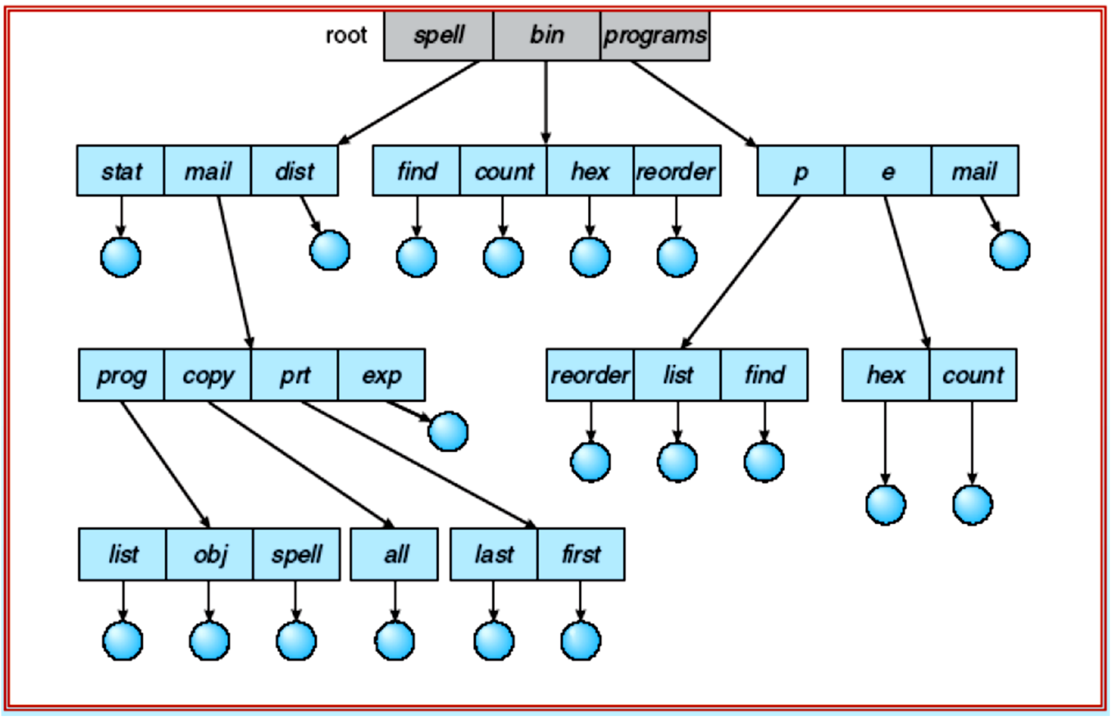
Chapter 3
File System Interface

This chapter will discuss the following concepts:



3.1 File Concept

A file is a named collection of related information that is recorded on secondary storage. 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**. 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.

3.1.1 File Attributes

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.

3.1.2 File Operations

The operating system can provide system calls to perform each of the following six basic 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.
- **Writing a file:** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- **Reading a file:** To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated.
  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.
- **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 tile be reset to length zero and its file space released.

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. When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table,

3.1.3 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 3.1). In this way, the user and the operating system can tell from the name alone what the type of a file is. For example, most operating systems allow users to specify file names as a sequence of characters followed by a period and terminated by an extension of additional characters. File name examples include **(resume.doc**, **Scrver.java**, and **ReaderThread.c**). Only a file with a **.com**, **.exe**, or **.bat** extension can be executed.

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	Source code in various languages
Batch	bat, sh	Commands to 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, rar	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 3.1 Common file types.

3.2 Access Methods

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

3.2.1 Sequential Access

The simplest access method is sequential access. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

A read operation—read next—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write operation—write next—appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning. Sequential access, which is depicted in Figure 3.2, is based on a **tape model** of a file and works as well on sequential-access devices.

