File system

- **The file system consists of two distinct parts: a collection of *files,* each storing related data, and a *directory structure,* which organizes and provides information about all the files in the system.**

10.1 File Concept

- **Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks.** So that the computer system will be convenient to use, the operating system provides a uniform logical view of information storage.
- ***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.
- **Commonly, files represent programs (both source and object forms) and data**. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.
- **The information in a file is defined by its creator**. Many different types of information may be stored in a file-source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.
- ***A* file has a certain defined structure which depends on its type**. *A text* file is a sequence of characters organized into lines (and possibly pages). *A source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An *object* file is a sequence of bytes organized into blocks understandable by the system's linker. An *executable* file is a series of code sections that the loader can bring into memory and execute.

10.1.1 **File Attributes**

- *A* file is named, for the convenience of its human users, and is referred to by its name. *A* name is usually a string of characters, such as *example.c.* Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not.

A file's attributes vary from one operating system to another but typically consist of these:

1. **Name**: The symbolic file name is the only information kept in human readable form.
2. **Identifier**: This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
3. **Type**: This information is needed for systems that support different types of files.
4. **Location**: This information is a pointer to a device and to the location of the file on that device.
5. **Size**: The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
6. **Protection:** Access-control information determines who can do reading, writing, executing, and so on.
7. **Time, date, and user identification**: This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

10.1.2 File Operations

To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

- **Creating a file**. Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
- **Writing a file**. To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a *write* pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- **Reading a file**. To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a *read* pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated.
- **Repositioning within a file**. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/0. This file operation is also known as a file *seek.*
- **Deleting a file**. To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
- **Truncating a file**. The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged –except for file length-but lets the file be reset to length zero and its file space released.
- **Other common operations include *appending* new information to the end of an existing file and *renaming* an existing file**. These primitive operations can then be combined to perform other file operations.

Continued…

The implementation of the open() and close() operations is more complicated in an environment where several processes may open the file simultaneously.

several pieces of information are associated with an open file.

- **File pointer**: On systems that do not include a file offset as part of the read() and write() system calls, the system must track the last read write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
- **File-open count**: As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Because multiple processes may have opened a file, the system must wait for the last file to close before removing the open-file table entry. The file-open counter tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
- **Disk location of the file**: Most file operations require the system to modify data within the file. 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**: Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/0 requests.

10.1.3 File Types

- **The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a *.com, .exe,* or *.bat* extension can be *executed,* for instance. The *.com* and *.exe* files are two forms of binary executable files, whereas a *.bat* file is a containing, in ASCII format, commands to the operating system.**

Continued…

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

Figure 10.2 Common file types.

10.2 Access Methods

- Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. Other systems, such as those of IBM, support many access methods, and choosing the right one for a particular application is a major design problem.

Access methods are

1. Sequential access
2. Direct access
3. Other Access Methods

   Indexed access

10.2.1 Sequential Access

- **The simplest access method is sequential access. Information in the file is processed in order, one record after the other**. This mode of access is by far the most common; **for example, editors and compilers usually access files in this fashion.**
- **Reads and writes make up the bulk of the operations on a file**. A read operation-read *next-reads* the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write *operation-write* next-appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning; and on some systems, a program may be able to skip forward or backward *n* records for some integer n-perhaps only for *n* = 1. Sequential access, which is depicted in Figure 10.3, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

Continued…

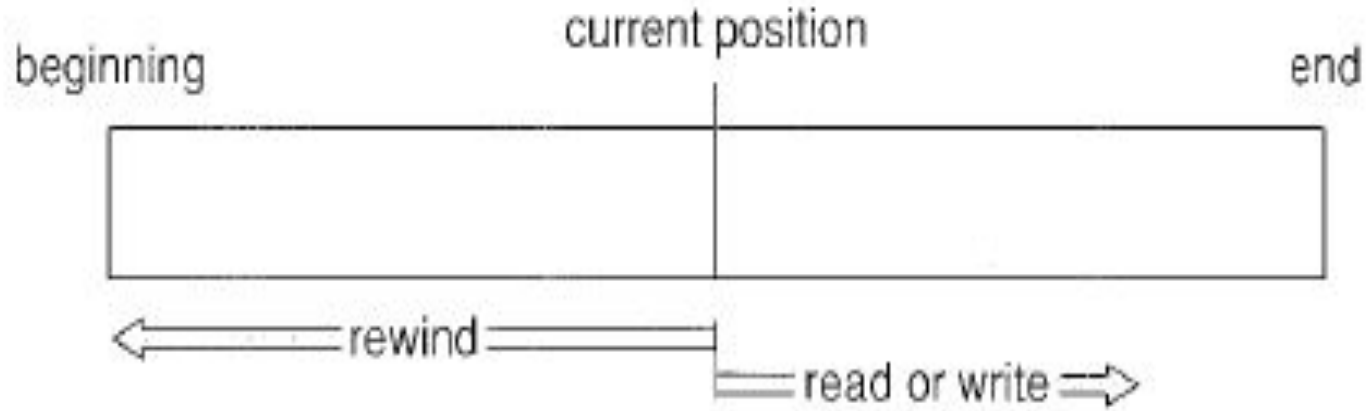


Figure 10.3 Sequential-access file.

10.2.2 Direct Access

- A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block.
- **For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.**
- Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.
- For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have *read n,* where *n* is the block number, rather than *read next,* and *write n* rather than *write next.*

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

Figure 10.4 Simulation of sequential access on a direct-access file.

