1 File-System Interface

- Since main memory is usually too small to accommodate all the data and programs permanently, the computer system must provide secondary storage to back up main memory.
- The file system provides the mechanism for on-line storage of and (multiple) access to both data and programs residing on the disks.
- A file is a collection of related information defined by its creator.
- *File Management System:* Bridges the gap between low-level disk organization (an array of blocks) and the user's views (a stream or collection of records) (mapped).
  - Some devices transfer a character or a block of characters at a time.
  - Some can be accessed only sequentially, others randomly.
  - Some transfer data synchronously, others asynchronously.
  - Some are dedicated, some shared.
  - They can be read-only or read-write.
- Also includes tools outside the kernel; formatting, recovery, defrag, consistency, backup utilities (system administration).
- In many ways, they are also the slowest major component of the computer.
- Files are managed by the OS. How they are *structured*, *named*, *accessed*, *used*, *protected*, and *implemented* are major topics in OS design.
- The file system consists of two distinct parts:
  1. **a collection of files**; each storing related data,
  2. **a directory structure**; which organizes and provides information about all the files in the system.
- Objectives for a file management system;
  - Provide a convenient naming system for files.
  - Provide a standardized set of I/O interface routines and provide access control for multiple users.

  - Guarantee that the data in the file are valid. Minimize or eliminate the potential for lost or destroyed data.
  - Optimize performance.
  - How do you keep one user from reading another's data?
  - How do you know which blocks are free?

1.1 File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.
- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- A file is a named collection of related information that is recorded on secondary storage, usually as a sequence of bytes, with two views:
  - Logical (programmer) view, as the users see it (how they are used and what properties they have.).
  - Physical (OS) view, as it actually resides on secondary storage.
- The information in a file is defined by its creator. 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.
- 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.

1.1.1 File Attributes

- When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file *example.c*, and another user might edit that file by specifying its name.

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 1: Some possible file attributes.

- The table of Fig. 1 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.
  - The first four attributes relate to the file's protection and tell who may access it and who may not.
  - The various times keep track of when the file was created, most recently accessed and most recently modified.
  - The current size tells how big the file is at present.
- A file's attributes vary from one OS to another but typically consist of these:
  - **Name**.

- **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**.
- **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.

- 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.

1.1.2 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.

- Six basic file operations. The OS can provide system calls to create, write, read, reposition, delete, and truncate files.
  - **Creating a file**. Two steps are necessary to create a file.
    1. Space in the file system must be found for the file.
    2. 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. 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. The system needs to keep a read pointer to the location in the file where the next read is to take place.
    * 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.

    * Both the read and write operations use this same pointer, saving space and reducing system complexity.
  - **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/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.

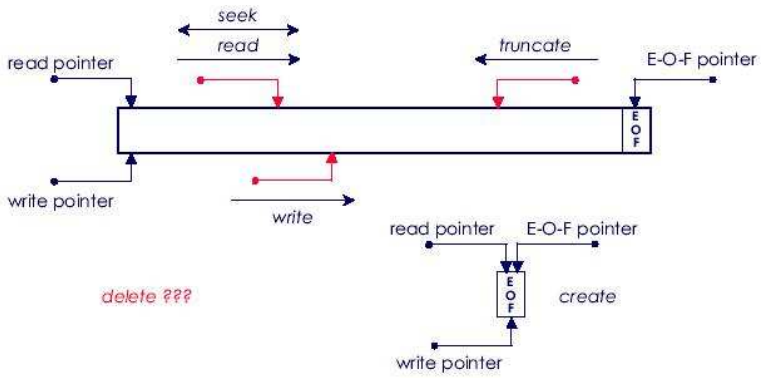

Figure 2: File operations.

- These six basic operations comprise the minimal set of required file operations.
- These primitive operations can then be combined to perform other file operations (i.e., copying).
- The OS 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.
  - When the file is no longer being actively used, it is closed by the process, and the OS removes its entry from the open-file table.