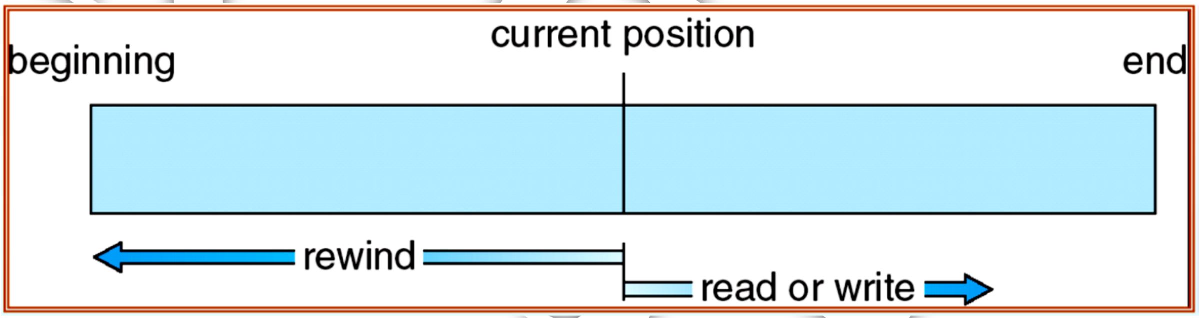
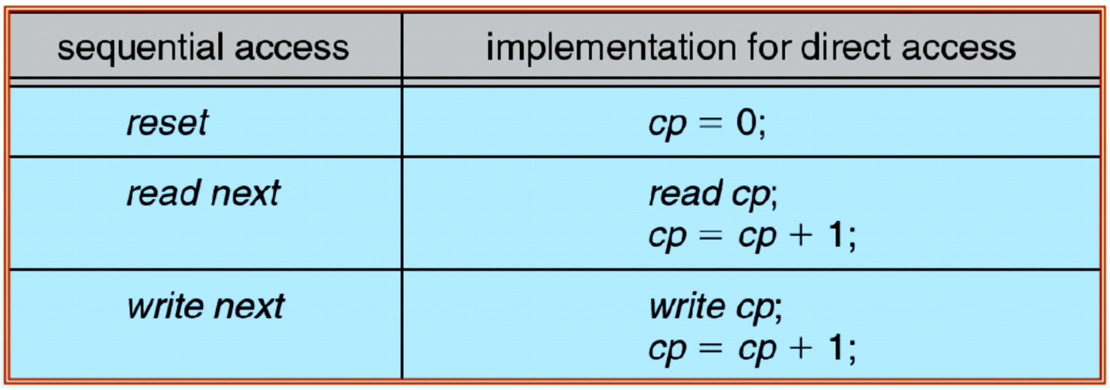


Figure 3.2 Sequential-access file.

3.2.2 Direct 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. The direct-access method is based on a **disk model** of a file. For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

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. 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 Figure 3.3.



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

Figure 3.3 Simulation of sequential access on a direct-accsss file.

3.2.3 Other Access Methods

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an **index** for the file. The index, like an index in the back of a book, contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record. From this search, we learn exactly which block contains the desired record and access that block. This structure allows us to search a large file doing little I/O.

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. Figure 3.4 shows a similar situation as implemented by index and relative files.

