CHAPTER 12

FILE MANAGEMENT

In most applications, the file is the central element. With the exception of real-time applications and some other specialized applications, the input to the application is by means of a file, and in virtually all applications, output is saved in a file for long-term storage and for later access by the user and by other programs.

Files have a life outside of any individual application that uses them for input and/or output. Users wish to be able to access files, save them, and maintain the integrity of their contents. To aid in these objectives, virtually all operating systems provide file management systems. Typically, a file management system consists of system utility programs that run as privileged applications. However, at the very least, a file management system needs special services from the operating system; at the most, the entire file management system is considered part of the operating system. Thus, it is appropriate to consider the basic elements of file management in this book.

We begin with an overview, followed by a look at various file organization schemes. Although file organization is generally beyond the scope of the operating system, it is essential to have a general understanding of the common alternatives to appreciate some of the design tradeoffs involved in file management. The remainder of this chapter looks at other topics in file management.

12.1 OVERVIEW

Files and File Systems

From the user's point of view, one of the most important parts of an operating system is the file system. The file system provides the resource abstractions typically associated with secondary storage. The file system permits users to create data collections, called files, with desirable properties, such as

- **Long-term existence:** Files are stored on disk or other secondary storage and do not disappear when a user logs off.
- **Sharable between processes:** Files have names and can have associated access permissions that permit controlled sharing.
- **Structure:** Depending on the file system, a file can have an internal structure that is convenient for particular applications. In addition, files can be organized into hierarchical or more complex structure to reflect the relationships among files.

Any file system provides not only a means to store data organized as files, but a collection of functions that can be performed on files. Typical operations include the following:

- **Create:** A new file is defined and positioned within the structure of files.
- **Delete:** A file is removed from the file structure and destroyed.
- **Open:** An existing file is declared to be "opened" by a process, allowing the process to perform functions on the file.
- **Close:** The file is closed with respect to a process, so that the process no longer may perform functions on the file, until the process opens the file again.

- **Read:** A process reads all or a portion of the data in a file.
- **Write:** A process updates a file, either by adding new data that expands the size of the file or by changing the values of existing data items in the file.

Typically, a file system maintains a set of attributes associated with the file. These include owner, creation time, time last modified, access privileges, and so on.

File Structure

Four terms are in common use when discussing files:

- Field
- Record
- File
- Database

A **field** is the basic element of data. An individual field contains a single value, such as an employee's last name, a date, or the value of a sensor reading. It is characterized by its length and data type (e.g., ASCII string, decimal). Depending on the file design, fields may be fixed length or variable length. In the latter case, the field often consists of two or three subfields: the actual value to be stored, the name of the field, and, in some cases, the length of the field. In other cases of variable-length fields, the length of the field is indicated by the use of special demarcation symbols between fields.

A **record** is a collection of related fields that can be treated as a unit by some application program. For example, an employee record would contain such fields as name, social security number, job classification, date of hire, and so on. Again, depending on design, records may be of fixed length or variable length. A record will be of variable length if some of its fields are of variable length or if the number of fields may vary. In the latter case, each field is usually accompanied by a field name. In either case, the entire record usually includes a length field.

A **file** is a collection of similar records. The file is treated as a single entity by users and applications and may be referenced by name. Files have file names and may be created and deleted. Access control restrictions usually apply at the file level. That is, in a shared system, users and programs are granted or denied access to entire files. In some more sophisticated systems, such controls are enforced at the record or even the field level.

Some file systems are structured only in terms of fields, not records. In that case, a file is a collection of fields.

A **database** is a collection of related data. The essential aspects of a database are that the relationships that exist among elements of data are explicit and that the database is designed for use by a number of different applications. A database may contain all of the information related to an organization or project, such as a business or a scientific study. The database itself consists of one or more types of files. Usually, there is a separate database management system that is independent of the operating system, although that system may make use of some file management programs.

Users and applications wish to make use of files. Typical operations that must be supported include the following:

- `Retrieve_All`: Retrieve all the records of a file. This will be required for an application that must process all of the information in the file at one time. For example, an application that produces a summary of the information in the file would need to retrieve all records. This operation is often equated with the term *sequential processing*, because all of the records are accessed in sequence.
- `Retrieve_One`: This requires the retrieval of just a single record. Interactive, transaction-oriented applications need this operation.
- `Retrieve_Next`: This requires the retrieval of the record that is "next" in some logical sequence to the most recently retrieved record. Some interactive applications, such as filling in forms, may require such an operation. A program that is performing a search may also use this operation.
- `Retrieve_Previous`: Similar to `Retrieve_Next`, but in this case the record that is "previous" to the currently accessed record is retrieved.
- `Insert_One`: Insert a new record into the file. It may be necessary that the new record fit into a particular position to preserve a sequencing of the file.
- `Delete_One`: Delete an existing record. Certain linkages or other data structures may need to be updated to preserve the sequencing of the file.
- `Update_One`: Retrieve a record, update one or more of its fields, and rewrite the updated record back into the file. Again, it may be necessary to preserve sequencing with this operation. If the length of the record has changed, the update operation is generally more difficult than if the length is preserved.
- `Retrieve_Few`: Retrieve a number of records. For example, an application or user may wish to retrieve all records that satisfy a certain set of criteria.

The nature of the operations that are most commonly performed on a file will influence the way the file is organized, as discussed in Section 12.2.

It should be noted that not all file systems exhibit the sort of structure discussed in this subsection. On UNIX and UNIX-like systems, the basic file structure is just a stream of bytes. For example, a C program is stored as a file but does not have physical fields, records, and so on.

File Management Systems

A file management system is that set of system software that provides services to users and applications in the use of files. Typically, the only way that a user or application may access files is through the file management system. This relieves the user or programmer of the necessity of developing special-purpose software for each application and provides the system with a consistent, well-defined means of controlling its most important asset. [GROS86] suggests the following objectives for a file management system:

- To meet the data management needs and requirements of the user, which include storage of data and the ability to perform the aforementioned operations
- To guarantee, to the extent possible, that the data in the file are valid

- To optimize performance, both from the system point of view in terms of overall throughput and from the user's point of view in terms of response time
- To provide I/O support for a variety of storage device types
- To minimize or eliminate the potential for lost or destroyed data
- To provide a standardized set of I/O interface routines to user processes
- To provide I/O support for multiple users, in the case of multiple-user systems

With respect to the first point, meeting user requirements, the extent of such requirements depends on the variety of applications and the environment in which the computer system will be used. For an interactive, general-purpose system, the following constitute a minimal set of requirements:

1. Each user should be able to create, delete, read, write, and modify files.
2. Each user may have controlled access to other users' files.
3. Each user may control what types of accesses are allowed to the user's files.
4. Each user should be able to restructure the user's files in a form appropriate to the problem.
5. Each user should be able to move data between files.
6. Each user should be able to back up and recover the user's files in case of damage.
7. Each user should be able to access his or her files by name rather than by numeric identifier.

These objectives and requirements should be kept in mind throughout our discussion of file management systems.

File System Architecture One way of getting a feel for the scope of file management is to look at a depiction of a typical software organization, as suggested in Figure 12.1. Of course, different systems will be organized differently, but this

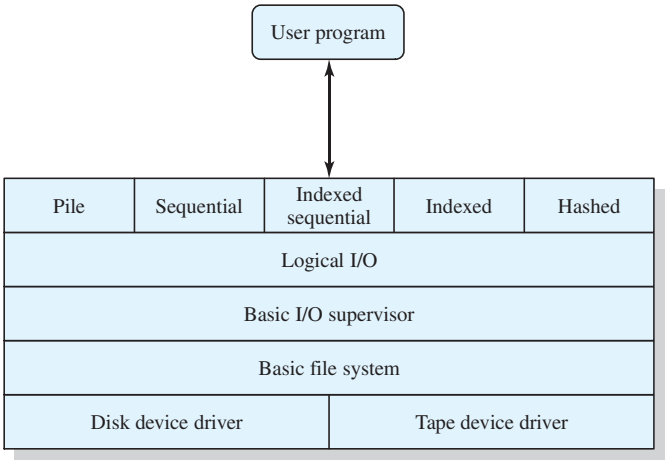


Figure 12.1 File System Software Architecture

organization is reasonably representative. At the lowest level, **device drivers** communicate directly with peripheral devices or their controllers or channels. A device driver is responsible for starting I/O operations on a device and processing the completion of an I/O request. For file operations, the typical devices controlled are disk and tape drives. Device drivers are usually considered to be part of the operating system.

The next level is referred to as the **basic file system**, or the **physical I/O** level. This is the primary interface with the environment outside of the computer system. It deals with blocks of data that are exchanged with disk or tape systems. Thus, it is concerned with the placement of those blocks on the secondary storage device and on the buffering of those blocks in main memory. It does not understand the content of the data or the structure of the files involved. The basic file system is often considered part of the operating system.

The **basic I/O supervisor** is responsible for all file I/O initiation and termination. At this level, control structures are maintained that deal with device I/O, scheduling, and file status. The basic I/O supervisor selects the device on which file I/O is to be performed, based on the particular file selected. It is also concerned with scheduling disk and tape accesses to optimize performance. I/O buffers are assigned and secondary memory is allocated at this level. The basic I/O supervisor is part of the operating system.

Logical I/O enables users and applications to access records. Thus, whereas the basic file system deals with blocks of data, the logical I/O module deals with file records. Logical I/O provides a general-purpose record I/O capability and maintains basic data about files.

The level of the file system closest to the user is often termed the **access method**. It provides a standard interface between applications and the file systems and devices that hold the data. Different access methods reflect different file structures and different ways of accessing and processing the data. Some of the most common access methods are shown in Figure 12.1, and these are briefly described in Section 12.2.

File Management Functions Another way of viewing the functions of a file system is shown in Figure 12.2. Let us follow this diagram from left to right. Users and application programs interact with the file system by means of commands for creating and deleting files and for performing operations on files. Before performing any operation, the file system must identify and locate the selected file. This requires the use of some sort of directory that serves to describe the location of all files, plus their attributes. In addition, most shared systems enforce user access control: Only authorized users are allowed to access particular files in particular ways. The basic operations that a user or application may perform on a file are performed at the record level. The user or application views the file as having some structure that organizes the records, such as a sequential structure (e.g., personnel records are stored alphabetically by last name). Thus, to translate user commands into specific file manipulation commands, the access method appropriate to this file structure must be employed.

Whereas users and applications are concerned with records or fields, I/O is done on a block basis. Thus, the records or fields of a file must be organized as a sequence of

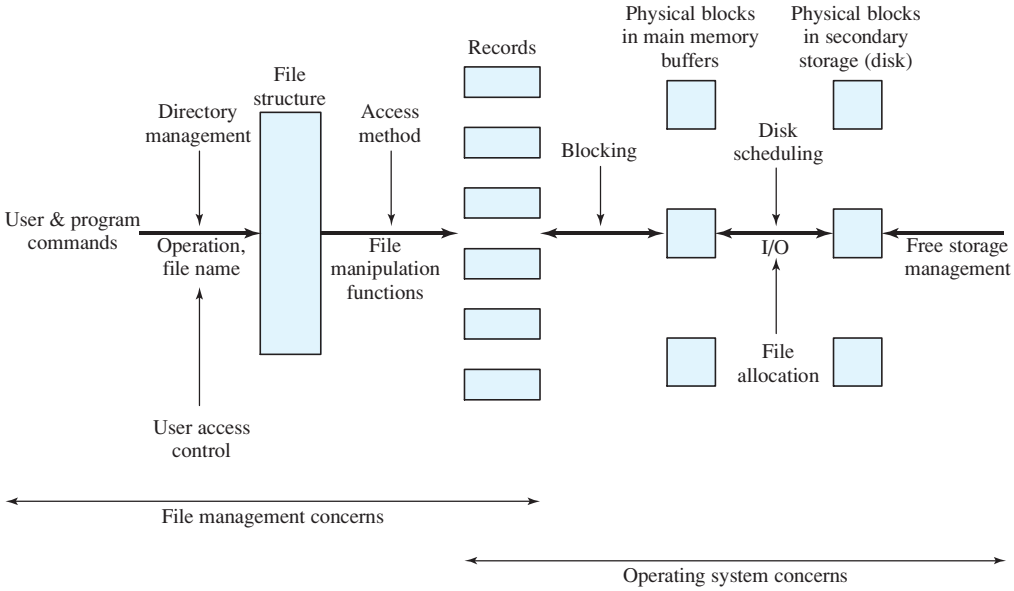


Figure 12.2 Elements of File Management

blocks for output and unblocked after input. To support block I/O of files, several functions are needed. The secondary storage must be managed. This involves allocating files to free blocks on secondary storage and managing free storage so as to know what blocks are available for new files and growth in existing files. In addition, individual block I/O requests must be scheduled; this issue was dealt with in Chapter 11. Both disk scheduling and file allocation are concerned with optimizing performance. As might be expected, these functions therefore need to be considered together. Furthermore, the optimization will depend on the structure of the files and the access patterns. Accordingly, developing an optimum file management system from the point of view of performance is an exceedingly complicated task.

Figure 12.2 suggests a division between what might be considered the concerns of the file management system as a separate system utility and the concerns of the operating system, with the point of intersection being record processing. This division is arbitrary; various approaches are taken in various systems.

In the remainder of this chapter, we look at some of the design issues suggested in Figure 12.2. We begin with a discussion of file organizations and access methods. Although this topic is beyond the scope of what is usually considered the concerns of the operating system, it is impossible to assess the other file-related design issues without an appreciation of file organization and access. Next, we look at the concept of file directories. These are often managed by the operating system on behalf of the file management system. The remaining topics deal with the physical I/O aspects of file management and are properly treated as aspects of operating system design. One such issue is the way in which logical records are organized into physical blocks. Finally, there are the related issues of file allocation on secondary storage and the management of free secondary storage.

12.2 FILE ORGANIZATION AND ACCESS

In this section, we use the term *file organization* to refer to the logical structuring of the records as determined by the way in which they are accessed. The physical organization of the file on secondary storage depends on the blocking strategy and the file allocation strategy, issues dealt with later in this chapter.