- Most systems require that the programmer open a file explicitly with the *open*() system call before that file can be used.
  - The *open*() operation takes a file name and searches the directory, copying the directory entry into the open-file table.
  - This call can also accept access-mode information (create, read-only, read-write, append-only, and so on). This mode is checked against the file's permissions. If the request mode is allowed, the file is opened for the process.
  - The *open*() system call typically returns a pointer to the entry in the open-file table. This pointer, not the actual file name, is used in all I/O operations.

- The implementation of the *open*() and *close*() operations is more complicated in an environment where several processes may open the file at the same time. This may occur in a system where several different applications open the same file at the same time.

- Typically, the OS uses two levels of internal tables:
  1. **A per-process table**. The per-process table tracks all files that a process has open. For instance, the current file pointer for each file is found here. Access rights to the file and accounting information can also be included.
  2. **A system-wide table**. 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.

- Typically, the open-file table also has an open count associated with each file to indicate how many processes have e 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.

- In summary, several pieces of information are associated with an open file.
  - **File pointer**.
  - **File-open count**.
  - **Disk location of 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 OS can allow or deny subsequent I/O requests.
- Some OSs provide facilities for locking an open file (or sections of a file). File locks allow one process to lock a file and prevent other processes from gaining access to it. File locks are useful for files that are shared by several processes -for example, a system log file that can be accessed and modified by a number of processes in the system.

1.1.3 An Example Program Using File System Calls

- A simple UNIX program that copies one file from its source file to a destination file (see Fig. 3). The program has minimal functionality and even worse error reporting.

```
copyfile abc xyz
```

- The copy loop. It starts by trying to read in 4 KB of data to buffer. It does this by calling the library procedure *read*, which actually invokes the *read system call*.
- The call to write outputs the buffer to the destination file.
- When the entire file has been processed, the first call beyond the end of file will return 0 to *rd_count* which will make it exit the loop. At this point the two files are closed and the program exits with a status indicating normal termination.

1.1.4 File Types

- A common technique for implementing file types is to include the type as part of the file name (see Fig. 4).

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);             /* syntax error if argc is not 3 */

