CHAPTER

11 File Systems

Learning Objectives

After reading this chapter, you should be able to:

- Explain the concept of file, directory, file system, and file management system.
- Describe how data is stored in files and directories.
- Describe the organization of a file system into directories.
- Explain how file systems are constructed on disks.
- Discuss the management of space in file systems.

11.1 Introduction

Users make use of computers to create-, store-, retrieve-, and manipulate information. They see information in the form of programs and data (henceforth referred to as data only for this chapter). Data are stored in a variety of physical storage devices that have different physical characteristics. Storage devices include disks, floppies, CDs, tapes, flashes, etc., and they can retain data across power disconnections.

As stated earlier, where usage is concerned I/O devices are the most complex hardware units in computer systems. It was noted in Chapter 10 that a device driver interfaces each device with the operating system. Device drivers provide uniform interface functionalities to enable access contents from devices in the raw format. Nevertheless, it is difficult and inconvenient for application developers to use the driver interface to manipulate raw data stored in devices. Data processing becomes even more difficult if devices are used to store data from many users and are accessed simultaneously by them.

In this chapter, we will study how the operating system helps users access data from storage devices. It implements abstract software objects called files to store data in storage devices. Management of these software objects is the theme of study in this chapter. We particularly study how the

≫ Although programs and data may be two different entities for users, but for the management of file systems they are one and the same and are treated simply as byte strings.

≫ The file abstraction provides a uniform logical view of physical contents from a wide variety of storage devices that have different physical characteristics.

operating system constructs file systems on disks, stores them there, organize them into directories, and manages these entities.

11.2 File Systems

Users create- and store data in a computer for later retrieval and manipulation. The computer system normally stores data in storage devices so that the data survives power disconnections. Users associate a name to a piece of data so that they can identify that particular piece later by using its name. An operating system implements a software layer on the top of the I/O subsystem (device drivers) for users to access data with ease from storage devices. The software layer is called the *file management system* (FMS) and sometimes, naively called the *file system*.[1] In this chapter we use FMS to mean the file management system, and file system to indicate where data is stored in devices such as the disk. The FMS provides an abstraction of software objects called *files*, (see Fig. 11.1).

An FMS organizes a particular storage area as a collection of "access units" called blocks. Those blocks collectively store a single file system. The FMS applies operations on blocks to manipulate the file system. The collection of all the blocks of a file system is called a *partition* if all blocks belong to a single storage medium, or a *volume*. Volume is a generic term, and a volume may contain blocks from a part of a storage medium, or the entire storage medium, or several storage mediums. Information such as the name, size, etc., about the partition or the volume is vital for any file system and is, therefore, kept in a special block called the superblock, (discussed in Sections 11.5 and 11.10). Some file management systems create in-memory superblocks on the fly for faster access.

Users always see their stored data in terms of files. The file abstraction helps users to handle their data in device-independent manner. For them, a file is a collection of related records and they can access those records atleast sequentially. The FMS carves out file objects from device-dependent physical signal patterns stored on the device storage mediums. It implements necessary functionalities so that users can create, manipulate-, store-, retrieve-, and organize files with relative ease. One fundamental objective of the FMS is to map the user view of files to the system view of blocks. Usually, a collection of blocks is assigned to hold the data of a single file. Maintaining which group of blocks belongs to which file is a primary function of the FMS. An additional duty of the FMS is to protect files from unauthorized accesses.

The FMS maintains a collection of file objects in storage devices. The file objects are collectively called a file system. A file system is a user-level view of a storage device. It stores file contents in storage devices, and the FMS is responsible for deciding where in the device the contents reside.

As mentioned above, the file is a device-independent concept. Loosely, a file stores a collection of related data in the form of a sequence of bytes. It is

≫ The concept of the file is well-known in our day-to-day life. We use files to maintain various records containing useful data. Minimally, it has a name for identification, space to store records, a predefined location for convenient access, and, of course, suitable access restrictions. Similarly, a file in a computer system has a name for its identification, space to store data, a location for convenient access, access restrictions, and other attributes and support mechanisms.

≫ The concepts of partition and volume define the logical boundary of the storage system for hosting a file system.

≫ There may be many different FMSes in a computer system; each manages a specific set of files. Here we consider only a single FMS.

[1] Many authors use the term file system to mean both the management system and the stored data. But, for ease in the exposition of the subject material, we treat them differently in this book. Some authors use the term 'file manager' for what we call FMS.

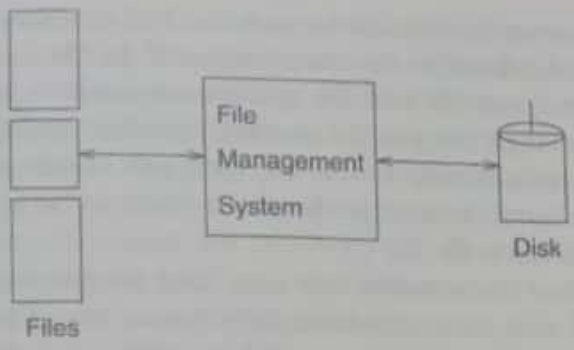

Figure 11.1: *An abstract view of a file management system.*

the (user) applications that see structures in the byte sequence. The applications can read and write the bytes; they may specify starting positions from where to read or write bytes and the number of bytes to read/write. The FMS helps applications in storing and retrieving bytes from the files. It maps (abstract) files to different devices in a device-dependent manner even though file is a device-independent concept. Users identify a particular piece of data by a filename and a relative- or logical position within the file. The FMS translates the filename and the relative positions into the appropriate physical locations in the devices. It helps applications to transfer data between application spaces and files.

≫ To manipulate files, users do not need to know the physical properties of storage devices.

Users always see data in storage devices in the abstraction of files. As far as users are concerned, files are units of data storage. As mentioned previously, users associate a name (filename) to a piece of stored data so that they can identify the data later using the filename. A filename is merely some symbolic information to identify a particular file in the file system. Names are generally human readable character strings.

A typical (modern) file system maintains a file as an unnamed object. That is, the name is not a part of the file content proper, and the file management system does not assign the file object a user-comprehensible name. The FMS stores files as abstract unnamed objects, and helps users set up a correspondence between filenames and file objects by a different means. It manages a collection of files and implements a set of standard operations to access file contents and to organize files. Applications manipulate files by invoking file operations without much concern about where the files are physically stored, and how they are stored in the various physical mediums. The FMS translates file operations into invocations of appropriate routines in the I/O manager (device driver).

≫ The FMS contains files, directories, and control information (metadata), and supports convenient and secured access on files and directories. Files and directories are the most basic abstractions and the organization of these two units as a tree is the most visible part of the FMS.

The FMS is the most visible component of an operating system to end-users. Its duties include (1) organization of files; (2) execution of file operations; (3) synchronization of file operations; (4) protection of file contents; and (5) management of space in the file system.

