Chap 12: File Systems

BY PRATYUSA MUKHERJEE, ASSISTANT PROFESSOR (I)

KIIT DEEMED TO BE UNIVERSITY

File Concept

- Computers store information on various storage media - magnetic tapes, magnetic disks and optical disks.
- To ensure convenience of use, OS provides a uniform logical view of stored informtion.
- **Thus the OS abstracts from the physical properties of its storage devices to define a logical storage unit called file.**
- These files are mapped to the physical devices by the OS.
- These storage devices are usually non-volatile so the contents are persistent between system boots.

What is a File?

- It is a named collection of related information that is recorded on a secondary storage.
- From a user's perspective, **a file is the smallest allotment of logical secondary storage**. So we can say that data cannot be written onto secondary storage unless they are within a file.
- Files represent **programs** (both source and object forms) **and data**.
- Data files can be - numeric, alphabetic, alphanumeric or binary.
- Files can be free form such as text files or rigidly formatted.
- **It is nothing but a sequence of bits, bytes, lines or records whose meaning is defined by it's creator and user**.
- Hence the concept of file is avidly general.

Contents of a File

- The information in a file is defined by its creator.
- Files can be source or executable programs, numeric or text data, photos, audio, video etc.
- Every file has a particular structure depending upon its type.
- **Text File**: sequence of characters organized into lines and then into pages.
- **Source File**: sequence of functions, each of which is further organized as declarations followed by executable statements.
- **Executable File**: series of code sections that the loader can bring into memory and execute.

File Name

- A file is named for convenience of use by humans.
- A name is usually a string of characters followed by an extension. Some systems differentiate between upper and lower case in file names whereas some do not.
- When a file is named, it becomes independent of the process, user and even the system that created it. So files can be created by one person and edited by someone else by specifying its name.
- The file’s owner can write a file created by him to a USB disk, send it as an email attachment or copy it across a network but whatever he does the file name is the same unless it is changed.

File Attributes

- **Name** – The only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system. It is in non-human-readable form
- **Type** – needed for systems that support different types
- **Location** – pointer to a device and to the file location on that device
- **Size** – current file size (or possibly the max allowed size)
- **Protection** – acess controls information about who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring. It keeps a track of time of creation, last modification and use.
- Many variations have come up including extended file attributes such as file checksum or character encoding

Where is information about files kept ? How is it accessed ?

File Operations

A file is an abstract data type and in order to perform various operations on files, the OS invokes system calls.

- **Creating a file**: First make sure there is space in the file system for the new file. Second make an entry for the new file in the directory.
- **Writing a file**: To write a file, a system call is made specifying both the file name and its contents. Given the filename, system searches the directory to find its location and uses a **writer pointer** to the location in the file where the next write is to take place. As soon as a write occurs, the writer pointer is updated.
- **Reading a file**: To read from a file, a system call specifies the file name and where in the memory the next block of the file should we put. Given the filename, system searches the directory to find it and uses a **read pointer** to the location in the file where the next read is to take place. As soon as a read occurs, the read pointer is updated.

All processes mostly read or write to a file, thus the **current operation location is kept as a per process current file position pointer**. Both read and write can hence use the same pointer, saving space and reducing system complexity.

- **Repositioning within a File (Seek)** : The directory is searched with the file name and the current file position pointer is repositioned to the new value. This operation need not involve any actual I/O.

- **Deleting a File**: To delete a file, the directory is searched with the file name. After finding it, we release all the file space so that it can be reused by other files and then the erase the directory entry.

- **Truncating a File**: User may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the entire file and recreate it, this operation allows all attributes to remain unchanged except for its length. It lets the file be reset to length zero and release the file space.

- **Other common operations** : append new info to the end of an existing file, rename an existing file, create a copy of the file, copy the file to an I/O device, create a new file and then read from an old one and write into this etc.

Open (Fi) & Close (Fi) & Open-file Table

- **Open(Fi)** – search the directory structure on disk for entry Fi, and move the content of entry to memory
- **Close (Fi)** – move the content of entry Fi in memory to directory structure on disk.
- All file operation involve searching the directory for the appropriate file name.
- To avoid this constant searching, many systems use open( ) system call before a file is first used.
- The OS keeps a **open-file table** containing information about all files.
- When a file operation is requested, the file is specified via an index to this table and hence no searching is needed.
- When the file is no longer used, it is closed by the process and the OS removes its entry from the open-file table.
- **create( ) and delete( ) system calls work with closed files rather than open files.**

What other facilities does open( ) provide? How to implement open( ) and close( ) in multiprocess emvironment?

Open Count

- **OS maintains a 2 level tables: per process table and system wide table where the first one points to the second one.**
- Per process table tracks all files that a particular process opens. system wide table contains process independent information.
- Once a file has been opened by a process, the system wide table includes an entry for the file.
- **When another process executes an open ( ), a new entry is simply added to the process's open file table pointing to the appropriate entry in the system wide file.**
- Hence the open file table also has a parameter **open count** associated with each file to indicate how many processes have that file open.
- **Each close( ) decreases this open count and when the open count value reaches zero, the file is no longer in use and its entry is removed from the open file table.**

Information associated with Open File

- **Open-file table**: tracks open files
- **File pointer**: pointer to last read/write location as a current-file-position-pointer, per process that has the file open. It is unique to each process operating on the file and hence is kept separate from file attributes.
- **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 It helps to reuse open-file table entries
- **Disk location of the file**: cache of data access information so that info needed to locate the file on the disk is kept in memory and the system does not have to read it from the disk for each operation.
- **Access rights**: per-process access mode information so that OS can allow or deny subsequent I/O requests