    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY);    /* open the source file */
    if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE); /* create the destination file */
    if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break;           /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                  /* no error on last read */
        exit(0);
    else
        exit(5);                        /* error on last read */
}
```

Figure 3: A simple program to copy a file.

  - The name is split into two parts -a name and an extension, usually separated by a period character.
  - In this way, the user and the OS can tell from the name alone what the type of a file is.

- 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.

- Application programs also use extensions to indicate file types in which they are interested. For example, assemblers expect source files to have

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 4: Common file types.

an *.asm* extension, and the Microsoft Word word processor expects its files to end with a *.doc* extension.

- Because these extensions are not supported by the OS, they can be considered as "hints" to the applications that operate on them.
- Many file systems support names as long as 255 characters. Some file systems distinguish between upper and lower case letters (Case (in)sensitivity).
- Windows 95 and Windows 98 both use the MS-DOS file system, and thus inherit many of its properties, such as how file names are constructed. In addition, Windows NT and Windows 2000 support the MS-DOS file system and thus also inherit its properties. However, the latter two systems also have a native file system (NTFS) that has different properties (such as file names in Unicode).
- Consider the Mac OS X OS. In this system, each file has a type, such as TEXT (for text file) or APPL (for application).
  - Each file also has a creator attribute containing the name of the program that created it.
  - This attribute is set by the OS during the *create*() call, so its use is enforced and supported by the system.

- The UNIX system uses a crude **magic number** stored at the beginning of some files to indicate roughly the type of the file -executable program, batchfile (or shell script), PostScript file, and so on.
  - Not all files have magic numbers, so system features cannot be based solely on this information.
  - UNIX does not record the name of the creating program, either. UNIX does allow file-name-extension hints, but these extensions are neither enforced nor depended on by the OS (interpreted by tools); they are meant mostly to aid users in determining the type of contents of the file.
  - Extensions can be used or ignored by a given application, but that is up to the application's programmer.
  - In contrast, Windows is aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system (Interpreted by OS).
- UNIX also has *character* and *block special* files (Device Files).
  - Character special files are related to input/output and used to model serial I/O devices such as terminals, printers, and networks.
  - Block special files are used to model disks.
- Other files are binary files, which just means that they are not ASCII files. Usually, they have some internal structure known to programs that use them (see Fig. 5).
- Every OS must recognize at least one file type; its own executable file. A simple executable binary file taken from a version of UNIX is seen in Fig. 5a .
  - Although technically the file is just a sequence of bytes, the operating system will only execute a file if it has the proper format.
  - It has five sections: header, text, data, relocation bits, and symbol table.
  - The header starts with a so-called magic number, identifying the file as an executable file (to prevent the accidental execution of a file not in this format).
  - Then come the sizes of the various pieces of the file, the address at which execution starts, and some flag bits.

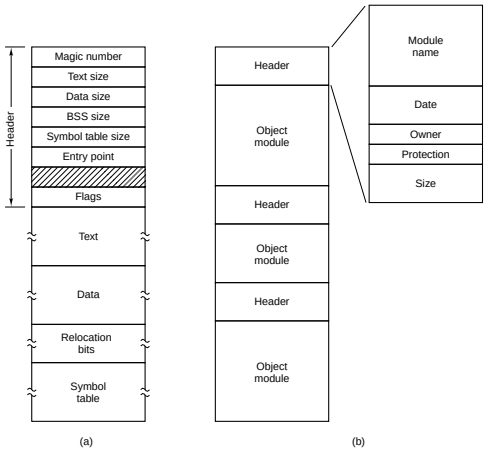

Figure 5: (a) An executable file. (b) An archive.

  - Beyond this header, executable files are typically divided into subsections (the text and data of the program itself).
  - Try the following commands:

    ```
    readelf -S exe_file
    objdump -h exe_file
    ```

- Second example of a binary file is an archive, also from UNIX (see Fig. 5b).
  - It consists of a collection of library procedures (modules) compiled but not linked.
  - Each one is prefaced by a header telling its name, creation date, owner, protection code, and size.

1.1.5 Internal File Structure

- Files can be structured in any of several ways. Three common possibilities are depicted in Fig. 6.
  - **Stream of Bytes**. The file in Fig. 6a is an *unstructured* sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Both UNIX and Windows use this approach.
  - **Records**. The first step up in structure is shown in Fig. 6b. A file is a sequence of *fixed-length records*, each with some internal structure.

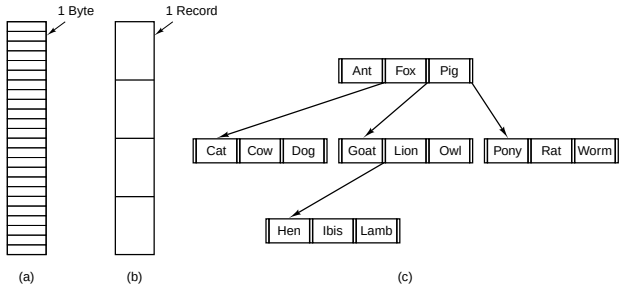

Figure 6: Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

  - **Tree of Records**. The third kind of file structure is shown in Fig. 6c. In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record.

- Internally, locating an offset within a file can be complicated for the OS.

- Disk systems typically have a well-defined block size determined by the size of a sector. All disk I/O 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. Packing a number of logical records into physical blocks is a common solution to this problem.