In choosing a file organization, several criteria are important:

- Short access time
- Ease of update
- Economy of storage
- Simple maintenance
- Reliability

The relative priority of these criteria will depend on the applications that will use the file. For example, if a file is only to be processed in batch mode, with all of the records accessed every time, then rapid access for retrieval of a single record is of minimal concern. A file stored on CD-ROM will never be updated, and so ease of update is not an issue.

These criteria may conflict. For example, for economy of storage, there should be minimum redundancy in the data. On the other hand, redundancy is a primary means of increasing the speed of access to data. An example of this is the use of indexes.

The number of alternative file organizations that have been implemented or just proposed is unmanageably large, even for a book devoted to file systems. In this brief survey, we will outline five fundamental organizations. Most structures used in actual systems either fall into one of these categories or can be implemented with a combination of these organizations. The five organizations, the first four of which are depicted in Figure 12.3, are as follows:

- The pile
- The sequential file
- The indexed sequential file
- The indexed file
- The direct, or hashed, file

Table 12.1 summarizes relative performance aspects of these five organizations.[1]

The Pile

The least-complicated form of file organization may be termed the *pile*. Data are collected in the order in which they arrive. Each record consists of one burst of data. The purpose of the pile is simply to accumulate the mass of data and save it. Records may have different fields, or similar fields in different orders. Thus, each field should be self-describing, including a field name as well as a value. The length of each field

[1]The table employs the "big-O" notation, used for characterizing the time complexity of algorithms. Appendix D explains this notation.

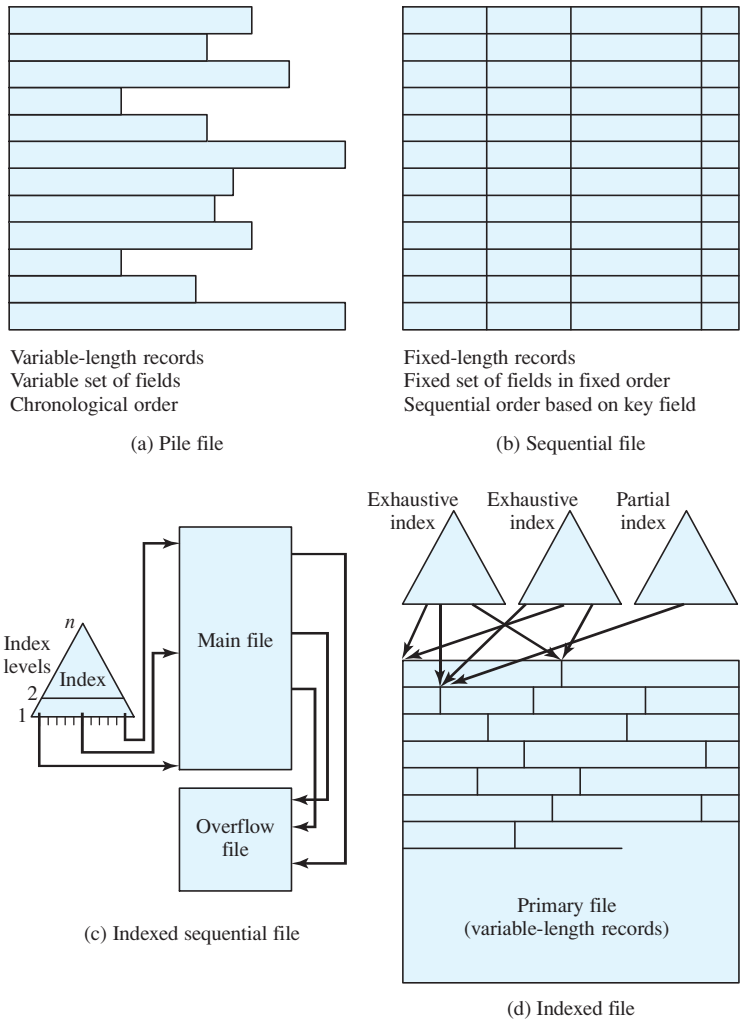


Figure 12.3 Common File Organizations

must be implicitly indicated by delimiters, explicitly included as a subfield, or known as default for that field type.

Because there is no structure to the pile file, record access is by exhaustive search. That is, if we wish to find a record that contains a particular field with a particular value, it is necessary to examine each record in the pile until the desired record is found or the entire file has been searched. If we wish to find all records that contain a particular field or contain that field with a particular value, then the entire file must be searched.

Pile files are encountered when data are collected and stored prior to processing or when data are not easy to organize. This type of file uses space well when the stored data vary in size and structure, is perfectly adequate for exhaustive searches,

Table 12.1 Grades of Performance for Five Basic File Organizations [WIED87]

	Space Attributes		Update Record Size		Retrieval		
File Method	**Variable**	**Fixed**	**Equal**	**Greater**	**Single record**	**Subset**	**Exhaustive**
Pile	A	B	A	E	E	D	B
Sequential	F	A	D	F	F	D	A
Indexed sequential	F	B	B	D	B	D	B
Indexed	B	C	C	C	A	B	D
Hashed	F	B	B	F	B	F	E

A = Excellent, well suited to this purpose — $\approx O(r)$
B = Good — $\approx O(o \times r)$
C = Adequate — $\approx O(r \log n)$
D = Requires some extra effort — $\approx O(n)$
E = Possible with extreme effort — $\approx O(r \times n)$
F = Not reasonable for this purpose — $\approx O(n^{>1})$

where
r = size of the result
o = number of records that overflow
n = number of records in file

and is easy to update. However, beyond these limited uses, this type of file is unsuitable for most applications.

The Sequential File

The most common form of file structure is the sequential file. In this type of file, a fixed format is used for records. All records are of the same length, consisting of the same number of fixed-length fields in a particular order. Because the length and position of each field are known, only the values of fields need to be stored; the field name and length for each field are attributes of the file structure.

One particular field, usually the first field in each record, is referred to as the **key field**. The key field uniquely identifies the record; thus key values for different records are always different. Further, the records are stored in key sequence: alphabetical order for a text key, and numerical order for a numerical key.

Sequential files are typically used in batch applications and are generally optimum for such applications if they involve the processing of all the records (e.g., a billing or payroll application). The sequential file organization is the only one that is easily stored on tape as well as disk.

For interactive applications that involve queries and/or updates of individual records, the sequential file provides poor performance. Access requires the sequential search of the file for a key match. If the entire file, or a large portion of the file, can be brought into main memory at one time, more efficient search techniques are possible. Nevertheless, considerable processing and delay are encountered to access a record in a large sequential file. Additions to the file also present problems. Typically, a sequential

file is stored in simple sequential ordering of the records within blocks. That is, the physical organization of the file on tape or disk directly matches the logical organization of the file. In this case, the usual procedure is to place new records in a separate pile file, called a log file or transaction file. Periodically, a batch update is performed that merges the log file with the master file to produce a new file in correct key sequence.

An alternative is to organize the sequential file physically as a linked list. One or more records are stored in each physical block. Each block on disk contains a pointer to the next block. The insertion of new records involves pointer manipulation but does not require that the new records occupy a particular physical block position. Thus, some added convenience is obtained at the cost of additional processing and overhead.

The Indexed Sequential File

A popular approach to overcoming the disadvantages of the sequential file is the indexed sequential file. The indexed sequential file maintains the key characteristic of the sequential file: records are organized in sequence based on a key field. Two features are added: an index to the file to support random access, and an overflow file. The index provides a lookup capability to reach quickly the vicinity of a desired record. The overflow file is similar to the log file used with a sequential file but is integrated so that a record in the overflow file is located by following a pointer from its predecessor record.

In the simplest indexed sequential structure, a single level of indexing is used. The index in this case is a simple sequential file. Each record in the index file consists of two fields: a key field, which is the same as the key field in the main file, and a pointer into the main file. To find a specific field, the index is searched to find the highest key value that is equal to or precedes the desired key value. The search continues in the main file at the location indicated by the pointer.

To see the effectiveness of this approach, consider a sequential file with 1 million records. To search for a particular key value will require on average one-half million record accesses. Now suppose that an index containing 1000 entries is constructed, with the keys in the index more or less evenly distributed over the main file. Now it will take on average 500 accesses to the index file followed by 500 accesses to the main file to find the record. The average search length is reduced from 500,000 to 1000.

Additions to the file are handled in the following manner: Each record in the main file contains an additional field not visible to the application, which is a pointer to the overflow file. When a new record is to be inserted into the file, it is added to the overflow file. The record in the main file that immediately precedes the new record in logical sequence is updated to contain a pointer to the new record in the overflow file. If the immediately preceding record is itself in the overflow file, then the pointer in that record is updated. As with the sequential file, the indexed sequential file is occasionally merged with the overflow file in batch mode.

The indexed sequential file greatly reduces the time required to access a single record, without sacrificing the sequential nature of the file. To process the entire file sequentially, the records of the main file are processed in sequence until a pointer to the overflow file is found, then accessing continues in the overflow file until a null pointer is encountered, at which time accessing of the main file is resumed where it left off.

To provide even greater efficiency in access, multiple levels of indexing can be used. Thus the lowest level of index file is treated as a sequential file and a higher-level index file is created for that file. Consider again a file with 1 million

records. A lower-level index with 10,000 entries is constructed. A higher-level index into the lower level index of 100 entries can then be constructed. The search begins at the higher-level index (average length = 50 accesses) to find an entry point into the lower-level index. This index is then searched (average length = 50) to find an entry point into the main file, which is then searched (average length = 50). Thus the average length of search has been reduced from 500,000 to 1000 to 150.

The Indexed File

The indexed sequential file retains one limitation of the sequential file: effective processing is limited to that which is based on a single field of the file. For example, when it is necessary to search for a record on the basis of some other attribute than the key field, both forms of sequential file are inadequate. In some applications, the flexibility of efficiently searching by various attributes is desirable.

To achieve this flexibility, a structure is needed that employs multiple indexes, one for each type of field that may be the subject of a search. In the general indexed file, the concept of sequentiality and a single key are abandoned. Records are accessed only through their indexes. The result is that there is now no restriction on the placement of records as long as a pointer in at least one index refers to that record. Furthermore, variable-length records can be employed.

Two types of indexes are used. An exhaustive index contains one entry for every record in the main file. The index itself is organized as a sequential file for ease of searching. A partial index contains entries to records where the field of interest exists. With variable-length records, some records will not contain all fields. When a new record is added to the main file, all of the index files must be updated.

Indexed files are used mostly in applications where timeliness of information is critical and where data are rarely processed exhaustively. Examples are airline reservation systems and inventory control systems.

The Direct or Hashed File

The direct, or hashed, file exploits the capability found on disks to access directly any block of a known address. As with sequential and indexed sequential files, a key field is required in each record. However, there is no concept of sequential ordering here.

The direct file makes use of hashing on the key value. This function was explained in Appendix 8A. Figure 8.27b shows the type of hashing organization with an overflow file that is typically used in a hash file.

Direct files are often used where very rapid access is required, where fixed-length records are used, and where records are always accessed one at a time. Examples are directories, pricing tables, schedules, and name lists.

12.3 FILE DIRECTORIES

Contents

Associated with any file management system and collection of files is a file directory. The directory contains information about the files, including attributes, location, and ownership. Much of this information, especially that concerned with

storage, is managed by the operating system. The directory is itself a file, accessible by various file management routines. Although some of the information in directories is available to users and applications, this is generally provided indirectly by system routines.

Table 12.2 suggests the information typically stored in the directory for each file in the system. From the user's point of view, the directory provides a mapping between file names, known to users and applications, and the files themselves. Thus, each file entry includes the name of the file. Virtually all systems deal with different types of files and different file organizations, and this information is also provided. An important category of information about each file concerns its storage, including its location and size. In shared systems, it is also important to provide information that is used to control access to the file. Typically, one user is the owner of the file and may grant certain access privileges to other users. Finally,

Table 12.2 Information Elements of a File Directory

	Basic Information
File Name	Name as chosen by creator (user or program). Must be unique within a specific directory.
File Type	For example: text, binary, load module, etc.
File Organization	For systems that support different organizations
	Address Information
Volume	Indicates device on which file is stored
Starting Address	Starting physical address on secondary storage (e.g., cylinder, track, and block number on disk)
Size Used	Current size of the file in bytes, words, or blocks
Size Allocated	The maximum size of the file
	Access Control Information
Owner	User who is assigned control of this file. The owner may be able to grant/deny access to other users and to change these privileges.
Access Information	A simple version of this element would include the user's name and password for each authorized user.
Permitted Actions	Controls reading, writing, executing, transmitting over a network
	Usage Information
Date Created	When file was first placed in directory
Identity of Creator	Usually but not necessarily the current owner
Date Last Read Access	Date of the last time a record was read
Identity of Last Reader	User who did the reading
Date Last Modified	Date of the last update, insertion, or deletion
Identity of Last Modifier	User who did the modifying
Date of Last Backup	Date of the last time the file was backed up on another storage medium
Current Usage	Information about current activity on the file, such as process or processes that have the file open, whether it is locked by a process, and whether the file has been updated in main memory but not yet on disk

usage information is needed to manage the current use of the file and to record the history of its usage.

Structure

The way in which the information of Table 12.2 is stored differs widely among various systems. Some of the information may be stored in a header record associated with the file; this reduces the amount of storage required for the directory, making it easier to keep all or much of the directory in main memory to improve speed.

The simplest form of structure for a directory is that of a list of entries, one for each file. This structure could be represented by a simple sequential file, with the name of the file serving as the key. In some earlier single-user systems, this technique has been used. However, it is inadequate when multiple users share a system and even for single users with many files.

To understand the requirements for a file structure, it is helpful to consider the types of operations that may be performed on the directory:

- **Search:** When a user or application references a file, the directory must be searched to find the entry corresponding to that file.
- **Create file:** When a new file is created, an entry must be added to the directory.
- **Delete file:** When a file is deleted, an entry must be removed from the directory.
- **List directory:** All or a portion of the directory may be requested. Generally, this request is made by a user and results in a listing of all files owned by that user, plus some of the attributes of each file (e.g., type, access control information, usage information).
- **Update directory:** Because some file attributes are stored in the directory, a change in one of these attributes requires a change in the corresponding directory entry.

