FILE SYSTEMS

FILE CONCEPT

FILE: *A* file is a named collection of related information that is recorded on secondary storage.
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 which depends on its type.

- *A text* file is a sequence of characters organized into lines.
- *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.

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
When a file is named, it becomes independent of the process, the user, and even the system that created it.

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

- **Name:** The symbolic file name is the only information kept in human readable form.
- **Identifier:** This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type:** This information is needed for systems that support different types of files.
- **Location:** This information is a pointer to a device and to the location of the file on that device.
- **Size:** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection:** Access-control information determines who can do reading, writing, executing, and so on.

- **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.

The information about all files is kept in the directory structure, which also resides on secondary storage. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes.

File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.

1. **Creating a file:** Two steps are necessary to create a file,
   i. Space in the file system must be found for the file.
   ii. An entry for the new file must be made in the directory.

2. **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.

3. **Reading a file:** To read from a file, we use a system call that specifies the name of the file and where 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. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current file-position pointer.

4. **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 files seek.
5. **Deleting a file:** To delete a file, search the directory for the named file. Having found the associated directory entry, then release all file space, so that it can be reused by other files, and erase the directory entry.

6. **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 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.

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 actively. The operating system keeps a small table, called the **open file table** containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required.

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

The operating system uses two levels of internal tables:

1. A per-process table
2. A system-wide table

The per-process table tracks all files that a process has open. Stored in this table is information regarding the use of the file by the process.

Each entry in the per-process table in turn points to a **system-wide open-file table**. The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file.

When another process executes an open() call a new entry is simply added to the process's open-file table pointing to the appropriate entry in the system-wide table. Typically, the open-file table also has an *open count* associated with each file to indicate how many processes have the file open. Each close() decreases this *open count*, and when the *open count* reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.

Several pieces of information are associated with an open file.

1. **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.

2. **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.
3. **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.
4. **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.

File Types

- The operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways.
- A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts-**a name and an extension**, usually separated by a period character
- The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.

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

File Structure

- File types also can be used to indicate the internal structure of the file. For instance source and object files have structures that match the expectations of the programs that read them. Certain files must conform to a required structure that is understood by the operating system.
  For example: the operating system 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.

- The operating system support multiple file structures: the resulting size of the operating system also increases. If the operating system defines five different file structures, it needs to contain the code to support these file structures.
- It is necessary to define every file as one of the file types supported by the operating system. When new applications require information structured in ways not supported by the operating system, severe problems may result.

Example: assume that a system supports two types of files: text files and executable binary files. If user wants to protect the contents by encryption methods, none of the above mentioned file formats can be used.

Some operating systems impose (and support) a minimal number of file structures.

Example: The Macintosh operating system supports a minimal number of file structures. It expects files to contain two parts: a resource fork and data fork.

- **The resource fork** contains information of interest to the user.
- **The data fork** contains program code or data

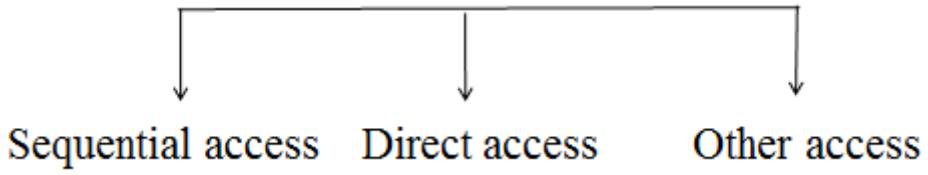
Internal File Structure

Internally, locating an offset within a file can be complicated for the operating system. Disk systems typically have a well-defined block size determined by the size of a sector. All disk I/0 is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

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 of the common methods are:



1. Sequential methods

- The simplest access method is sequential methods. Information in the file is processed in order, one record after the other.
- Reads and writes make up the bulk of the operations on a file.
- A read operation (next-reads) reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location
- The write operation (write next) appends to the end of the file and advances to the end of the newly written material.

- 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.

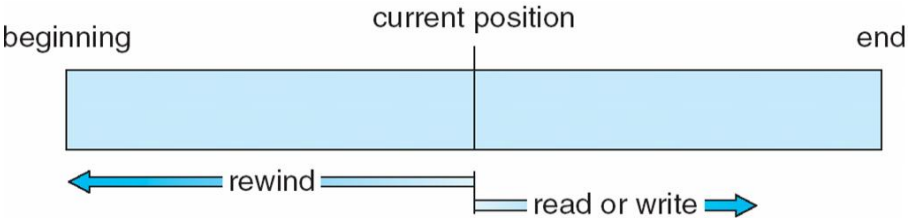


Figure: Sequential-access file.