11.3 File Attributes

A file stores user data. The data is called the *primary content* of the file which users are mostly interested in. For the purpose of file management, the FMS

creates- and stores additional data for each file. This data is called file *metadata* or *control data* and describes the characteristics of the file. File metadata is also stored in the file system. In most file systems, a metadata is an unnamed object that represents a single file, and the metadata describes various properties of the file by way of characterizing the file. The FMS uses metadata to identify a file, and uses it as a handle to manage the file. In short, as far as the FMS is concerned, the metadata is *the* file.

≫ Metadata is called inode in UNIX systems. It acts like a pointer to the file content proper. The primary content of a file is defined by the file owner, but the metadata is defined by the FMS. The FMS, in general, may not be concerned with the primary content, and it treats the content as a finite sequence of bytes. Metadata is considered out-of-band data that are used to describe the properties of the file.

The content of file metadata may vary from one file management system to another. For ease in manipulating information stored in metadata, it is structured into a set of small objects called *file attributes* as shown in Fig. 11.2. An attribute represents information about some entity. Metadata normally comprises the following attributes: (1) type, (2) size, (3) physical location, (4) protection information, (5) various timestamps (creation time, last read/write time, etc), (6) owner, (7) group, and (8) the number of links. The type identifies the class of the file. The owner is the user identification representing the current owner of the file; the group identifies the current group of the file. The size value specifies the current length of the file. The location identifies the position in a device where the file content proper resides. The protection field stores access control information. The timestamps keep track of which operations are applied and the times at which they were applied. Users cannot directly manipulate metadata objects the way they do the file content proper. There are system calls to set- and get file attribute values. For example, the **chmod** command in UNIX changes the file protection information.

≫ Most file management systems store only a fixed set of a priori known attributes. The recent trend is to store "user defined" attributes as well in file metadata. An attribute is simply a name and a value pair. These extended attributes help store additional information about a file. For example, if a file stores a song, the composer information can be stored as a user-defined file attribute.

11.4 File Types

A user may like to group a few files and want to store them somewhere in isolation from her other files. Also, the system might want to store files of different users separately as distinct collections. We need a folder to hold such files separately. A *folder*, most popularly called a *directory*, itself is a special file that stores information about the contained files.

Directories unlike ordinary files do not store user data proper, but contain search information about contained files and subdirectories. Certainly,

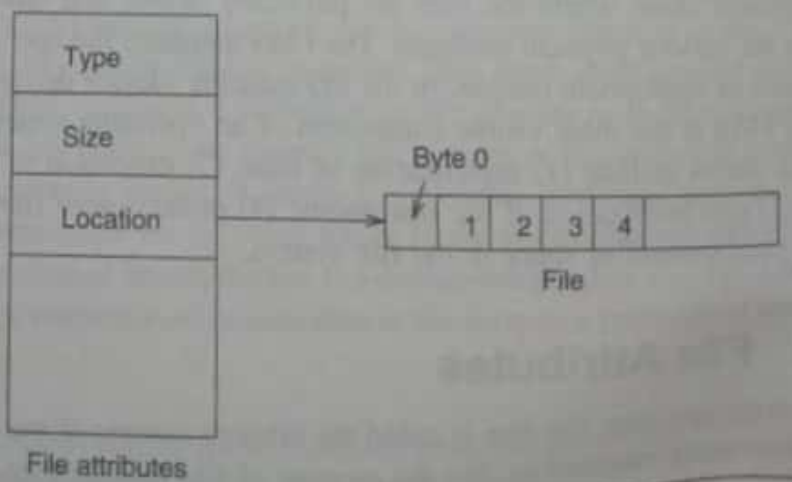

Figure 11.2: *File metadata structured as various attributes.*

directories and ordinary files are used for different purposes; directories are used to organize ordinary files and directories. Thus, the FMS may need to know what kind of file it is handling. If the FMS has correct information about the kind of a file, it can handle it in a certain way.

The FMS normally categorizes files into a few different classes or *file types*. A file may or may not have certain internal structure depending on its type. UNIX systems support the following six file types: (1) regular file, (2) directory, (3) symbolic link, (4) device (character device or block device) file, (5) FIFO (pipe and named pipe), and (6) socket. The FMS implements different operations for different file types.

A regular file is an ordinary file that contains unformatted data, and the FMS is not involved in interpreting its content. Most files in a file system are of this type. Users of these files are responsible for maintaining the internal structure of the contents. Contents of other file types might be interpreted by the FMS. Depending on the type of the file, its content is structured in the way the FMS understands. For example, in UNIX systems, a directory contains a list of entries, where each entry consists of a filename and a reference to the file metadata. This is the only way UNIX systems associate names to file objects even though file objects are nameless. We studied pipe, named pipe, and socket in Chapter 6. We discuss the other file types later in this chapter.

≫ Some FMSes are sensitive to filenames, but most are not. They use file types to handle files.

11.5 Operations

A file system is a user-level view of one storage unit or a partition of a storage unit or a volume comprising many storage units. The FMS encapsulates the physical characteristics and organizations of the storage recording mediums. It provides a logical view of a storage unit as a set of files. The FMS implements many operations that applications invoke through system calls. The operations are grouped into file operations, file-system operations, and other operations.

≫ Some systems have subtypes for regular files, and the subtype information is encoded in filenames and/or content. For example, .exe, .com, .jpg are handled differently in Windows. In UNIX executable files store a 16-bit magic number at the very beginning of the files.

11.5.1 File Operations

For management purposes, a file is treated like an abstract object that is accessed by a predefined set of operations. File operations supported by a typical FMS include (1) create, (2) delete, (3) truncate, (4) rename, (5) open, (6) read, (7) write, (8) append, (9) reposition, (10) close, etc. There are other operations for different types of files. Data in one file can be manipulated independently of data in other files: data can be added to or deleted from one file without affecting the data contained in other files. Some commonly used file operations are discussed briefly in the following subsections.

≫ The FMS provides some minimum level of operations for file system users. Sophisticated operations are developed by users themselves using basic operations. For example, copying a file into another is a utility in UNIX: the name of the utility is **cp**.

Create

Users must be able to create files and directories. The create operation requires the name of the file/directory being created, the name of its container

directory, and may require some file attribute values. This operation creates an empty file/directory, and the FMS allocates space for the metadata object, and creates an entry in the container directory. The directory entry contains the name of the new file/directory, and a reference to the metadata. The FMS initializes file attributes (in the metadata), especially of the file owner, time of its creation, and permission attributes. Depending on the space allocation policy of the FMS, the create operation may or may not allocate space to hold the file content proper. If space is allocated, the metadata records the address of the physical location of the file.

≫ Depending on the operating system and the FMS, there may be a restriction on names. In UNIX systems, a file or directory name is a sequence of ASCII characters without containing a '/' or null character. Most FMSes put a limit on the length of the file/directory name. For example, Windows XP permits a maximum of 255 characters.