The simple list is not suited to supporting these operations. Consider the needs of a single user. The user may have many types of files, including word-processing text files, graphic files, spreadsheets, and so on. The user may like to have these organized by project, by type, or in some other convenient way. If the directory is a simple sequential list, it provides no help in organizing the files and forces the user to be careful not to use the same name for two different types of files. The problem is much worse in a shared system. Unique naming becomes a serious problem. Furthermore, it is difficult to conceal portions of the overall directory from users when there is no inherent structure in the directory.

A start in solving these problems would be to go to a two-level scheme. In this case, there is one directory for each user, and a master directory. The master directory has an entry for each user directory, providing address and access control information. Each user directory is a simple list of the files of that user. This arrangement means that names must be unique only within the collection of files of a single user, and that the file system can easily enforce access restriction on directories. However, it still provides users with no help in structuring collections of files.

A more powerful and flexible approach, and one that is almost universally adopted, is the hierarchical, or tree-structure, approach (Figure 12.4). As before, there

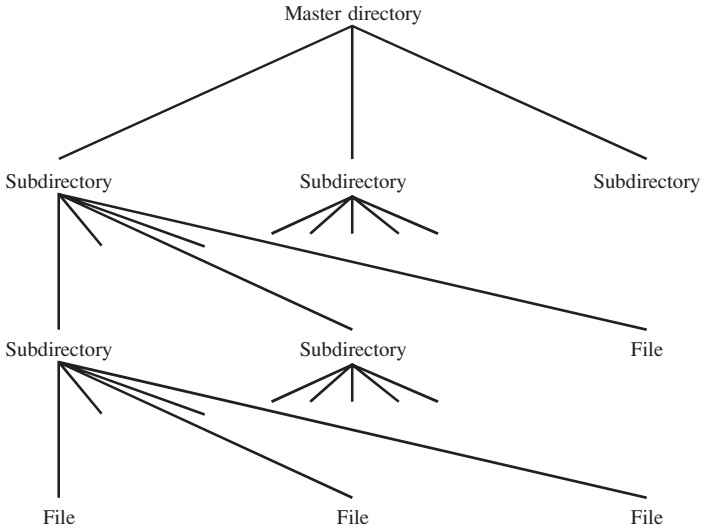


Figure 12.4 **Tree-Structured Directory**

is a master directory, which has under it a number of user directories. Each of these user directories, in turn, may have subdirectories and files as entries. This is true at any level: That is, at any level, a directory may consist of entries for subdirectories and/or entries for files.

It remains to say how each directory and subdirectory is organized. The simplest approach, of course, is to store each directory as a sequential file. When directories may contain a very large number of entries, such an organization may lead to unnecessarily long search times. In that case, a hashed structure is to be preferred.

Naming

Users need to be able to refer to a file by a symbolic name. Clearly, each file in the system must have a unique name in order that file references be unambiguous. On the other hand, it is an unacceptable burden on users to require that they provide unique names, especially in a shared system.

The use of a tree-structured directory minimizes the difficulty in assigning unique names. Any file in the system can be located by following a path from the root or master directory down various branches until the file is reached. The series of directory names, culminating in the file name itself, constitutes a **pathname** for the file. As an example, the file in the lower left-hand corner of Figure 12.5 has the pathname `User_B/Word/Unit_A/ABC`. The slash is used to delimit names in the sequence. The name of the master directory is implicit, because all paths start at that directory. Note that it is perfectly acceptable to have several files with the same file name, as long as they have unique pathnames, which is equivalent to saying that the same file name may be used in different directories. In our example, there is another file in the system with the file name ABC, but that has the pathname `/User_B/Draw/ABC`.

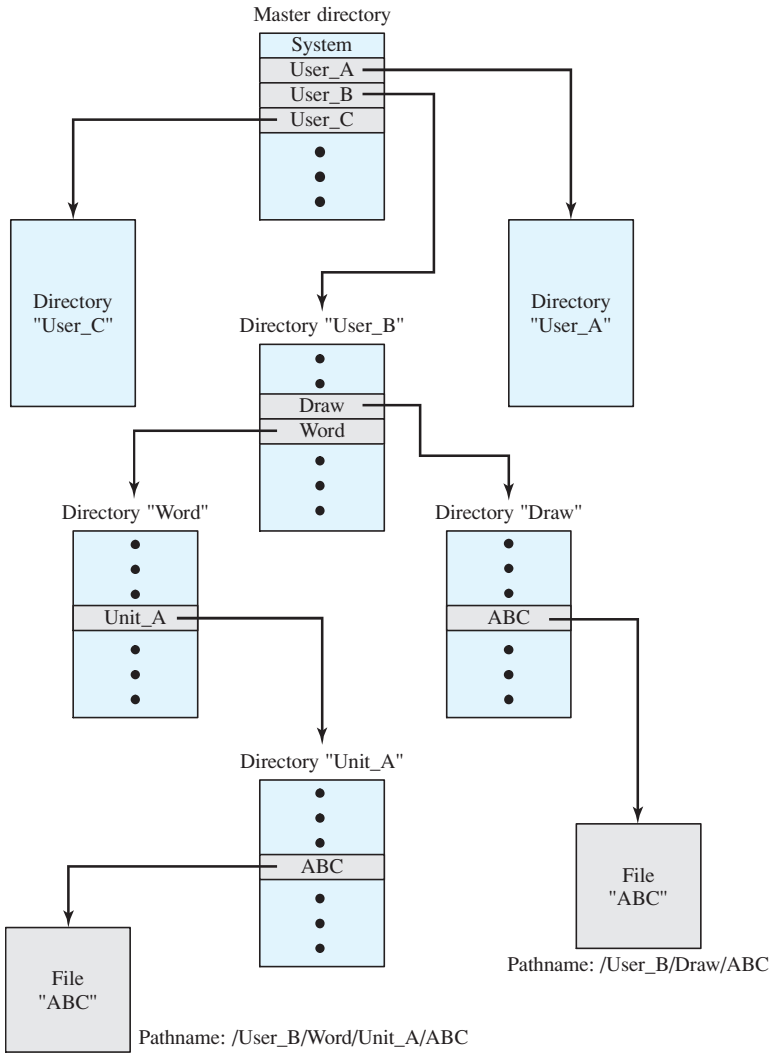


Figure 12.5 Example of Tree-Structured Directory

Although the pathname facilitates the selection of file names, it would be awkward for a user to have to spell out the entire pathname every time a reference is made to a file. Typically, an interactive user or a process has associated with it a current directory, often referred to as the **working directory**. Files are then referenced relative to the working directory. For example, if the working directory for user B is "Word," then the pathname `Unit_A/ABC` is sufficient to identify the file in the lower left-hand corner of Figure 12.5. When an interactive user logs on, or when a process is created, the default for the working directory is the user home directory. During execution, the user can navigate up or down in the tree to change to a different working directory.

12.4 FILE SHARING

In a multiuser system, there is almost always a requirement for allowing files to be shared among a number of users. Two issues arise: access rights and the management of simultaneous access.

Access Rights

The file system should provide a flexible tool for allowing extensive file sharing among users. The file system should provide a number of options so that the way in which a particular file is accessed can be controlled. Typically, users or groups of users are granted certain access rights to a file. A wide range of access rights has been used. The following list is representative of access rights that can be assigned to a particular user for a particular file:

- **None:** The user may not even learn of the existence of the file, much less access it. To enforce this restriction, the user would not be allowed to read the user directory that includes this file.
- **Knowledge:** The user can determine that the file exists and who its owner is. The user is then able to petition the owner for additional access rights.
- **Execution:** The user can load and execute a program but cannot copy it. Proprietary programs are often made accessible with this restriction.
- **Reading:** The user can read the file for any purpose, including copying and execution. Some systems are able to enforce a distinction between viewing and copying. In the former case, the contents of the file can be displayed to the user, but the user has no means for making a copy.
- **Appending:** The user can add data to the file, often only at the end, but cannot modify or delete any of the file's contents. This right is useful in collecting data from a number of sources.
- **Updating:** The user can modify, delete, and add to the file's data. This normally includes writing the file initially, rewriting it completely or in part, and removing all or a portion of the data. Some systems distinguish among different degrees of updating.
- **Changing protection:** The user can change the access rights granted to other users. Typically, this right is held only by the owner of the file. In some systems, the owner can extend this right to others. To prevent abuse of this mechanism, the file owner will typically be able to specify which rights can be changed by the holder of this right.
- **Deletion:** The user can delete the file from the file system.

These rights can be considered to constitute a hierarchy, with each right implying those that precede it. Thus, if a particular user is granted the updating right for a particular file, then that user is also granted the following rights: knowledge, execution, reading, and appending.

One user is designated as owner of a given file, usually the person who initially created a file. The owner has all of the access rights listed previously and may grant rights to others. Access can be provided to different classes of users:

- **Specific user:** Individual users who are designated by user ID.
- **User groups:** A set of users who are not individually defined. The system must have some way of keeping track of the membership of user groups.
- **All:** All users who have access to this system. These are public files.

Simultaneous Access

When access is granted to append or update a file to more than one user, the operating system or file management system must enforce discipline. A brute-force approach is to allow a user to lock the entire file when it is to be updated. A finer grain of control is to lock individual records during update. Essentially, this is the readers/writers problem discussed in Chapter 5. Issues of mutual exclusion and deadlock must be addressed in designing the shared access capability.

12.5 RECORD BLOCKING

As indicated in Figure 12.2, records are the logical unit of access of a structured file,[2] whereas blocks are the unit of I/O with secondary storage. For I/O to be performed, records must be organized as blocks.

There are several issues to consider. First, should blocks be of fixed or variable length? On most systems, blocks are of fixed length. This simplifies I/O, buffer allocation in main memory, and the organization of blocks on secondary storage. Next, what should the relative size of a block be compared to the average record size? The tradeoff is this: The larger the block, the more records that are passed in one I/O operation. If a file is being processed or searched sequentially, this is an advantage, because the number of I/O operations is reduced by using larger blocks, thus speeding up processing. On the other hand, if records are being accessed randomly and no particular locality of reference is observed, then larger blocks result in the unnecessary transfer of unused records. However, combining the frequency of sequential operations with the potential for locality of reference, we can say that the I/O transfer time is reduced by using larger blocks. The competing concern is that larger blocks require larger I/O buffers, making buffer management more difficult.

Given the size of a block, there are three methods of blocking that can be used:

- **Fixed blocking:** Fixed-length records are used, and an integral number of records are stored in a block. There may be unused space at the end of each block. This is referred to as internal fragmentation.
- **Variable-length spanned blocking:** Variable-length records are used and are packed into blocks with no unused space. Thus, some records must span two blocks, with the continuation indicated by a pointer to the successor block.

[2]As opposed to a file that is treated only as a stream of bytes, such as in the UNIX file system.

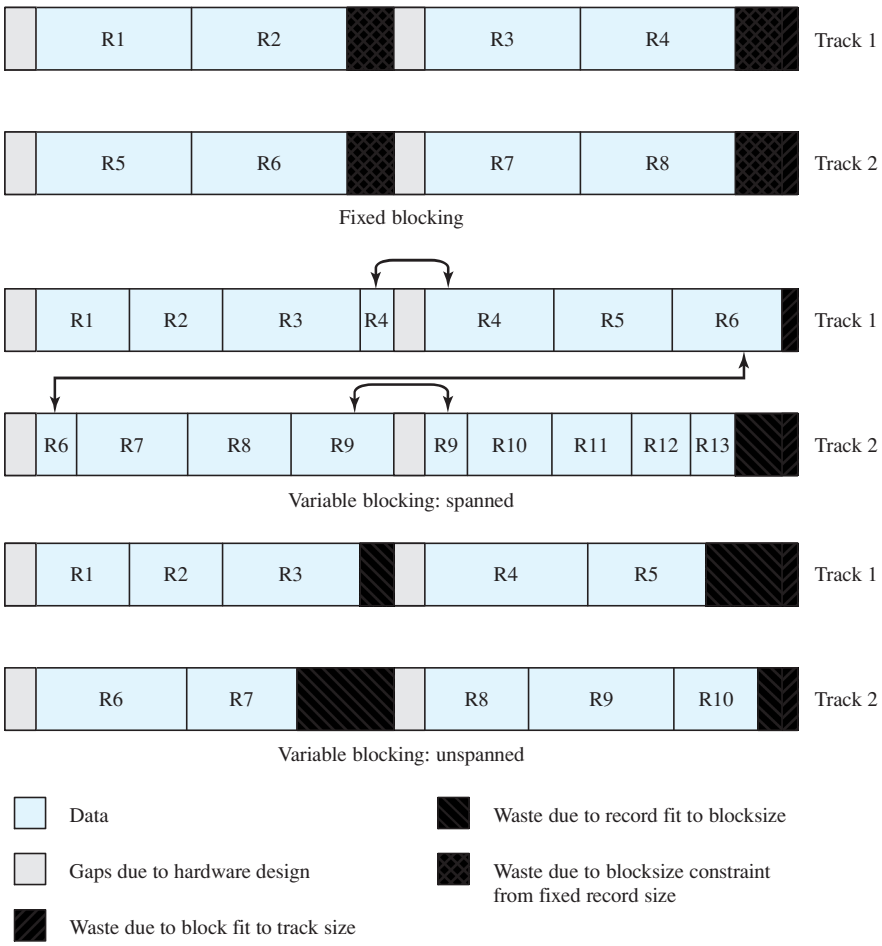


Figure 12.6 **Record Blocking Methods [WIED87]**

- **Variable-length unspanned blocking:** Variable-length records are used, but spanning is not employed. There is wasted space in most blocks because of the inability to use the remainder of a block if the next record is larger than the remaining unused space.

Figure 12.6 illustrates these methods assuming that a file is stored in sequential blocks on a disk. The figure assumes that the file is large enough to span two tracks.[3] The effect would not be changed if some other file allocation scheme were used (see Section 12.6).