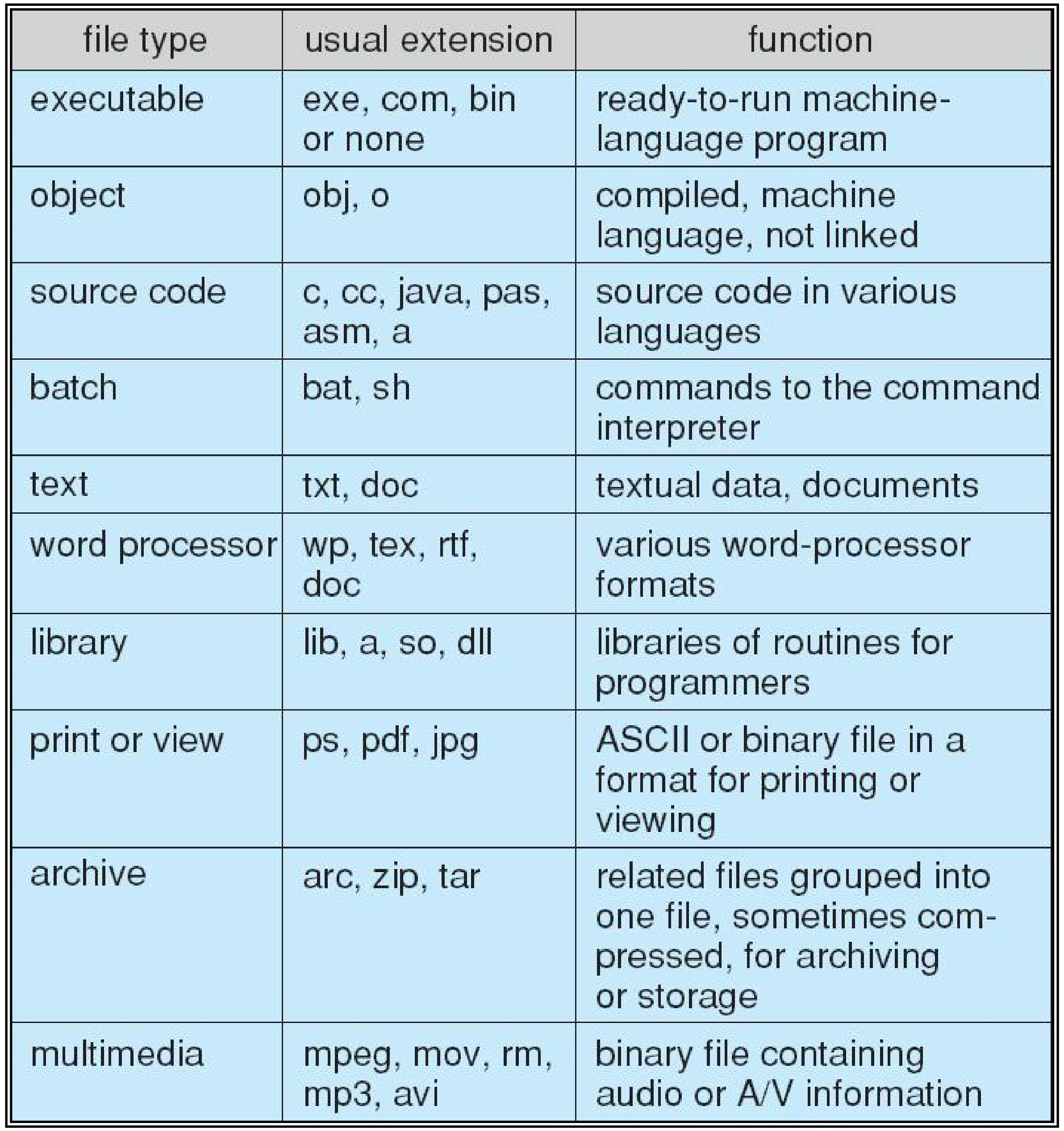
Open File Locking

- File locks allow one process to lock a file and prevent other processes from gaining access to it.
- Thus it is helpful for files that are shared by several processes.
- In a **shared lock or reader lock**, several processes can acquire the lock concurrently.
- In an **exclusive lock or writer lock**, only one process can acquire it at a time.
- Furthermore, OS may provide either mandatory or advisory file locking mechanism.
- **If a lock is mandatory**, then once a process acquires an exclusive lock, OS will prevent any other process from accessing the locked file. Thus OS ensures locking integrity. Eg: Windows OS.
- **If a lock is advisory**, then OS will not prevent other process from accessing the locked file. Thus here it is upto the software developers to ensure that locks appropriately acquired and released. Eg: UNIX OS.
- It is necessary to take care that if using mandatory locking, hold exclusive lock only while accessing a file in order to **prevent deadlock**

File Types

- If an OS recognizes the type of a file, it can operate on the file in reasonable ways.
- Mostly, the file type is included as a part of the file name - a name and an extension separated by a period.
- The type of a file further indicates the operation that can be done on it.
- Extensions are not always required. In that case, the application will look for a file with the given name and the extension it expects.

What are magic numbers?



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 com-pressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Structure

- File types also indicate the internal structure of the file.
- Each OS extends a system supported file structures with set of operations for manipulating files with those structures.
- But, just because the OS supports multiple file structures, the resulting size of the OS is cumbersome as it has to contain the codes to support these structures.
- Some OS impose a minimal number of file structures.
- However, all OS must support at least one structure - an executable file - so that system is able to load and run programs.

Internal File Structures

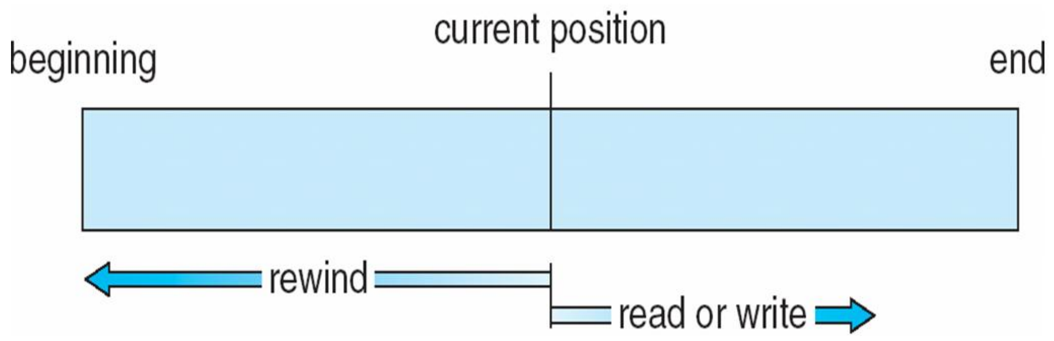
- For an OS, it is very difficult to locate an offset within a file
- Disk systems have a well defined block size determined by the size of the sector. Thus all disk I/O is performed in units of one block of physical record and all blocks are of same size.
- However, physical block size will never exactly match to the length of the desired logical record as logical records vary in length.
- So **the easiest solution is to pack a number of logical records into physical blocks.**
- The logical record size, physical block size and packing technique determines how many logical records are in each physical block. (**Who does this??**)
- **As disk space is always allocated in blocks, some portion of the last block of each file is generally wasted in order to keep everything in units of blocks**. (**Do you remember what is this??**)
- **The larger the blocksize, the larger will be the wastage**.