Creating a directory is a slightly more complex operation than creating a file. The system in addition initializes the directory content, especially search structures, if any, used to speed up searching the directory content. Some FMSes store one- or two default entries in an empty directory: one to reference its parent (named ".." in UNIX), and the other to reference itself (named "." in UNIX). The "." directory in UNIX acts as a back pointer to its own metadata object.

Delete

Users must be able to delete their files and directories. A delete operation requires the file/directory name to be deleted. The FMS frees up the space allocated to the file and the file metadata object, and removes the file entry from the container directory. The FMS never allows deleting the root of the file system (the root of a file system is the first directory created when the file system is initialized, see Section "Initialize"). A delete operation is carried out in two steps. In the first step, the filename is removed from the container directory and the file is marked for deletion. In the second step, the actual deletion is performed only when no applications access the file any further.

≫ All operating systems put a limit on how many open files a process can have at any time instant.

Open/Close

Before an application can access a file, it needs to open the file by way of indicating to the FMS that it is interested in the file. The open operation requires a filename to be opened and may require additional optional values. The operation first checks if the user has permission to access the file. If the user does have permission, the open operation returns an object called *file descriptor* that will be used as a handle by the application to apply further operations on the file. A file descriptor represents an interactive session between a process and an open file. The open operation also creates a temporary open file object in the kernel to mediate operations on the file. In most systems, the open operation creates a pointer called *file pointer* (which acts as an offset in the file), and sets it to 0. The file pointer indicates a relative position within the file from where the next read, write, and other positional operations become effective.

≫ The file descriptor is the external handle, and the temporary file object the internal handle to operate on the same file. The file descriptor is a symbolic reference to the file object. It is a small unsigned integer in UNIX.

When an application no longer requires a file, it closes the file. The close operation requires a file descriptor that was obtained in a previous open operation. The operation releases the file descriptor and other related resources allocated for the open file session.

≫ Most operating systems automatically close all open files when an application process terminates.

Reposition

This operation adjusts a file pointer to a new offset. The operation takes a file descriptor and an offset as parameters, and sets the associated file pointer to the offset. (Reposition is applicable for those FMSes that support direct access files. If arbitrary repositioning of file pointer is not allowed, the file can only be accessed sequentially. For example, files on tape devices are accessed sequentially. Otherwise, a file is a directly accessed file, and we can read or write any number of bytes from any relative position.)

Read

A read operation takes as parameters a file descriptor, a positive integer number, and a buffer address. The operation copies into the buffer those many number of consecutive bytes starting at the current file pointer position from the file. It repositions the file pointer past the last byte it read.

Write

A write operation takes as parameters a file descriptor, a byte string, and the size of the string. The byte string is written in the file identified by the file descriptor. It overwrites the file content starting at the current file pointer position within the file. It allocates more space to the file when it needs to write past the current last byte in the file. It repositions the file pointer after the last byte written.

Truncate

A truncate operation takes as its parameters a file descriptor and a positive integer number. It reduces the size of the corresponding file to the specified number. If needed, it frees up space from the file.

≫ Freeing space from the middle of a file is not supported by any FMS currently.

Memory Map

Some operating systems allow an application process to map a complete file or parts of the file into its private address space, (see Fig. 11.3). (File mapping can be done only for those files that reside on block devices and are accessed through the block-device interface.) Memory-mapping a file creates a region in the process address space, and each byte in the region corresponds to a byte in the file. For example, if a process maps a file at logical address 0x100, then the logical address 0x100 corresponds to the 0th offset in the file, 0x101 to

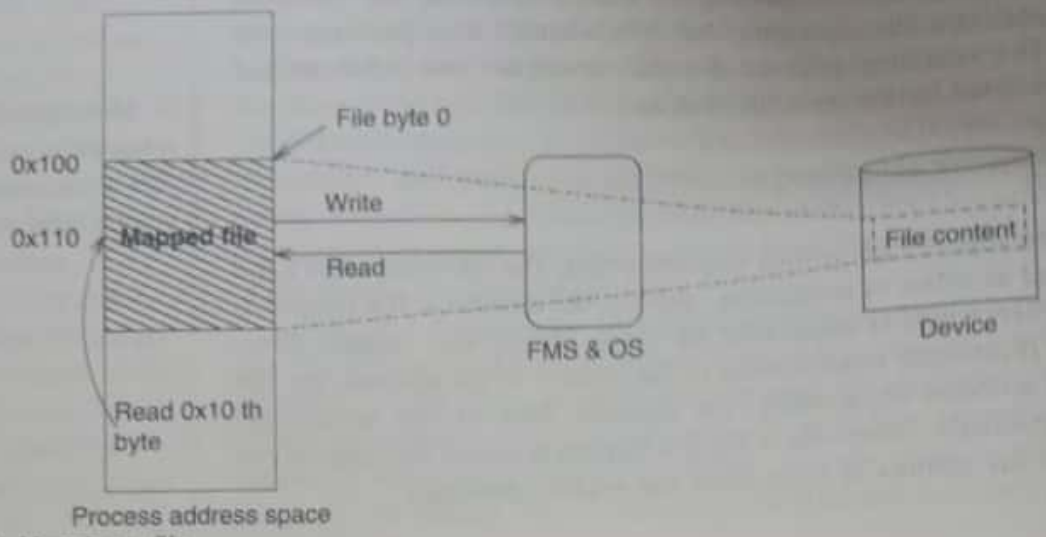

Figure 11.3: *Mapping a file in process address space.*

the 1st byte offset, and so forth. Ordinary memory reads and writes on the mapped sections by applications are treated by the system as file read and write operations, respectively. Thus, reading and writing the contents of a file become convenient for applications. When the mapped file is closed, all the modified data are written back to the file and the file is unmapped from the process address space.

≫ A file can be memory-mapped by several processes simultaneously at different logical addresses in their respective private address spaces. The operating system creates a shared memory region where the file is mapped. File mapping is a good way of doing interprocess communications in the user space.

11.5.2 File-system Operations

There are a few operations that are performed on a file system as a single object instead of on its files. The notable ones are file-system initialization, and their mount and unmount at runtime.

Initialize

Users should be able to create and initialize a file system out of a given storage partition. It is the very first operation executed on a (storage) partition to initialize an empty file system there. The operation initializes necessary physical structures (required for the management of the file system) on blocks in the partition, and links them together to form an empty file system. After the preliminary initialization, it creates the root directory. Unless the root is at a fixed location within the file system, a reference to the root is stored in the partition superblock. The superblock is always written at an a priori known location in the partition, and it records the state of the file system. The superblock also stores the file system specific management information. The superblock acts as the anchor to the file system. It is the most important data structure for the FMS. Once a file system is initialized it is for the FMS to maintain the integrity of the management data structures.

Mount

A computer system may have many file systems installed in its storage devices. There is only one file system that will be treated as the "system root" file system. The system knows the default FMS and the location of the root file system. Other file systems are mounted on the file tree hanging from the root file system. Once all the file systems are mounted, they collectively form a "single" file tree.