Fixed blocking is the common mode for sequential files with fixed-length records. Variable-length spanned blocking is efficient of storage and does not limit

[3]Recall from Appendix 11A that the organization of data on a disk is in a concentric set of rings, called *tracks*. Each track is the same width as the read/write head.

the size of records. However, this technique is difficult to implement. Records that span two blocks require two I/O operations, and files are difficult to update, regardless of the organization. Variable-length unspanned blocking results in wasted space and limits record size to the size of a block.

The record-blocking technique may interact with the virtual memory hardware, if such is employed. In a virtual memory environment, it is desirable to make the page the basic unit of transfer. Pages are generally quite small, so that it is impractical to treat a page as a block for unspanned blocking. Accordingly, some systems combine multiple pages to create a larger block for file I/O purposes. This approach is used for VSAM files on IBM mainframes.

12.6 SECONDARY STORAGE MANAGEMENT

On secondary storage, a file consists of a collection of blocks. The operating system or file management system is responsible for allocating blocks to files. This raises two management issues. First, space on secondary storage must be allocated to files, and second, it is necessary to keep track of the space available for allocation. We will see that these two tasks are related; that is, the approach taken for file allocation may influence the approach taken for free space management. Further, we will see that there is an interaction between file structure and allocation policy.

We begin this section by looking at alternatives for file allocation on a single disk. Then we look at the issue of free space management, and finally we discuss reliability.

File Allocation

Several issues are involved in file allocation:

1. When a new file is created, is the maximum space required for the file allocated at once?
2. Space is allocated to a file as one or more contiguous units, which we shall refer to as portions. That is, a **portion** is a contiguous set of allocated blocks. The size of a portion can range from a single block to the entire file. What size of portion should be used for file allocation?
3. What sort of data structure or table is used to keep track of the portions assigned to a file? An example of such a structure is a **file allocation table (FAT)**, found on DOS and some other systems.

Let us examine these issues in turn.

Preallocation versus Dynamic Allocation A preallocation policy requires that the maximum size of a file be declared at the time of the file creation request. In a number of cases, such as program compilations, the production of summary data files, or the transfer of a file from another system over a communications network, this value can be reliably estimated. However, for many applications, it is difficult if not impossible to estimate reliably the maximum potential size of the file. In those cases, users and application programmers would tend to overestimate file size

so as not to run out of space. This clearly is wasteful from the point of view of secondary storage allocation. Thus, there are advantages to the use of dynamic allocation, which allocates space to a file in portions as needed.

Portion Size The second issue listed is that of the size of the portion allocated to a file. At one extreme, a portion large enough to hold the entire file is allocated. At the other extreme, space on the disk is allocated one block at a time. In choosing a portion size, there is a tradeoff between efficiency from the point of view of a single file versus overall system efficiency. [WIED87] lists four items to be considered in the tradeoff:

1. Contiguity of space increases performance, especially for `Retrieve_Next` operations, and greatly for transactions running in a transaction-oriented operating system.
2. Having a large number of small portions increases the size of tables needed to manage the allocation information.
3. Having fixed-size portions (for example, blocks) simplifies the reallocation of space.
4. Having variable-size or small fixed-size portions minimizes waste of unused storage due to overallocation.

Of course, these items interact and must be considered together. The result is that there are two major alternatives:

- **Variable, large contiguous portions:** This will provide better performance. The variable size avoids waste, and the file allocation tables are small. However, space is hard to reuse.
- **Blocks:** Small fixed portions provide greater flexibility. They may require large tables or complex structures for their allocation. Contiguity has been abandoned as a primary goal; blocks are allocated as needed.

Either option is compatible with preallocation or dynamic allocation. In the case of variable, large contiguous portions, a file is preallocated one contiguous group of blocks. This eliminates the need for a file allocation table; all that is required is a pointer to the first block and the number of blocks allocated. In the case of blocks, all of the portions required are allocated at one time. This means that the file allocation table for the file will remain of fixed size, because the number of blocks allocated is fixed.

With variable-size portions, we need to be concerned with the fragmentation of free space. This issue was faced when we considered partitioned main memory in Chapter 7. The following are possible alternative strategies:

- **First fit:** Choose the first unused contiguous group of blocks of sufficient size from a free block list.
- **Best fit:** Choose the smallest unused group that is of sufficient size.
- **Nearest fit:** Choose the unused group of sufficient size that is closest to the previous allocation for the file to increase locality.

It is not clear which strategy is best. The difficulty in modeling alternative strategies is that so many factors interact, including types of files, pattern of file

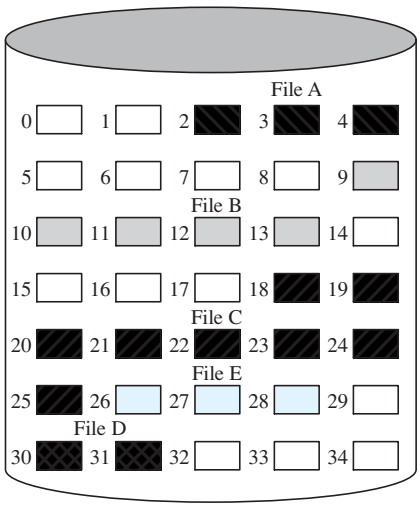
Table 12.3 File Allocation Methods

	Contiguous	Chained	Indexed	
Preallocation?	Necessary	Possible	Possible	
Fixed or variable size portions?	Variable	Fixed blocks	Fixed blocks	Variable
Portion size	Large	Small	Small	Medium
Allocation frequency	Once	Low to high	High	Low
Time to allocate	Medium	Long	Short	Medium
File allocation table size	One entry	One entry	Large	Medium

access, degree of multiprogramming, other performance factors in the system, disk caching, disk scheduling, and so on.

File Allocation Methods Having looked at the issues of preallocation versus dynamic allocation and portion size, we are in a position to consider specific file allocation methods. Three methods are in common use: contiguous, chained, and indexed. Table 12.3 summarizes some of the characteristics of each method.

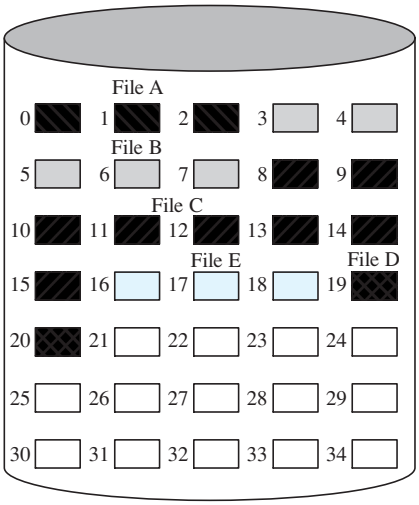
With **contiguous allocation**, a single contiguous set of blocks is allocated to a file at the time of file creation (Figure 12.7). Thus, this is a preallocation strategy, using variable-size portions. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. Contiguous allocation is the best from the point of view of the individual sequential file. Multiple blocks can be read in at a time to improve I/O performance for sequential processing. It is also easy to retrieve a single block. For example, if a file starts at block b, and the ith block of the file is wanted, its location on secondary storage is simply $b + i - 1$. Contiguous allocation presents some problems. External fragmentation will occur, making it difficult to find contiguous blocks of space of sufficient length. From time to time, it will be necessary to perform a compaction algorithm to free up additional



File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Figure 12.7 **Contiguous File Allocation**



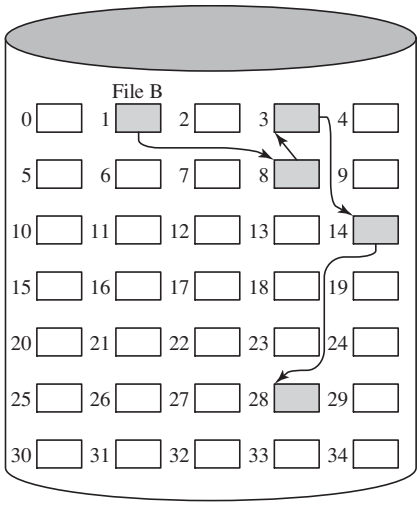
File Allocation Table

File Name	Start Block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

Figure 12.8 **Contiguous File Allocation (After Compaction)**

space on the disk (Figure 12.8). Also, with preallocation, it is necessary to declare the size of the file at the time of creation, with the problems mentioned earlier.

At the opposite extreme from contiguous allocation is **chained allocation** (Figure 12.9). Typically, allocation is on an individual block basis. Each block contains a pointer to the next block in the chain. Again, the file allocation table needs just a single entry for each file, showing the starting block and the length of the file. Although preallocation is possible, it is more common simply to allocate blocks as needed. The selection of blocks is now a simple matter: any free block can be added to a chain. There is no external fragmentation to worry about because only one



File Allocation Table

File Name	Start Block	Length
• • •	• • •	• • •
File B	1	5
• • •	• • •	• • •

Figure 12.9 **Chained Allocation**

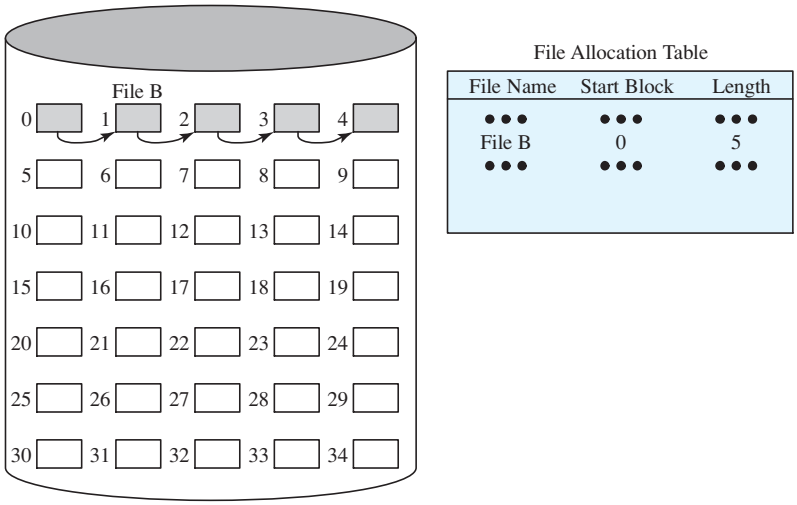


Figure 12.10 **Chained Allocation (After Consolidation)**

block at a time is needed. This type of physical organization is best suited to sequential files that are to be processed sequentially. To select an individual block of a file requires tracing through the chain to the desired block.

One consequence of chaining, as described so far, is that there is no accommodation of the principle of locality. Thus, if it is necessary to bring in several blocks of a file at a time, as in sequential processing, then a series of accesses to different parts of the disk are required. This is perhaps a more significant effect on a single-user system but may also be of concern on a shared system. To overcome this problem, some systems periodically consolidate files (Figure 12.10).

Indexed allocation addresses many of the problems of contiguous and chained allocation. In this case, the file allocation table contains a separate one-level index for each file; the index has one entry for each portion allocated to the file. Typically, the file indexes are not physically stored as part of the file allocation table. Rather, the file index for a file is kept in a separate block, and the entry for the file in the file allocation table points to that block. Allocation may be on the basis of either fixed-size blocks (Figure 12.11) or variable-size portions (Figure 12.12). Allocation by blocks eliminates external fragmentation, whereas allocation by variable-size portions improves locality. In either case, file consolidation may be done from time to time. File consolidation reduces the size of the index in the case of variable-size portions, but not in the case of block allocation. Indexed allocation supports both sequential and direct access to the file and thus is the most popular form of file allocation.

Free Space Management

Just as the space that is allocated to files must be managed, so the space that is not currently allocated to any file must be managed. To perform any of the file allocation techniques described previously, it is necessary to know what blocks on the disk are available. Thus we need a **disk allocation table** in addition to a file allocation table. We discuss here a number of techniques that have been implemented.

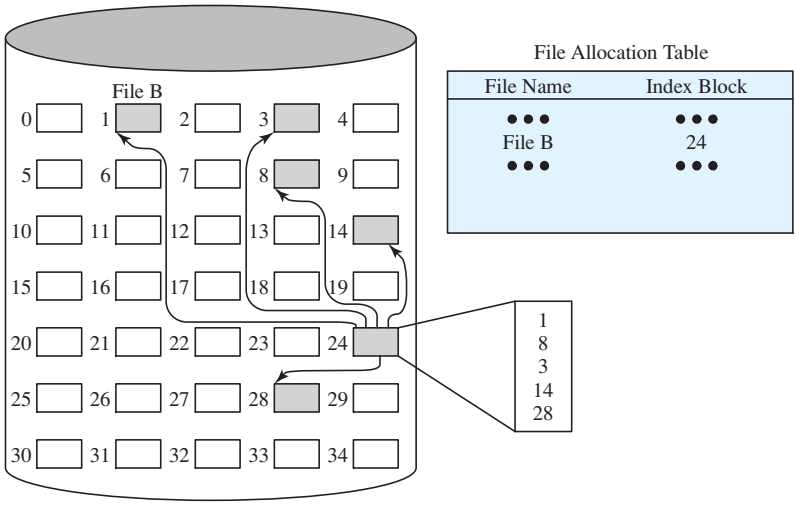


Figure 12.11 **Indexed Allocation with Block Portions**

Bit Tables This method uses a vector containing one bit for each block on the disk. Each entry of a 0 corresponds to a free block, and each 1 corresponds to a block in use. For example, for the disk layout of Figure 12.7, a vector of length 35 is needed and would have the following value:

00111000011111000011111111111011000

A bit table has the advantage that it is relatively easy to find one or a contiguous group of free blocks. Thus, a bit table works well with any of the file allocation methods outlined. Another advantage is that it is as small as possible.

