# UNIT IV

**File System and Secondary Storage File Concept**

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 according to 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.

* *File Attributes*

A file has certain other attributes, which 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 those systems that support different types.
* **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.

* *File Operations*
* **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. The directory entry records the name of the file and the location in the file system, and possibly other information.
* **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 location of the file. 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 directory 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. A given process is usually only reading or writing a given file, and the current operation location is kept as a per-process current-file-position pointer. Both the read and write operations use this same pointer, saving space and reducing the system complexity.
* **Repositioning within a file:** The directory is searched for the appropriate entry, and the current-file-position is set to a given value. Repositioning within a file does not need to involve any actual I/O. 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.

These six basic operations certainly comprise the minimal set of required file operations. Other common operations include ***appending new*** *information* to the end of an existing file and ***renaming*** *an existing file. These primitive* operations may then be combined to perform other file operations. For instance, creating a *copy of a file, or copying the file to another I/O device, such as a* printer or a display, may be accomplished by creating a new file, and reading from the old and writing to the new.

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 used before that file is first used actively. The operating system keeps a small table containing information about all open files (the **open-file table). When a file operation is requested, the** file is specified via an index into this table, so no searching is required. When the file is no longer actively used, it is *closed by the process and the operating* system removes its entry in the open-file table.

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 open a file, the system must wait for the last file to close before removing the open-file table entry. This 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 to avoid having 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/O requests.
* *File Types*

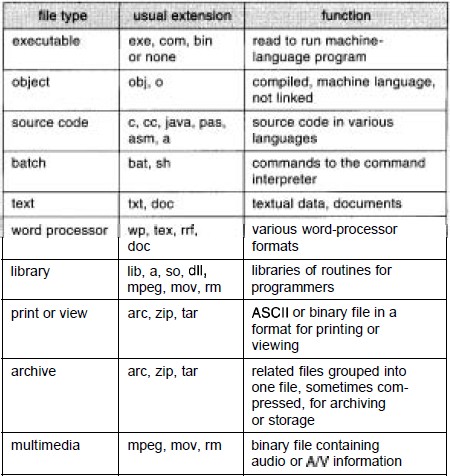
A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts-a name and an extension, usually separated by a period character.

Figure: Common file types.

* *File Structure*

File types also can be used to indicate the internal structure of the file. 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. If OS supports multiple file structures; the resulting size of OS is large. If the OS defines 5 different file structures, it needs to contain the code to support these file structures. All OS must support at least one structure that of an executable file so that the system is able to load and run programs.

**Internal File Structure:**

In UNIX OS, defines all files to be simply stream of bytes. Each byte is individually addressable by its offset from the beginning or end of the file. In this case, the logical record size is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks, say 512 bytes per block.

The logical record size, physical block size, packing determines how many logical records

are in each physical block. The packing can be done by the user’s application program or OS. A file may be considered a sequence of blocks. If each block were 512 bytes, a file of 1949 bytes would be allocated 4 blocks (2048 bytes). The last 99 bytes would be wasted. It is called internal fragmentation all file systems suffer from internal fragmentation, the larger the block size, the greater the internal fragmentation.

## 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. The following are the different access methods.

* *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.

The bulk of the operations on a file is reads and writes. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write 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 is depicted in Figure below.

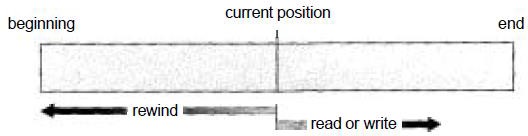
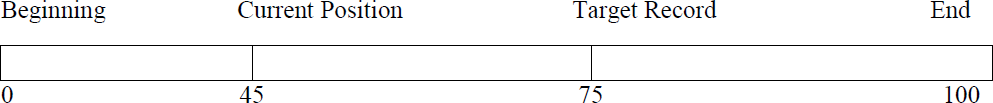


Figure: Sequential access file

Eg : A file consisting of 100 records, the current position of read/write head is 45th record, suppose we want to read the 75th record then, it access sequentially from 45, 46, 47 74,

75. So the read/write head traverse all the records between 45 to 75.



* *Direct Access*

Another method is **direct access (or relative access). A file is made up of fixedlength 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. A direct-access file allows arbitrary blocks to be read or written. 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.

* *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.

With large files, the index file itself may become too large to be keep in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files, which would point to the actual data items. The following figure shows a situation as implemented by VMS (Virtual Memory Storage) index and relative files.

