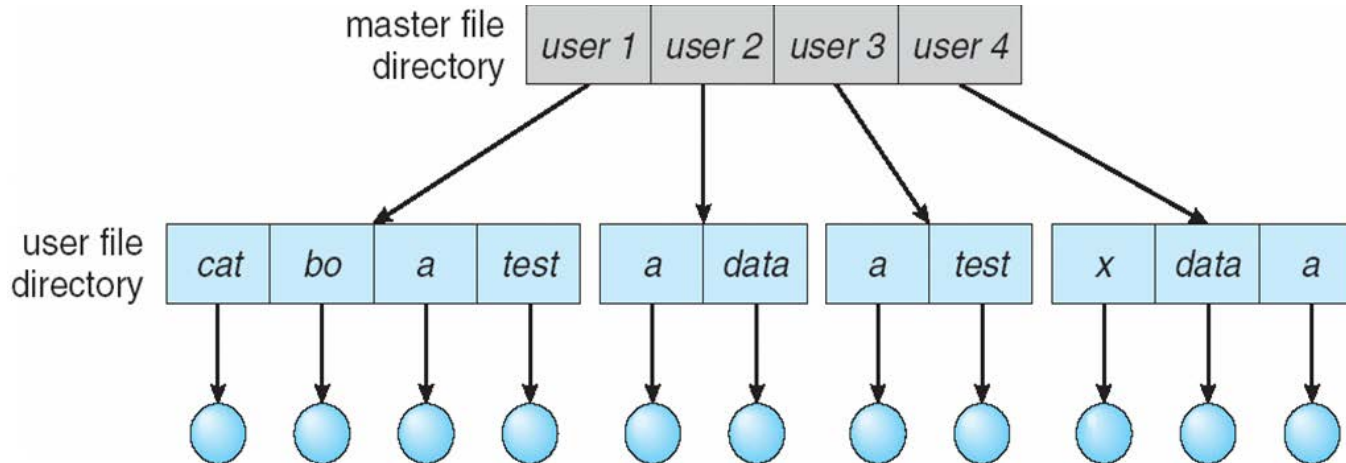
- Let us start with the simplest one: A single-level directory for all users



- All files are contained in the same directory
- **Pros:** Easy to support and understand
- **Cons:** Naming problem and Grouping problem
  - As the number of files increases, difficult to remember unique names
  - As the number of users increase, users must use unique names for files

# Two-Level Directory

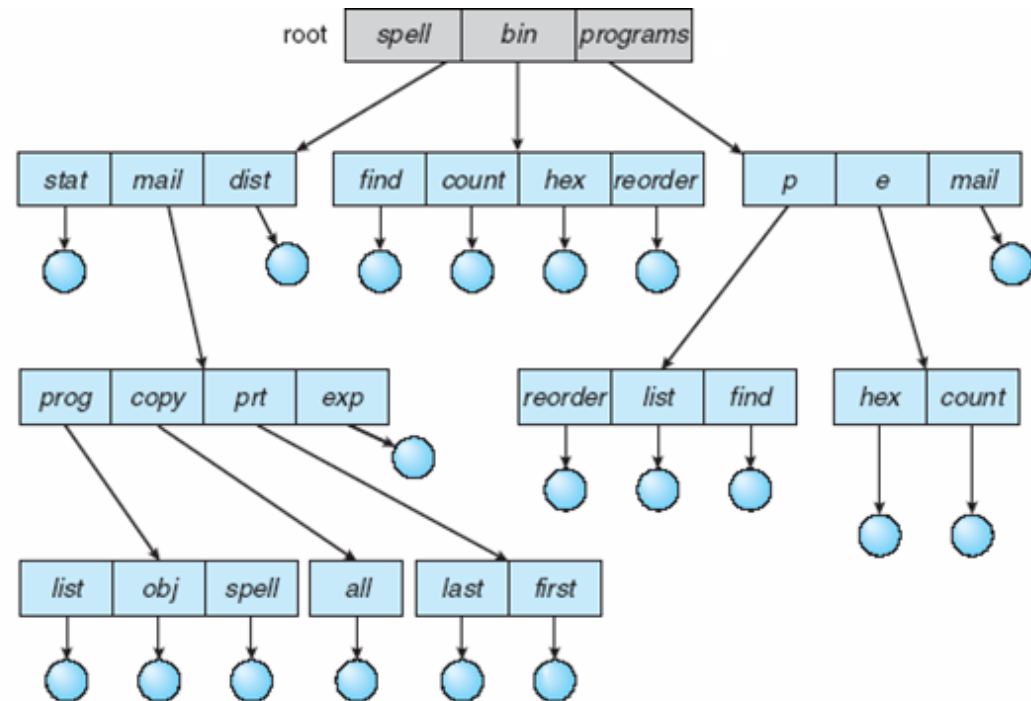
- Introduced to remove naming problem among users
- A separate directory for each user



