

File-System Interface



For most users, the file system is the most visible aspect of a general-purpose operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system. Most file systems live on storage devices, which we described in Chapter 11 and will continue to discuss in the next chapter. In this chapter, we consider the various aspects of files and the major directory structures. We also discuss the semantics of sharing files among multiple processes, users, and computers. Finally, we discuss ways to handle file protection, necessary when we have multiple users and want to control who may access files and how files may be accessed.

CHAPTER OBJECTIVES

- Explain the function of file systems.
- Describe the interfaces to file systems.
- Discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures.
- Explore file-system protection.

13.1 File Concept

Computers can store information on various storage media, such as NVM devices, HDDs, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of stored information. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent between system reboots.

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.

Because files are **the** method users and applications use to store and retrieve data, and because they are so general purpose, their use has stretched beyond its original confines. For example, UNIX, Linux, and some other operating systems provide a proc file system that uses file-system interfaces to provide access to system information (such as process details).

The information in a file is defined by its creator. Many different types of information may be stored in a file—source or executable programs, numeric or text data, photos, music, video, and so on. A file has a certain defined structure, which depends on its type. A **text fil** is a sequence of characters organized into lines (and possibly pages). A **source fil** is a sequence of functions, each of which is further organized as declarations followed by executable statements. An **executable fil** is a series of code sections that the loader can bring into memory and execute.

13.1.1 File Attributes

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as `example.c`. Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not. When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file `example.c`, and another user might edit that file by specifying its name. The file's owner might write the file to a USB drive, send it as an e-mail attachment, or copy it across a network, and it could still be called `example.c` on the destination system. Unless there is a sharing and synchronization method, that second copy is now independent of the first and can be changed separately.

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

- **Name.** The symbolic file name is the only information kept in human-readable form.
- **Identifie**. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type.** This information is needed for systems that support different types of files.
- **Location.** This information is a pointer to a device and to the location of the file on that device.

- **Size.** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection.** Access-control information determines who can do reading, writing, executing, and so on.
- **Timestamps and user identification.** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

Some newer file systems also support **extended file attributes**, including character encoding of the file and security features such as a file checksum. Figure 13.1 illustrates a **file info window** on macOS that displays a file's attributes.

The information about all files is kept in the directory structure, which resides on the same device as the files themselves. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes. It may take more than a kilobyte to record this information for each file. In a system with many files, the size of the directory itself may be megabytes or gigabytes. Because directories must match the volatility of the files, like files, they must be stored on the device and are usually brought into memory piecemeal, as needed.

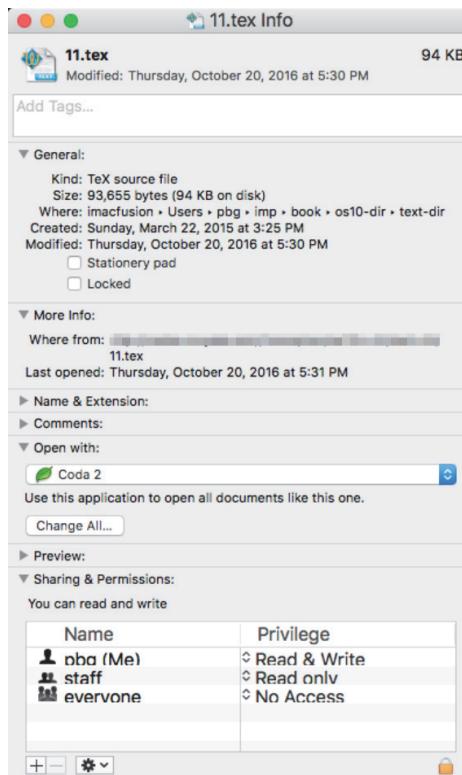


Figure 13.1 A file info window on macOS.

13.1.2 File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let's examine what the operating system must do to perform each of these seven basic file operations. It should then be easy to see how other similar operations, such as renaming a file, can be implemented.