Continued…

- **The block number provided by the user to the operating system is normally is a relative block number. A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the absolute disk address may be 14703 for the first block and 3192 for the second**. The use of relative block numbers allows the operating system to decide where the file should be placed. and helps to prevent the user from accessing portions of the file system that may not be part of her file. Some systems start their relative block numbers at 0; others start at 1.

10.2.3 Other Access Methods

- **Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file.** The index like an index in the back of a book contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

Example

- **For example, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index.** The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads. Figure 10.5 shows a similar situation as implemented by VMS index and relative files.

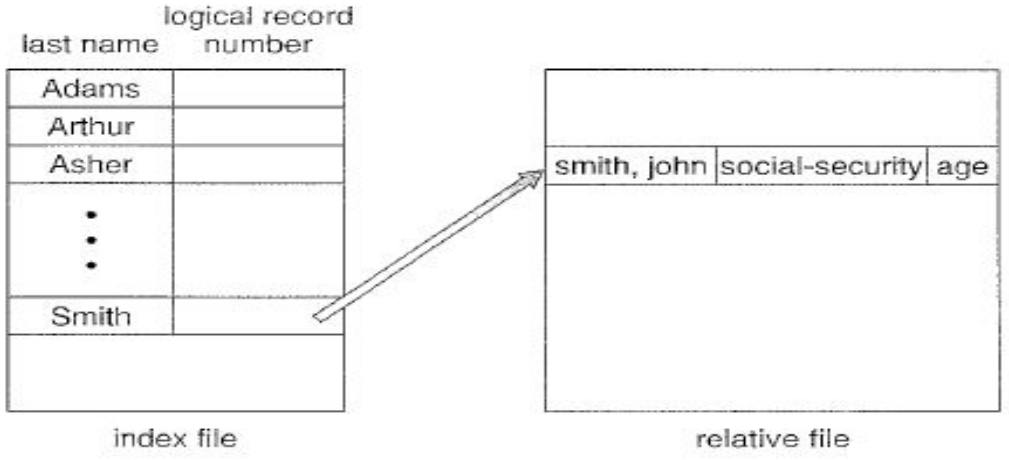
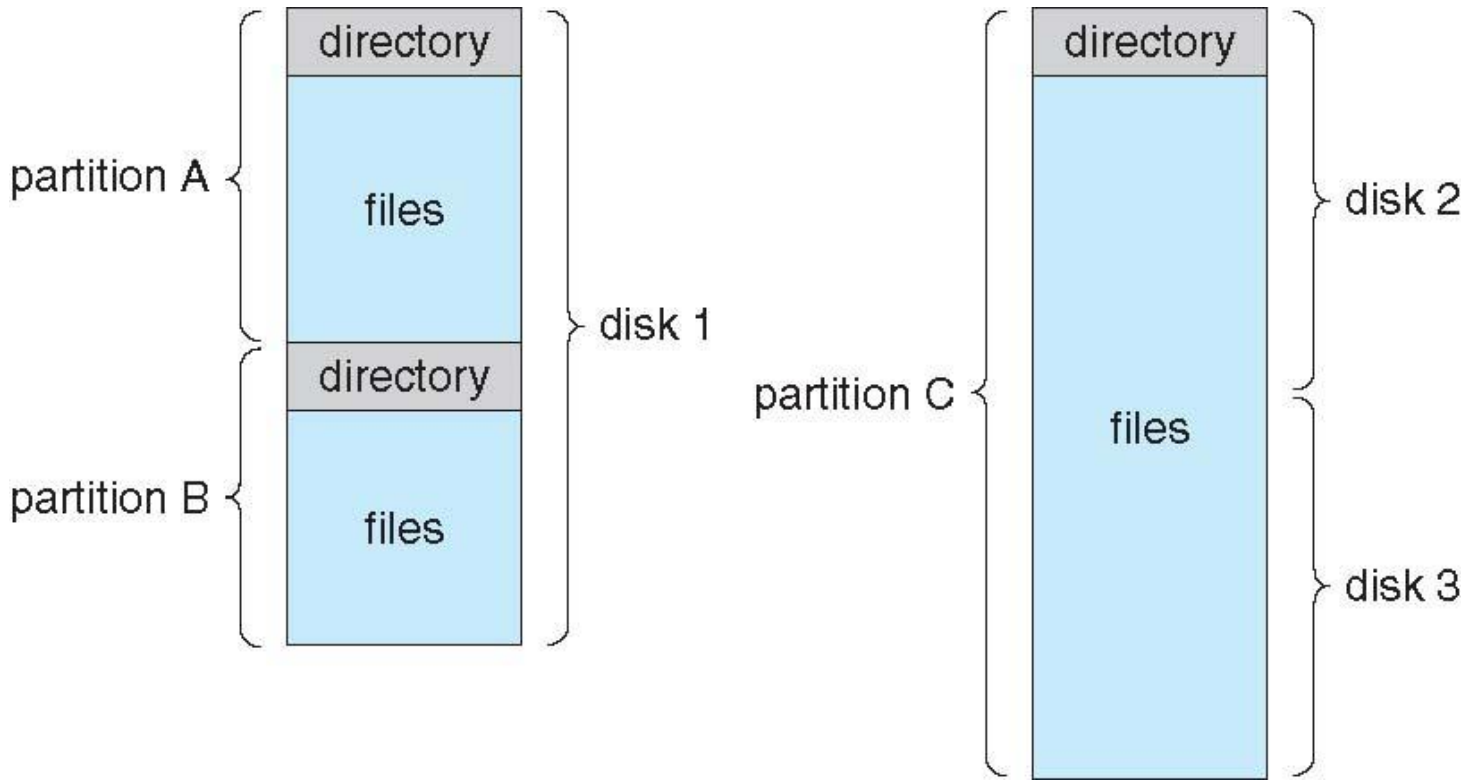


Figure 10.5 Example of index and relative files.