Access Methods

- Information in the file can be accessed in several ways.
- Some systems provide only one access method while others support many methods.
- It is thus important to choose the right method for a particular application
- The various methods are:
  - ☐Sequential Access
  - ☐Direct Access
  - ☐Other Methods

Sequential Access

- **Info. in a file is processed in order, one record after the other.**
- It is the most common and simplest method.
- **Read_next( )** : reads the next portion of a 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.
- **Such files can be reset / rewinded to the beginning and also skipped forward or backward for n records (mostly n = 1)**
- Eg: tape models



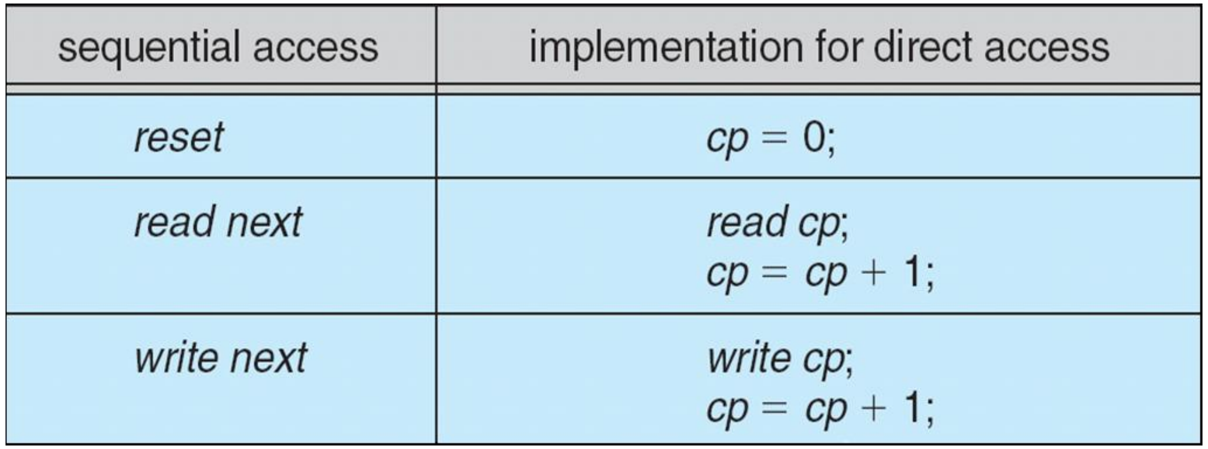
Direct Access or Relative Access

- **Here a file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order.**
- Thus it is based on a disk model of a file as disks allow random access to any file block
- So, the file is viewed as a numbered sequence of blocks or records and there is no restriction on the order of reading or writing.
- It is helpful for immediate access to large amounts of information (**How??**)
- Here, the file operations must be modified to include the block number as a parameter. **So instead of read_next( ) we have read (n) and instead of write_next( ) we have write (n)**. (**What is this n??**)
- Also to continue with previous sequential operations, we can add an operation **position_file (n)**. So first we do position_file(n) and then read_next ( ) or write_next ( ).

- The block number provided by the user to the OS is normally a **relative block number** even if the absolute disk address are different.
- Thus it is an index relative to the beginning of the line and it starts from 0 or 1.
- **The use of relative block numbers allow to OS to decide where the file should be placed thus solving the allocation problem and preventing the user from accessing portions of file system that is not a part of his file**.

How does a system satisfy a request for record N in a file?

- **We can easily simulate sequential access on a direct access file by simple keeping a variable cp that defines the current position**.