- **Creating a fil** . Two steps are necessary to create a file. First, space in the file system must be found for the file. We discuss how to allocate space for the file in Chapter 14. Second, an entry for the new file must be made in a directory.
- **Opening a fil** . Rather than have all file operations specify a file name, causing the operating system to evaluate the name, check access permissions, and so on, all operations except create and delete require a file `open()` first. If successful, the `open` call returns a file handle that is used as an argument in the other calls.
- **Writing a fil** . To write a file, we make a system call specifying both the open file handle and the information to be written to the file. The system must keep a **write pointer** to the location in the file where the next write is to take place if it is sequential. The write pointer must be updated whenever a write occurs.
- **Reading a fil** . To read from a file, we use a system call that specifies the file handle and where (in memory) the next block of the file should be put. Again, the system needs to keep a **read pointer** to the location in the file where the next read is to take place, if sequential. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **current-file-position pointer**. Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **Repositioning within a file** . The current-file-position pointer of the open file is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file **seek**.
- **Deleting a fil** . To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase or mark as free the directory entry. Note that some systems allow **hard links**—multiple names (directory entries) for the same file. In this case the actual file contents is not deleted until the last link is deleted.
- **Truncating a fil** . The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length. The file can then be reset to length zero, and its file space can be released.

These seven basic operations comprise the minimal set of required file operations. Other common operations include appending new information to the end of an existing file and renaming an existing file. These primitive operations can then be combined to perform other file operations. For instance, we can create a copy of a file by creating a new file and then reading from the old and writing to the new. We also want to have operations that allow a user to get and set the various attributes of a file. For example, we may want to have operations that allow a user to determine the status of a file, such as the file's length, and to set file attributes, such as the file's owner.

As mentioned, most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an `open()` system call be made before a file is first used. The operating system keeps a table, called the **open-file table**, containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required. When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table, potentially releasing locks. `create()` and `delete()` are system calls that work with closed rather than open files.

Some systems implicitly open a file when the first reference to it is made. The file is automatically closed when the job or program that opened the file terminates. Most systems, however, require that the programmer open a file explicitly with the `open()` system call before that file can be used. The `open()` operation takes a file name and searches the directory, copying the directory entry into the open-file table. The `open()` call can also accept access-mode information—create, read-only, read-write, append-only, and so on. This mode is checked against the file's permissions. If the request mode is allowed, the file is opened for the process. The `open()` system call typically returns a pointer to the entry in the open-file table. This pointer, not the actual file name, is used in all I/O operations, avoiding any further searching and simplifying the system-call interface.

The implementation of the `open()` and `close()` operations is more complicated in an environment where several processes may open the file simultaneously. This may occur in a system where several different applications open the same file at the same time. Typically, the operating system uses two levels of internal tables: a per-process table and a system-wide table. The per-process table tracks all files that a process has open. Stored in this table is information regarding the process's use of the file. For instance, the current file pointer for each file is found here. Access rights to the file and accounting information can also be included.

Each entry in the per-process table in turn points to a system-wide open-file table. The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file. When another process executes an `open()` call, a new entry is simply added to the process's open-file table pointing to the appropriate entry in the system-wide table. Typically, the open-file table also has an **open count** associated with each file to indicate how many processes have the file open. Each `close()` decreases this open count, and when the open count reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.

FILE LOCKING IN JAVA

In the Java API, acquiring a lock requires first obtaining the `FileChannel` for the file to be locked. The `lock()` method of the `FileChannel` is used to acquire the lock. The API of the `lock()` method is

```
FileLock lock(long begin, long end, boolean shared)
```

where `begin` and `end` are the beginning and ending positions of the region being locked. Setting `shared` to `true` is for shared locks; setting `shared` to `false` acquires the lock exclusively. The lock is released by invoking the `release()` of the `FileLock` returned by the `lock()` operation.

The program in Figure 13.2 illustrates file locking in Java. This program acquires two locks on the file `file.txt`. The lock for the first half of the file is an exclusive lock; the lock for the second half is a shared lock.

In summary, several pieces of information are associated with an open file.