10.3 Directory structure

- **Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization involves the use of directories**. In this section, we explore the topic of directory structure.
- 10.3.1 Storage Structure
- A disk (or any storage device that is large enough) can be used in its entirety for a file system. Sometimes, though, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things, such as swap space or unformatted (raw) disk space.
- These parts are known variously as **partitions, slices,** or (in the IBM world) **minidisks.** A file system can be created on each of these parts of the disk. As we shall see in the next chapter, the parts can also be combined to form larger structures known as **volumes,** and file systems can be created on these as well.
- Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents.**

A Typical File-system Organization

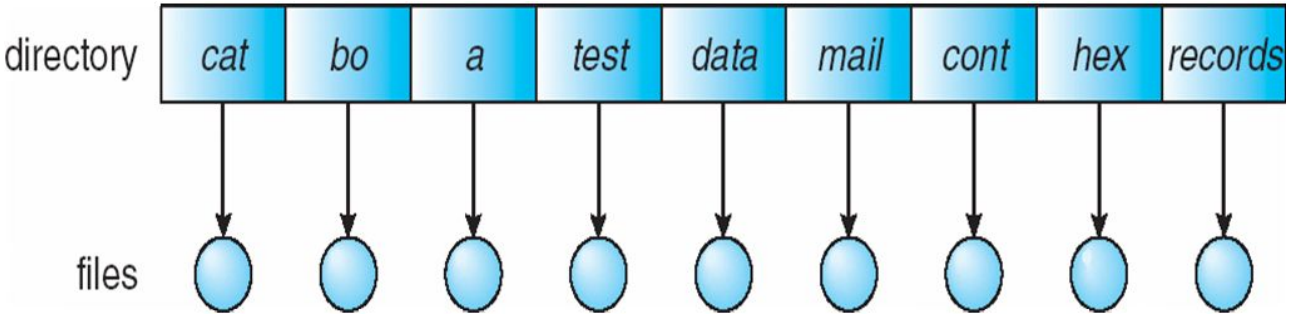


10.3.2 Directory Overview

- When considering a particular directory structure, we need to keep in mind the **operations that are to be performed on a directory:**
- **Search** for a file. We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names and similar names may indicate a relationship between files, we may want to be able 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, we want to be able to remove it from the directory.
- **List a directory**. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Rename a file**. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes.
- **Traverse the file system**. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape. This technique provides a backup copy in case of system failure.

10.3.3 Single-Level Directory

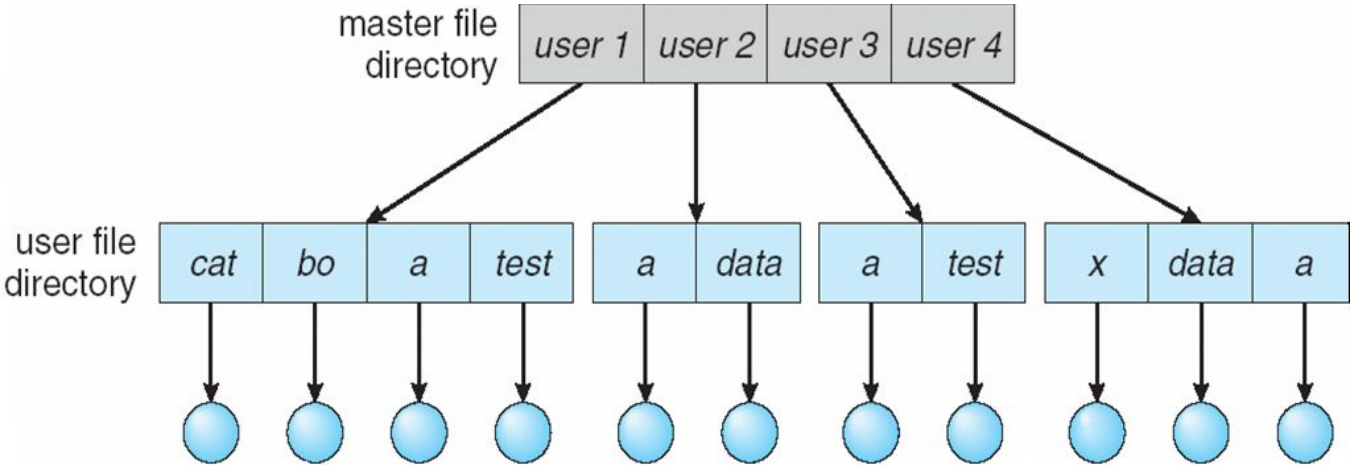
- **The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand**.