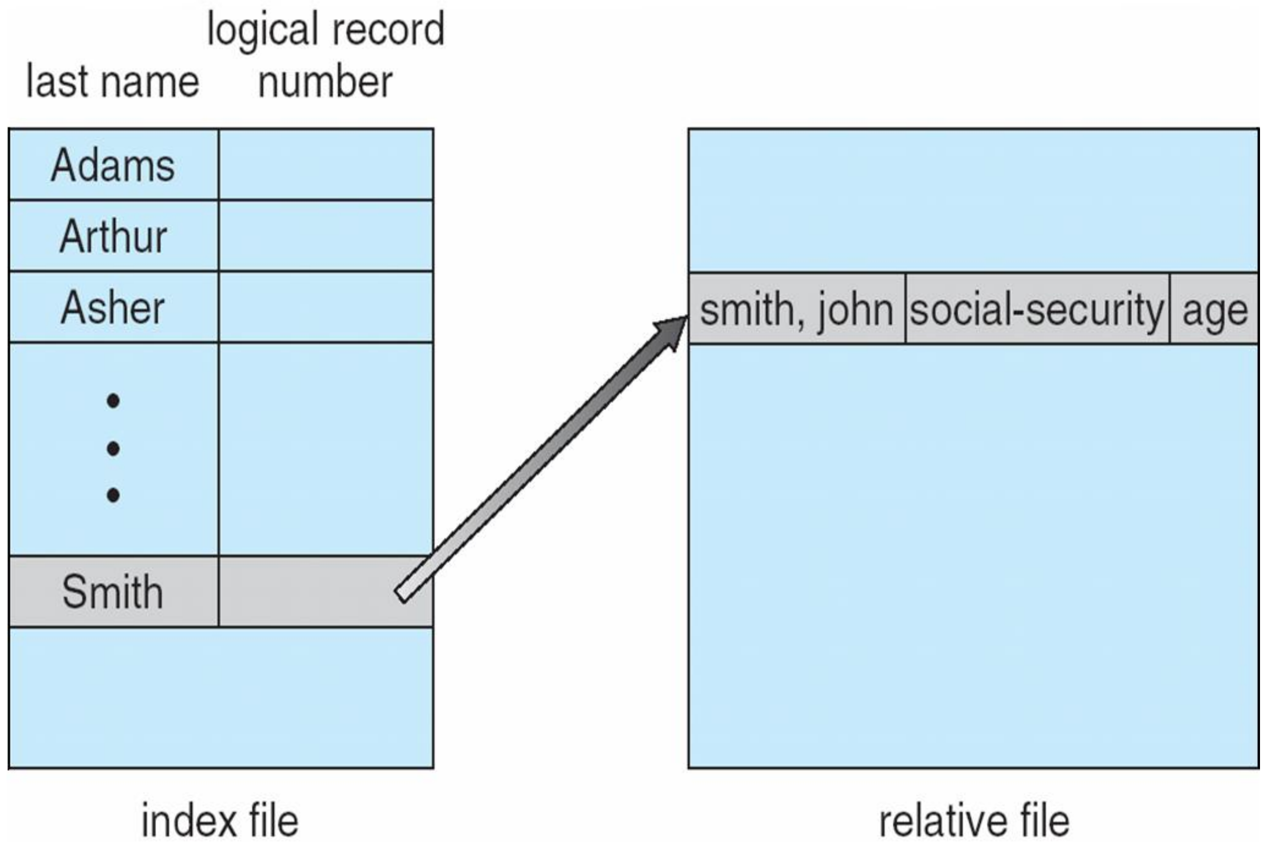
sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

- **Remember however, simulating a direct access file on a sequential access file is extremely inefficient and clumsy.**

Other Access Methods

- These can be built on top of a direct access method.
- Generally involve creation of an **index** for the file which contains pointers to various block.
- To find a record in the file, first search the index and then use the pointer to access the file directly and find the desired record.
- With large files, the index file file itself may become too large to be kept into the memory. So we can create an index for the index table.
- So the primary index file contains pointers to secondary index file which finally points to the actual location.

Example of Index and Relative Files



Directory and Disk Structure

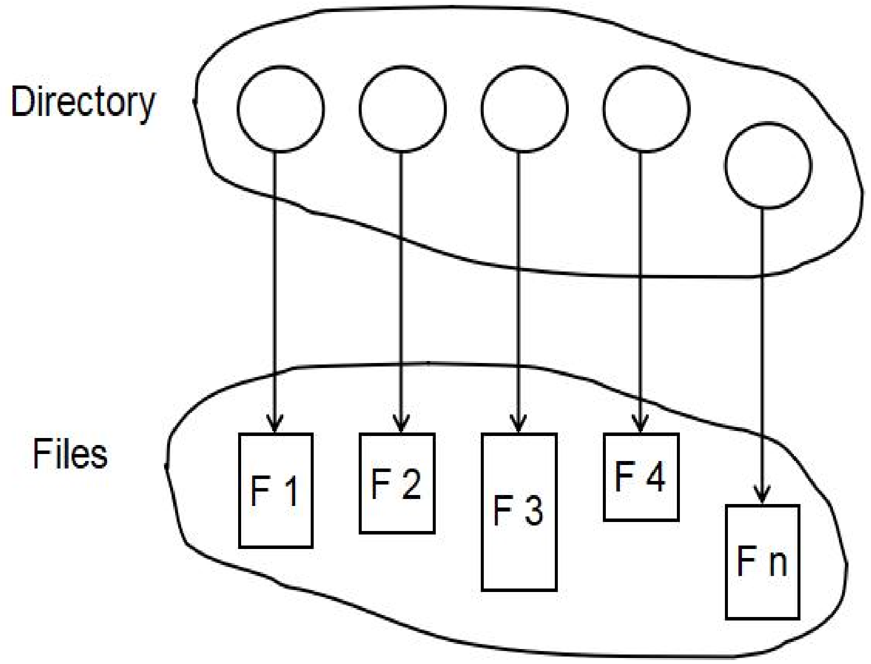
- Each computer can have vast number of files stored on random access storage devices.
- A storage device can be used in its entirety for file system else it can also be sub-divided for finer control.
- So a disk can be partitioned into quarters each storing separate file systems.
- Storage devices can also be collected together into RAID sets to provide protection from failure of a single disk.
- **Benefits of partitioning**:
  - ☐limit the size of individual files
  - ☐put multiple file system types on same device
  - ☐leave part of the device available for others