A file system must be mounted on the system file tree before its files can be accessed by users. A given file system can only be mounted on a directory that is accessible through the current file tree. The mounted file system hides everything underlying the previous directory as long as it remains mounted. The previous directory is called a *mount point*. Figure 11.4 displays a scenario, in UNIX, where a given file system is mounted on the "**/usr**" directory of the root file tree. The previous contents (if any), the owner, and the mode of the "**/usr**" directory become invisible as long as this file system remains mounted on the directory. The "**/usr**" directory now refers to the root directory of the mounted file system. If one changes her current directory to the "**/usr**" directory, she unwittingly moves to the root of the newly mounted file system. The operating system will channel her file-operations to the FMS of the mounted file system.

A mount point is represented by a kernel data structure that holds superblock information of the mounted file system. A mount point acts as a redirection agent or gateway to the mounted file system. The operating system maintains a runtime table, called *mount table*, that contains information pertaining to all active mount points. When a mount point directory is accessed, the mount table is searched for information about the mounted file system. We need information of the device partition that holds the mounted file system. From the table, we get the major and minor numbers of the device, which are used to obtain the appropriate device driver that interfaces the device.

≫ There are exceptions. Microsoft Windows MSDOS maintains a forest of files trees. There are no parents for the a:\ and c:\ directories. In recent Microsoft Windows, 'My Computer' is the parent of those directories.

Unmount

An unmount operation detaches a previously mounted file system from the system file tree. The unmount operation flushes data caches to the storage partition holding the mounted file system. It marks the superblock of the

≫ At the time of mounting a file system, the operating system normally verifies the integrity of the file system, and tries to recover it in case its data is corrupted.

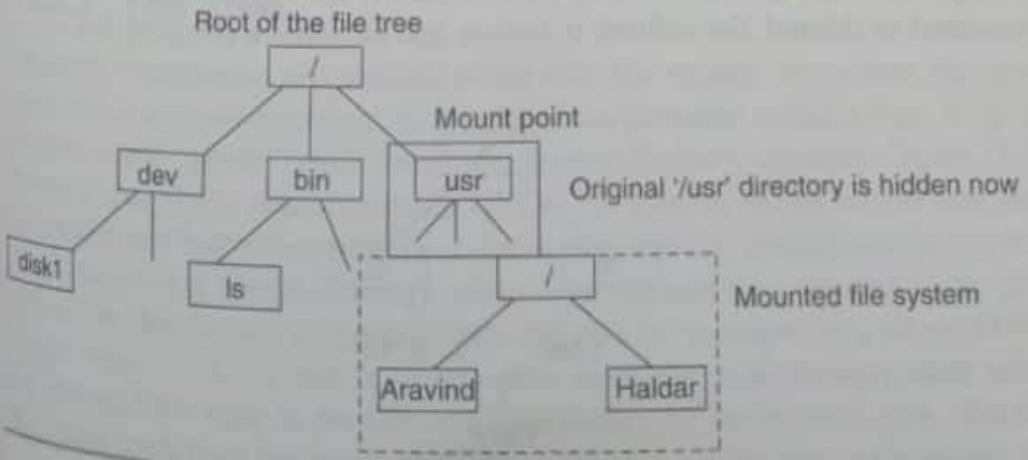

Figure 11.4: *Mounting a file system.*

mounted file system to indicate that the file system is shutdown normally. The operating system must not access the partition space unless it mounts the file system again. After a successful unmount operation, the original hidden directory becomes active again.

11.5.3 Extended Operations

In the previous two subsections, we discussed some necessary operations that almost all FMSes support. Some FMSes, in addition, support other operations. For example, UNIX/Linux systems support links. Three kinds of links are presented below.

≫ Note that a filename is not a file, but a (symbolic) reference to a file. In many systems, a single file may have many different names, and they all refer to the same metadata object representing the file. The FMS maps those different filenames to the same metadata object.

Hardlink

A hardlink, sometimes called an alias, is another name of a given file. The link file does not have its own metadata, and instead, it references the metadata of the given file. Although the file is the same, it has different filenames. Applications can access the same file by using different filenames. See Figure 11.5 where A/B/a.tex and A/C/b.tex are "the" same file. Even if the original file is moved or renamed, the link refers to the same file. Hardlinks are always within the same file system, and do not cross the file system boundary. Hardlinks cannot be set up to non-existent files. File operations have the same effect on the file irrespective of the aliases applied.

Symbolic Link

≫ If you are familiar with Windows, the shortcuts are symbolic links.

It does not make sense to set up a hardlink from one file system to another file system because the two file systems may be widely different. Some file systems, however, do allow setting up softlinks (called symbolic links) between two file systems as well as within the same file system. A softlink is a special file that simply refers to another file. Unlike hardlinks that reference the same metadata, a softlink has its own metadata and stores the name of the referred file. See Figure 11.6 where A/B/a.tex and A/C/b.tex are two different files, but A/C/b.tex stores a reference to ../B/a.tex, i.e., to A/B/a.tex. The operating system redirects all operations on a softlink to the referred file. If the original file is moved or renamed or deleted, the softlink is broken and becomes a *dangling link*.

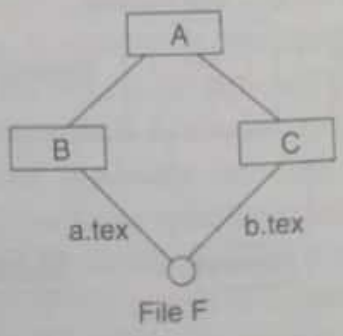

Figure 11.5: *Using hardlink in a file system.*

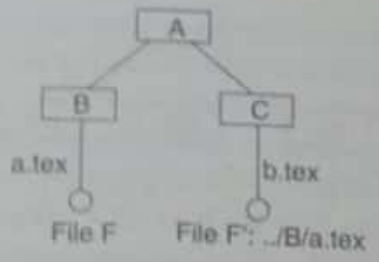

Figure 11.6: *Using softlink in a file system.*

Note that the referred file can even be non-existent. In this case operations on the softlink will fail.

Dynamic Link

A dynamic link is a softlink with special properties. Like the softlink, it contains a reference to another file and that reference is stored as a text string. However, for a dynamic link, the same text may be interpreted differently in different contexts. For example, the text may contain environment variables whose values would determine the appropriate filename at the time of opening the file, and, hence, may refer to different files at different times depending on the user and her environment when she examines the file. In Linux, **/proc/self** is a dynamic link; it is equivalent to the **/proc/process-id** directory.

11.6 File Organization

A file system stores multiple files. As mentioned previously, a file and its metadata are usually treated as nameless objects. But, users refer to files by their names. The FMS does the mapping from filenames to file objects. Also, where there are a large number of files, we need a way to organize files so that for a given filename, we can locate the file with relative ease. The file system uses specialized files called directories (or folders) to organize files. A directory is a special file that stores information about the subdirectories and files it holds; it stores the names of the files/subdirectories. The primary function of a directory is to help the system map filenames to file objects.