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.
- Example: if 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 such as Databases, where searching becomes easy and fast.
- 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.* An alternative approach is to retain *read next* and *write next,* as with sequential access, and to add an operation *position file to n,* where *n* is the block number. Then, to affect a *read n,* we would *position to n* and then *read 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: Simulation of sequential access on a direct-access file.

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**, is 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.

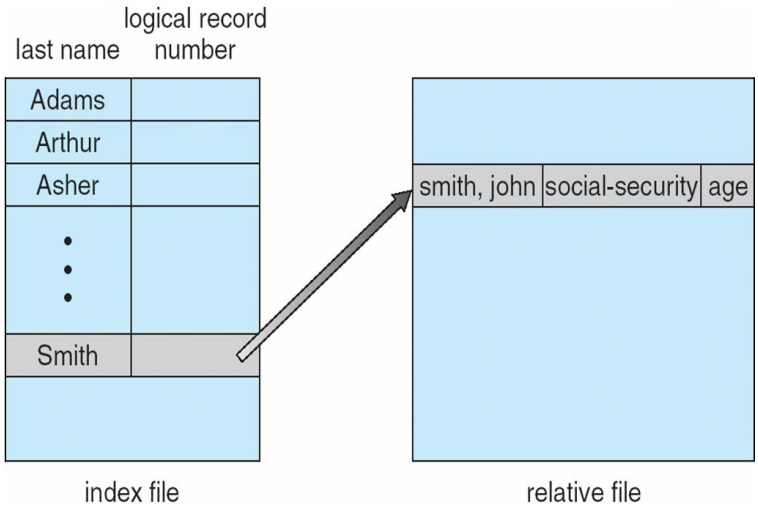


Figure: Example of index and relative files

DIRECTORY AND DISK STRUCTURE

- Files are stored on random-access storage devices, including hard disks, optical disks, and solid state (memory-based) disks.
- A storage device can be used in its entirety for a file system. It can also be subdivided for finer-grained control
- Disk can be subdivided into partitions. Each disks or partitions can be RAID protected against failure.
- Partitions also known as minidisks or slices. Entity containing file system known as a volume. 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**

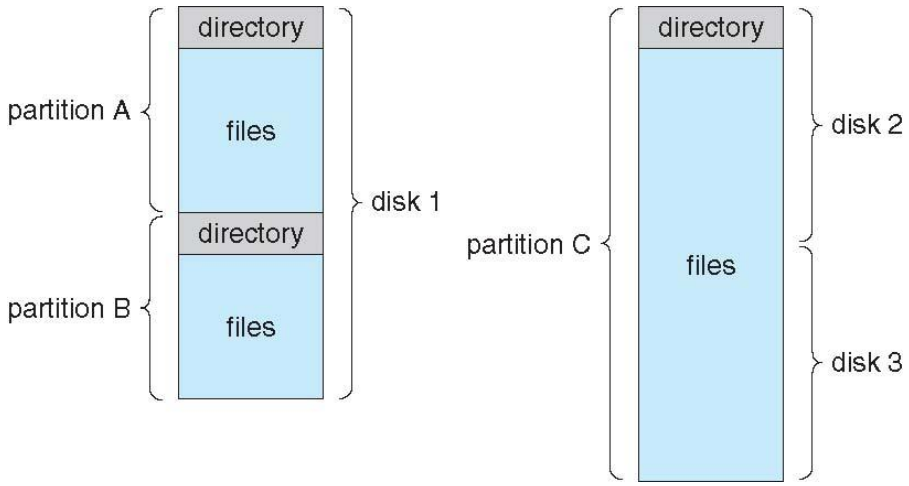


Figure: A Typical File-system Organization

Directory Overview

The directory can be viewed as a symbol table that translates file names into their directory entries. A directory contains information about the files including attributes location and ownership.

To consider a particular directory structure, certain operations on the directory have to be considered:

- **Search for a file:** Directory structure is searched for a particular file in directory. Files have symbolic names and similar names may indicate a relationship between files. Using this similarity it will be easy to find all whose name matches a particular pattern.
- **Create a file:** New files needed to be created and added to the directory.
- **Delete a file:** When a file is no longer needed, then it is able to remove it from the directory.