- 1st level -- contains list of user directories
- 2nd level -- contains user files
- System files kept in separate directory or Level 1.
- Need to specify **Path name**
- **Pros:**
  - Can have the same file name for different user
  - Efficient searching
- **Cons:** No grouping capability

# Tree Structured Directories

- Arbitrary depth of directories
  - leaf nodes are files
  - interior nodes are directories.
- Efficient Searching
- Grouping Capability
- **Current Directory**
  - =working directory
  - `cd /spell/mail/prog, cd ..`
  - `dir, ls`
- MS-DOS uses a tree structured directory



# Tree-Structured Directories (Cont)

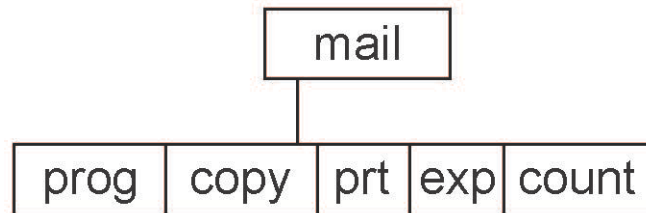
- **Absolute** or **relative** path name
  - Absolute from root
  - Relative paths from current working directory pointer.
- Creating a new file is done in current directory
- Delete a file

```
rm <file-name>
```

- Creating a new subdirectory is done in current directory

```
mkdir <dir-name>
```

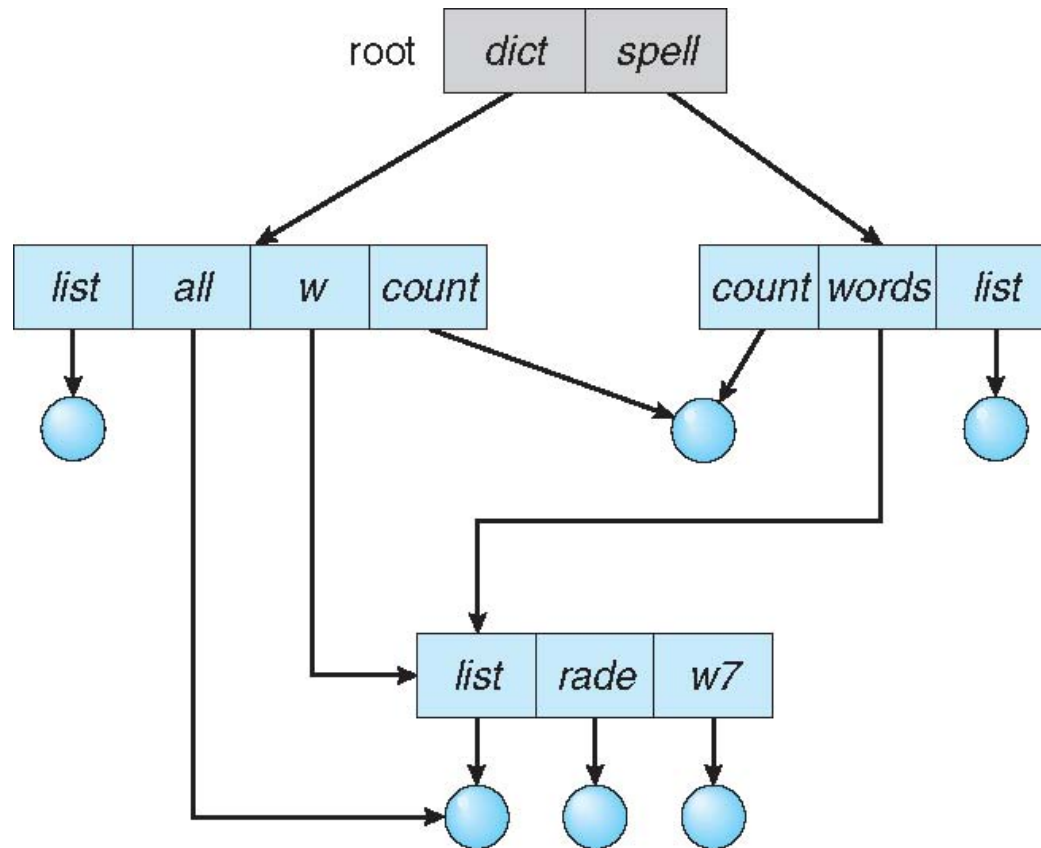
Example: if in current directory `/mail`