Volume

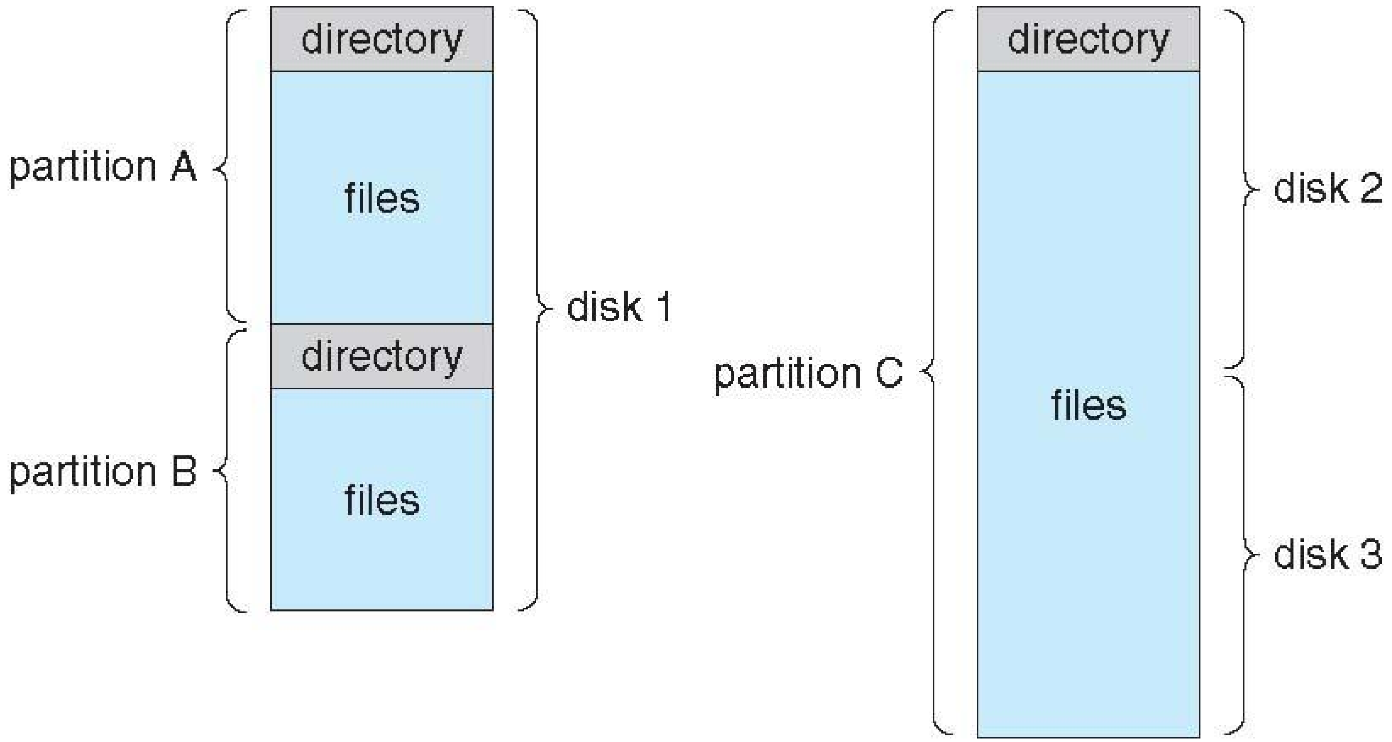
- Any entity containing a file system is generally known as a **volume**.
- Volume may be a subset of a device, a whole device or multiple devices linked together in RAID.
- Each volume can be thought of a virtual disk
- Volumes can also store multiple OS thus allowing a system to boot and run more than one OS
- Each volume that contains file systems also contain information about these files.
- This information is kept in entries in **device directory or volume table of contents**.
- The directory records info. such as name, location, size and type for all files on that volume

Directory Structure

- A collection of nodes containing information about all files
- Both the directory structure and the files reside on disk



A Typical File-system Organization



Types of File Systems

We mostly talk of general-purpose file systems. But systems frequently have may file systems, some general- and some special- purpose

Consider Solaris has

- **tmpfs** – memory-based volatile FS for fast, **temporary** I/O. Contents erased if system reboots or crashes
- **objfs** – **Virtual** FS essentially an interface into kernel memory to get kernel symbols for debugging
- **ctfs** – **contract** file system for managing daemons to dictate which process starts when the system boots and must continue to run during operation.
- **lofs** – **loopback** file system allows one FS to be accessed in place of another
- **procfs** – kernel interface to process structures. It is a virtual FS that presents info. on all processes as a file system
- **ufs, zfs** – general purpose file systems

Directory Overview

Directory is basically a symbol table that translates file names into their directory entries.

Organization of a directory must allow efficient location, names enabling convenient use, logical grouping of files by their properties.

The various organizations methods of a directory must allow users various operations like to insert entries, delete entries, search an entry and list all entries in it.

- **Search a file**: We might have to search a directory to find the entry for particular file. We may also want to find all files whose names match a particular pattern.
- **Create a file**: New files need to be created and added to the directory.
- **Delete a file**: When a file is no longer needed, it must be removed from the directory.

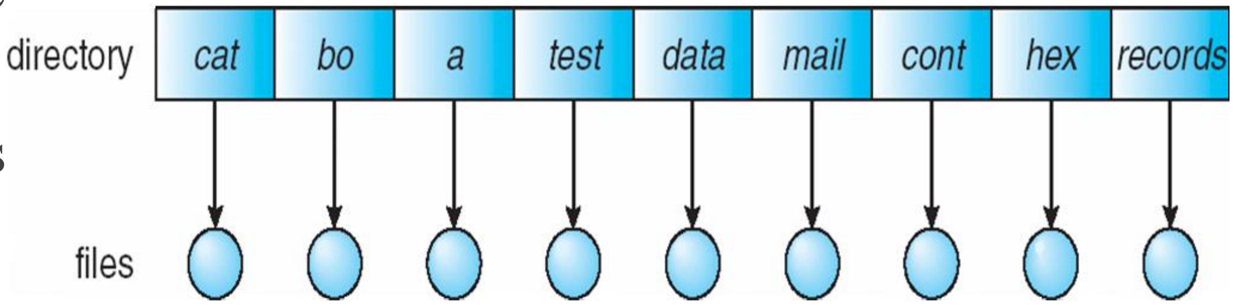
- **List a directory**: List the files in a directory and the contents of the directory entry for each file in the list.

- **Rename a file**: We must be able to change the file name when the contents are changed as both are related. Renaming must also allow its position within the directory structure to be changed.

- **Traverse a file system**: Access every directory and every file within a directory structure. We must save the contents and structure of entire file system at regular intervals to magnetic tape. This also provides a backup in case of failures. In case a file is no longer needed, copy the file to a tape and the disk space of this file can be released for reuse by other files.

Single Level Directory Structure

- The simplest, easy to support and understand.
- In this structure, all files are contained in the same directory.
- When number of files increase or system has more than one user problem occurs.
- Since all files are in the same directory, they must have unique names using 255 characters. But if 2 users call their file by same name, this rule is violated.
- Even a single user may find it difficult to remember all file names as number of files increase.