- **List a directory:** It is 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, It is possible to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system:** User may wish to access every directory and every file within a directory structure. To provide reliability the contents and structure of the entire file system is saved at regular intervals.

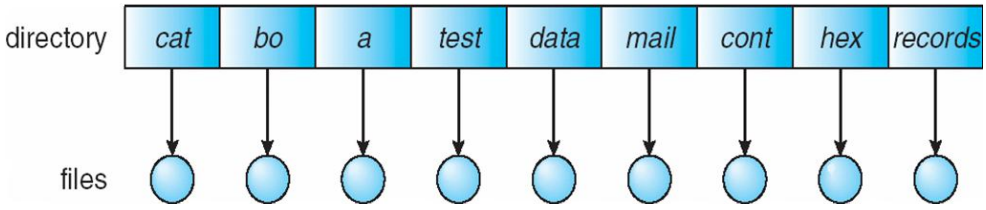
The most common schemes for defining the logical structure of a directory are described below

1. Single-level Directory

2. Two-Level Directory
3. Tree-Structured Directories
4. Acyclic-Graph Directories
5. General Graph Directory

1. 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

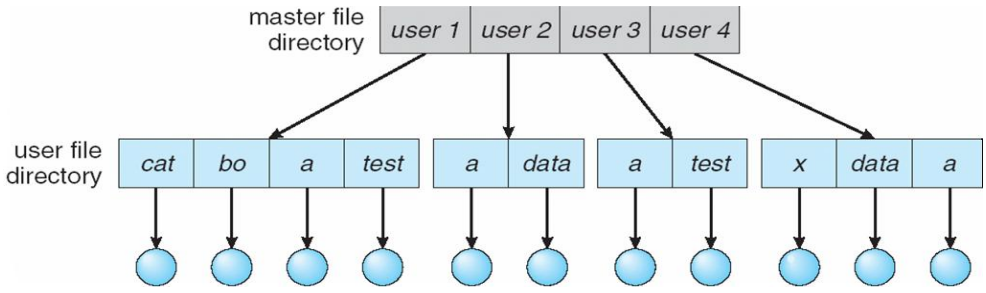


A single-level directory has significant limitations, when the number of files increases or when the system has more than one user.

- As directory structure is single, uniqueness of file name has to be maintained, which is difficult when there are multiple users.
- 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.
- It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

2. Two-Level Directory

- In the two-level directory structure, each user has its own **user file directory** (UFD). The UFDs have similar structures, but each lists only the files of a single user.
- When a user refers to a particular file, only his own UFD is searched. 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.
- 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.



Advantage:

1. No filename-collision among different users.
2. Efficient searching.

Disadvantage

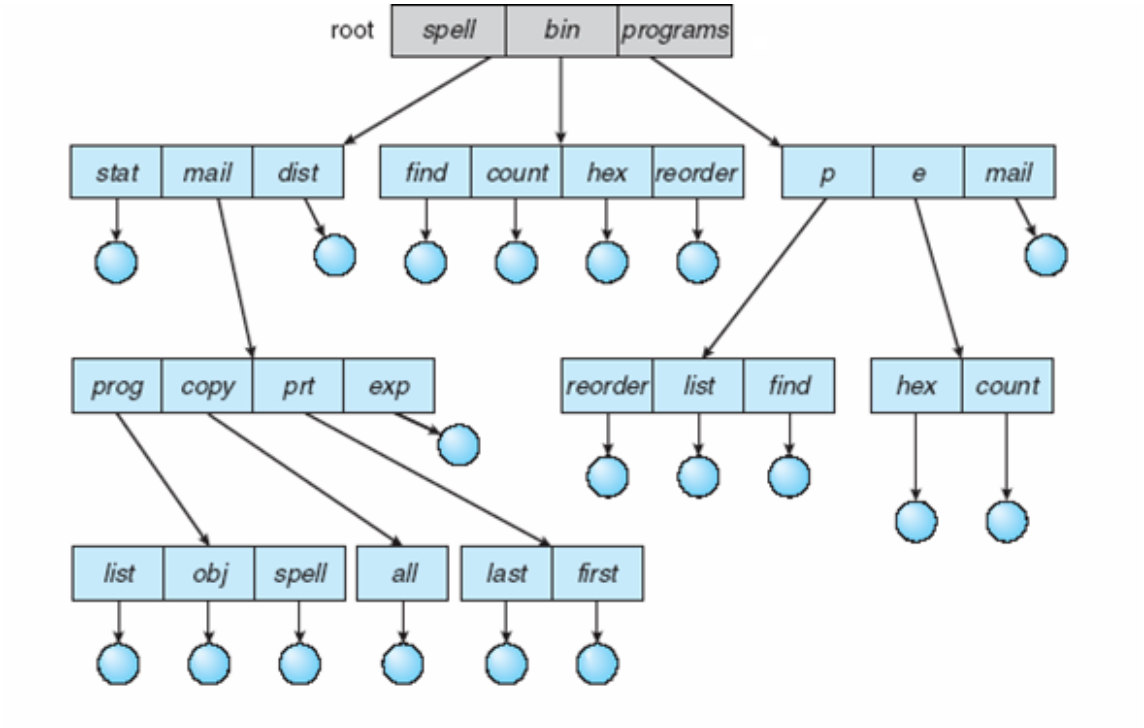
1. Users are isolated from one another and can't cooperate on the same task.