- For example, the UNIX OS defines all files to be simply streams 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- as necessary.

- The file may be considered to be a sequence of blocks. All the basic I/O functions operate in terms of blocks.

- Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. If each block were 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted.

- The waste incurred to keep everything in units of blocks (instead of bytes) is **internal fragmentation**. All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

1.2 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.

1.2.1 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 beginning current position 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).

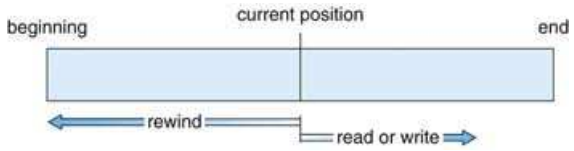

Figure 7: Sequential-access file.

- Sequential access, which is depicted in Fig. 7, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

1.2.2 Direct (Random) Access

- Another method is **direct access** (or **relative access**).
- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order. 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.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.
- Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type.
- For the direct-access method, the file operations must be modified to include the block number as a parameter.
- The block number provided by the user to the OS is normally 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 actual absolute disk address of the block may be 14703 for the first block and 3192 for the second.
- The use of relative block numbers allows the OS to decide where the file should be placed (called the allocation problem) and helps *to prevent the user from accessing portions of the file system that may not be part of her file.*
- We can easily simulate sequential access on a direct-access file by simply keeping a variable *cp* that defines our current position, as shown in Fig. 8. Simulating a direct-access file on a sequential-access file, however, is extremely inefficient.
- Modern OSs have all their files are automatically random access.

1.2.3 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.

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

Figure 8: Simulation of sequential access on a direct-access file.

  - To find a record in the file, we first search the index to learn exactly which block contains the desired record
  - and then use the pointer to access the file directly and to find the desired record.

- This structure allows us to search a large file doing little I/O. But, with large files, the index file itself may become too large to be kept 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.
  - For example, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index. 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.
  - The secondary index blocks point to the actual file blocks. 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 (see Fig. 9)

1.3 Directory Structure

The file systems of computers, then, can be extensive. 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.

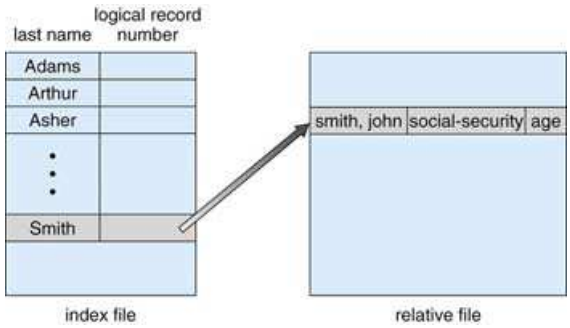

Figure 9: Example of index and relative files.

1.3.1 Storage Structure

- Sometimes, 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. We simply refer to a chunk of storage that holds a file system 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**.
- The device directory (more commonly known simply as a directory) records information-such as name, location, size, and type-for all files on that volume. Figure 10 shows a typical file-system organization.

1.3.2 Directory Overview

- To keep track of files, file systems normally have directories or folders. Usually, *a directory is itself a file.*
- The directory can be viewed as a symbol table that translates file names into their directory entries.
- A typical directory entry contains information (attributes, location, ownership) about a file.

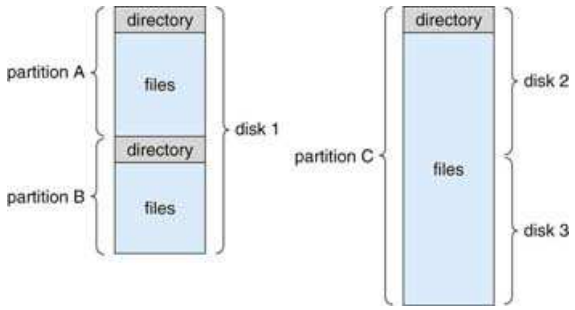

Figure 10: A typical file-system organization.

- We want to be able
  - to insert entries,
  - to delete entries,
  - to search for a named entry,
  - to list all the entries in the directory.
- 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.
  - **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 (backup copy).

1.3.3 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 (see Fig. 11).

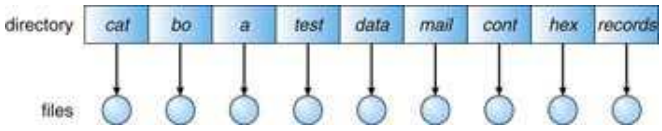

Figure 11: Single-level directory.

- On early personal computers, this system was common, in part because there was only one user. The world's first supercomputer, the CDC 6600, also had only a single directory for all files, even though it was used by many users at once.
- A single-level directory has significant limitations, 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.
- 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.

1.3.4 Two-Level Directory

- The standard solution to limitations of single-level directory is to create a separate directory for each user.
- 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 (see Fig. 12).
- when a user refers to a particular file, only his own UFD is searched (create a file, delete a file?).

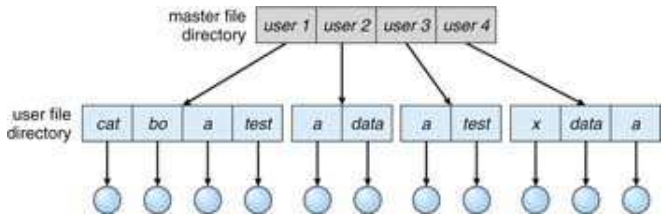

Figure 12: Two-level directory structure.

- 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.
- 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. To name a file uniquely, a user must know the path name of the file desired.
- Additional syntax is needed to specify the volume of a file. For instance, in MS-DOS a volume is specified by a letter followed by a colon. Thus, a file specification might be