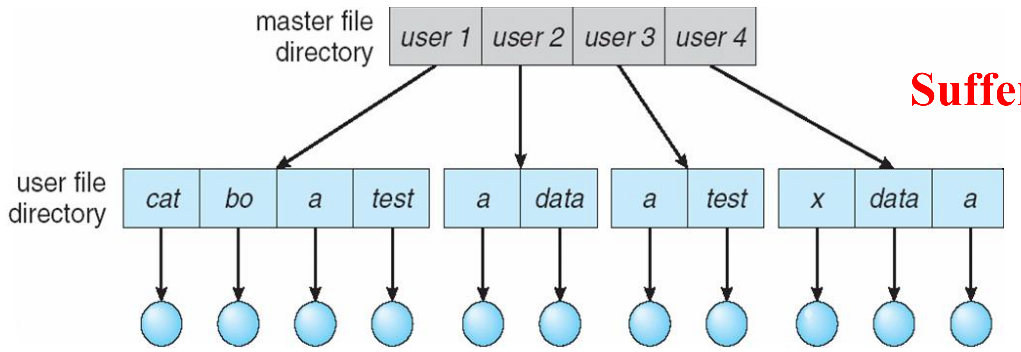


Suffers from Naming Problem and Grouping Problem

Two Level Directory Structure

- To avoid confusion of file names among various users, it is better to create a separate directory for each user.
- **Here each user has his own user file directory (UFD) which have similar structures listing the files of a single user**.
- **When a user logs in, the system's master file directory (MFD) is searched that is indexed by user name or account number and each entry points to the UFD for that user**.
- When a user refers to a particular file, only his UFD is searched. Hence different users may have files with same name as long as all file names within each UFD are unique. Also while deletion, the OS searches in the local UFD so that another user's file is not accidently deleted.
- Thus **2 level directory solves the problem of name collision as it isolates users**.
- However, when users want to cooperate on some task and access each other's files, it poses a problem.

- To discard this problem, one user must have the ability to name a file in another user’s directory.
- To name a particular file uniquely in a 2 level directory, we nned to use both the user name and file name. Thus, it is an inverted tree of height 2.
- The root of the tree is MFD and its direct descendants are the UFDs. The descendants of the tree are the files themselves.
- So the files are the leaves of the tree.
- So specifying a user name and a file name defines a path in the tree from the root (MFD) to a leaf (specific file)
- **Thus, a username and file name defines a path name.**
- So to name a file uniquely, a user must know the path name of the file desired.
- Each system has its own syntax for naming files in directories other than the user’s own.
- Additional syntax is needed to specify the volume of a file. Volume name is also a part of the directory name. First name is the volume and rest is directory and file.
- The sequence of directories searched when a file is named is called the **search path**

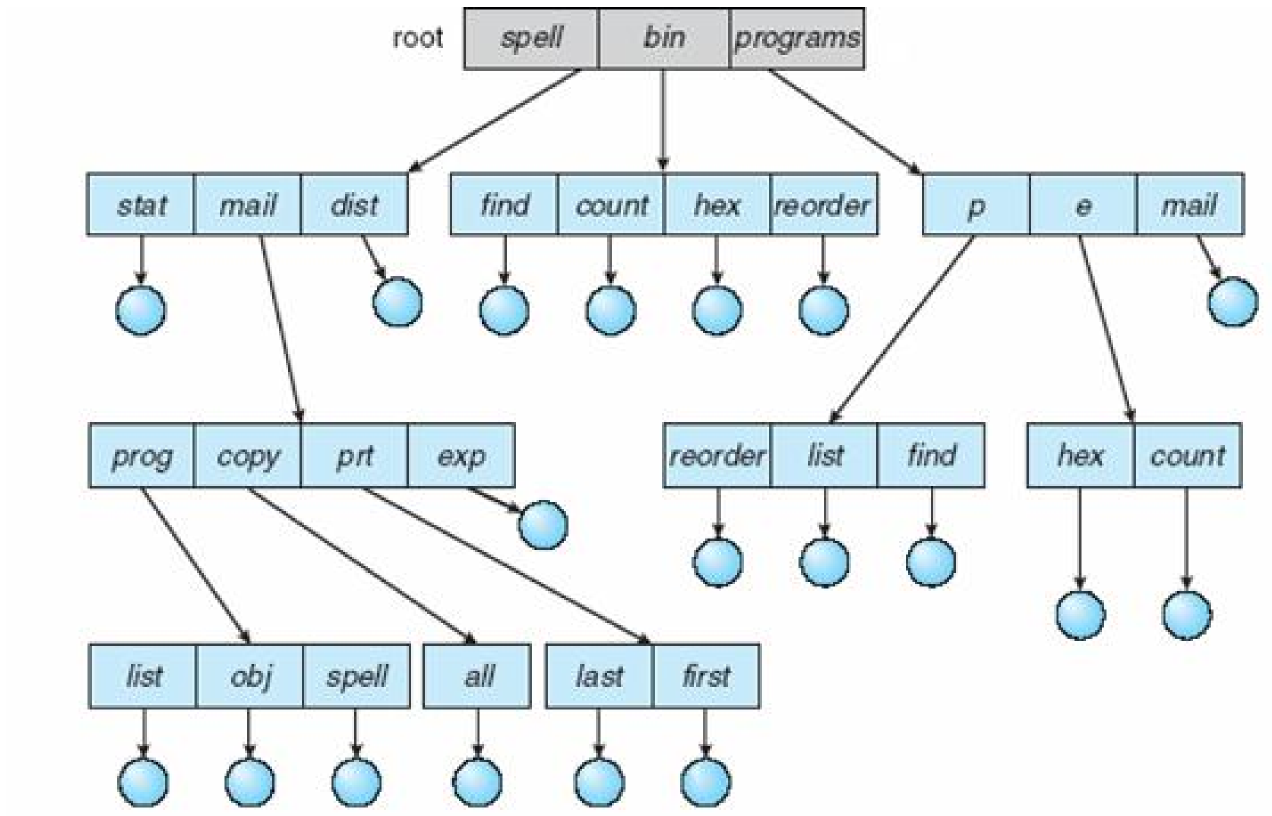


Suffers from Grouping Problem

Tree-Structured Directories

- 2 level directory is restricted to height 2.
- Tree structured directories extend the structure to a tree of arbitrary height.
- So users are free to create own subdirectories and organize their files accordingly within a directory.
- Thus a directory is simply another file treated in a special way.
- One bit in each directory entry defines the entry as a file (0) or a subdirectory (1).
- Each process has its own current directory which contains most of the files of current interest.
- When a reference is made, the current directory is searched. If it is not there, users specify the path name or change the current directory to the directory holding it. All this is done by apt. system calls change_directory ( ) and then open ( ) to search that directory