≫ UNIX and its variants maintain a single virtual tree whereas DOS/Windows might allow multiple trees to exist in different drives. The tree grows when files and/or directories are created, and shrinks when files and/or directories are deleted. Mounting a file system on a branch of the tree alters the tree by overlaying that branch with the new file system. Many file systems are mounted at boot time.

11.6.1 File Tree

Directory organization has evolved along with file system. In modern file systems, directories are organized as a hierarchical structure called a *tree*, or in a directed acyclic graph. Tree is the most common directory structure. Figure 11.7 shows a typical file organization in UNIX. All internal nodes on the tree are directories, and leaf nodes are files or directories. The file system tree (or acyclic graph) begins at a special directory called *root*, referred to as the root of the file system. All the files and directories under the root are managed by a single FMS. Every directory or file has a unique path from the root through other subdirectories. Each node in the tree can be specified by its absolute path, that is, the unique path from the root to the node, or by the relative path with respect to

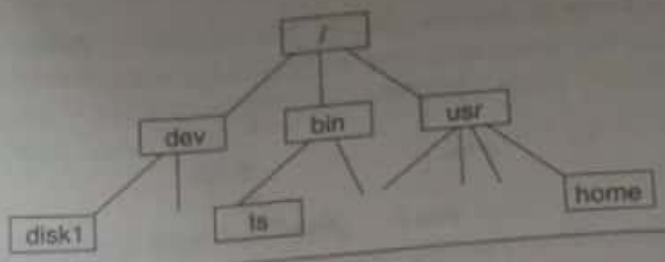

Figure 11.7: *UNIX file organization.*

another node in the tree. For a given directory node, the node itself is referenced by a special name called "." (the dot directory), and its parent by ".." (the dot-dot directory). Every two nodes in a path are separated by a special character, normally "/". The filename "/" itself identifies the root directory. For example, /usr/home/aravind indicates a unique directory in the file tree of Fig. 11.8.

Some file systems allow directories to have shared files (and, possibly, shared subdirectories). Figure 11.8 presents a typical example where a file *F* is shared by two directories. The two directories refer to the file by two different names: a.tex and b.tex. As explained earlier, there is actually a single file with two direct references or hardlinks to it. By shared we mean that changes made through one filename are immediately visible to the other filename. Sharing is not possible in tree-structured file systems. In UNIX systems, we can set up a shared file by executing 'ln' (hardlink) command line utility. In these systems, it is difficult to keep the file organization free of cycles.

11.6.2 Directory Management

A directory is very much like an ordinary file represented by a metadata object with the content stored in it as in the ordinary file. The type attribute value in metadata distinguishes directories from ordinary- and other files. Directory is the only file type, whose content is managed by the FMS itself. However, the content is determined by users, and not by the FMS. Directory content is structured by the FMS for ease in maintaining the content and for faster retrieval of data. It was

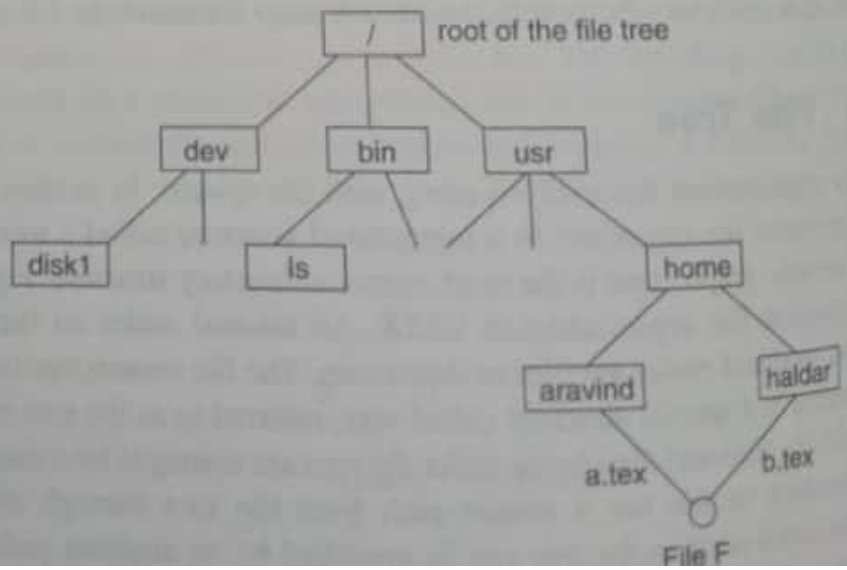

Figure 11.8: *Sharing files in a file system.*

mentioned above that directories are used only to organize files and directories. Directory management is the key to reliability of the file system and performance of the FMS. Content of a directory is referenced in many file operations such as searching for a file, creating/deleting a file, renaming a file, listing the directory, etc. In a directory search, a filename is used as the search key.

A directory is nothing but a list of pairs, each containing a file name and a pointer to the metadata representing the file. The metadata pointer is the address of a location in the storage device, and the filename is a variable length character string, see Fig. 11.9. Organizing the list is of the essence for performance. The simplest approach is to store the pairs linearly one after another without any ordering. Searching the directory for a filename would require a linear search through the content list, which is time consuming. If the directory contains a large number of entries, the search time can be a significant overhead. To insert a ⟨name, metadata pointer⟩ pair, the FMS first searches the name in the directory. If the name is present, either it returns an error or it updates the metadata pointer. Otherwise, it inserts the pair at the end of the list. To delete a pair, the FMS first searches the name in the directory. If the name is present, it deletes the entry by specially marking the entry or compacting the remaining entries.

Most FMSes do not manage directory entries as unsorted lists. They organize the directory content so that lookup is very fast. They build different structures such as hashing, B-tree, B^+-tree on the entries, or at least store entries as a sorted list. Readers may consult books on data structures and algorithms to learn more about search data structures. Data structures used to organize directory content must have efficient lookup and reasonable insertion/deletion costs.