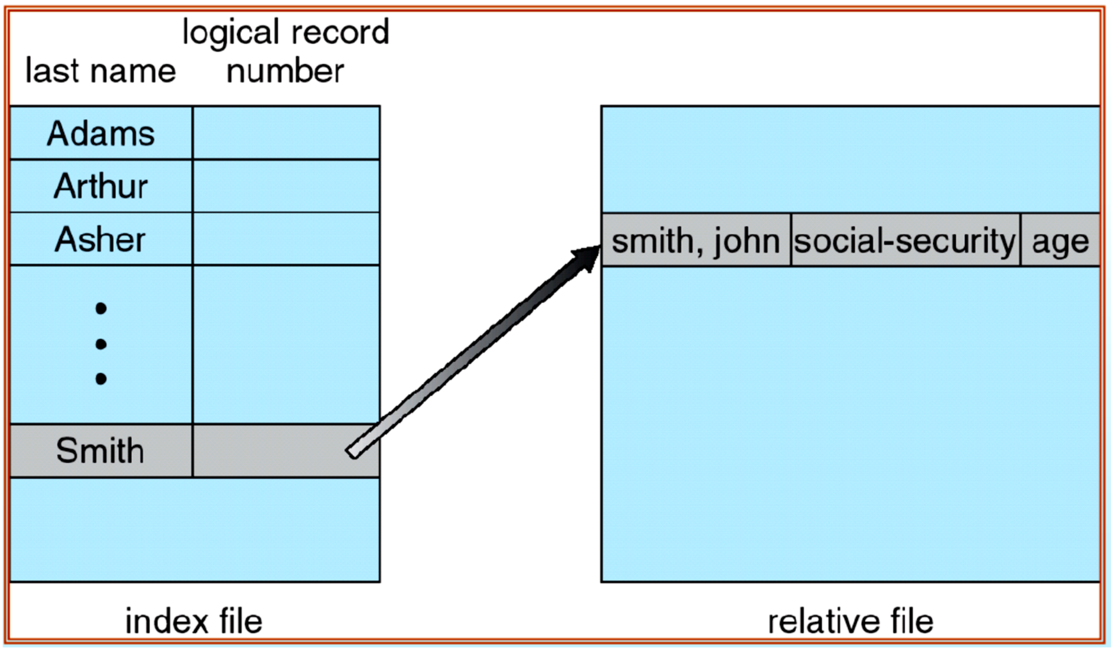


Figure 3.4 Example of index and relative files.

3.3 Directory Structure

3.3.1 Storage Structure

A disk (or any storage device that is large enough) can be used in its entirety for a file system. Sometimes, though, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things, such as swap space or unformatted (**raw**) disk space. These parts are known variously as **partitions, slices**, or (in the IBM world) **minidisks**.

A file system can be created on each of these parts of the disk. The parts can also be combined to form larger structures known as **volumes**, and file systems can be created on these as well. Volumes can also store multiple operating systems, allowing a system to boot and run more than one.

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 3.5 shows a typical file-system organization.

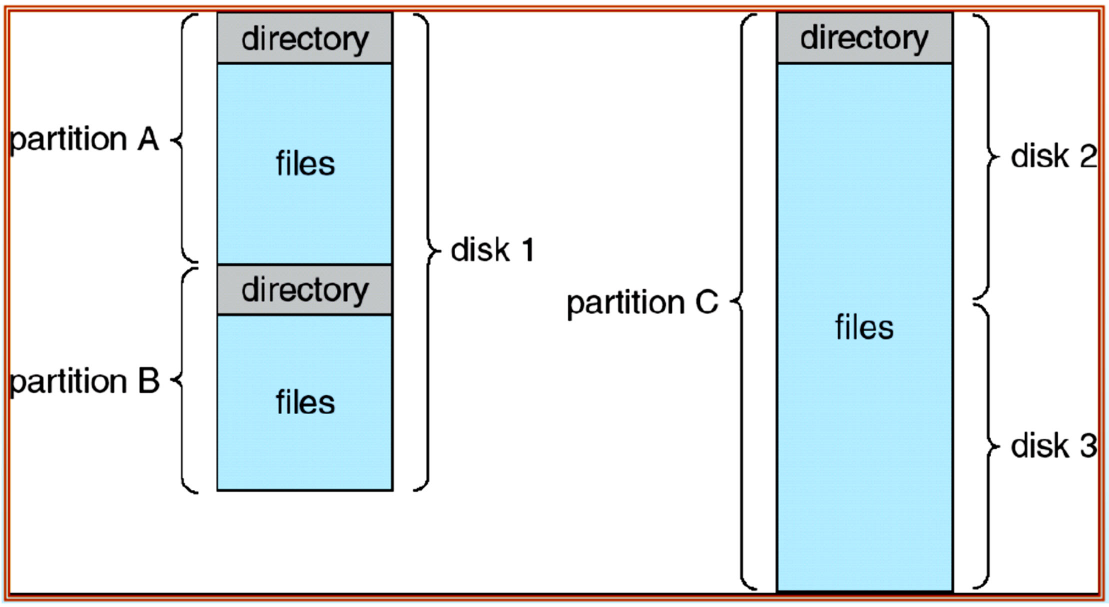


Figure 3.5 typical file-system organization.