- **A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names**. If two users call their data file *test,* then the unique-name rule is violated. For example, in one programming class, 23 students called the program for their second assignment *progl;* another 11 called *i\ assign!.* Although file names are generally selected to reflect the content of the file, they are often limited in length, complicating the task of making file names unique.
- **Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.**

10.3.4 Two-Level Directory

- In the two-level directory structure, each user has his own user file directory (UFD). The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The MFD 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 own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique**. ***To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.***

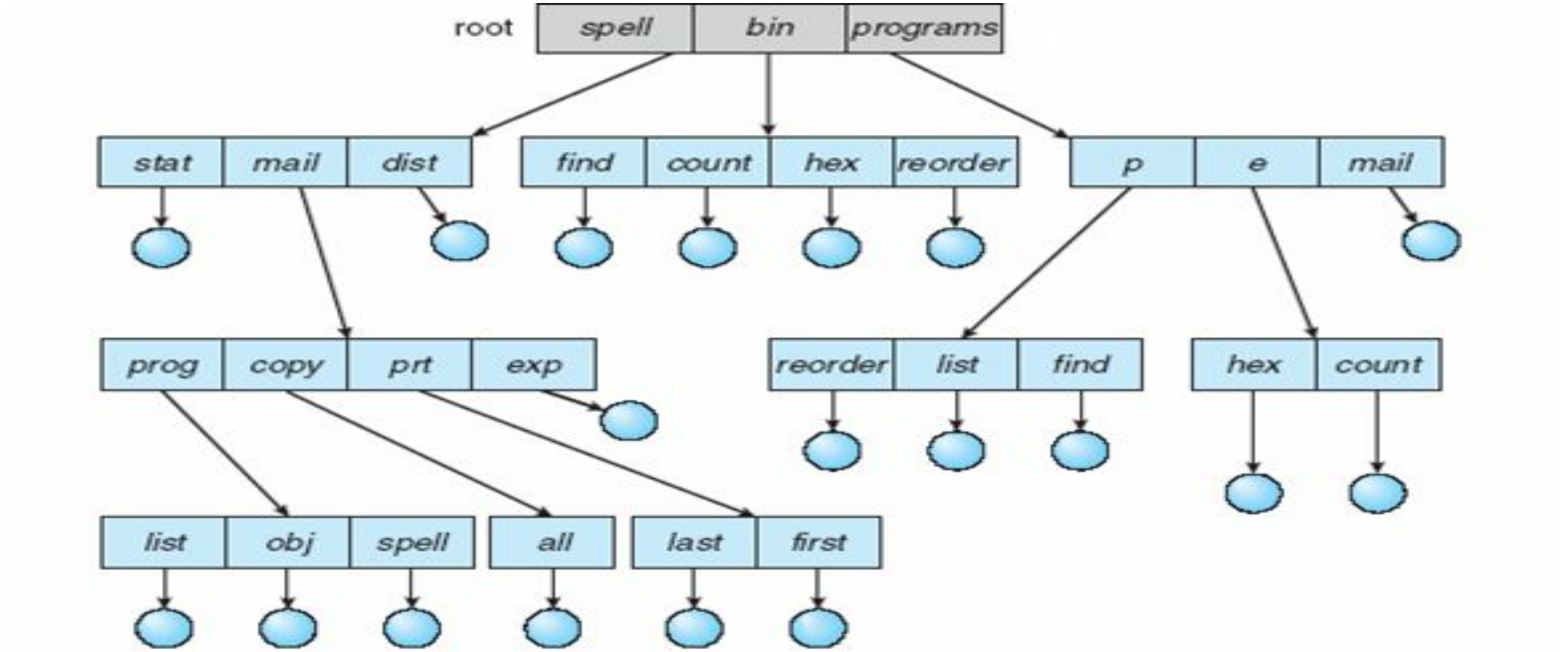
Continued…

Disadvantage: Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users *want* to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.

- If access is to be permitted, one user must have the ability to name a file in another user's directory. To name a particular file uniquely in a two-level directory, we must give both the user name and the file name. A two-level directory can be thought of as a tree, or an inverted tree, of height 2. The root of the tree is the MFD. Its direct descendants are the UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file).
- Thus, a user name and a file name define a *path name.* Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired.
- For example, if user A wishes to access her own test file named *test,* she can simply refer to *test.* To access the file named *test* of user B (with directory-entry name *userb),* however, she might have to refer to */userb/test.* Every system has its own syntax for naming files in directories other than the user's own.

10.3.5 Tree-Structured Directories

- This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.



Continued.

- **A directory (or subdirectory) contains a set of files or subdirectories**. A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.
- **In normal use, each process has a current directory**. The **current directory** should contain most of the files that are of current interest to the process. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file.
- **The initial current directory of the login shell of a user is designated when the user job starts or the user logs in**. **The operating system searches the accounting file (or some other predefined location) to find an entry for this user (for accounting purposes). In the accounting file is a pointer to (or the name of) the user's initial directory**. This pointer is copied to a local variable for this user that specifies the user's initial current directory. From that shell, other processes can be spawned. The current directory of any subprocess is usually the current directory of the parent when it was spawned.

Continued..

- **Path names can be of two types: *absolute* and *relative.* An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path. A relative path name defines a path from the current directory**.
- For example, in the tree-structured file system of Figure 10.9, if the current directory is ***root/spell'/mail,*** then the relative path name ***prt/first*** refers to the same file as does the absolute path name ***root/spcll/mail/prt/first.***
- Allowing a user to define her own subdirectories permits her to impose a structure on her files. This structure might result in separate directories for files associated with different topics (for example, a subdirectory was created to hold the text of this book) or different forms of information (for example, the directory *programs* may contain source programs; the directory *bin* may store ail the binaries).

continued..