  ```
  C:\userb\test
  ```

1.3.5 Tree-Structured Directories

- Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height (see Fig. 13).

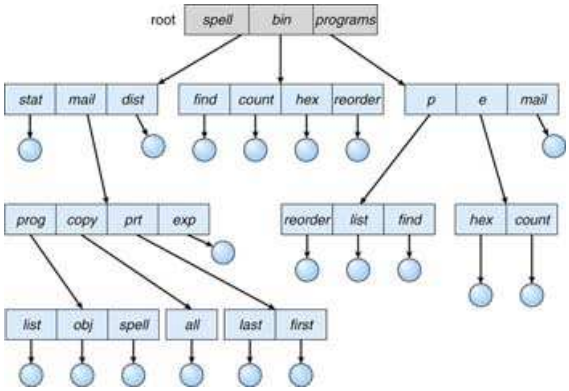

Figure 13: Tree-structured directory structure.

- 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.
- 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),
  - as a subdirectory (1).
- Path names can be of two types: **absolute** and **relative**
  1. An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path.
  2. A relative path name defines a path from the current directory.
- With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users.
  - For example, user B can access a file of user A by specifying its path names.
  - User B can specify either an absolute or a relative path name.
  - Alternatively, user B can change her current directory to be user A's directory and access the file by its file names.

1.3.6 Acyclic-Graph Directories

- The acyclic graph is a natural generalization of the tree-structured directory scheme.
- 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** (a graph with no cycles) allows directories to share subdirectories and files (see Fig. 14).

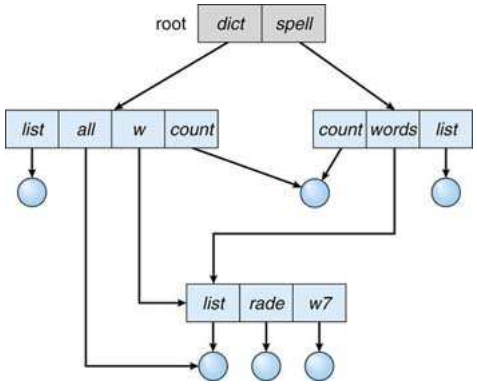

Figure 14: Acyclic-graph directory structure.

- The same file or subdirectory may be in two different directories.
- 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.
- 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.

- 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.
- We resolve the link by using that path name to locate the real file. Links are easily identified by their format in the directory entry and are effectively named indirect pointers.
- Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal.
- A link is clearly different from the original directory entry; thus, the two are not equal. A major problem with duplicate directory entries is maintaining consistency when a file is modified.
- Several problems must be considered carefully for an acyclic-graph directory structure.
  - A file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file.
  - 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, these dangling pointers may point into the middle of other files.
- In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle.
  - The deletion of a link need not affect the original file; only the link is removed.
  - If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling.
- We can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist and can fail to resolve the link name; the access is treated just as with any other illegal file name.

- In the case of UNIX, symbolic links are left when a file is deleted, and it is up to the user to realize that the original file is gone or has been replaced. Microsoft Windows (all flavours) uses the same approach.
- Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. The trouble with this approach is the variable and potentially large size of the file-reference list.
- However, we really do not need to keep the entire list -we need to keep only a count of the number of references.
  - Adding a new link or directory entry increments the reference count;
  - Deleting a link or entry decrements the count.
  - When the count is 0, the file can be deleted; there are no remaining references to it.
- The UNIX OS uses this approach for non-symbolic links (or **hard links**), keeping a reference count in the file information block (or **inode**).