3. Tree Structured Directories

- 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.
- A directory 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.

Two types of path-names:

1. Absolute path-name: begins at the root.
2. Relative path-name: defines a path from the current directory.



How to delete directory?

1. To delete an empty directory: Just delete the directory.
2. To delete a non-empty directory:
   - First, delete all files in the directory.
   - If any subdirectories exist, this procedure must be applied recursively to them.

Advantage:

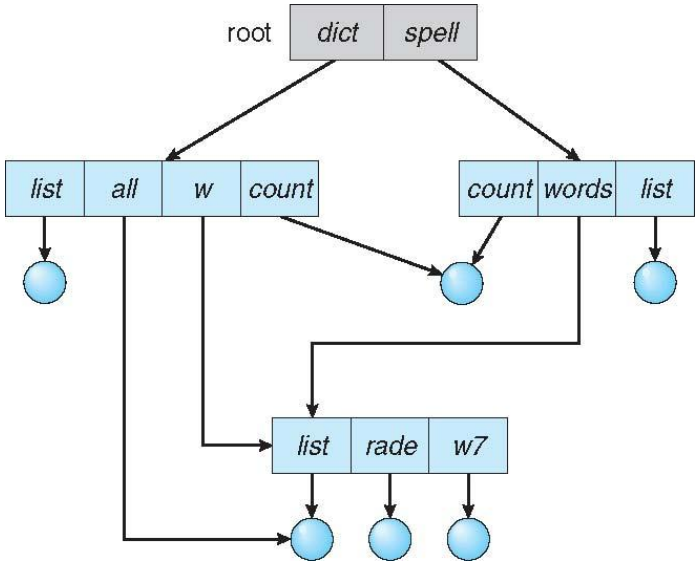
1. Users can be allowed to access the files of other users.

Disadvantages:

1. A path to a file can be longer than a path in a two-level directory.
2. Prohibits the sharing of files (or directories).

4. Acyclic Graph 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 is a graph with no cycles. It allows directories to share subdirectories and files



- The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

Two methods to implement shared-files (or subdirectories):

1. Create a new directory-entry called a link. A link is a pointer to another file (or subdirectory).
2. Duplicate all information about shared-files in both sharing directories.

Two problems:

1. A file may have multiple absolute path-names.
2. Deletion may leave dangling-pointers to the non-existent file.

Solution to deletion problem:

1. Use back-pointers: Preserve the file until all references to it are deleted.
2. With symbolic links, remove only the link, not the file. If the file itself is deleted, the link can be removed.

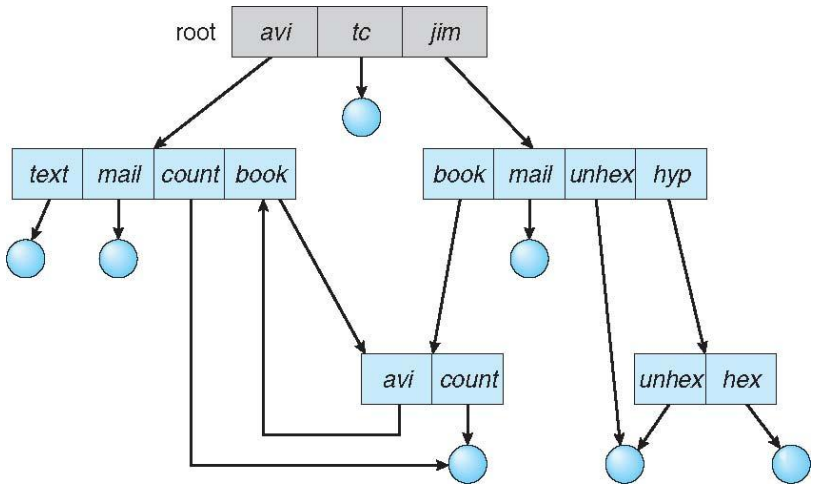
5. General Graph Directory

Problem: If there are cycles, we want to avoid searching components twice.
Solution: Limit the no. of directories accessed in a search.