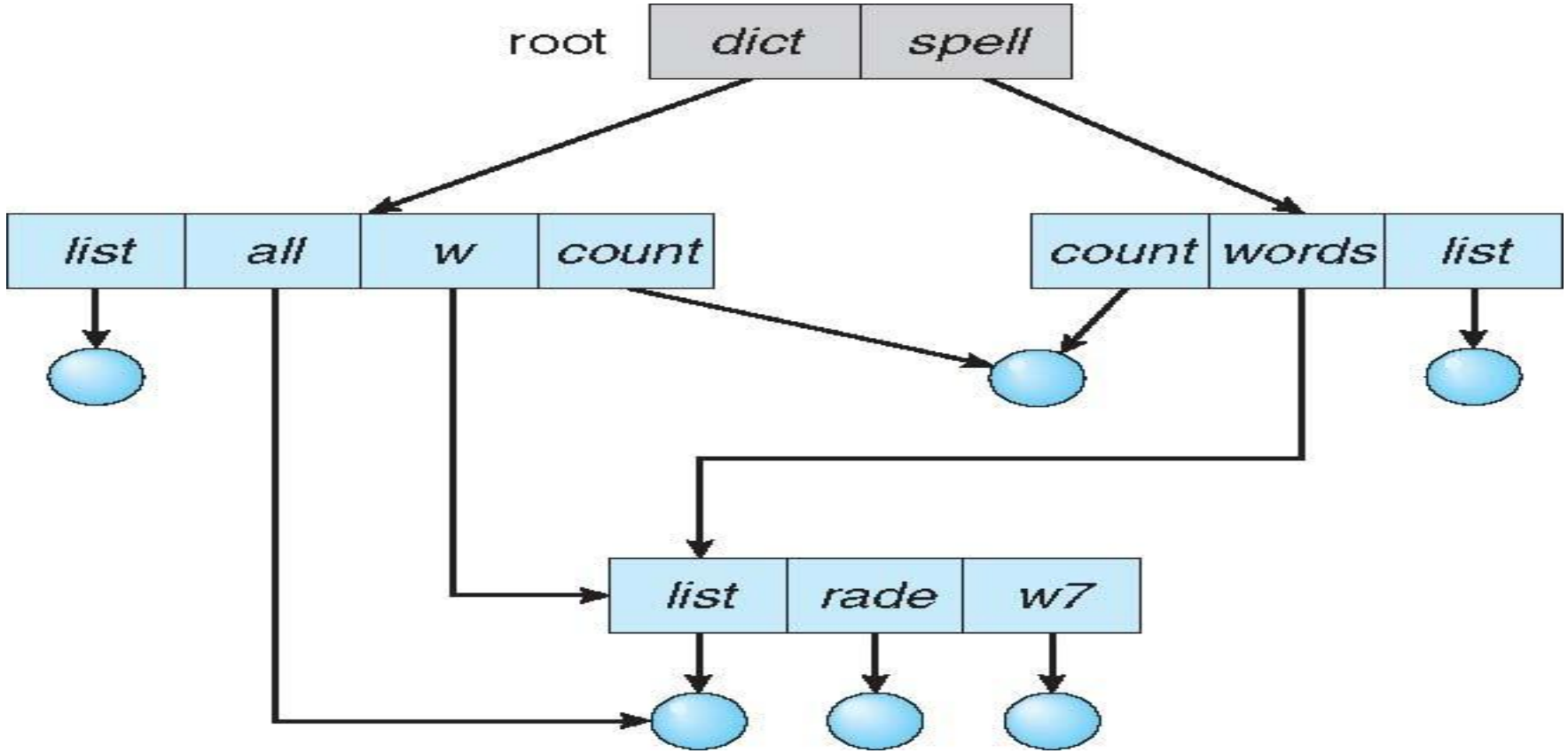
- **An interesting policy decision in a tree-structured directory concerns how to handle the deletion of a directory.** If a directory is empty, its entry in the directory that contains it can simply be deleted. However, suppose the directory to be deleted is not empty but contains several files or subdirectories. One of two approaches can be taken. Some systems, such as MS-DOS, will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist, this procedure must be applied recursively to them, so that they can be deleted also. **This approach can result in a substantial amount of work. An alternative approach, such as that taken by the UNIX rm command**, is to provide an option: When a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted. Either approach is fairly easy to implement; the choice is one of policy.
- A path to a file in a tree-structured directory can be longer than a path in a two-level directory

10.3.6 Acyclic-Graph Directories

- **Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. The common subdirectory should be *shared. A* shared directory *or* file will exist in the file system in two (or more) places at once**.
- **A tree structure prohibits the sharing of files or directories. An acyclic graph —that is, a graph with no cycles—allows directories to share subdirectories and files**.
- **It is important to note that a shared file (or directory) is not the same as two copies of the file**. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, only *one* actual file exists, so any changes made by one person are immediately visible to the other. Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.
- **When people are working as a team, all the files they want to share can be put into one directory. The UFD of each team member will contain this directory of shared files as a subdirectory.**

Continued..

- **Shared files and subdirectories can be implemented in several ways. A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a link**. **A link is effectively a pointer to another file or subdirectory.** For example, a link may be implemented as an absolute or a relative path name. When a reference to a file is made, we search the directory. If the directory entry is marked as a link, then the name of the real file is included in the link information.