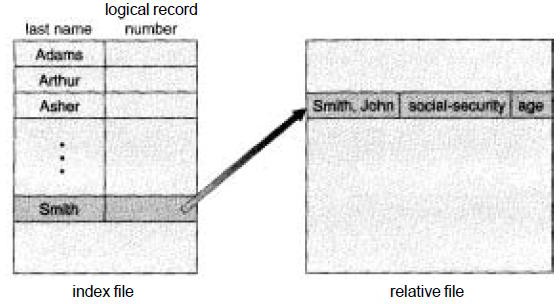


Figure: Example of relative and index files.

**Directory Structure**

Sometimes the file system consisting of millions of files, at that situation it is very hard to manage the files. To manage these files grouped these files and load one group into one partition. Each partition is called a directory. A directory structure provides a mechanism for organizing many files in the file system.

* ***Operations on the directories***
* **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 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, the name must be changeable 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:** 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. This saving often consists of copying all files to magnetic tape. This technique provides a backup copy in case of system failure or if the file is simply no longer in use. In this case, the file can be copied to tape, and the disk space of that file released for reuse by another file.
* ***Directory Structure:***

The most common schemes for defining the structure of the directory are:

## 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 (Figure below).

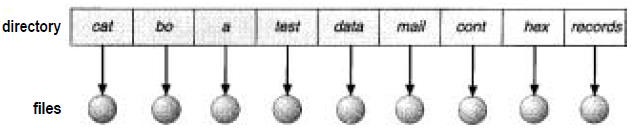
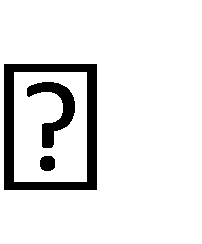


Figure: Single level directory

Advantages:

* Since it is a single directory, so its implementation is very easy.
* If the files are smaller in size, searching will become faster.
*  The operations like file creation, searching, deletion, updating is very easy in such a directory structure.

Disadvantages:

* There may chance of name collision because two files cannot have the same name.
* Searching will become time taking if the directory is large.
* The same type of files cannot be grouped together.

## Two-Level Directory

The disadvantage of single level directory is the confusion of files names between different users. The solution for this problem is to create a directory for each user as shown in figure below.

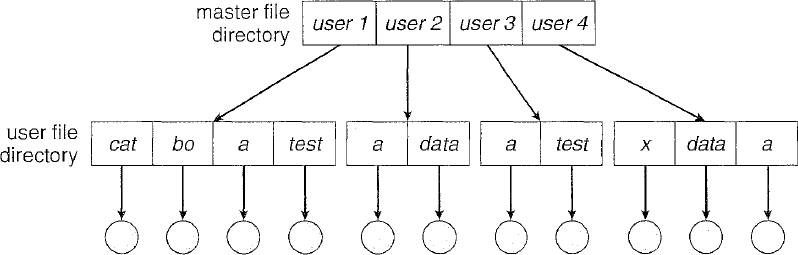


Figure: Two-level directory structure.

In the two-level directory structure, each user has her own **user file directory (UFD). Each**

**UFD has a similar structure, but 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.

## Advantages:

* We can give full path like /User-name/directory-name.
* Different users can have same directory as well as file name.
* Searching of files become more easy due to path name and user-grouping.

## Disadvantages:

* A user is not allowed to share files with other users.
* Searching will become time taking if the directory is large.
* Two files of the same type cannot be grouped together in the same user.

1. **Tree- Structured Directory**

The tree-structure allows user to create their own subdirectories and organize their files accordingly. The tree has a root directory. Every file in the systems has a unique path name. A path is the path from the root through all the subdirectories to a specified file. A directory contains a set of files and or sub-directories.

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.

Path names can be of two types: *absolute path names or relative path names.* 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 shown in the following figure, 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/spell/mail/prt/first.*

**Advantages:**

User can access other user’s files by specifying the path name.

User can create his own sub- directories.

Searching becomes very easy. We can use both absolute and relative paths.

**Disadvantages:**

Every file does not fit into the hierarchical model.

We cannot share files

It is inefficient, because accessing a file may go under multiple directories.