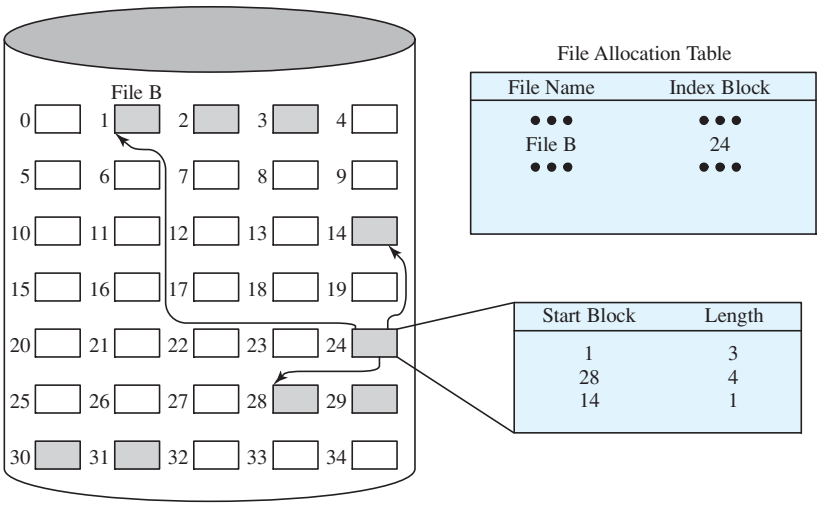


Figure 12.12 **Indexed Allocation with Variable-Length Portions**

However, it can still be sizable. The amount of memory (in bytes) required for a block bitmap is

$$\frac{\text{disk size in bytes}}{8 \times \text{file system block size}}$$

Thus, for a 16-Gbyte disk with 512-byte blocks, the bit table occupies about 4 Mbytes. Can we spare 4 Mbytes of main memory for the bit table? If so, then the bit table can be searched without the need for disk access. But even with today's memory sizes, 4 Mbytes is a hefty chunk of main memory to devote to a single function. The alternative is to put the bit table on disk. But a 4-Mbyte bit table would require about 8000 disk blocks. We can't afford to search that amount of disk space every time a block is needed, so a bit table resident in memory is indicated.

Even when the bit table is in main memory, an exhaustive search of the table can slow file system performance to an unacceptable degree. This is especially true when the disk is nearly full and there are few free blocks remaining. Accordingly, most file systems that use bit tables maintain auxiliary data structures that summarize the contents of subranges of the bit table. For example, the table could be divided logically into a number of equal-size subranges. A summary table could include, for each subrange, the number of free blocks and the maximum-sized contiguous number of free blocks. When the file system needs a number of contiguous blocks, it can scan the summary table to find an appropriate subrange and then search that subrange.

Chained Free Portions The free portions may be chained together by using a pointer and length value in each free portion. This method has negligible space overhead because there is no need for a disk allocation table, merely for a pointer to the beginning of the chain and the length of the first portion. This method is suited to all of the file allocation methods. If allocation is a block at a time, simply choose the free block at the head of the chain and adjust the first pointer or length value. If allocation is by variable-length portion, a first-fit algorithm may be used: The headers from the portions are fetched one at a time to determine the next suitable free portion in the chain. Again, pointer and length values are adjusted.

This method has its own problems. After some use, the disk will become quite fragmented and many portions will be a single block long. Also note that every time you allocate a block, you need to read the block first to recover the pointer to the new first free block before writing data to that block. If many individual blocks need to be allocated at one time for a file operation, this greatly slows file creation. Similarly, deleting highly fragmented files is very time consuming.

Indexing The indexing approach treats free space as a file and uses an index table as described under file allocation. For efficiency, the index should be on the basis of variable-size portions rather than blocks. Thus, there is one entry in the table for every free portion on the disk. This approach provides efficient support for all of the file allocation methods.

Free Block List In this method, each block is assigned a number sequentially and the list of the numbers of all free blocks is maintained in a reserved portion of the disk. Depending on the size of the disk, either 24 or 32 bits will be needed to

store a single block number, so the size of the free block list is 24 or 32 times the size of the corresponding bit table and thus must be stored on disk rather than in main memory. However, this is a satisfactory method. Consider the following points:

1. The space on disk devoted to the free block list is less than 1% of the total disk space. If a 32-bit block number is used, then the space penalty is 4 bytes for every 512-byte block.
2. Although the free block list is too large to store in main memory, there are two effective techniques for storing a small part of the list in main memory.
   a. The list can be treated as a push-down stack (Appendix 1B) with the first few thousand elements of the stack kept in main memory. When a new block is allocated, it is popped from the top of the stack, which is in main memory. Similarly, when a block is deallocated, it is pushed onto the stack. There only has to be a transfer between disk and main memory when the in-memory portion of the stack becomes either full or empty. Thus, this technique gives almost zero-time access most of the time.
   b. The list can be treated as a FIFO queue, with a few thousand entries from both the head and the tail of the queue in main memory. A block is allocated by taking the first entry from the head of the queue and deallocated by adding it to the end of the tail of the queue. There only has to be a transfer between disk and main memory when either the in-memory portion of the head of the queue becomes empty or the in-memory portion of the tail of the queue becomes full.

In either of the strategies listed in the preceding point (stack or FIFO queue), a background thread can slowly sort the in-memory list or lists to facilitate contiguous allocation.

Volumes

The term *volume* is used somewhat differently by different operating systems and file management systems, but in essence a volume is a logical disk. [CARR05] defines a volume as follows:

Volume: A collection of addressable sectors in secondary memory that an OS or application can use for data storage. The sectors in a volume need not be consecutive on a physical storage device; instead they need only appear that way to the OS or application. A volume may be the result of assembling and merging smaller volumes.

In the simplest case, a single disk equals one volume. Frequently, a disk is divided in to partitions, with each partition functioning as a separate volume. It is also common to treat multiple disks as a single volume of partitions on multiple disks as a single volume.

Reliability

Consider the following scenario:

1. User A requests a file allocation to add to an existing file.
2. The request is granted and the disk and file allocation tables are updated in main memory but not yet on disk.
3. The system crashes and subsequently restarts.
4. User B requests a file allocation and is allocated space on disk that overlaps the last allocation to user A.
5. User A accesses the overlapped portion via a reference that is stored inside A's file.

This difficulty arose because the system maintained a copy of the disk allocation table and file allocation table in main memory for efficiency. To prevent this type of error, the following steps could be performed when a file allocation is requested:

1. Lock the disk allocation table on disk. This prevents another user from causing alterations to the table until this allocation is completed.
2. Search the disk allocation table for available space. This assumes that a copy of the disk allocation table is always kept in main memory. If not, it must first be read in.
3. Allocate space, update the disk allocation table, and update the disk. Updating the disk involves writing the disk allocation table back onto disk. For chained disk allocation, it also involves updating some pointers on disk.
4. Update the file allocation table and update the disk.
5. Unlock the disk allocation table.

This technique will prevent errors. However, when small portions are allocated frequently, the impact on performance will be substantial. To reduce this overhead, a batch storage allocation scheme could be used. In this case, a batch of free portions on the disk is obtained for allocation. The corresponding portions on disk are marked "in use." Allocation using this batch may proceed in main memory. When the batch is exhausted, the disk allocation table is updated on disk and a new batch may be acquired. If a system crash occurs, portions on the disk marked "in use" must be cleaned up in some fashion before they can be reallocated. The technique for cleanup will depend on the file system's particular characteristics.

12.7 FILE SYSTEM SECURITY

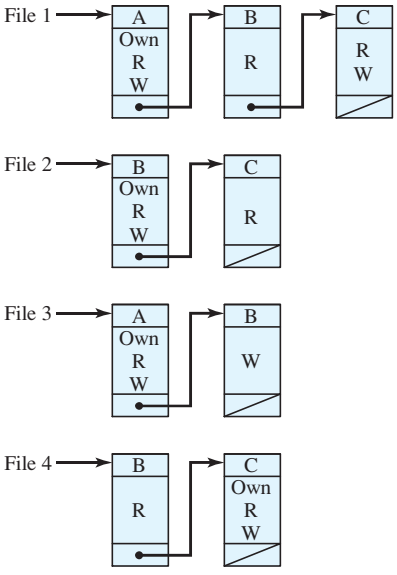
Following successful logon, the user has been granted access to one or a set of hosts and applications. This is generally not sufficient for a system that includes sensitive data in its database. Through the user access control procedure, a user can be identified to the system. Associated with each user, there can be a profile that specifies permissible operations and file accesses. The operating system can then enforce rules based on the user profile. The database management system, however, must control access to specific records or even portions of records. For example, it may be

permissible for anyone in administration to obtain a list of company personnel, but only selected individuals may have access to salary information. The issue is more than just one of level of detail. Whereas the operating system may grant a user permission to access a file or use an application, following which there are no further security checks, the database management system must make a decision on each individual access attempt. That decision will depend not only on the user's identity but also on the specific parts of the data being accessed and even on the information already divulged to the user.

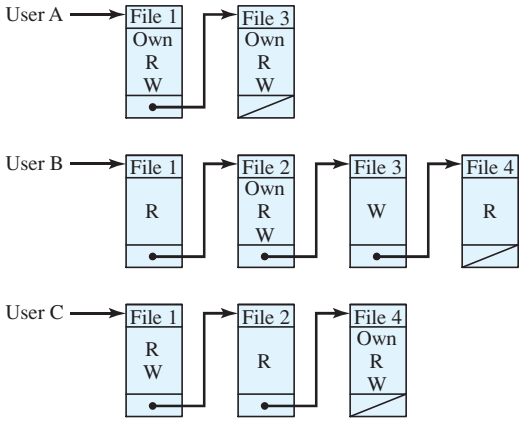
A general model of access control as exercised by a file or database management system is that of an **access matrix** (Figure 12.13a, based on a figure in [SAND94]). The basic elements of the model are as follows:

	File 1	File 2	File 3	File 4	Account 1	Account 2
User A	Own R W		Own R W		Inquiry Credit	
User B	R	Own R W	W	R	Inquiry Debit	Inquiry Credit
User C	R W	R		Own R W		Inquiry Debit

(a) Access matrix



(b) Access control lists for files of part (a)



(c) Capability lists for files of part (a)

Figure 12.13 Example of Access Control Structures

- **Subject:** An entity capable of accessing objects. Generally, the concept of subject equates with that of process. Any user or application actually gains access to an object by means of a process that represents that user or application.
- **Object:** Anything to which access is controlled. Examples include files, portions of files, programs, segments of memory, and software objects (e.g., Java objects).
- **Access right:** The way in which an object is accessed by a subject. Examples are read, write, execute, and functions in software objects.

One dimension of the matrix consists of identified subjects that may attempt data access. Typically, this list will consist of individual users or user groups, although access could be controlled for terminals, hosts, or applications instead of or in addition to users. The other dimension lists the objects that may be accessed. At the greatest level of detail, objects may be individual data fields. More aggregate groupings, such as records, files, or even the entire database, may also be objects in the matrix. Each entry in the matrix indicates the access rights of that subject for that object.

In practice, an access matrix is usually sparse and is implemented by decomposition in one of two ways. The matrix may be decomposed by columns, yielding **access control lists** (Figure 12.13b). Thus for each object, an access control list lists users and their permitted access rights. The access control list may contain a default, or public, entry. This allows users that are not explicitly listed as having special rights to have a default set of rights. Elements of the list may include individual users as well as groups of users.

Decomposition by rows yields **capability tickets** (Figure 12.13c). A capability ticket specifies authorized objects and operations for a user. Each user has a number of tickets and may be authorized to loan or give them to others. Because tickets may be dispersed around the system, they present a greater security problem than access control lists. In particular, the ticket must be unforgeable. One way to accomplish this is to have the operating system hold all tickets on behalf of users. These tickets would have to be held in a region of memory inaccessible to users.

Network considerations for data-oriented access control parallel those for user-oriented access control. If only certain users are permitted to access certain items of data, then encryption may be needed to protect those items during transmission to authorized users. Typically, data access control is decentralized, that is, controlled by host-based database management systems. If a network database server exists on a network, then data access control becomes a network function.

12.8 UNIX FILE MANAGEMENT

In the UNIX file system, six types of files are distinguished:

- **Regular, or ordinary:** Contains arbitrary data in zero or more data blocks. Regular files contain information entered in them by a user, an application program, or a system utility program. The file system does not impose any internal structure to a regular file but treats it as a stream of bytes.

- **Directory:** Contains a list of file names plus pointers to associated inodes (index nodes), described later. Directories are hierarchically organized (Figure 12.4). Directory files are actually ordinary files with special write protection privileges so that only the file system can write into them, while read access is available to user programs.
- **Special:** Contains no data, but provides a mechanism to map physical devices to file names. The file names are used to access peripheral devices, such as terminals and printers. Each I/O device is associated with a special file, as discussed in Section 11.8.
- **Named pipes:** As discussed in Section 6.7, a pipe is an interprocess communications facility. A pipe file buffers data received in its input so that a process that reads from the pipe's output receives the data on a first-in-first-out basis.
- **Links:** In essence, a link is an alternative file name for an existing file.
- **Symbolic links:** This is a data file that contains the name of the file it is linked to.

In this section, we are concerned with the handling of ordinary files, which correspond to what most systems treat as files.

Inodes

Modern UNIX operating systems support multiple file systems but map all of these into a uniform underlying system for supporting file systems and allocating disk space to files. All types of UNIX files are administered by the OS by means of inodes. An inode (index node) is a control structure that contains the key information needed by the operating system for a particular file. Several file names may be associated with a single inode, but an active inode is associated with exactly one file, and each file is controlled by exactly one inode.