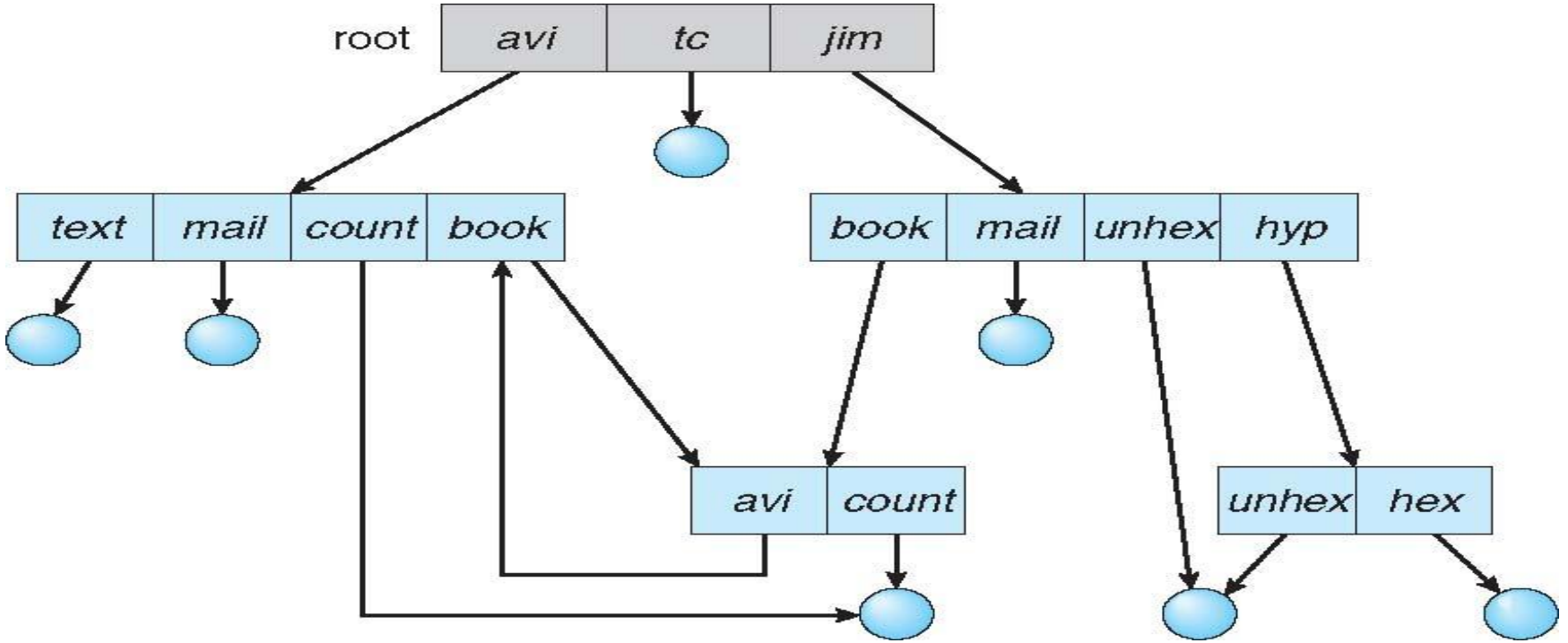


Continued..

- **An acyclic-graph directory structure is more flexible than is a simple tree structure, but it is also more complex.** Several problems must be considered carefully.
- **Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused? One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file**. Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files.
- Another approach to deletion is to preserve the file until all references to it are deleted.

10.3.7 General Graph Directory

- A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. **However, when we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure.**



Continued..

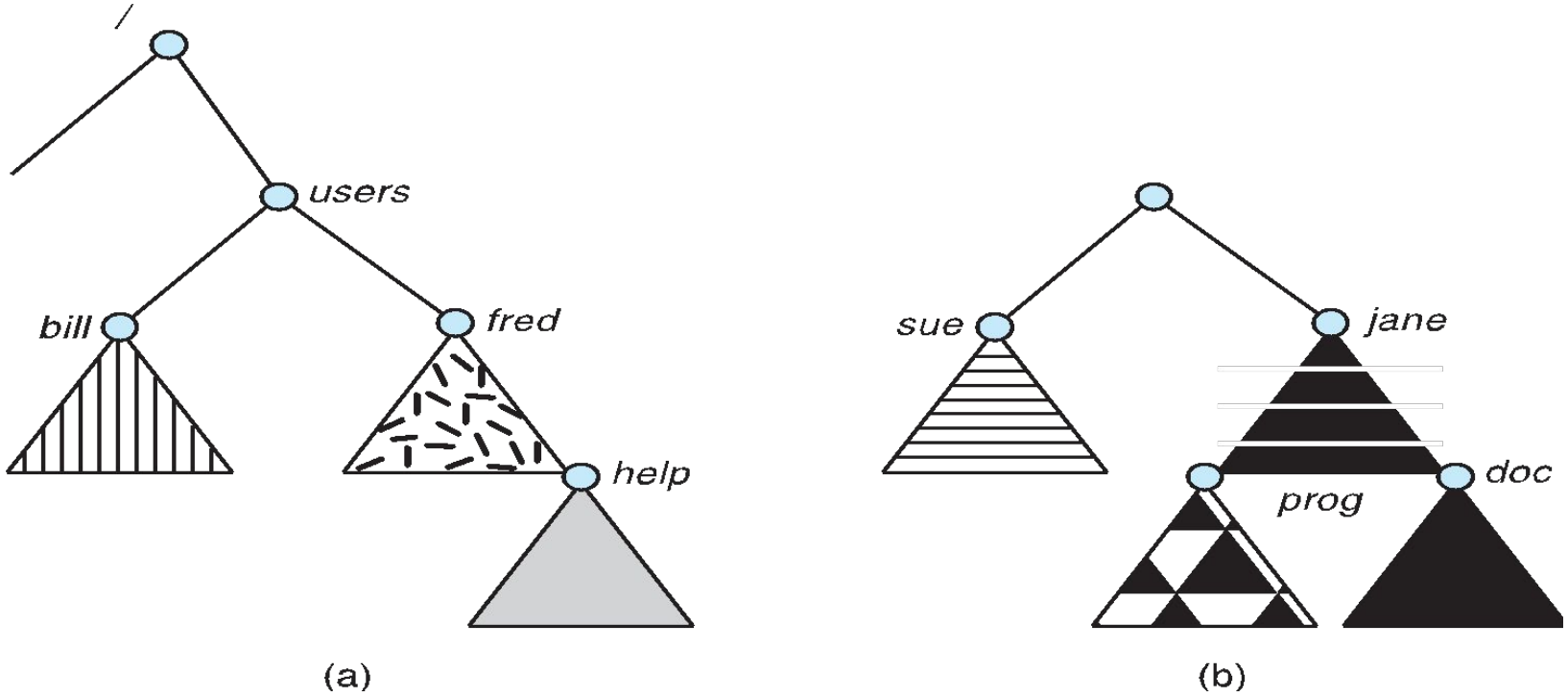
- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - **Garbage collection**
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK.