- **An important concern is how to handle the deletion of a directory in this structure**.
- If a directory is empty, its entry in the directory that contains it is simple deleted.
- But suppose it is not empty and has several files or subdirectories problem occurs.
- One approach is do not delete a directory unless empty. So first delete all files in that directory. Also delete all subdirectories in it recursively then finally delete the directory. Too cumbersome.
- Otherwise use rm command. Thus easy and covenient but pretty dangerous.
- Thus structure also enables user's to access files of other users by specifying the path names



No Grouping and Naming Problem

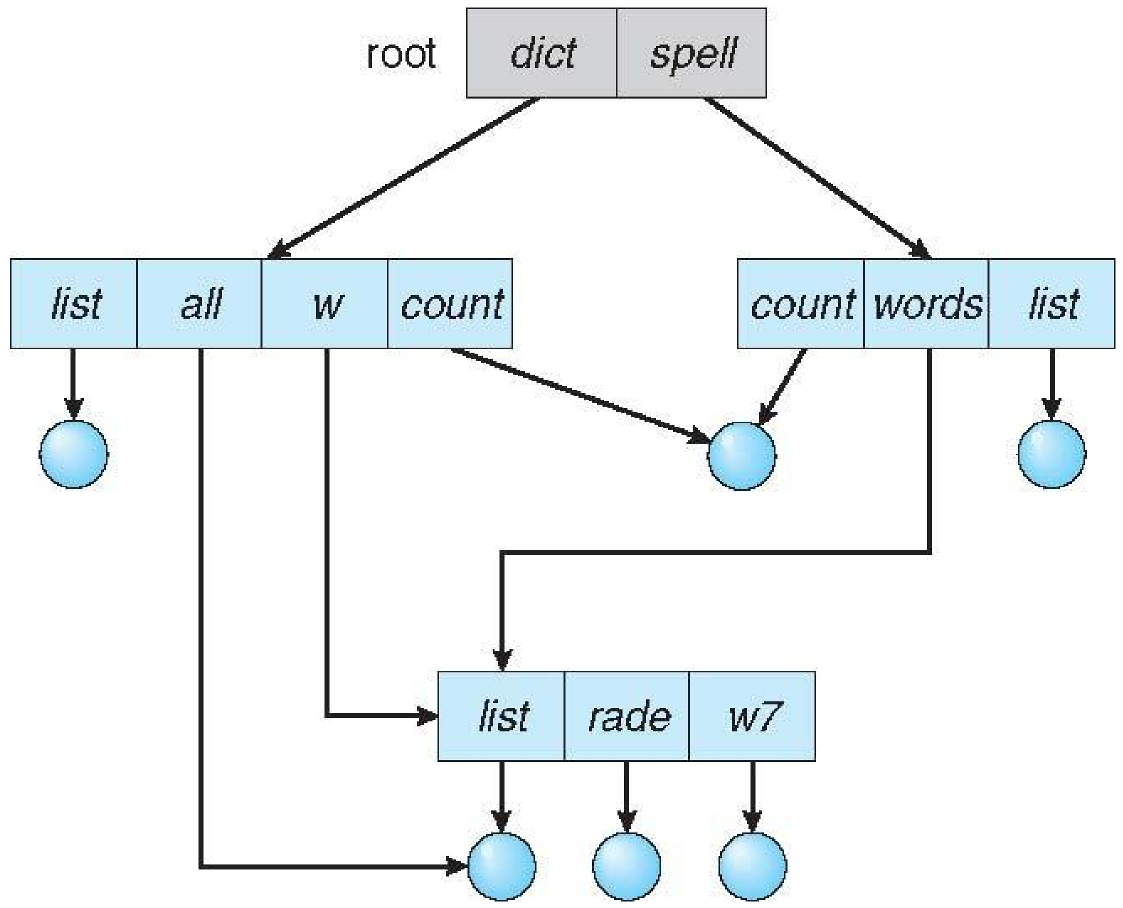
Path Name Types

- ☐ **Absolute Path Name**: Begins at the root and follows a path down to the specified file, thus giving the directory names on the path followed.
- ☐ **Relative Path Name**: Defines a path from the current directory.

- For Eg:
  - Absolute path : root/spell/mail/prt/first.
  - If currently in mail sub-directory (means root/spell/mail is already traversed) then relative path name is prt/first

Acyclic Graph Directories

- Consider two users working on a joint project.
- The files associated to this project must be stored in subdirectory separated from other project files and other files of the two users.
- **Both users are equally responsible for the project so both want the subdirectory to be in their directories.**
- **Thus this common subdirectory is shared by the two users.**
- **So a shared directory or files exists in two or more places at once.**
- **Tree structures prohibit sharing files or directories.**
- **Acyclic graph, graph with no cycles allows sgaring directories or files**
- **Thus, acyclic graph is a natural generalization of the tree structured directory scheme.**



Can you see so may files and directories being shared??

- **Remember a shared file is not the same as two copies of the file.**
- Each user can view the copy of the final but changes in one is not reflected in the other.
- With a shared file, only one actual file exists, changes by one person are immediately visible to others.
- **Sharing is important for subdirectories. So when a new file is created, it will automatically appear in all shared subdirectories**
- **Thus it is better to put all files to be shared into one directory.**
- The UFD of each user will contain this directory of shared files as a subdirectory.