Deleting “mail”  $\Rightarrow$  deleting the **entire subtree** rooted by “mail”

# Acyclic-Graph Directories

- Have shared subdirectories and files



# Acyclic Graph Directories

---

- Generalization of the tree-structured directory scheme
- Acyclic graphs **allow sharing**
  - A shared directory (or file) exists in the file system in two (or more) places at once.
  - Convenient for several reasons, e.g.:
    - ▶ 2 programmers working on the same project might want to share a subdirectory
    - ▶ both want the subdirectory to be in their own directories

# Acyclic Graph Directories

---

- Sharing can be implemented differently
- Implementation via **soft links (=symbolic links)**
  - Links are pointers to other files or subdirectories
    - ▶ May be implemented as an absolute or a relative path name
    - ▶ We **resolve** the link by using that path name to locate the real file
    - ▶ A link can point to a file/subfolder on a totally different volume
- Implementation via **hard links**
  - Duplicate all information about shared files in the corresponding **directory entries**
  - Original and copy are **indistinguishable**
  - Need to maintain consistency if one of them is modified.
  - A link cannot point to a different volume

# Comparison of Hard Link and Symbolic Link

---

Item	Hard Link	Symbolic Link
Name resolution	Faster. A hard link contains a direct reference to the object.	Slower. A symbolic link contains a path name to the object, which must be resolved to find the object.
Object existence	Required. An object must exist in order to create a hard link to it.	Optional. A symbolic link can be created when the object it refers to does not exist.
Object deletion	Restricted. All hard links to an object must be unlinked (removed) to delete the object.	Unrestricted. An object can be deleted even if there are symbolic links referring to it.
Dynamic objects (attributes change)	Slower. Many of the attributes of an object are stored in each hard link. Changes to a dynamic object, therefore, are slower as the number of hard links to the object increases.	Faster. Changes to a dynamic object are not affected by symbolic links.
Static objects (attributes do not change)	Faster. For a static object, name resolution is the primary performance concern. Name resolution is faster when hard links are used.	Slower. Name resolution is slower when symbolic links are used.
Scope	Restricted. Hard links cannot cross file systems.	Unrestricted. Symbolic links can cross file systems.