Problem: With cycles, the reference-count may be non-zero even when it is no longer possible to refer to a directory (or file). (A value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted).
Solution: Garbage-collection scheme can be used to determine when the last reference has been deleted.

Garbage collection involves
1. First pass traverses the entire file-system and marks everything that can be accessed.

2. A second pass collects everything that is not marked onto a list of free-space



FILE SYSTEM MOUNTING

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

Mount Point: The location within the file structure where the file system is to be attached.

The mounting procedure:

- The operating system is given the name of the device and the mount point.
- 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
- The operating system notes in its directory structure that a file system is mounted at the specified mount point.

To illustrate file mounting, consider the file system shown in figure. The triangles represent sub-trees of directories that are of interest

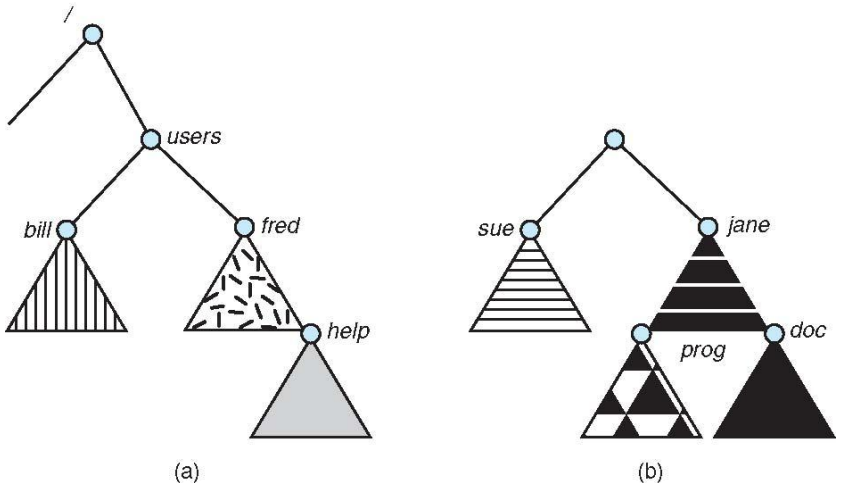
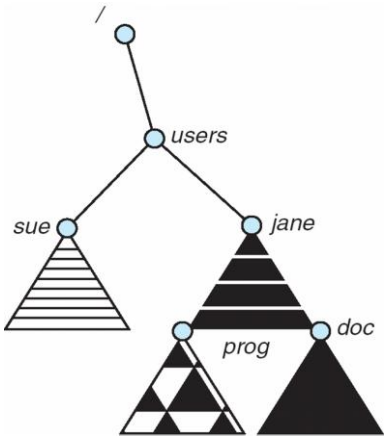


Figure 1(a) shows an existing file system, while Figure 1(b) shows an un-mounted volume residing on */device/dsk.* At this point, only the files on the existing file system can be accessed.



Above figure shows the effects of mounting the volume residing on */device/dsk* over */users.* If the volume is un-mounted, the file system is restored to the situation depicted in Figure 1.

FILE SHARING

- Sharing of files on multi-user systems is desirable.
- Sharing may be done through a protection scheme.
- On distributed systems, files may be shared across a network.
- Network File-system (NFS) is a common distributed file-sharing method.

Multiple Users

File-sharing can be done in 2 ways:

1. The system can allow a user to access the files of other users by default or
2. The system may require that a user specifically grant access.

To implement file-sharing, the system must maintain more file- & directory-attributes than on a single-user system.
Most systems use concepts of file owner and group.

1. Owner

- The user who may change attributes & grant access and has the most control over the file (or directory).
- Most systems implement owner attributes by managing a list of user-names and user IDs

2. Group

- The group attribute defines a subset of users who can share access to the file.

- Group functionality can be implemented as a system-wide list of group-names and group IDs.

- Exactly which operations can be executed by group-members and other users is definable by the file's owner.
- The owner and group IDs of files are stored with the other file-attributes and can be used to allow/deny requested operations.