To implement shared files or subdirectories:

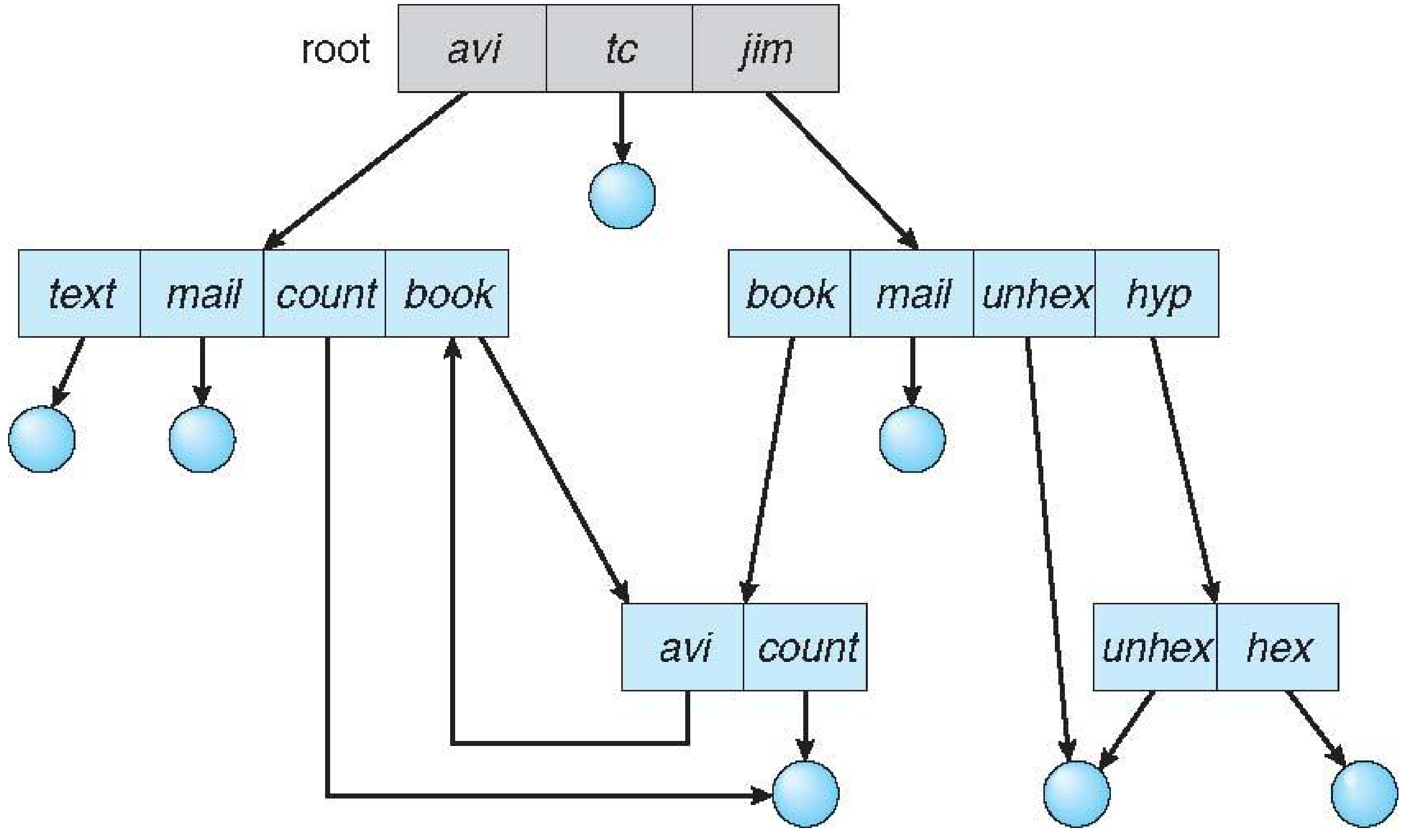
- Many systems create a new directory entry called a **link**. A link is effectively a pointer to another file or subdirectory. So link and original directory entry are different.
- Another approach is to **duplicate all info** about shared files in both sharing directory. So both entries are identical.
- **Problem in Duplication: Original and copy are indistinguishable. Maintaining consistency is difficult.**

Issues in Acyclic Graph Directories:

- Although flexible but more complex.
- A file now can have multiple absolute paths. Thus distinct file names may lead to same file (aliasing problem)
- Shared structues may be traversed, found, backed up etc. more than once.
- **Important question is when can the space allocated to a shared file be deallocated or reused**. So, remove the file whenever any one user deletes it thus leaving dangling pointers and non-existent files. If remaining file pointers contain actual disk address, space is reused for other files and dangling pointers may point into the middle of other files.
- **Deletion is easier if implementation using links as deletion of a link need not affect original file. If the original entry is deleted, the space is deallocated and links are left dangling. Later if user wants he can remove those links also.**
- **Another approach to delete is to preserve a file untill all reference to it are deleted. When reference count is 0 delete the file.**

General Graph Directory

- Serious problem with using an acyclic graph structures is to ensure that there are no cycles.
- **If we use 2 level directory and allow users to create subdirectory, a tree structured directory is formed.**
- **When we add links, the tree structure is destroyed, resulting in a simple graph.**
- **Existence of cycles lead to searching or traversing any component twice.**
- **It may also lead to an infinite loop continually searchin through the cycle and never terminating.**
- **If cycles exist, reference count may not be 0 even when it is no longer possible to refer to a directory or file. This anomaly is due to self referencing.**
- In such cases, we use **garbage collection scheme** to find when last reference has been deleted and disk space can be allocated.



Can you see a cycle??

So a quick recap :-

- ☐ 2 level D: tree of height 2
- ☐ tree D: arbitary height
- ☐ acyclic D: sharing allowed, no cycles
- ☐ general graph D: cycles with sharing

File System Mounting

- **Just as a file must be opened before using, a file system must be mounted before it is available to processes on the system.**
- The mount procedure is straight forward. OS is given the name of the device and mount point.
- **Mount point** is the location within the file structure where the file system is to be attached.
- Typically, a mount point is an empty directory.
- Mounting a file system in a particular location will give its path name.
- OS verifies that device contains a valid file system by asking device driver to read device directory and verify that directory has the expected format.
- Finally OS notes in its directory that a file system is mounted at the mount point.

Read the example of mounting from book. If possible also read file sharing briefly