# Acyclic Graph Directories

---

## ■ Naming

- A single file may have multiple absolute path names
- Two different names for the same file

## ■ Traversal

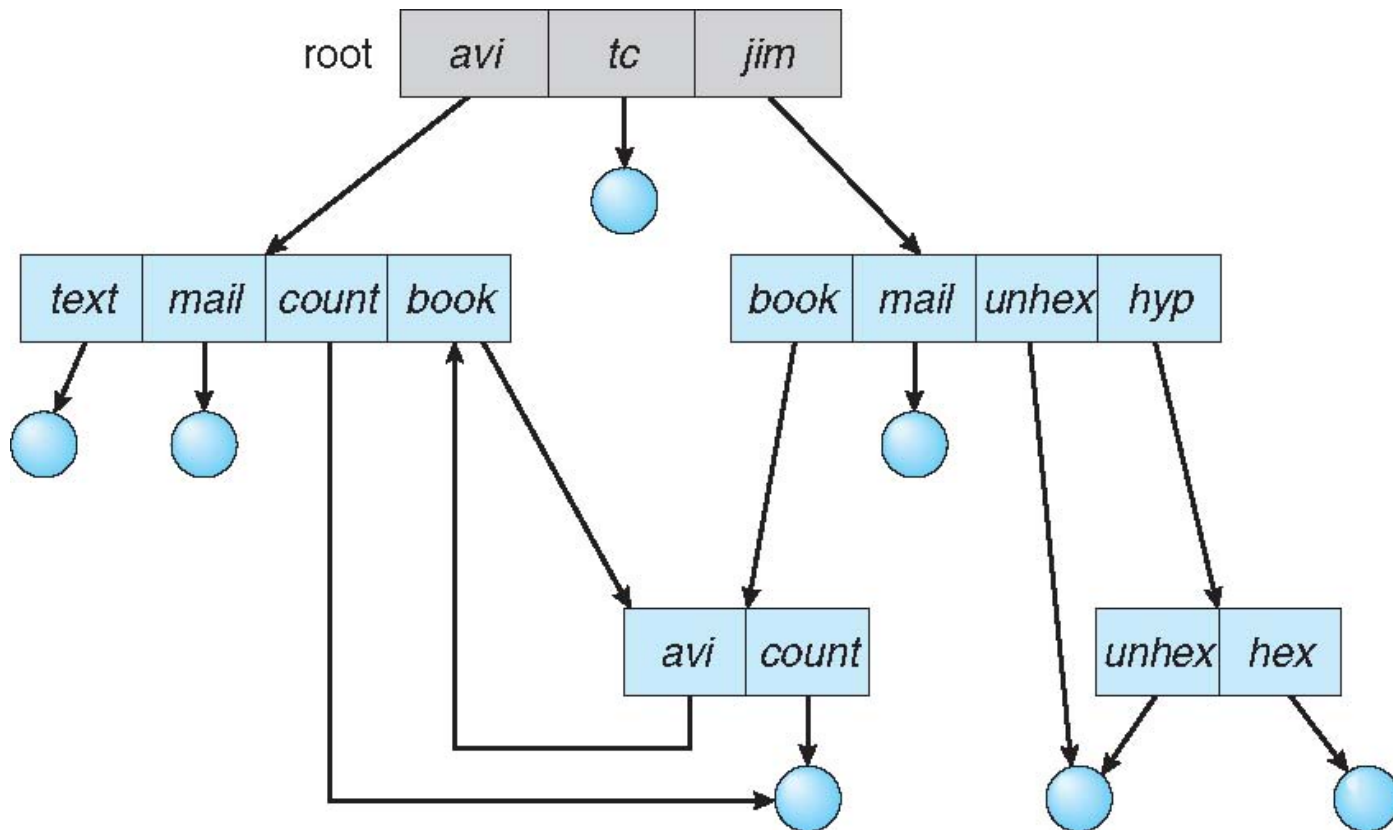
- Ensure that shared data structures are traversed only once.
  - ▶ For efficiency

## ■ Deletion

- Removing file when someone deletes it may leave dangling pointers.
- Preserve file until all references to it are deleted
  - ▶ Keep a **list of all references** to a file or
  - ▶ Keep a **count** of the number of references
    - Called *reference count*
    - When count = 0, file can be deleted

# General Graph Directory

- A problem with an acyclic-graph is ensuring that there are no cycles.
- When we add links
  - the tree structure is destroyed
  - resulting in a simple graph structure (**general graph directory**)



# General Graph Directory

---

- The primary advantages of an **acyclic graph** is the simplicity of the algorithms
  1. For traversing the graph and **(no infinite loops)**
  2. For determining when there are no more references to a file.
  