11.7 Device File

Many operating systems treat peripheral devices as special files and allow many file related system calls on the special files. In UNIX systems,

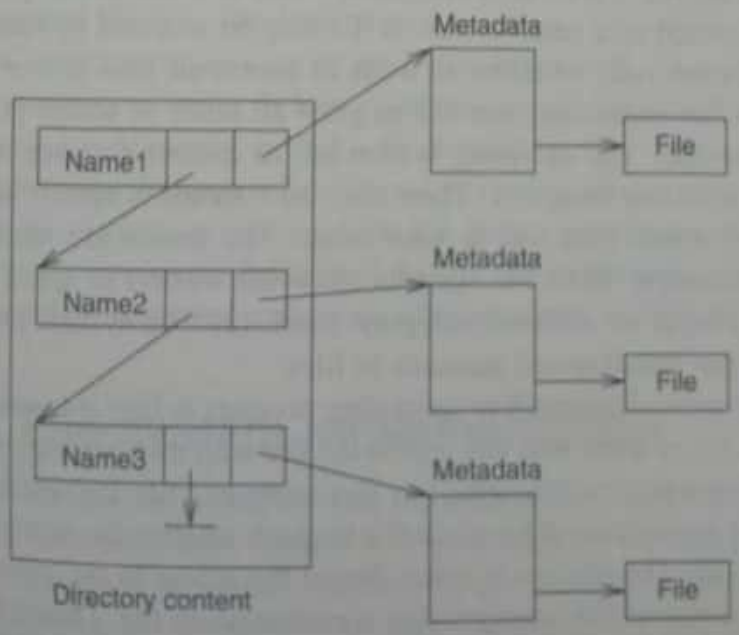

Figure 11.9: *A typical directory organization.*

character- and block devices are represented by *device files* that normally are in the /dev directory of the root file system. As mentioned in Section 10.7 on page 299, a device file acts as an interface to the device (actually, the device driver) it represents, and is the user-visible portion of the device. Users apply file operations on a device file to manipulate the device via the device driver.

≫ A device filename acts as a gateway to a device driver that controls the device rather than to the device itself. Users cannot store data to or retrieve it from a device file like they do with a regular file.

A device file specifies whether it is a block or character device (see Section 10.4.2 on page 282). All non-network I/O devices are classified into these two broad categories. Each category is also sub-classified into different types of devices. For example, ordinary disks and CDs are block devices, while keyboards are character devices. A device file stores two integer numbers: major- and minor numbers in UNIX systems. The major number identifies the device type and the device driver that handles operations on the device. The minor number identifies a particular device within a set of devices that the driver handles.

Operations on device files are executed through system calls. System calls referring to a device file are translated into invocations of appropriate functions of the device driver identified by the file major number. Information pertaining to a device file is normally stored in its metadata object, and file content proper is empty. A single device can have multiple filenames—some of which may offer slightly different characteristics—all mapping to the same actual device through different drivers. For example, a disk is a block device, but may be treated as a character device where the read/write are done on a byte-by-byte basis.

≫ Normal file ownership- and access control are applicable to device files too.

11.8 File Protection

In multiuser systems, we have to protect data belonging to one user from others. Protection deals with improper access to files but does not deal with damage to the physical media holding the files. The file is the unit of data protection, that is, the same kind of protection is applicable to the entire content of the file. Each file is created by some user and at any point of time the file is owned by a specific user. A file may be accessed by many users. It is generally too risky to allow all users to access all files in every possible manner. A file owner may not like to grant all kinds of access privileges to every other user. The operating system has to control accesses to files, for which we need two things: (1) There must be a means to specify which users may access which files and in what mode. The modes are normally read, write, and execute. Most file systems allow file owners to specify different access privileges to different category users. (2) The system must have a means to trap unauthorized accesses to files.

A very general approach to controlling accesses to files is to associate with each file a list of users who can access the file with their respective modes of access. When a process belonging to a user accesses a file, the operating system consults the access control list of the file to check whether the user is allowed to access the file. The process is either denied the access to the given file, or is permitted to execute the corresponding operation on the file. Though the scheme

is very general it is costly, as we need to maintain a long list of users and their access rights as a part of file metadata. Many systems implement a file protection scheme that approximates the access control list. UNIX systems divide all users into three classes for access control purpose on each file: (1) the file owner, (2) the file-user group: users who share the file and need similar accesses to the file, and (3) the file universe: all users in the system except the file owner and the file group members. The access control list stores permission information for these three classes. The success of this scheme depends on controlling membership of the user groups. Not everyone has rights to change a user group membership. In UNIX systems only superusers can create groups or change group memberships. For each file, there is a user group attribute in the file metadata which specifies the current group of the file. For each class of users, the file metadata maintains three bits of information indicating read, write, and execution rights. The meanings of these three bits are a little bit different for directories: (1) read permission means one can list the content of a directory, (2) write permission means one can add new entries into or delete existing entries from the directory, and (3) execution permission means one can change to make the directory her current directory.

≫ Each user is a member of at least one group, but can be a member of many groups. UNIX systems create user-groups for the purpose of selective sharing of system resources.

≫ Protecting a directory automatically protects all subdirectories and files under that directory.

11.9 Concurrency Control

In previous sections, we discussed about file sharing by two directories, and file accessing by different classes of users. There is another important issue that the FMS has to handle, and that is to handle concurrent accesses to the same file by multiple processes. A read operation on a file does not alter the file content, but a write operation does. What semantics does the FMS guarantee under concurrent accesses (read and write) to a file or directory? What happens when a process writes to a file that is being read or written by other concurrent processes at the same time? Consistency semantics is an important criterion in designing an FMS.

In UNIX systems, a write to a file is immediately visible to all the processes that have opened the file. The FMS provides a single view of the file to all processes. Concurrent writes may be arbitrarily interleaved. These systems do not ensure atomicity of writes. If finer synchronization is wanted, the processes need to do what is necessary explicitly by themselves. UNIX systems support file locking interfaces (system calls) that may be used for this purpose. A process can lock an entire file or a section of the file to prevent others from applying conflicting operations on the (section of the) file. (For general synchronization problems and their solutions, readers are advised to refer to Chapter 7.)

11.10 File-system Implementation

In the previous sections, we dealt with files, directories, and file systems as abstract entities—they were treated as logical objects. We also discussed logical connections between these objects. At the highest level, users always see a storage device as a collection of files, organized in directories. At the

lowest level, a storage device stores raw (device-dependent) physical signals on the device recording media. We need to bridge the gap between these two extremes. The FMS does the logical to physical mapping, and helps users in manipulating their files.

In this section, we will see how the FMS organizes file objects, i.e., maps on physical devices. Direct-access devices such as the disk are used to build online file systems. At the lowest level, a raw device medium is partitioned into a number of blocks, called *physical blocks* or simply *p-blocks* in this section; for example, in disk devices, a p-block consists of a couple of consecutive sectors on the same track. Blocks are units of space allocation and deallocation, and units of data transfer between storage devices and the host system. Blocks can be overwritten in-place. Each block has its own address (in the device space), and the operating system can access the block by providing the block address to the I/O controller that manages the device.

≫ File systems mostly reside on secondary storage devices such as the disk. As stated in Chapter 10, a file system occupies an entire partition of a storage device. A few operating systems use a part of the main memory to implement temporary file systems. The **/proc** is an example of such file system in Linux. The **/proc** file system contains a hierarchy of special files reflecting the current state of the kernel information about the system.