The attributes of the file as well as its permissions and other control information are stored in the inode. The exact inode structure varies from one UNIX implementation to another. The FreeBSD inode structure, shown in Figure 12.14, includes the following data elements:

- The type and access mode of the file
- The file's owner and group-access identifiers
- The time that the file was created, when it was most recently read and written, and when its inode was most recently updated by the system
- The size of the file in bytes
- A sequence of block pointers, explained in the next subsection
- The number of physical blocks used by the file, including blocks used to hold indirect pointers and attributes
- The number of directory entries the reference the file
- The kernel and user setable flags that describe the characteristics of the file
- The generation number of the file (a randomly selected number assigned to the inode each time that the latter is allocated to a new file; the generation number is used to detect references to deleted files)

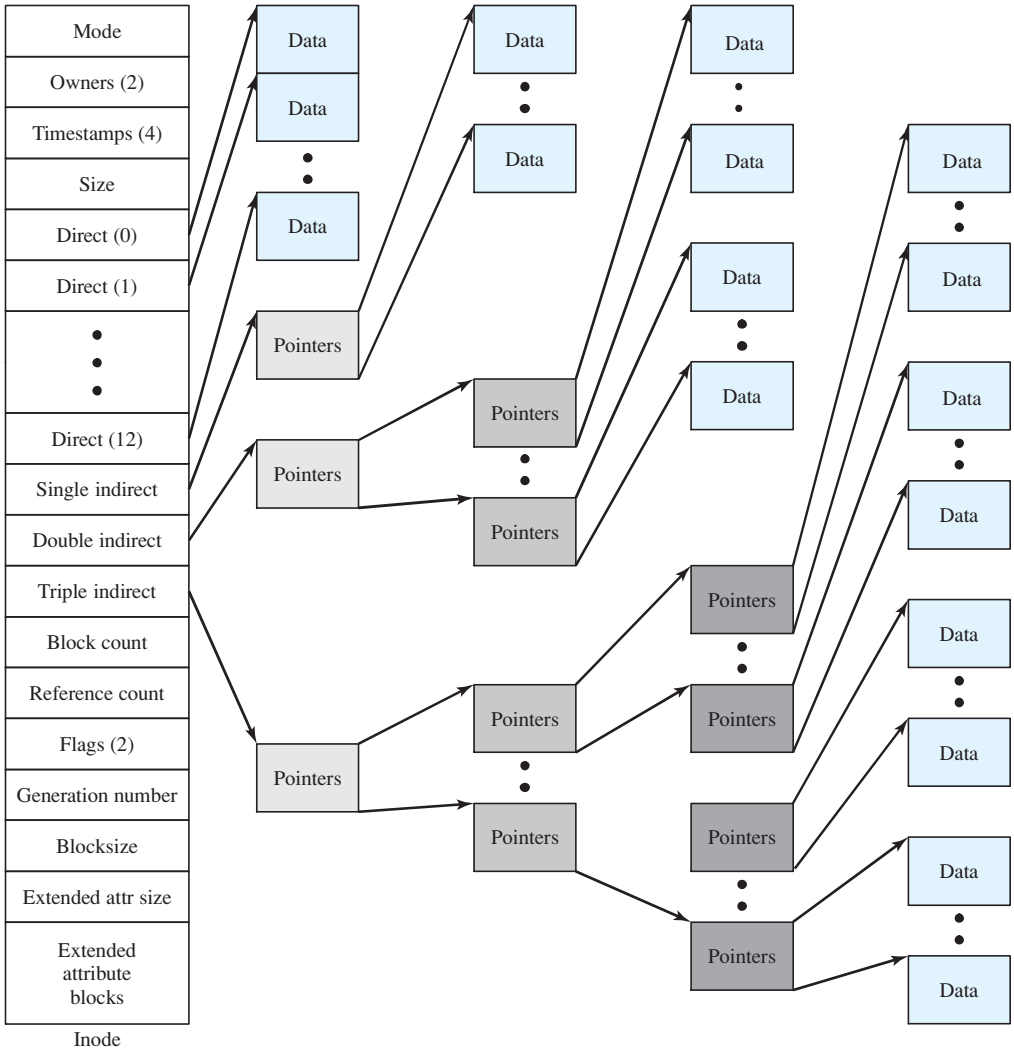


Figure 12.14 **Structure of FreeBSD inode and File**

- The blocksize of the data blocks referenced by the inode (typically the same as, but sometimes larger than, the file system blocksize)
- The size of the extended attribute information
- Zero or more extended attribute entries

The blocksize value is typically the same as, but sometimes larger than, the file system blocksize. On traditional UNIX systems, a fixed blocksize of 512 bytes was used. FreeBSD has a minimum blocksize of 4096 bytes (4 Kbytes); the blocksize can be any power of 2 greater than or equal to 4096. For typical file systems, the blocksize is 8 Kbytes or 16 Kbytes. The default FreeBSD blocksize is 16 Kbytes.

Extended attribute entries are variable-length entries used to store auxiliary data that is separate from the contents of the file. The first two extended attributes defined for FreeBSD deal with security. The first of these support access control lists; this is described in Chapter 15. The second defined extended attribute supports the use of security labels, which are part of what is known as a mandatory access control scheme, also described in Chapter 15.

On the disk, there is an inode table, or inode list, that contains the inodes of all the files in the file system. When a file is opened, its inode is brought into main memory and stored in a memory-resident inode table.

File Allocation

File allocation is done on a block basis. Allocation is dynamic, as needed, rather than using preallocation. Hence, the blocks of a file on disk are not necessarily contiguous. An indexed method is used to keep track of each file, with part of the index stored in the inode for the file. In all UNIX implementations, the inode includes a number of direct pointers and three indirect pointers (single, double, triple).

The FreeBSD inode includes 120 bytes of address information that is organized as fifteen 64-bit addresses, or pointers. The first 12 addresses point to the first 12 data blocks of the file. If the file requires more than 12 data blocks, one or more levels of indirection is used as follows:

- The thirteenth address in the inode points to a block on disk that contains the next portion of the index. This is referred to as the single indirect block. This block contains the pointers to succeeding blocks in the file.
- If the file contains more blocks, the fourteenth address in the inode points to a double indirect block. This block contains a list of addresses of additional single indirect blocks. Each of single indirect blocks, in turn, contains pointers to file blocks.
- If the file contains still more blocks, the fifteenth address in the inode points to a triple indirect block that is a third level of indexing. This block points to additional double indirect blocks.

All of this is illustrated in Figure 12.14. The total number of data blocks in a file depends on the capacity of the fixed-size blocks in the system. In FreeBSD, the minimum block size is 4 Kbyte, and each block can hold a total of 512 block addresses. Thus, the maximum size of a file with this block size is over 500 GB (Table 12.4).

Table 12.4 Capacity of a FreeBSD File with 4 kByte Block Size

Level	Number of Blocks	Number of Bytes
Direct	12	48K
Single Indirect	512	2M
Double Indirect	$512 \times 512 = 256K$	1G
Triple Indirect	$512 \times 256K = 128M$	512G

This scheme has several advantages:

1. The inode is of fixed size and relatively small and hence may be kept in main memory for long periods.
2. Smaller files may be accessed with little or no indirection, reducing processing and disk access time.
3. The theoretical maximum size of a file is large enough to satisfy virtually all applications.

Directories

Directories are structured in a hierarchical tree. Each directory can contain files and/or other directories. A directory that is inside another directory is referred to as a subdirectory. As was mentioned, a directory is simply a file that contains a list of file names plus pointers to associated inodes. Figure 12.15 shows the overall structure. Each directory entry (dentry) contains a name for the associated file or subdirectory plus an integer called the i-number (index number). When the file or directory is accessed, its i-number is used as an index into the inode table.

Volume Structure

A UNIX file system resides on a single logical disk or disk partition and is laid out with the following elements:

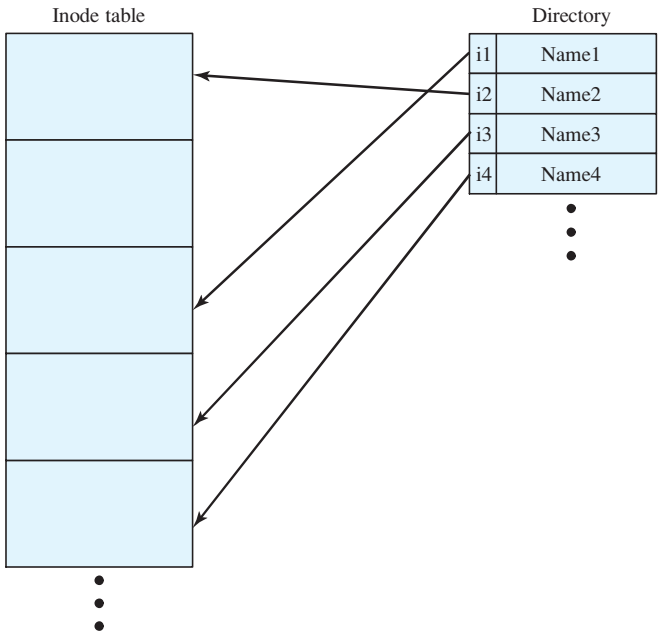


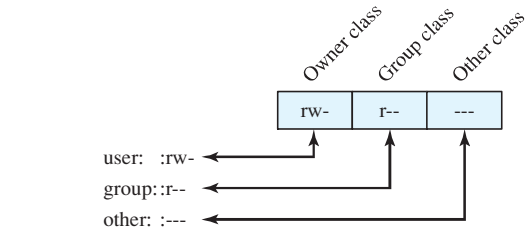
Figure 12.15 UNIX Directories and Inodes

- **Boot block:** Contains code required to boot the operating system
- **Superblock:** Contains attributes and information about the file system, such as partition size, and inode table size
- **Inode table:** The collection of inodes for each file
- **Data blocks:** Storage space available for data files and subdirectories

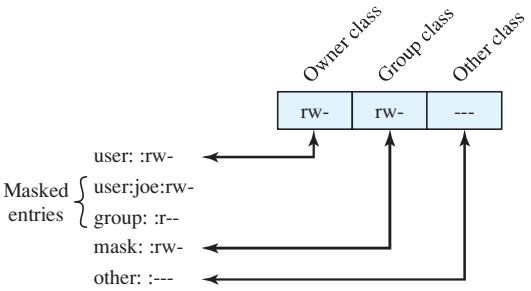
Traditional UNIX File Access Control

Most UNIX systems depend on, or at least are based on, the file access control scheme introduced with the early versions of UNIX. Each UNIX user is assigned a unique user identification number (user ID). A user is also a member of a primary group, and possibly a number of other groups, each identified by a group ID. When a file is created, it is designated as owned by a particular user and marked with that user's ID. It also belongs to a specific group, which initially is either its creator's primary group, or the group of its parent directory if that directory has SetGID permission set. Associated with each file is a set of 12 protection bits. The owner ID, group ID, and protection bits are part of the file's inode.

Nine of the protection bits specify read, write, and execute permission for the owner of the file, other members of the group to which this file belongs, and all other users. These form a hierarchy of owner, group, and all others, with the highest relevant set of permissions being used. Figure 12.16a shows an example in which the file owner has read and write access; all other members of the file's group have



(a) Traditional UNIX approach (minimal access control list)



(b) Extended access control list

Figure 12.16 UNIX File Access Control

read access, and users outside the group have no access rights to the file. When applied to a directory, the read and write bits grant the right to list and to create/rename/delete files in the directory.[4] The execute bit grants to right to search the directory for a component of a filename.

The remaining three bits define special additional behavior for files or directories. Two of these are the "set user ID" (SetUID) and "set group ID" (SetGID) permissions. If these are set on an executable file, the operating system functions as follows. When a user (with execute privileges for this file) executes the file, the system temporarily allocates the rights of the user's ID of the file creator, or the file's group, respectively, to those of the user executing the file. These are known as the "effective user ID" and "effective group ID" and are used in addition to the "real user ID" and "real group ID" of the executing user when making access control decisions for this program. This change is only effective while the program is being executed. This feature enables the creation and use of privileged programs that may use files normally inaccessible to other users. It enables users to access certain files in a controlled fashion. Alternatively, when applied to a directory, the SetGID permission indicates that newly created files will inherit the group of this directory. The SetUID permission is ignored.

The final permission bit is the "Sticky" bit. When set on a file, this originally indicated that the system should retain the file contents in memory following execution. This is no longer used. When applied to a directory, though, it specifies that only the owner of any file in the directory can rename, move, or delete that file. This is useful for managing files in shared temporary directories.

One particular user ID is designated as "superuser." The superuser is exempt from the usual file access control constraints and has systemwide access. Any program that is owned by, and SetUID to, the "superuser" potentially grants unrestricted access to the system to any user executing that program. Hence great care is needed when writing such programs.

This access scheme is adequate when file access requirements align with users and a modest number of groups of users. For example, suppose a user wants to give read access for file X to users A and B and read access for file Y to users B and C. We would need at least two user groups, and user B would need to belong to both groups in order to access the two files. However, if there are a large number of different groupings of users requiring a range of access rights to different files, then a very large number of groups may be needed to provide this. This rapidly becomes unwieldy and difficult to manage, even if possible at all.[5] One way to overcome this problem is to use access control lists, which are provided in most modern UNIX systems.

A final point to note is that the traditional UNIX file access control scheme implements a simple protection domain structure. A domain is associated with the user, and switching the domain corresponds to changing the user ID temporarily.

---

[4]Note that the permissions that apply to a directory are distinct from those that apply to any file or directory it contains. The fact that a user has the right to write to the directory does not give the user the right to write to a file in that directory. That is governed by the permissions of the specific file. The user would, however, have the right to rename the file.

[5]Most UNIX systems impose a limit on the maximum number of groups any user may belong to, as well as to the total number of groups possible on the system.

Access Control Lists in UNIX

Many modern UNIX and UNIX-based operating systems support access control lists, including FreeBSD, OpenBSD, Linux, and Solaris. In this section, we describe the FreeBSD approach, but other implementations have essentially the same features and interface. The feature is referred to as extended access control list, while the traditional UNIX approach is referred to as minimal access control list.

FreeBSD allows the administrator to assign a list of UNIX user IDs and groups to a file by using the setfacl command. Any number of users and groups can be associated with a file, each with three protection bits (read, write, execute), offering a flexible mechanism for assigning access rights. A file need not have an ACL but may be protected solely by the traditional UNIX file access mechanism. FreeBSD files include an additional protection bit that indicates whether the file has an extended ACL.

FreeBSD and most UNIX implementations that support extended ACLs use the following strategy (e.g., Figure 12.16b):

1. The owner class and other class entries in the 9-bit permission field have the same meaning as in the minimal ACL case.
2. The group class entry specifies the permissions for the owner group for this file. These permissions represent the maximum permissions that can be assigned to named users or named groups, other than the owning user. In this latter role, the group class entry functions as a mask.
3. Additional named users and named groups may be associated with the file, each with a 3-bit permission field. The permissions listed for a named user or named group are compared to the mask field. Any permission for the named user or named group that is not present in the mask field is disallowed.