3.3.2 Directory Overview

The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, we see that the directory itself can be organized in many ways. 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:**
- **Create a file:**
- **Delete a file:**
- **List a directory:**

- **Rename a file:**
- **Traverse the file system:** We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape. This technique provides a **backup** copy in case of system failure.

In the following sections, we describe the most common schemes for defining the logical structure of a directory.

3.3.3 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 (Figure 3.6).

A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have **unique names**. If two users call their data file test, then the unique-name rule is violated.

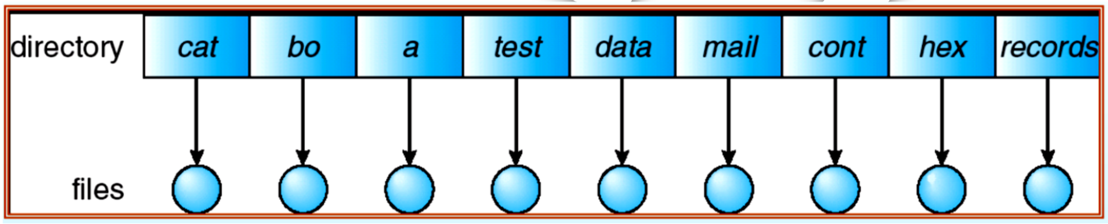


Figure 3.6 Single-level directory.

3.3.4 Two-Level Directory

As we have seen, a single-level directory often leads to confusion of file names among different users. The standard solution is to create a separate directory for each user.

In the two-level directory structure, each user has 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 (Figure 3.7). 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.

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.

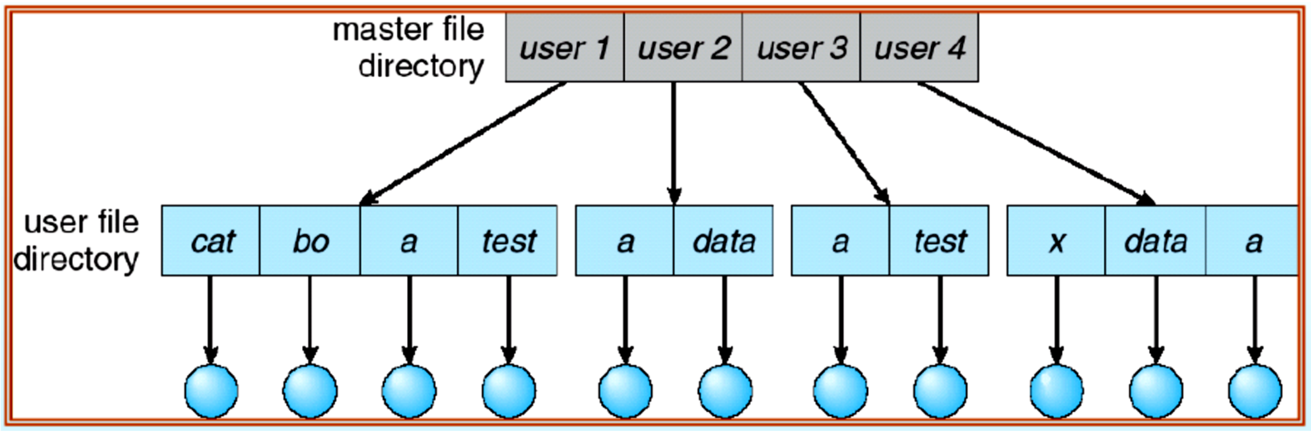


Figure 3.7 Two-level directory structure.

3.3.5 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 (Figure 3.8). 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 (or subdirectory) contains a set of files or subdirectories. All directories have the same internal format.

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.

Path names can be of two types: **absolute** and **relative**. An **absolute path name** begins at the **root** and follows a path down to the specified file, giving the directory names on the path. A **relative path name** defines a path from the current directory. For example, in the tree-structured file system of Figure 3.8, if the current directory is root/spell/mail, then the relative path name prt/first refers to the same file as does the absolute path name root/spcll/mail/prt/first.