10.4 File-System Mounting

- **Just as a file must be *opened* before it is used, a file system must be *mounted* before it can be available to processes on the system**. More specifically, the directory structure can be built out of multiple volumes, which must be mounted to make them available within the file-system name space.
- **The mount procedure is straightforward**. The operating system is given the name of the device and the **mount point**—the location within the file structure where the file system is to be attached. **Typically, a mount point is an empty directory**. For instance, on a UNIX system, a file system containing a user's home directories might be mounted as */home;* then, to access the directory structure within that file system, we could precede the directory names with *ftiome,* as in */homc/janc.* Mounting that file system under */users* would result in the path name */users/jane,* which we could use to reach the same directory.
- **Next, the operating system verifies that the device contains a valid file system**. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format.

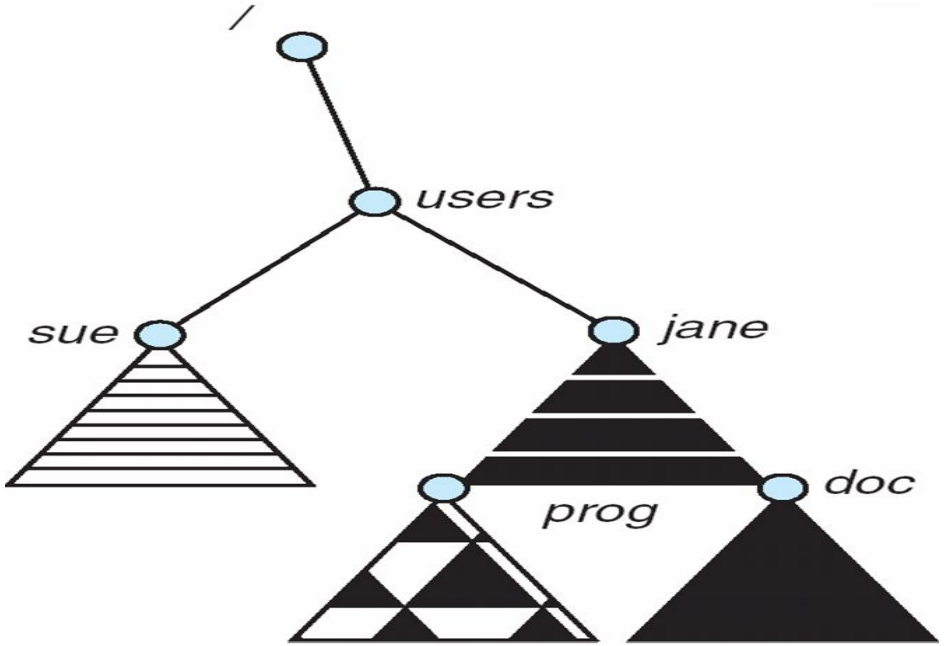
Continued..

- Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems as appropriate.
- A unmounted file system (fig b) is mounted at a **mount point(fig a).**



Continued..

- directory structure, switching among file systems as appropriate. To illustrate file mounting, consider the file system depicted in Figure, where the triangles represent subtrees of directories that are of interest. Figure (a) shows an existing file system, while Figure (b) shows an unmounted volume residing on */device'/dsk.* At this point, only the files on the existing file system can be accessed. Below Figure shows the effects of mounting the volume residing on */device/dsk* over */users*
-



Continued…

- file sharing
  - Multiple users
  - Remote File Systems
    - Client server model
    - Distributed information system
    - Failure modes
- Protection
  - Access control

10.5 file sharing

- 10.5.1 Multiple Users
- **When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent**. Given a directory structure that allows files to be shared by users, the system must mediate the file sharing.
- **The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files.** These are the issues of access control and protection.
- **To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system**. Although many approaches have been taken to this requirement historically, most systems have evolved to use the concepts of file (or directory) owner (or user) and group.
- **The owner is the user who can change attributes and grant access and who has the most control over the file**. The group attribute defines a subset of users who can share access to the file.
- **For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations.**
- **The owner and group IDs of a given file (or directory) are stored with the other file attributes. When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file. Likewise, the group IDs can be compared. The result indicates which permissions are applicable**. The system then applies those permissions to the requested operation and allows or denies it