- Most OSs that support a hierarchical directory system have two special entries in every directory, "." and "..", generally pronounced "dot" and "dotdot". Dot refers to the current directory; dotdot refers to its parent. To see how these are used, consider the UNIX file tree of Fig. 15.

1.4 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.
- The mount procedure is straightforward. The OS 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. Next, the OS verifies that the device contains a valid file system.
- Finally, the OS notes in its directory structure that a file system is mounted at the specified mount point.

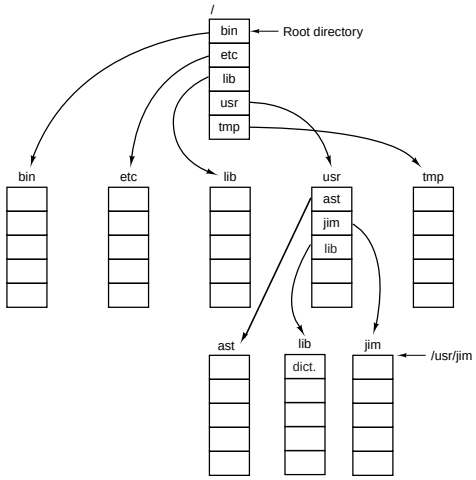

Figure 15: A UNIX directory tree.

- This scheme enables the OS to traverse its directory structure, switching among file systems as appropriate.
- To illustrate file mounting, consider the file system depicted in Fig. 16, where the triangles represent sub-trees of directories that are of interest. At this point, only the files on the existing file system can be accessed.

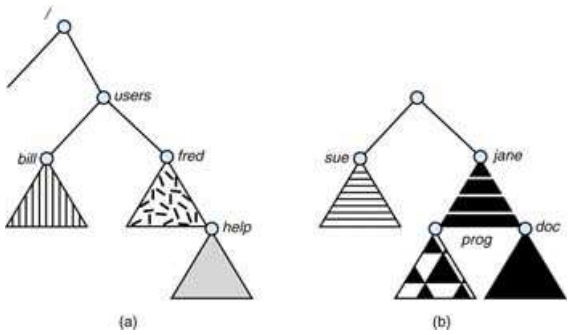

Figure 16: File system. (a) Existing system. (b) Unmounted volume.

- Figure 17 shows the effects of mounting the volume residing on $/dev/disk$ over $/users$.

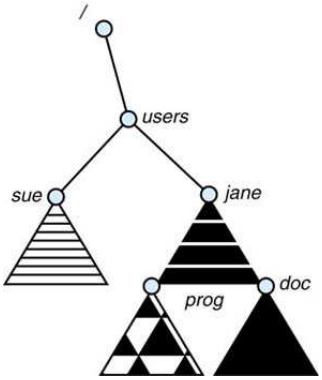

Figure 17: Mount point.

1.5 File Sharing

We explored the motivation for file sharing and some of the difficulties involved in allowing users to share files. Such file sharing is very desirable for users who want to collaborate and to reduce the effort required to achieve a computing goal.

1.5.1 Multiple Users

- When an OS accommodates multiple users, the issues of file sharing, file naming, and file protection become pre-eminent.
- To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system.
- 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, and all other users can execute another subset of operations.
- 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 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.

1.6 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 (copy disk files to tape).
- File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost.

1.6.1 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.
  - Alternatively, we could provide free access with no protection.
- Both approaches are too extreme for general use. What is needed is **controlled access**.
- Several different types of operations may be controlled:
  - **Read**.
  - **Write**.
  - **Execute**. Load the file into memory and execute it.
  - **Append**. Write new information at the end of the file.

  - **Delete**.
  - **List**. List the name and attributes of the file.

- Other operations, such as renaming, copying, and editing the file, may also be controlled. These higher-level functions may be implemented by a system program that makes lower-level system calls.

1.6.2 Access Control

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

- 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.

- 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 a group, or work group.
  - **Universe**. All other users in the system constitute the universe.

- With this more limited protection classification, only three fields are needed to define protection. Often, each field is a collection of bits, and each bit either allows or prevents the access associated with it.

- For example, the UNIX system defines three fields of 3 bits each$-rwx$, where r controls read access, w controls write access, and x controls execution.
- In this scheme, nine bits per file are needed to record protection information.