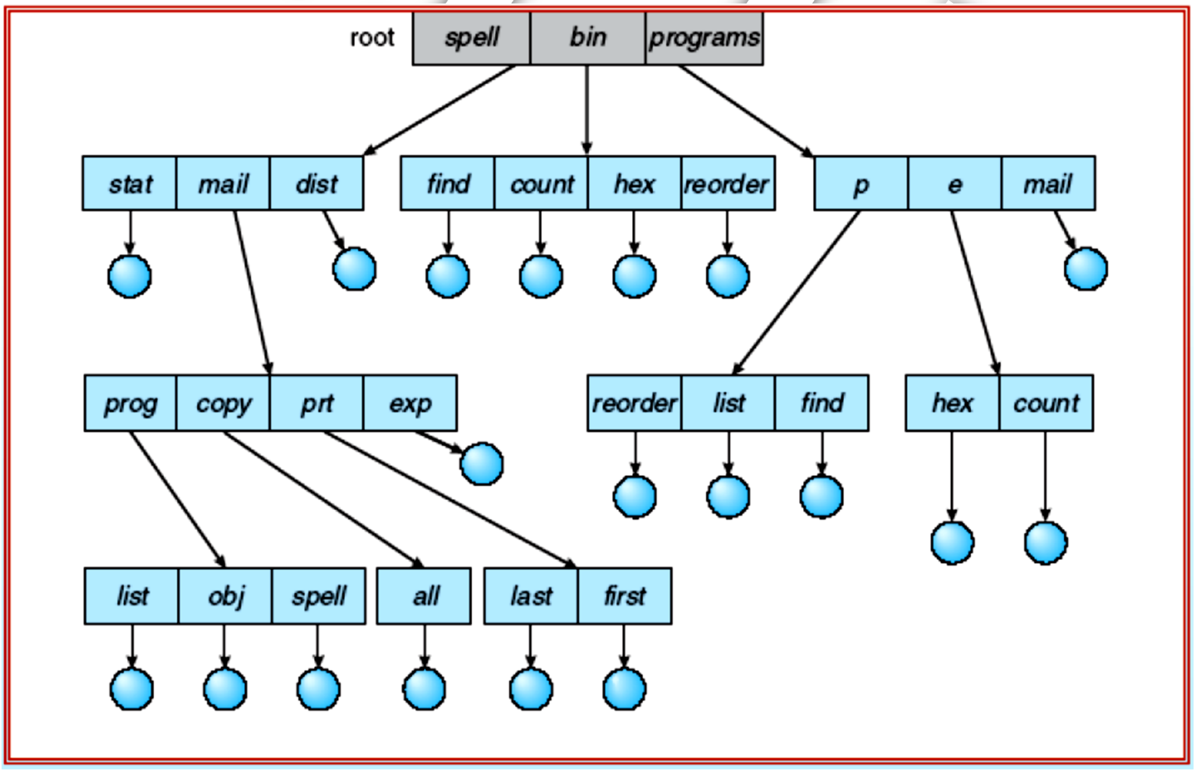


Figure 3.8 Tree-structured directory structure.

3.3.6 Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. The common subdirectory should be shared. **A shared directory** or file will exist in the file system in two (or more) places at once.

Acyclic graph —that is, a graph with no cycles—allows directories to share subdirectories and files (Figure 3.9). 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. Sharing is particularly important for subdirectories; a new file created by one person will automatically appears in all the shared subdirectories.

Several problems must be considered carefully. A file may now have multiple absolute path names. Another problem involves deletion. When can the space allocated to a shared file be de-allocated 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.