10.5.2 Remote File Systems

- Through the evolution of network and file technology, remote file-sharing methods have changed. **The first implemented method involves manually transferring files between machines via programs like ftp. The second major method uses a distributed file system (DFS) in which remote directories are visible from a local machine. In some ways, the third method, the World Wide Web, is a reversion to the first. A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for ftp) are used to transfer files.**
- **ftp is used for both anonymous and authenticated access. Anonymous access allows a user to transfer files without having an account on the remote system.** The World Wide Web uses anonymous file exchange almost exclusively.

10.5.2.1 The Client- Server Model

- **Remote file systems allow a computer to mount one or more file systems from one or more remote machines. In this case, the machine containing the files is the server, and the machine seeking access to the files is the client. The client-server relationship is common with networked machines.**
- **Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients. A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client-server facility.**
- **The server usually specifies the available files on a volume or directory level**. Client identification is more difficult**. A client can be specified by a network name or other identifier, such as an IP address**, but these can be spoofed, or imitated.
- **As a result of spoofing, an unauthorized client could be allowed access to the server. More secure solutions include secure authentication of the client via encrypted keys.**
- **In the case of UNIX and its network file system (NFS), authentication takes place via the client networking information, by default**. In this scheme, the user's IDs on the client and server must match. If they do not, the server will be unable to determine access rights to files.

10.5.2.2 Distributed Information Systems

- **To make client-server systems easier to manage, distributed information systems, also known as distributed naming services, provide unified access to the information needed for remote computing.**
- The domain name system (DNS) provides host-name-to-network-address translations for the entire Internet (including the World Wide Web). Before DNIS became widespread, files containing the same information were sent via e-mail or ftp between all networked hosts.
- **Other distributed information systems provide user name/password/user ID/group ID space for a distributed facility.** UNIX systems have employed a wide variety of distributed-information methods. Sun Microsystems introduced yellow pages (since renamed network information service, or NIS), and most of the industry adopted its use.
- In the case of Microsofts common internet file system (CIFS), network information is used in conjunction with user authentication (user name and password) to create a network login that the server uses to decide whether to allow or deny access to a requested file system.

10.5.2.3 Failure Modes

- **Local file systems can fail for a variety of reasons, including failure of the disk containing the file system, corruption of the directory structure or other disk-management information (collectively called metadata), disk-controller failure, cable failure, and host-adapter failure**. **User or systems-administrator failure can also cause files to be lost or entire directories or volumes to be deleted. Many of these failures will cause a host to crash and an error condition to be displayed, and human intervention will be required to repair the damage.**
- Remote file systems have even more failure modes. **Because of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems.**
- **In the case of networks, the network can be interrupted between two hosts. Such interruptions can result from hardware failure, poor hardware configuration**, or networking implementation issues.
- **Now consider a partitioning of the network, a crash of the server, or even a scheduled shutdown of the server. Suddenly, the remote file system is no longer reachable**. This scenario is rather common, so it would not be appropriate for the client system to act as it would if a local file system were lost.
- Rather, the system can either terminate all operations to the lost server or delay operations until the server is again reachable.

failure semantics are defined and implemented as part of the remote-file-system protocol

- **Consistency semantics represent an important criterion for evaluating any file system that supports file sharing. These semantics specify how multiple users of a system are to access a shared file simultaneously. In particular, they specify when modifications of data by one user will be observable by other users.**
- **For the following discussion, we assume that a series of file accesses (that is, reads and writes) attempted by a user to the same file is always enclosed between the open() and close() operations. The series of accesses between the open() and close () operations makes up a file session**. To illustrate the concept, we sketch several prominent examples of consistency semantics.
- **10.5.3.1 UNIX Semantics**

The UNIX file system uses the following consistency semantics:

• Writes to an open file by a user are visible immediately to other users that have this file open.

• One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, regardless of their origin.

10.5.3.2 Session Semantics

The Andrew file system (AFS) uses the following consistency semantics:

• Writes to an open file by a user are not visible immediately to other users that have the same file open.

• Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes

Continued..

10.5.3.3 Immutable-Shared-Files Semantics

- **A unique approach is that of immutable shared files. Once a file is declared as shared by its creator, it cannot be modified.**
- **An immutable file has two key properties:**
- Its name may not be reused, and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed.
- The implementation of these semantics in a distributed system is simple, because the sharing is disciplined (read-only).

10.6 Protection

- When information is stored in a computer system, we want to keep it safe from physical damage (reliability) and improper access (protection).
- Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed.
- Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multiuser system, however, other mechanisms are needed.
- 10.6.1 Types of Access
- The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access. Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is controlled access.

Continued..

- Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:
- Read. Read from the file.
- Write. Write or rewrite the file.
- Execute. Load the file into memory and execute it.
- Append. Write new information at the end of the file.
- Delete. Delete the file and free its space for possible reuse.
- List. List the name and attributes of the file.

10.6.2 Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a. file or directory. **The most general scheme to implement identity dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user.**

- When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

Continued…

- To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:
- Owner. The user who created the file is the owner.
- Group. A set of users who are sharing the file and need similar access is a group, or work group.
- Universe. All other users in the system constitute the universe.