- **File pointer.** On systems that do not include a file offset as part of the `read()` and `write()` system calls, the system must track the last read-write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
- **File-open count.** As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Multiple processes may have opened a file, and the system must wait for the last file to close before removing the open-file table entry. The file-open count tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
- **Location of the file.** Most file operations require the system to read or write data within the file. The information needed to locate the file (wherever it is located, be it on mass storage, on a file server across the network, or on a RAM drive) is kept in memory so that the system does not have to read it from the directory structure for each operation.
- **Access rights.** Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

Some operating systems provide facilities for locking an open file (or sections of a file). File locks allow one process to lock a file and prevent other processes from gaining access to it. File locks are useful for files that are shared by several processes—for example, a system log file that can be accessed and modified by a number of processes in the system.

File locks provide functionality similar to reader-writer locks, covered in Section 7.1.2. A **shared lock** is akin to a reader lock in that several processes can acquire the lock concurrently. An **exclusive lock** behaves like a writer lock; only one process at a time can acquire such a lock. It is important to note that not

```

import java.io.*;
import java.nio.channels.*;

public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;

    public static void main(String args[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;

        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");

            // get the channel for the file
            FileChannel ch = raf.getChannel();

            // this locks the first half of the file - exclusive
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);

            /** Now modify the data . . . */

            // release the lock
            exclusiveLock.release();

            // this locks the second half of the file - shared
            sharedLock = ch.lock(raf.length()/2+1, raf.length(), SHARED);

            /** Now read the data . . . */

            // release the lock
            sharedLock.release();
        } catch (java.io.IOException ioe) {
            System.err.println(ioe);
        }
        finally {
            if (exclusiveLock != null)
                exclusiveLock.release();
            if (sharedLock != null)
                sharedLock.release();
        }
    }
}

```

Figure 13.2 File-locking example in Java.

all operating systems provide both types of locks: some systems provide only exclusive file locking.

Furthermore, operating systems may provide either **mandatory** or **advisory** file-locking mechanisms. With mandatory locking, once a process acquires an exclusive lock, the operating system will prevent any other process from

accessing the locked file. For example, assume a process acquires an exclusive lock on the file `system.log`. If we attempt to open `system.log` from another process—for example, a text editor—the operating system will prevent access until the exclusive lock is released. Alternatively, if the lock is advisory, then the operating system will not prevent the text editor from acquiring access to `system.log`. Rather, the text editor must be written so that it manually acquires the lock before accessing the file. In other words, if the locking scheme is mandatory, the operating system ensures locking integrity. For advisory locking, it is up to software developers to ensure that locks are appropriately acquired and released. As a general rule, Windows operating systems adopt mandatory locking, and UNIX systems employ advisory locks.

The use of file locks requires the same precautions as ordinary process synchronization. For example, programmers developing on systems with mandatory locking must be careful to hold exclusive file locks only while they are accessing the file. Otherwise, they will prevent other processes from accessing the file as well. Furthermore, some measures must be taken to ensure that two or more processes do not become involved in a deadlock while trying to acquire file locks.

13.1.3 File Types

When we design a file system—indeed, an entire operating system—we always consider whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a user tries to output the binary-object form of a program. This attempt normally produces garbage; however, the attempt can succeed if the operating system has been told that the file is a binary-object program.

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an extension, usually separated by a period (Figure 13.3). In this way, the user and the operating system can tell from the name alone what the type of a file is. Most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension made up of additional characters. Examples include `resume.docx`, `server.c`, and `ReaderThread.cpp`.

The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a `.com`, `.exe`, or `.sh` extension can be executed, for instance. The `.com` and `.exe` files are two forms of binary executable files, whereas the `.sh` file is a **shell script** containing, in ASCII format, commands to the operating system. Application programs also use extensions to indicate file types in which they are interested. For example, Java compilers expect source files to have a `.java` extension, and the Microsoft Word word processor expects its files to end with a `.doc` or `.docx` extension. These extensions are not always required, so a user may specify a file without the extension (to save typing), and the application will look for a file with the given name and the extension it expects. Because these extensions are not supported by the operating system, they can be considered “hints” to the applications that operate on them.

Consider, too, the macOS operating system. In this system, each file has a type, such as `.app` (for application). Each file also has a creator attribute