When a process requests access to a file system object, two steps are performed. Step 1 selects the ACL entry that most closely matches the requesting process. The ACL entries are looked at in the following order: owner, named users, (owning or named) groups, others. Only a single entry determines access. Step 2 checks if the matching entry contains sufficient permissions. A process can be a member in more than one group; so more than one group entry can match. If any of these matching group entries contain the requested permissions, one that contains the requested permissions is picked (the result is the same no matter which entry is picked). If none of the matching group entries contains the requested permissions, access will be denied no matter which entry is picked.

12.9 LINUX VIRTUAL FILE SYSTEM

Linux includes a versatile and powerful file handling facility, designed to support a wide variety of file management systems and file structures. The approach taken in Linux is to make use of a **virtual file system (VFS)**, which presents a single, uniform file system interface to user processes. The VFS defines a common file model that is capable of representing any conceivable file system's general feature and behavior. The VFS assumes that files are objects in a computer's mass storage memory that share basic properties regardless of the target file system or the underlying processor

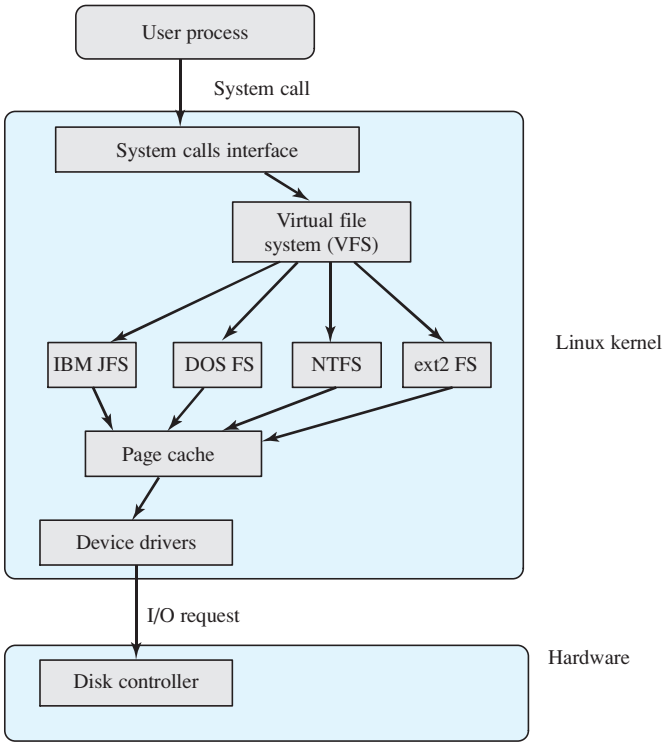


Figure 12.17 Linux Virtual File System Context

hardware. Files have symbolic names that allow them to be uniquely identified within a specific directory within the file system. A file has an owner, protection against unauthorized access or modification, and a variety of other properties. A file may be created, read from, written to, or deleted. For any specific file system, a mapping module is needed to transform the characteristics of the real file system to the characteristics expected by the virtual file system.

Figure 12.17 indicates the key ingredients of the Linux file system strategy. A user process issues a file system call (e.g., read) using the VFS file scheme. The VFS converts this into an internal (to the kernel) file system call that is passed to a mapping function for a specific file system [e.g., IBM's Journaling File System (JFS)]. In most cases, the mapping function is simply a mapping of file system functional calls from one scheme to another. In some cases, the mapping function is more complex. For example, some file systems use a file allocation table (FAT), which stores the position of each file in the directory tree. In these file systems, directories are not files. For such file systems, the mapping function must be able to construct dynamically, and when needed, the files corresponding to the directories. In any case, the original user file system call is translated into a call that is native to the target file system. The target file system software is then invoked to perform the requested function on a file or directory under its control and secondary storage. The results of the operation are then communicated back to the user in a similar fashion.

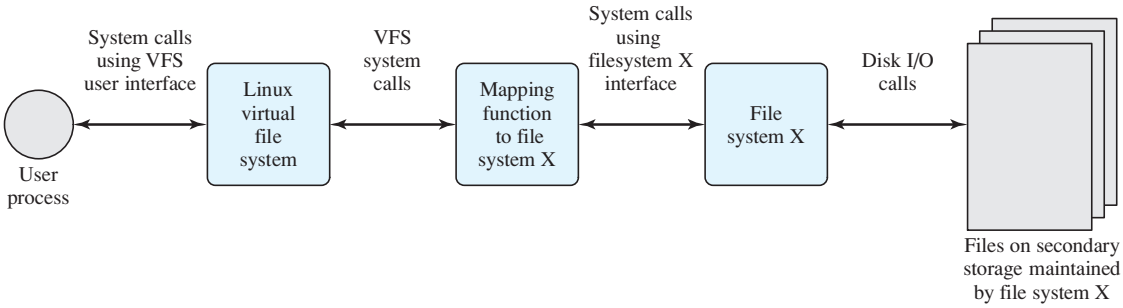


Figure 12.18 **Linux Virtual File System Concept**

Figure 12.18 indicates the role that VFS plays within the Linux kernel. When a process initiates a file-oriented system call (e.g., read), the kernel calls a function in the VFS. This function handles the file-system-independent manipulations and initiates a call to a function in the target file system code. This call passes through a mapping function that converts the call from the VFS into a call to the target file system. The VFS is independent of any file system, so the implementation of a mapping function must be part of the implementation of a file system on Linux. The target file system converts the file system request into device-oriented instructions that are passed to a device driver by means of page cache functions.

VFS is an object-oriented scheme. Because it is written in C, rather than a language that supports object programming (such as C++ or Java), VFS objects are implemented simply as C data structures. Each object contains both data and pointers to file-system-implemented functions that operate on data. The four primary object types in VFS are as follows:

- **Superblock object:** Represents a specific mounted file system
- **Inode object:** Represents a specific file
- **Dentry object:** Represents a specific directory entry
- **File object:** Represents an open file associated with a process

This scheme is based on the concepts used in UNIX file systems, as described in Section 12.7. The key concepts of UNIX file system to remember are the following. A file system consists of a hierarchal organization of directories. A directory is the same as what is knows as a folder on many non-UNIX platforms and may contain files and/or other directories. Because a directory may contain other directories, a tree structure is formed. A path through the tree structure from the root consists of a sequence of directory entries, ending in either a directory entry (dentry) or a file name. In UNIX, a directory is implemented as a file that lists the files and directories contained within it. Thus, file operations can be performed on either files or directories.

The Superblock Object

The superblock object stores information describing a specific file system. Typically, the superblock corresponds to the file system superblock or file system control block, which is stored in a special sector on disk.

The superblock object consists of a number of data items. Examples include the following:

- The device that this file system is mounted on
- The basic block size of the file system
- Dirty flag, to indicate that the superblock has been changed but not written back to disk
- File system type
- Flags, such as a read-only flag
- Pointer to the root of the file system directory
- List of open files
- Semaphore for controlling access to the file system
- List of superblock operations

The last item on the preceding list refers to an operations object contained within the superblock object. The operations object defines the object methods (functions) that the kernel can invoke against the superblock object. The methods defined for the superblock object include the following:

- `read_inode`: Read a specified inode from a mounted file system.
- `write_inode`: Write given inode to disk.
- `put_inode`: Release inode.
- `delete_inode`: Delete inode from disk.
- `notify_change`: Called when inode attributes are changed.
- `put_super`: Called by the VFS on unmount to release the given superblock.
- `write_super`: Called when the VFS decides that the superblock needs to be written to disk.
- `statfs`: Obtain file system statistics.
- `remount_fs`: Called by the VFS when the file system is remounted with new mount options.
- `clear_inode`: Release inode and clear any pages containing related data.

The Inode Object

An inode is associated with each file. The inode object holds all the information about a named file except its name and the actual data contents of the file. Items contained in an inode object include owner, group, permissions, access times for a file, size of data it holds, and number of links.

The inode object also includes an inode operations object that describes the file system's implemented functions that the VFS can invoke on an inode. The methods defined for the inode object include the following:

- `create`: Creates a new inode for a regular file associated with a dentry object in some directory
- `lookup`: Searches a directory for an inode corresponding to a file name

- `mkdir`: Creates a new inode for a directory associated with a dentry object in some directory

The Dentry Object

A dentry (directory entry) is a specific component in a path. The component may be either a directory name or a file name. Dentry objects facilitate access to files and directories and are used in a dentry cache for that purpose. The dentry object includes a pointer to the inode and superblock. It also includes a pointer to the parent dentry and pointers to any subordinate dentrys.

The File Object

The file object is used to represent a file opened by a process. The object is created in response to the open() system call and destroyed in response to the close() system call. The file object consists of a number of items, including the following:

- Dentry object associated with the file
- File system containing the file
- File objects usage counter
- User's user ID
- User's group ID
- File pointer, which is the current position in the file from which the next operation will take place

The file object also includes an inode operations object that describes the file system's implemented functions that the VFS can invoke on a file object. The methods defined for the file object include read, write, open, release, and lock.

12.10 WINDOWS FILE SYSTEM

The developers of Windows designed a new file system, the New Technology File System (NTFS), that is intended to meet high-end requirements for workstations and servers. Examples of high-end applications include the following:

- Client/server applications such as file servers, compute servers, and database servers
- Resource-intensive engineering and scientific applications
- Network applications for large corporate systems

This section provides an overview of NTFS.

Key Features of NTFS

NTFS is a flexible and powerful file system built, as we shall see, on an elegantly simple file system model. The most noteworthy features of NTFS include the following:

- **Recoverability:** High on the list of requirements for the new Windows file system was the ability to recover from system crashes and disk failures. In the

WINDOWS/LINUX COMPARISON	
Windows	**Linux**
Windows supports a variety of file systems, including the legacy FAT/FAT32 file systems from DOS/Windows and formats common to CDs and DVDs	Linux supports a variety of file systems, including Microsoft file systems, for compatibility and interoperation
The most common file system used in Windows is NTFS, which has many advanced features related to security, encryption, compression, journaling, change notifications, and indexing built in	The most common file systems are Ext2, Ext3, and IBM's JFS journaling file system
NTFS uses logging of metadata to avoid having to perform file system checks after crashes	With Ext3, journaling of changes allows file system checks to be avoided after crashes
Windows file systems are implemented as device drivers, and can be stacked in layers, as with other device drivers, due to the object-oriented implementation of Windows I/O. Typically NTFS is sandwiched between 3rd party filter drivers, which implement functions like anti-virus, and the volume management drivers, which implement RAID	Linux file systems are implemented using the Virtual File System (VFS) technique developed by Sun Microsystems. File systems are plug-ins in the VFS model, which is similar to the general object-oriented model used for block and character devices
The file systems depend heavily on the I/O system and the CACHE manager. The cache manager is a virtual file cache that maps regions of files into kernel virtual-address space	Linux uses a page cache which keeps copies of recently used pages in memory. Pages are organized per 'owner': most commonly an inode for files and directories, or the inode of the block device for file system metadata
The CACHE manager is implemented on top of the virtual memory system, providing a unified caching mechanism for both pages and file blocks	The Linux virtual memory system builds memory-mapping of files on top of the page cache facility
Directories, bitmaps, file and file system metadata, are all represented as files by NTFS, and thus all rely on unified caching by the CACHE manager	The Common File System model of VFS treats directory entries and file inodes and other file system metadata, such as the superblock, separately from file data with special caching for each category. File data can be stored in the cache twice, once for the 'file' owner and once for the 'block device' owner
Pre-fetching of disk data uses sophisticated algorithms which remember past access patterns of code and data for applications, and initiate asynchronous pagefault operations when applications launch, move to the foreground, or resume from the system hibernate power-off state	Pre-fetching of disk data uses read-ahead of files that are being accessed sequentially

event of such failures, NTFS is able to reconstruct disk volumes and return them to a consistent state. It does this by using a transaction processing model for changes to the file system; each significant change is treated as an atomic action that is either entirely performed or not performed at all. Each transaction that was in process at the time of a failure is subsequently backed out or

brought to completion. In addition, NTFS uses redundant storage for critical file system data, so that failure of a disk sector does not cause the loss of data describing the structure and status of the file system.

- **Security:** NTFS uses the Windows object model to enforce security. An open file is implemented as a file object with a security descriptor that defines its security attributes. The security descriptor is persisted as an attribute of each file on disk.
- **Large disks and large files:** NTFS supports very large disks and very large files more efficiently than most other file systems, including FAT.
- **Multiple data streams:** The actual contents of a file are treated as a stream of bytes. In NTFS it is possible to define multiple data streams for a single file. An example of the utility of this feature is that it allows Windows to be used by remote Macintosh systems to store and retrieve files. On Macintosh, each file has two components: the file data and a resource fork that contains information about the file. NTFS treats these two components as two data streams.
- **Journaling:** NTFS keeps a log of all changes made to files on the volumes. Programs, such as desktop search, can read the journal to identify what files have changed.
- **Compression and Encryption:** Entire directories and individual files can be transparently compressed and/or encrypted.

NTFS Volume and File Structure

NTFS makes use of the following disk storage concepts:

- **Sector:** The smallest physical storage unit on the disk. The data size in bytes is a power of 2 and is almost always 512 bytes.
- **Cluster:** One or more contiguous (next to each other on the same track) sectors. The cluster size in sectors is a power of 2.
- **Volume:** A logical partition on a disk, consisting of one or more clusters and used by a file system to allocate space. At any time, a volume consists of a file system information, a collection of files, and any additional unallocated space remaining on the volume that can be allocated to files. A volume can be all or a portion of a single disk or it can extend across multiple disks. If hardware or software RAID 5 is employed, a volume consists of stripes spanning multiple disks. The maximum volume size for NTFS is 2^{64} bytes.

The cluster is the fundamental unit of allocation in NTFS, which does not recognize sectors. For example, suppose each sector is 512 bytes and the system is configured with two sectors per cluster (one cluster = 1K bytes). If a user creates a file of 1600 bytes, two clusters are allocated to the file. Later, if the user updates the file to 3200 bytes, another two clusters are allocated. The clusters allocated to a file need not be contiguous; it is permissible to fragment a file on the disk. Currently, the maximum file size supported by NTFS is 2^{32} clusters, which is equivalent to a maximum of 2^{48} bytes. A cluster can have at most 2^{16} bytes.