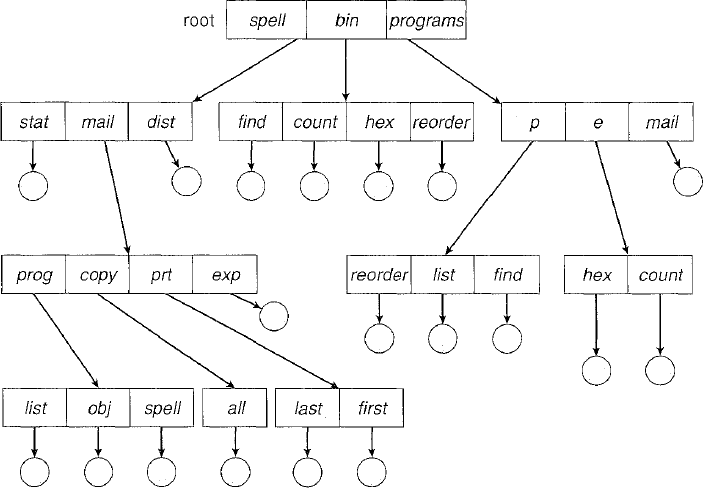


Figure: Tree-structured directory structure.

1. **Acyclic-Graph Directories**

A shared directory or file will exist in the file system in two or more places at once. A shared directory or file is not the same as two copies of the file. With a shared file there is only one actual field and any changes made by one person would be immediately visible to the other.

An acyclic graph allows directories to have shared sub-directories and files. The same file or sub- directory may be in two or more process exists in the file system at a time. An acyclic graph directory structure is more flexible than a simple structure but it is also more complex.

**Advantages:**

* We can share files.
* Searching is easy due to different-different paths
* Allow multiple directories to contain same file.

**Disadvantages:**

* We share the files via linking; in case of deleting it may create the problem.
* Need to be cautions of dangling pointers when files are deleted.

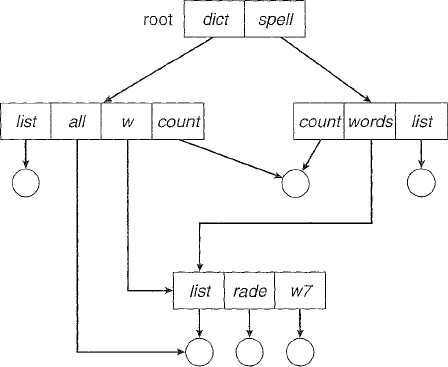


Figure: Acyclic-graph directory structure.

## General Graph Directory

When we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure (Figure below).

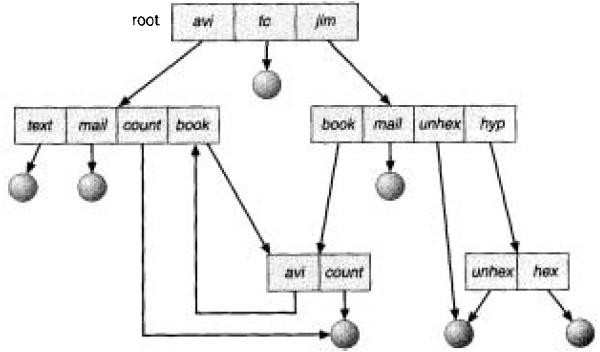


Figure: General graph directory.

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to

traverse the graph and to determine when there are no more references to a file. We want to avoid traversing shared sections of an acyclic graph twice, mainly for performance reasons. If we have just searched a major shared subdirectory for a particular file, without finding it, we want to avoid searching that subdirectory again; the second search would be a waste of time. If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance.

## 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 partitions, 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 location within the file structure at which to attach the file system (or mount point). Typically, a mount point is an empty directory at which the mounted file system will be attached.

To illustrate file mounting, consider the file system depicted in Figure below, where the triangles represent subtrees of directories that are of interest. In Figure (a), an existing file system is shown, while in Figure (b), an unmounted partition residing on */device/dsk is shown. At this point, only the* files on the existing file system can be accessed.

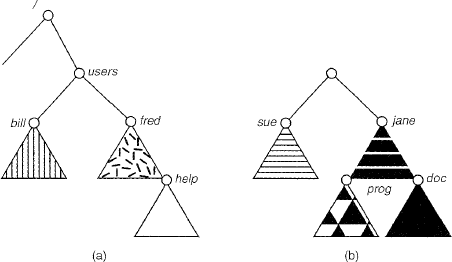


Figure: File system. (a) Existing. (b) Unmounted partition.

In the following Figure, the effects of the mounting of the partition residing on

*/device/dsk over /users are shown. If* the partition is unmounted, the file system is restored to

the situation depicted in the above figure.