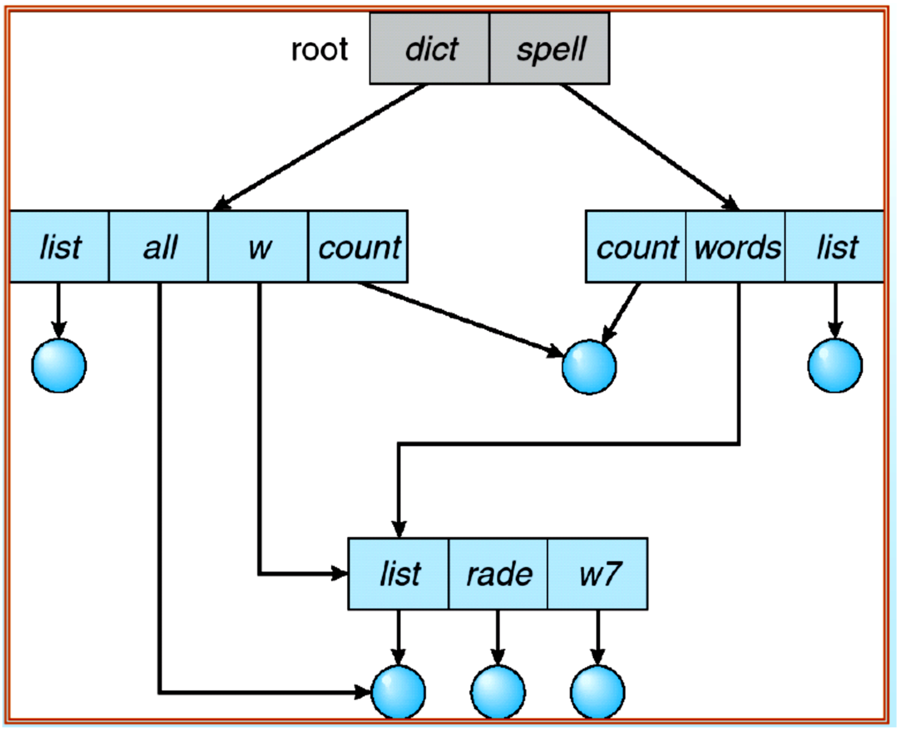


Figure 3.9 Acyclic-graph directory structure.

3.3.7 General Graph Directory

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. When we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure (Figure 3.10).

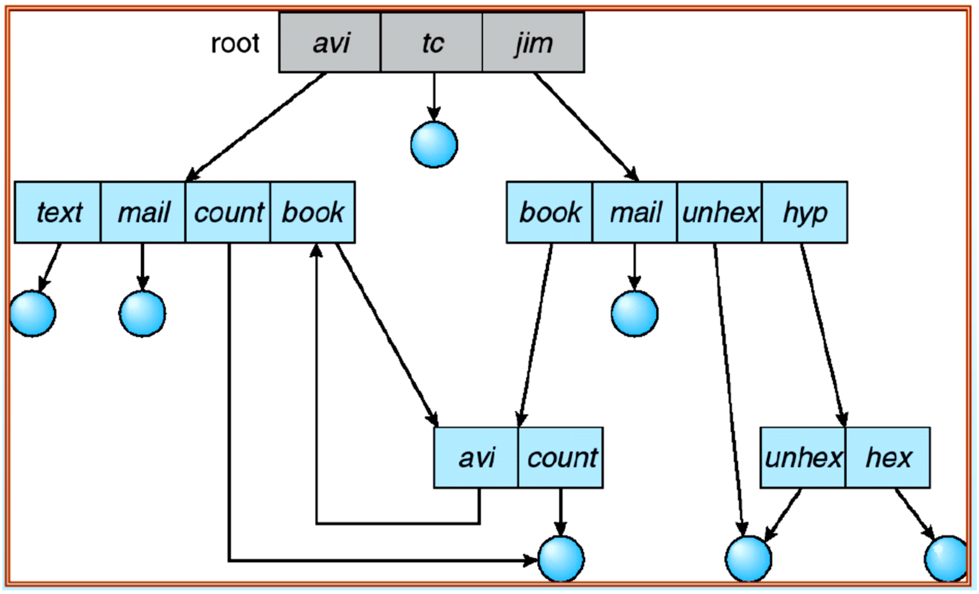


Figure 3.10 General graph directory.

3.4 Protection

When information is stored in a computer system, we want to keep it safe from physical damage (**reliability**) and improper access (**protection**). Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multiuser system, however, other mechanisms are needed.

3.4.1 Types of Access

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 tree its space for possible reuse.
- **List:** List the name and attributes of the file.

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

3.4.2 Access Control

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

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

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.

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.

The most common recent approach is to combine access-control lists with the more general **owner**, **group**, and **universe** access control scheme just described.

3.4.3 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 in the same way. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file.

End of chapter 3