Table 12.5 Windows NTFS Partition and Cluster Sizes

Volume Size	Sectors per Cluster	Cluster Size
512 Mbyte	1	512 bytes
512 Mbyte–1 Gbyte	2	1K
1 Gbyte–2 Gbyte	4	2K
2 Gbyte–4 Gbyte	8	4K
4 Gbyte–8 Gbyte	16	8K
8 Gbyte–16 Gbyte	32	16K
16 Gbyte–32 Gbyte	64	32K
> 32 Gbyte	128	64K

The use of clusters for allocation makes NTFS independent of physical sector size. This enables NTFS to support easily nonstandard disks that do not have a 512-byte sector size and to support efficiently very large disks and very large files by using a larger cluster size. The efficiency comes from the fact that the file system must keep track of each cluster allocated to each file; with larger clusters, there are fewer items to manage.

Table 12.5 shows the default cluster sizes for NTFS. The defaults depend on the size of the volume. The cluster size that is used for a particular volume is established by NTFS when the user requests that a volume be formatted.

NTFS Volume Layout NTFS uses a remarkably simple but powerful approach to organizing information on a disk volume. Every element on a volume is a file, and every file consists of a collection of attributes. Even the data contents of a file is treated as an attribute. With this simple structure, a few general-purpose functions suffice to organize and manage a file system.

Figure 12.19 shows the layout of an NTFS volume, which consists of four regions. The first few sectors on any volume are occupied by the **partition boot sector** (although it is called a sector, it can be up to 16 sectors long), which contains information about the volume layout and the file system structures as well as boot startup information and code. This is followed by the **master file table (MFT)**, which contains information about all of the files and folders (directories) on this NTFS volume. In essence, the MFT is a list of all files and their attributes on this NTFS volume, organized as a set of rows in a relational database structure.

Following the MFT is a region, typically about 1 Mbyte in length, containing **system files**. Among the files in this region are the following:

- **MFT2:** A mirror of the first three rows of the MFT, used to guarantee access to the MFT in the case of a single-sector failure

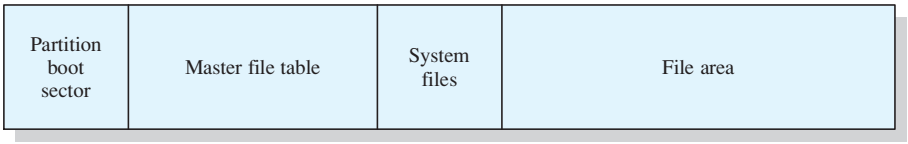


Figure 12.19 **NTFS Volume Layout**

- **Log file:** A list of transaction steps used for NTFS recoverability
- **Cluster bit map:** A representation of the volume, showing which clusters are in use
- **Attribute definition table:** Defines the attribute types supported on this volume and indicates whether they can be indexed and whether they can be recovered during a system recovery operation

Master File Table

The heart of the Windows file system is the MFT. The MFT is organized as a table of 1024-byte rows, called records. Each row describes a file on this volume, including the MFT itself, which is treated as a file. If the contents of a file are small enough, then the entire file is located in a row of the MFT. Otherwise, the row for that file contains partial information and the remainder of the file spills over into other available clusters on the volume, with pointers to those clusters in the MFT row of that file.

Each record in the MFT consists of a set of attributes that serve to define the file (or folder) characteristics and the file contents. Table 12.6 lists the attributes that may be found in a row, with the required attributes indicated by shading.

Recoverability

NTFS makes it possible to recover the file system to a consistent state following a system crash or disk failure. The key elements that support recoverability are as follows (Figure 12.20):

- **I/O manager:** Includes the NTFS driver, which handles the basic open, close, read, write functions of NTFS. In addition, the software RAID module FTDISK can be configured for use.

Table 12.6 Windows NTFS File and Directory Attribute Types

Attribute Type	Description
Standard information	Includes access attributes (read-only, read/write, etc.); time stamps, including when the file was created or last modified; and how many directories point to the file (link count).
Attribute list	A list of attributes that make up the file and the file reference of the MFT file record in which each attribute is located. Used when all attributes do not fit into a single MFT file record.
File name	A file or directory must have one or more names.
Security descriptor	Specifies who owns the file and who can access it.
Data	The contents of the file. A file has one default unnamed data attribute and may have one or more named data attributes.
Index root	Used to implement folders.
Index allocation	Used to implement folders.
Volume information	Includes volume-related information, such as the version and name of the volume.
Bitmap	Provides a map representing records in use on the MFT or folder.

Note: Colored rows refer to required file attributes; the other attributes are optional.

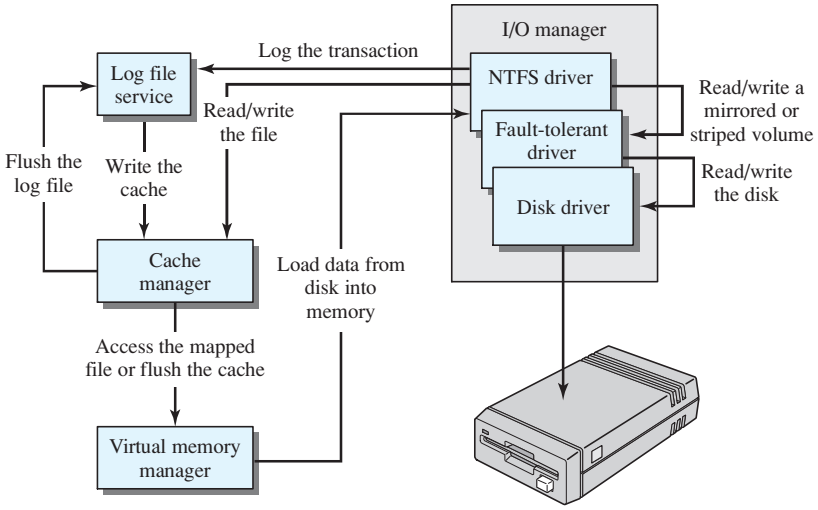


Figure 12.20 Windows NTFS Components

- **Log file service:** Maintains a log of file system metadata changes on disk. The log file is used to recover an NTFS-formatted volume in the case of a system failure (i.e., without having to run the file system check utility).
- **Cache manager:** Responsible for caching file reads and writes to enhance performance. The cache manager optimizes disk I/O.
- **Virtual memory manager:** The NTFS accesses cached files by mapping file references to virtual memory references and reading and writing virtual memory.

It is important to note that the recovery procedures used by NTFS are designed to recover file system metadata, not file contents. Thus, the user should never lose a volume or the directory/file structure of an application because of a crash. However, user data are not guaranteed by the file system. Providing full recoverability, including user data, would make for a much more elaborate and resource-consuming recovery facility.

The essence of the NTFS recovery capability is logging. Each operation that alters a file system is treated as a transaction. Each suboperation of a transaction that alters important file system data structures is recorded in a log file before being recorded on the disk volume. Using the log, a partially completed transaction at the time of a crash can later be redone or undone when the system recovers.

In general terms, these are the steps taken to ensure recoverability, as described in [RUSS05]:

1. NTFS first calls the log file system to record in the log file in the cache any transactions that will modify the volume structure.
2. NTFS modifies the volume (in the cache).

3. The cache manager calls the log file system to prompt it to flush the log file to disk.
4. Once the log file updates are safely on disk, the cache manager flushes the volume changes to disk.

12.11 SUMMARY

A file management system is a set of system software that provides services to users and applications in the use of files, including file access, directory maintenance, and access control. The file management system is typically viewed as a system service that itself is served by the operating system, rather than being part of the operating system itself. However, in any system, at least part of the file management function is performed by the operating system.

A file consists of a collection of records. The way in which these records may be accessed determines its logical organization, and to some extent its physical organization on disk. If a file is primarily to be processed as a whole, then a sequential file organization is the simplest and most appropriate. If sequential access is needed but random access to individual file is also desired, then an indexed sequential file may give the best performance. If access to the file is principally at random, then an indexed file or hashed file may be the most appropriate.

Whatever file structure is chosen, a directory service is also needed. This allows files to be organized in a hierarchical fashion. This organization is useful to the user in keeping track of files and is useful to the file management system in providing access control and other services to users.

File records, even when of fixed size, generally do not conform to the size of a physical disk block. Accordingly, some sort of blocking strategy is needed. A tradeoff among complexity, performance, and space utilization determines the blocking strategy to be used.

A key function of any file management scheme is the management of disk space. Part of this function is the strategy for allocating disk blocks to a file. A variety of methods have been employed, and a variety of data structures have been used to keep track of the allocation for each file. In addition, the space on disk that has not been allocated must be managed. This latter function primarily consists of maintaining a disk allocation table indicating which blocks are free.

12.12 RECOMMENDED READING

There are a number of good books on file management. The following all focus on file management systems but also address related operating system issues. Perhaps the most useful is [WIED87], which takes a quantitative approach to file management and deals with all of the issues raised in Figure 12.2, from disk scheduling to file structure. [LIVA90] emphasizes file structures, providing a good and lengthy survey with comparative performance analyses. [GROS86] provides a balanced look at issues relating to both file I/O and file access methods. It also contains general descriptions of all of the control structures needed by a file system. These provide a useful checklist in assessing a file system design. [FOLK98] emphasizes the

processing of files, addressing such issues as maintenance, searching and sorting, and sharing.

The Linux file system is examined in detail in [LOVE05] and [BOVE03]. A good overview is [RUBI97].

[CUST94] provides a good overview of the NT file system. [NAGA97] covers the material in more detail.

BOVE03 Bovet, D., and Cesati, M. *Understanding the Linux Kernel.* Sebastopol, CA: O'Reilly, 2003.

CUST94 Custer, H. *Inside the Windows NT File System.* Redmond, WA: Microsoft Press, 1994.

FOLK98 Folk, M., and Zoellick, B. *File Structures: An Object-Oriented Approach with C++.* Reading, MA: Addison-Wesley, 1998.

GROS86 Grosshans, D. *File Systems: Design and Implementation.* Englewood Cliffs, NJ: Prentice Hall, 1986.

LIVA90 Livadas, P. *File Structures: Theory and Practice.* Englewood Cliffs, NJ: Prentice Hall, 1990.

LOVE05 Love, R. *Linux Kernel Development.* Waltham, MA: Novell Press, 2005.

NAGA97 Nagar, R. *Windows NT File System Internals.* Sebastopol, CA: O'Reilly, 1997.

RUBI97 Rubini, A. "The Virtual File System in Linux." *Linux Journal*, May 1997.

WIED87 Wiederhold, G. *File Organization for Database Design.* New York: McGraw-Hill, 1987.

12.13 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

access method	file allocation	indexed sequential file
bit table	file allocation table	inode
block	file directory	key field
chained file allocation	file management system	pathname
contiguous file allocation	file name	pile
database	hashed file	record
disk allocation table	indexed file	sequential file
field	indexed file allocation	working directory
file		

Review Questions

12.1 What is the difference between a field and a record?

12.2 What is the difference between a file and a database?

12.3 What is a file management system?

12.4 What criteria are important in choosing a file organization?

12.5 List and briefly define five file organizations.

12.6 Why is the average search time to find a record in a file less for an indexed sequential file than for a sequential file?

12.7 What are typical operations that may be performed on a directory?

12.8 What is the relationship between a pathname and a working directory?

12.9 What are typical access rights that may be granted or denied to a particular user for a particular file?

12.10 List and briefly define three blocking methods.

12.11 List and briefly define three file allocation methods.

Problems

12.1 Define:

B = block size

R = record size

P = size of block pointer

F = blocking factor; expected number of records within a block

Give a formula for F for the three blocking methods depicted in Figure 12.6.

12.2 One scheme to avoid the problem of preallocation versus waste or lack of contiguity is to allocate portions of increasing size as the file grows. For example, begin with a portion size of one block, and double the portion size for each allocation. Consider a file of n records with a blocking factor of F, and suppose that a simple one-level index is used as a file allocation table.

a. Give an upper limit on the number of entries in the file allocation table as a function of F and n.

b. What is the maximum amount of the allocated file space that is unused at any time?

12.3 What file organization would you choose to maximize efficiency in terms of speed of access, use of storage space, and ease of updating (adding/deleting/modifying) when the data are

a. updated infrequently and accessed frequently in random order?

b. updated frequently and accessed in its entirety relatively frequently?

c. updated frequently and accessed frequently in random order?

12.4 Ignoring overhead for directories and file descriptors, consider a file system in which files are stored in blocks of 16K bytes. For each of the following file sizes, calculate the percentage of wasted file space due to incomplete filling of the last block: 41,600 bytes; 640,000 bytes; 4.064,000 bytes.

12.5 What are the advantages of using directories?

12.6 Directories can be implemented either as "special files" that can only be accessed in limited ways, or as ordinary data files. What are the advantages and disadvantages of each approach?

12.7 Some operating systems have a tree-structured file system but limit the depth of the tree to some small number of levels. What effect does this limit have on users? How does this simplify file system design (if it does)?

12.8 Consider a hierarchical file system in which free disk space is kept in a free space list.

a. Suppose the pointer to free space is lost. Can the system reconstruct the free space list?

b. Suggest a scheme to ensure that the pointer is never lost as a result of a single memory failure.

12.9 In UNIX System V, the length of a block is 1 Kbyte, and each block can hold a total of 256 block addresses. Using the inode scheme, what is the maximum size of a file?

12.10 Consider the organization of a UNIX file as represented by the inode (Figure 12.14). Assume that there are 12 direct block pointers, and a singly, doubly, and triply indirect pointer in each inode. Further, assume that the system block size and the disk sector size are both 8K. If the disk block pointer is 32 bits, with 8 bits to identify the physical disk and 24 bits to identify the physical block, then

a. What is the maximum file size supported by this system?

b. What is the maximum file system partition supported by this system?

c. Assuming no information other than that the file inode is already in main memory, how many disk accesses are required to access the byte in position 13,423,956?