- How do we guarantee no cycles?
  - Allow only links to file; not subdirectories
    - ▶ Limiting
  - Every time a new link is added, use a cycle detection algorithm to determine whether it is OK
    - ▶ **Computationally expensive**
    - ▶ High cost for disks specifically

# General Graph Directory

---

- How do we deal with [file deletion](#) in general graph directory?
  - A [garbage collection](#) scheme is needed, as even when file is not used its reference count can be above zero due to self-referencing or cycles
- Garbage collection involves:
  - **Pass 1:** Traversing the entire file system, marking everything that can be accessed.
  - **Pass 2:** Collecting everything that is not marked onto [a list of free space](#).
- A similar marking procedure can be used to ensure that a traversal or search will cover everything in the file system once and only once.
- Garbage collection for a disk-based file system, however, is
  - [extremely time consuming](#), and
  - is thus seldom attempted.
- How do we ensure an effective traversal in a general graph directory then?
  - One solution is to [ignore links in such a traversal](#)
  - Cycles are avoided, and no extra overhead is incurred.

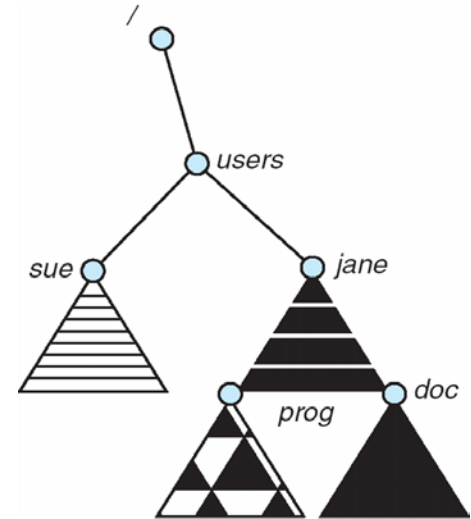
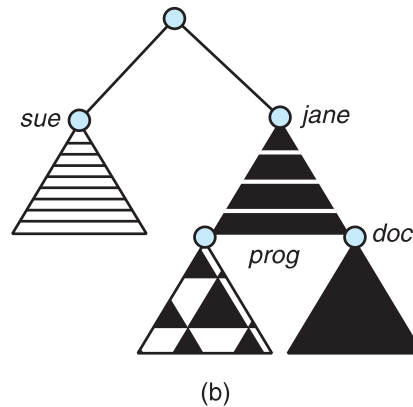
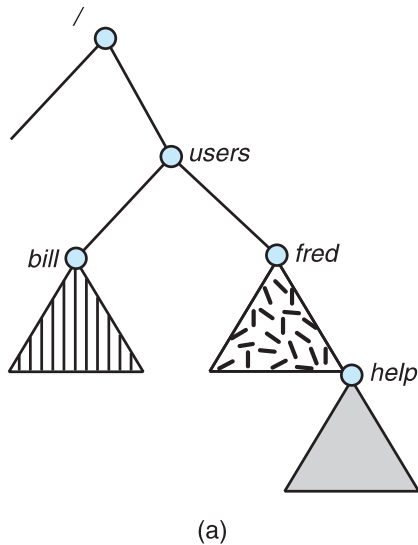
# File System Mounting

---

- File System must be **mounted** before it can be available to process on the system
- The OS is given
  - the **name of the device** and
  - the **mount point** -- location within file structure at which files attach
- OS verifies that the device contains a valid file system.
- OS notes in its directory structure that a file system is mounted at the specified mount point.