The FMS actually resides one level above the I/O subsystem (device drivers) in the operating system software architecture (see Fig. 3.1 on page 71). The FMS views a disk as a linear array of p-blocks. The blocks are numbered starting with zero and going up to the size of the array. The FMS manages its data structures on the top of this p-block array abstraction. Note that device drivers implement basic I/O and control routines to access p-blocks from devices. The driver directs I/O controllers what to do on devices. For the purpose of this section we assume that the drivers have routines to read and write individual p-blocks by their ordinal numbers or some other suitable addressing mode.

≫ Data transfer is done in units of blocks to improve I/O efficiency.

As mentioned previously, a file system may encompass a complete disk, or a partition (a subset of blocks) of a disk, or a volume comprising several disks. Without loss of generality, we will use partition to refer to the entire storage space that the file system can use. Some blocks in the partition store critical partition management information, and those are called *superblocks* or partition control blocks. A superblock usually contains information about the partition space size, file-system name, block size, address of the root directory, and so forth.

The FMS views a file or directory content as a finite sequence of logical blocks or simply *l-blocks*. We refer to *l*-blocks in a file by their ordinal position number, that is, the 0th *l*-block, the 1st *l*-block, and so forth. The *l*-blocks are stored in p-blocks, and each p-block holds exactly one *l*-block. We have to maintain information, for each file, on the mapping of *l*-blocks to p-blocks. This mapping information is stored in the p-blocks themselves, either in separate p-blocks or in the p-blocks that store the file content proper.

The FMS also needs to keep track of all the unallocated p-block in the file systems. When the need arises, it allocates free p-blocks to files. The information about which p-blocks are allocated and which are free is stored in the partition itself. Initially, all p-blocks are free except the few used as partition control blocks. In a nutshell, file contents, file metadata, file mapping information, and partition management information are stored in their entirety in the partition space itself. The file system is a self-contained system to be managed by the corresponding FMS. The FMS manipulates that information to maintain the integrity of the file system.

11.10.1 Physical Organization

Users view a file as a contiguous finite sequence of bytes (or records). This is a logical view. They may read or write the content at any relative position in the file. The FMS must be able to translate the relative position to a p-block address where the required data reside. The FMS uses the p-block address to instruct a device driver to access the required data from the p-block.

The content proper and the metadata of a file are stored in p-blocks. They are normally stored in different p-blocks. In UNIX systems, metadata are called inodes, and are stored in what we call inode blocks. (In UNIX systems, when a file system is initialized in a partition, inode blocks are reserved at a priori known p-blocks. The number of inodes limits the total number of files and directories in the file system.) Depending on the size of a p-block, many inodes may reside in the same inode block. One inode contains metadata of a single file. There is a metadata attribute, called *start address*, which stores a pointer to the starting position of the file object, that is, the address of the first p-block. As shown in Fig. 11.10, the metadata acts as a connection between file content and other data associated with the file.

As noted in Section "Initialize", every file system has a root directory that is created when the file system is initialized. The root metadata helps us access the content of the root directory and traverse the rest of the file system hierarchy. Consequently, the foremost task of the FMS is to find the root metadata in the file system. The root metadata resides either at a fixed location or a pointer to it is stored in a superblock. Superblocks always reside at a particular known location in the file system.

Most files require many p-blocks allocated to them. The p-blocks are normally allocated and deallocated on demand. In the following section we study a few well known space allocation schemes.

≫ The file and the process are two important abstractions of operating systems. Like process control blocks (PCB) hold information about processes in the system, the file system maintains file control blocks (FCB) to hold information about the files in the system. File allocation table (FAT) holds the FCB for the file systems of MS-DOS and Windows (up to ME), index nodes (or simply inodes) are the FCB for the files systems of UNIX and its variants.

11.10.2 Space Allocation

A single FMS manages the entire space in a device partition. The FMS keeps track of both the used space and the free space in the partition. Each file is allocated some space, and this space may grow or shrink in the lifetime of the file. When a file grows or a new file is created, the FMS finds and allocates appropriate amount of space to the file. As long as there is free space in the partition, a file can grow (subjected to its maximum size or quota requirements if there are any) or a new file can be created.

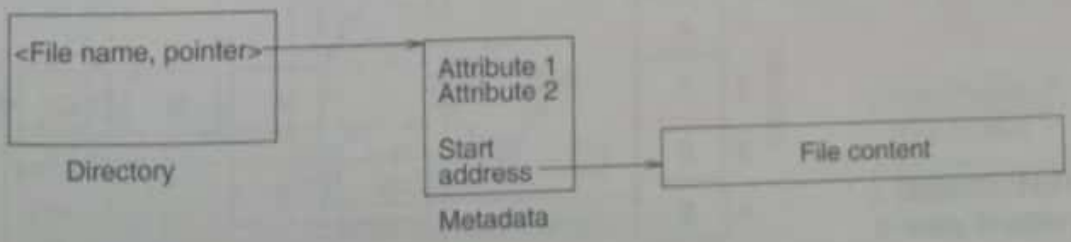

Figure 11.10: *Metadata and file content connection.*

An allocation policy is a set of rules that the FMS follows to decide the placement of the file data, directory content, and metadata in the device partition it manages. Where space management is concerned, we have to bear in mind two factors: (1) space allocation and deallocation must be simple to execute, and (2) data retrieval and update from and to files must be efficient. Many space management schemes have been designed and developed over the past years. Most popular schemes include contiguous allocation, linked allocation, and indexed allocation. These schemes, and their merits and demerits, are discussed in the following subsections. No single space-allocation scheme is suitable for all systems. Every scheme has its own space- and access time overheads. We have to evaluate these two factors before deciding on an allocation scheme.

Contiguous Allocation

The content of each file is stored in contiguous p-blocks. Here the term contiguous is used in the sense of the ordinal positions of the p-blocks (see Section "Physical Formatting"). If a file consists of multiple *l*-blocks and the file content is stored starting from *b*th p-block, then the 0th *l*-block is in *b*th p-block, the 1st *l*-block is in $b + 1$st p-block, and so forth, see Fig. 11.11. The FMS does not separate the logical view of a file from the physical allocation. To access the *n*th *l*-block, the FMS invokes appropriate driver routine against the $(b + n)$th p-block. The file metadata have two attributes for this purpose: the starting p-block address and the total number of blocks the file occupies.

Space management is a difficult problem. It is akin to the segment space management scheme we studied in Section 8.5. Finding contiguous space of a given size is a time consuming task. As shown in Section 8.5.4 on page 226, we may follow the first-fit, best-fit, and/or buddy allocation schemes. These allocation algorithms suffer from external fragmentation. Another major problem is determining the size of a file when it is created. Most of the time, except for file copying operations, there is not much a priori information available to estimate its size. If we allocate too much space, we have internal fragmentation. If we allocate insufficient space, the file soon fills up. Either

≫ In the disks, two consecutive p-blocks reside either consecutively on the same track or on two consecutive tracks on the same cylinder, or on two adjacent tracks. In the latter two cases, one is the last block on a track, and the other is the first block on the other track. Consequently, sequential accessing of a file does not require disk head movement, or requires movement by one track when needed.