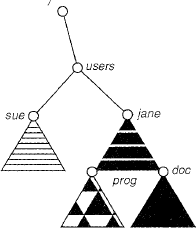
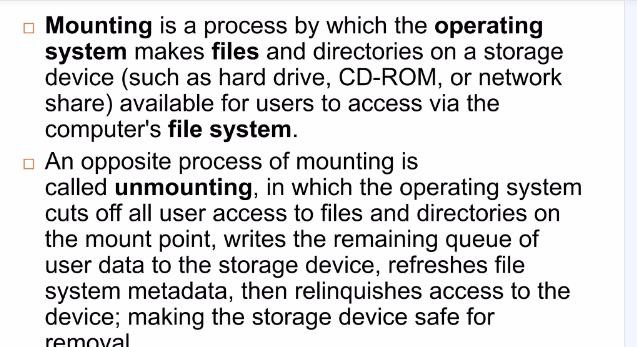


Figure: Mount point.



# File Sharing

* *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 either can allow a user to access the files of other users by default, or it may require that a user specifically grant access to the files. These are the issues of access control and protection.

Most systems have evolved to the concepts of file/directory ***owner (or user)*** *and* ***group.***

The owner is the user who may change attributes, grant access, and has the most control over the file or directory. The group attribute of a file is used to define a subset of users who may share access to the file.

Most systems implement owner attributes by managing a list of user names and associated **user identifiers (user IDS).** Likewise, group functionality can be implemented as a system-wide list of group names and **group identifiers.** Every user can be in one or more groups.

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 to 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.

* *Remote File Systems*

The advent of networks allowed communication between remote computers. Networking allows the sharing of resources spread within a campus or even around the world. One obvious resource to share is data, in the form of files. Through the evolution of network and file technology, file-sharing methods have changed. In the first implemented method, users manually transfer files between machines via programs like ftp. The second major method is a **distributed file system (DFS) in which remote directories are visible from the** 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. DFS involve a much tighter integration between the machine that is accessing the remote files and the machine providing the files.

1. **The Client-Server Model**

The machine containing the files is the *server, and the machine wanting 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. Files are usually specified on a partition or subdirectory level. A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client-server facility.

Clients can be specified by their network name or other identifier, such as *IP address,* but these can be **spoofed** (or imitated). An unauthorized client can spoof the server into deciding that it is authorized, and the unauthorized client could be allowed access. More secure solutions include secure authentication of the client to the server via encrypted keys. Unfortunately, with security comes many challenges, including ensuring compatibility of the client and server (they must use the same encryption algorithms) and secure key exchanges (intercepted keys could again allow unauthorized client access).

## Distributed Information Systems

To ease the management of client-server services, **distributed information systems,** also known as **distributed naming services, have been devised to provide** a unified access to the information needed for remote computing. **Domain name system (DNS) provides host-name- to-network-address translations for** the entire Internet (including the World Wide Web). Before DNS was invented and became widespread, files containing the same information were sent via email or ftp between all networked hosts. This methodology was not scalable.

Other distributed information systems provide *user name/password/user ID/group ID space for a distributed facility.*

## 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, or host adapter failure. User or systems-administrator failure can also cause files to be lost, or entire directories or partitions to be deleted. Many of these failures would cause a host to crash and an error condition to be displayed, and require human intervention to repair.

Some failures do not cause loss of data or loss of availability of data. **Redundant arrays of inexpensive disks (RAID) can prevent the loss of a disk** from resulting in the loss of data.

Remote file systems have more failure modes. By nature 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 the two hosts. This could be due to hardware failure or misconfiguration, or networking implementation issues at any of the involved hosts. Although some networks have built-in resiliency, including multiple paths between each host, many do not.

* *Consistency Semantics*

**Consistency semantics is an important criterion for evaluating any file system** that supports file sharing. It is a characterization of the system that specifies the semantics of multiple users accessing a shared file simultaneously. In particular, these semantics should specify when modifications of data by one user are observable by other users. The semantics are typically implemented as code with the file system.

* *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 at the same time.
  + 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.
* *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

simultaneously.

* + 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.
* *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, rather than the file being a container for variable information.

## Protection

When information is kept in a computer system, we want to keep it safe from physical damage (reliability) and improper access (protection).

* *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.**

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.

Other operations, such as renaming, copying, or editing the file, may also be controlled.

* *Access Control*

The most common approach to the protection problem is to make access dependent on the identity of the user. Various 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 the user name 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.

This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

* + Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
  + The directory entry, previously of fixed size, now needs to be of variable size, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list. 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 agroup, or work group.
* **Universe:** All other users in the system constitute the universe.
* *Other Protection Approaches*

Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled by a password. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file to only those users who know the password. This scheme, however, has several disadvantages. First, the number of passwords that a user needs to remember may become large, making the scheme impractical.