# File System Mounting

- A unmounted file system (i.e., Fig. (b)) is mounted at a **mount point**



# Protection

---

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

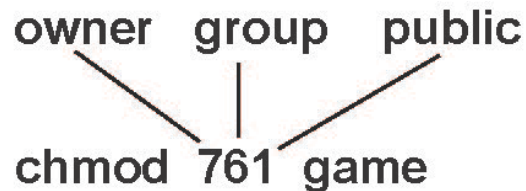
# Access lists and groups

---

- Associate each file/directory with **an access list**
  - Problem - length of access list...
- Solution - condensed version of list
  - Mode of access: read, write, execute
  - Three **classes** of users:
    1. **owner access** - user who created the file
    2. **groups access** - set of users who are sharing the file and need similar access
    3. **public access** - all other users
  - In UNIX
    - ▶ 3 fields (of length 3 bits) are used.
    - ▶ Fields are: user, group, others(u,g,o),
    - ▶ Bits are: read, write, execute (r,w,x).
    - ▶ E.g. `chmod 761 game`

# Access Lists and Groups (contd.)

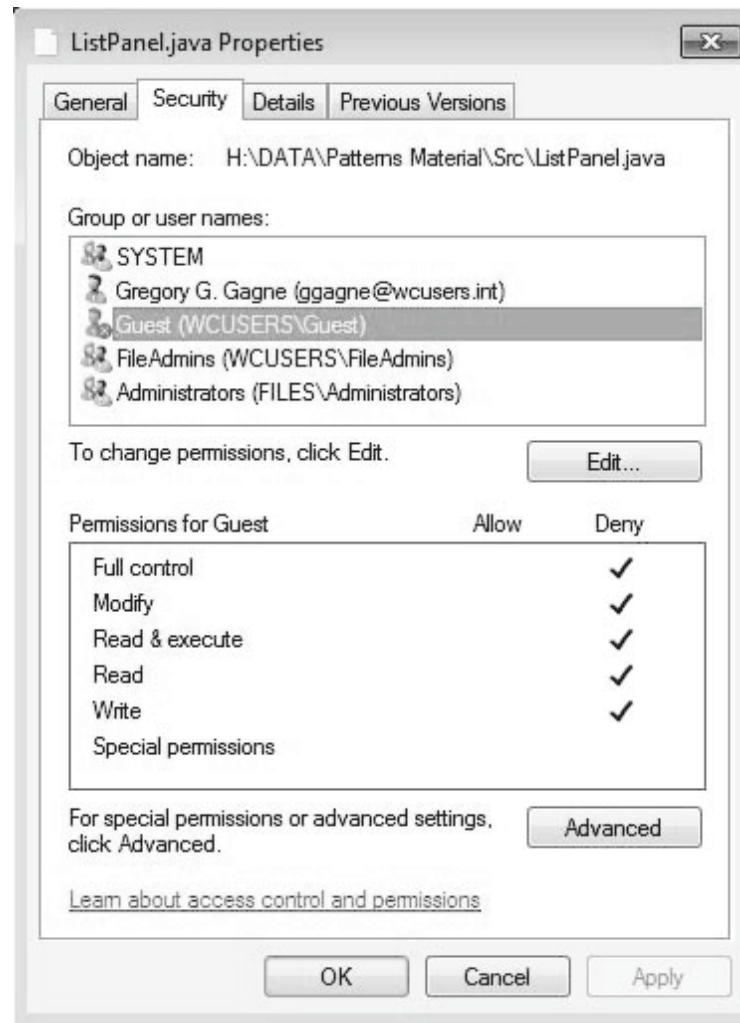
- Ask manager to create a group (unique name), say G,
  - and add some users to the group.
  - `chgrp G game`
- For a particular file (say *game*) or subdirectory, define an appropriate access.



- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

			RWX
a) <b>owner access</b>	7	⇒	1 1 1
			RWX
b) <b>group access</b>	6	⇒	1 1 0
			RWX
c) <b>public access</b>	1	⇒	0 0 1

# Windows 7 Access-Control List Management



# End of Chapter 11