Remote File Systems

It allows a computer to mount one or more file-systems from one or more remote-machines.

There are three methods:

1. Manually transferring files between machines via programs like ftp.
2. Automatically DFS (Distributed file-system): remote directories are visible from a local machine.
3. Semi-automatically via www (World Wide Web): A browser is needed to gain access to the remote files, and separate operations (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.

Client Server Model

- Allows clients to mount remote file-systems from servers.
- The machine containing the files is called the **server**. The machine seeking access to the files is called the **client**.
- A server can serve multiple clients, and A client can use multiple servers.
- The server specifies which resources (files) are available to which clients.
- A client can be specified by a network-name such as an IP address.

Disadvantage:

1. Client identification is more difficult.

• In UNIX and its NFS (network file-system), authentication takes place via the client networking information by default.

• Once the remote file-system is mounted, file-operation requests are sent to the server via the DFS protocol.

Distributed Information Systems

- Provides unified access to the information needed for remote computing.

- The DNS (domain name system) provides hostname-to-network address translations.
- Other distributed info. systems provide username/password space for a distributed facility

Failure Modes

- Local file-systems can fail for a variety of reasons such as failure of disk (containing the file-system), corruption of directory-structure & cable failure.
- Remote file-systems have more failure modes because of the complexity of network-systems.
- The network can be interrupted between 2 hosts. Such interruptions can result from hardware failure, poor hardware configuration or networking implementation issues.
- DFS protocols allow delaying of file-system operations to remote-hosts, with the hope that the remote-host will become available again.
- To implement failure-recovery, some kind of state information may be maintained on both the client and the server.

Consistency Semantics

- These represent an important criterion of evaluating file-systems that supports file-sharing. These 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 observed by other users.
- These semantics are typically implemented as code with the file-system.
- These are directly related to the process-synchronization algorithms.
- A successful implementation of complex sharing semantics can be found in the Andrew file-system (AFS).

UNIX Semantics

UNIX file-system (UFS) uses the following consistency semantics:

1. Writes to an open-file by a user are visible immediately to other users who have this file opened.
2. One mode of sharing allows users to share the pointer of current location into a file. Thus, the advancing of the pointer by one user affects all sharing users.

A file is associated with a single physical image that is accessed as an exclusive resource.

Contention for the single image causes delays in user processes.

Session Semantics

The AFS uses the following consistency semantics:

1. Writes to an open file by a user are not visible immediately to other users that have the same file open.
2. 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.

• A file may be associated temporarily with several (possibly different) images at the same time.

• Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay.
• Almost no constraints are enforced on scheduling accesses.

Immutable Shared Files Semantics

• Once a file is declared as shared by its creator, it cannot be modified.
• An immutable file has 2 key properties:
1. File-name may not be reused and
2. File-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

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.
- For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer.
- File owner/creator should be able to control what can be done and by whom.

Types of Access

Systems that do not permit access to the files of other users do not need protection. This is too extreme, so controlled-access is needed.

Following operations may be controlled:

1. ***Read:*** Read from the file.
2. ***Write:*** Write or rewrite the file.
3. ***Execute:*** Load the file into memory and execute it.
4. ***Append:*** Write new information at the end of the file.
5. ***Delete:*** Delete the file and tree its space for possible reuse.
6. ***List:*** List the name and attributes of the file.

Access Control

- Common approach to protection problem is to make access dependent on identity of user.
- Files can be associated with an ACL (access-control list) which specifies username and types of access for each user.

Problems:

1. Constructing a list can be tedious.
2. Directory-entry now needs to be of variable-size, resulting in more complicated space management.

Solution: These problems can be resolved by combining ACLs with an 'owner, group, universe' access control scheme

To reduce the length of the ACL, many systems recognize 3 classifications of users:

1. Owner: The user who created the file is the owner.
2. Group: A set of users who are sharing the file and need similar access is a group.
3. Universe: All other users in the system constitute the universe.

Other Protection Approaches

A password can be associated with each file.

Disadvantages:

1. The no. of passwords you need to remember may become large.
2. If only one password is used for all the files, then all files are accessible if it is discovered.
3. Commonly, only one password is associated with all of the user's files, so protection is all-or nothing.

In a multilevel directory-structure, we need to provide a mechanism for directory protection.
The directory operations that must be protected are different from the File-operations:

1. Control creation & deletion of files in a directory.
2. Control whether a user can determine the existence of a file in a directory.