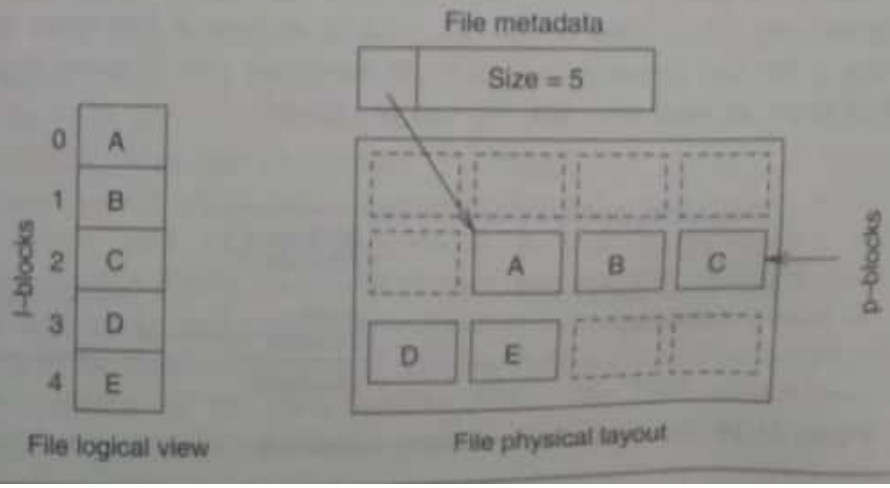

Figure 11.11: *Contiguous allocation of space to files.*

we do not expand the file further, or we have to relocate the file in a larger contiguous space. The latter can be a very time consuming task.

Linked Allocation

Under linked allocation, any *l*-block of any file can be stored in any p-block (see Fig. 11.12). This allocation scheme resembles the paging memory management scheme we discussed in Section 8.6. The FMS separates the logical view of a file from its physical allocation, that is, a file then may not be stored in contiguous p-blocks. The file metadata stores a pointer to the first p-block of the file. Each p-block points to the next p-block of the file. The last p-block of the file points to null. Thus, each file is a linked list of p-blocks that may be scattered in the device partition. The FMS defines the p-block structure. A p-block, in addition to storing an *l*-block, also stores file management information (such as the header, trailer, and/or p-block pointer). This management information, however, is not visible to file users who see only the file content proper.

≫ For robustness, we can have doubly linked allocation. But then, we have additional space overhead.

Linked allocation eliminates the external fragmentation problem. When a file is created, the file metadata object is allocated and initialized. The start-address pointer in the metadata object is set to null indicating an empty file. The file size is stored in the *size* attribute of the metadata: it is zero initially. When a write occurs to a file and we need to expand the file, a free p-block is allocated to it at the end of its p-block list. When a file is truncated, all the trailing p-blocks starting at a particular p-block are freed. A file can grow or shrink with relative ease. Space allocation is very fast as any p-block can be used for any *l*-block.

To read data from a file, we start reading from the first p-block pointed to by its metadata, and chase the p-block link pointers until we reach the required p-block. Although p-blocks are directly addressed blocks (as far as the disk is concerned), a file becomes effectively a sequentially accessed object. Supporting direct *l*-block access capability will be highly inefficient. To find the *n*th *l*-block, we have to read *n* p-blocks. Also, there is a considerable space overhead due to the header, trailer, and/or block pointers stored in each p-block.

≫ For better robustness the mapping information of files (i.e., *l*-blocks of each file) to p-blocks is stored in a table called (as noted earlier) the file allocation table (FAT) in MS-DOS and Windows (up to ME 2000). Here each file is mapped to a linked list of p-blocks. The FAT is simply a list of p-block numbers. The number of bits in each entry of the table is called the FAT size and it is one of the three values 12, 16, or 32.

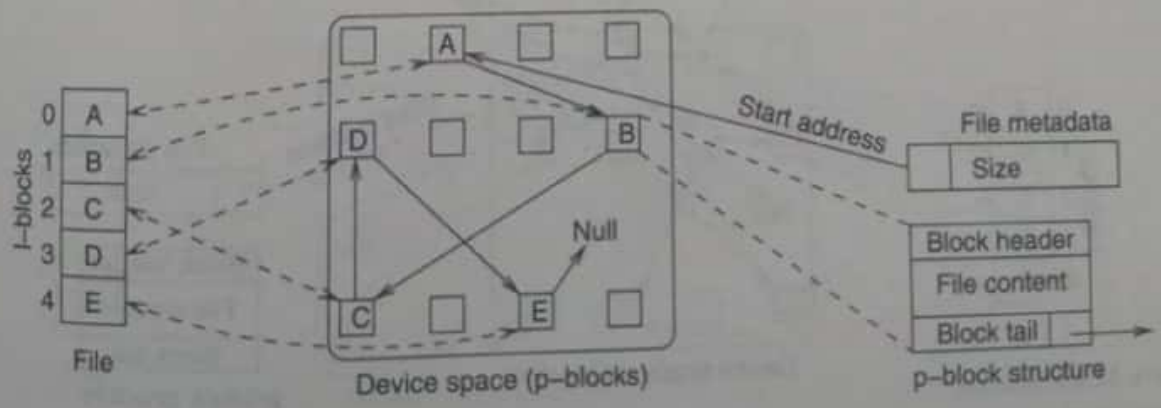

Figure 11.12: *Linked allocation of space to files.*

Extent Allocation

An extent-based space-allocation scheme is a compromise between the contiguous and linked allocation schemes. An *extent* is a contiguous chunk of p-blocks, and represents a contiguous region in the device. An extent is identified by the address of the first p-block and the number of consecutive blocks. The space allocation is done in variable-sized extents. When a file needs more space, another extent is allocated to the file. As in contiguous allocation, this scheme suffers from external fragmentation. In addition, it suffers from internal fragmentation due to larger extent of space.

Indexed Allocation

In the contiguous allocation scheme, each *l*-block in a file can be directly accessed. But the scheme has space management problems due to external fragmentation. The linked allocation scheme solves the external fragmentation problem, but its support of the direct *l*-block access is very inefficient. We have to have a scheme that accommodates benefits of both the schemes and minimizes their shortcomings.

Like in the linked allocation, any *l*-block can be stored in any p-block for the indexed allocation scheme. Unlike linked allocation, we have two types of p-blocks to hold, separately, the data blocks (d-blocks) and the index blocks (i-blocks), see Fig. 11.13. The i-blocks only contain pointers to d-blocks or to other i-blocks. These d-blocks do not store any block pointers. An i-block space is structured as an array of p-block addresses. All but the last array entry points to the d-blocks. The last entry points to another i-block. All i-blocks are arranged in a link list. The file metadata points to the first i-block of the list. As need arises we allocate more i- and d-blocks to the file. Implementation for direct access to *l*-blocks is much more efficient compared to that for the linked allocation. Most small files will have one- or two index

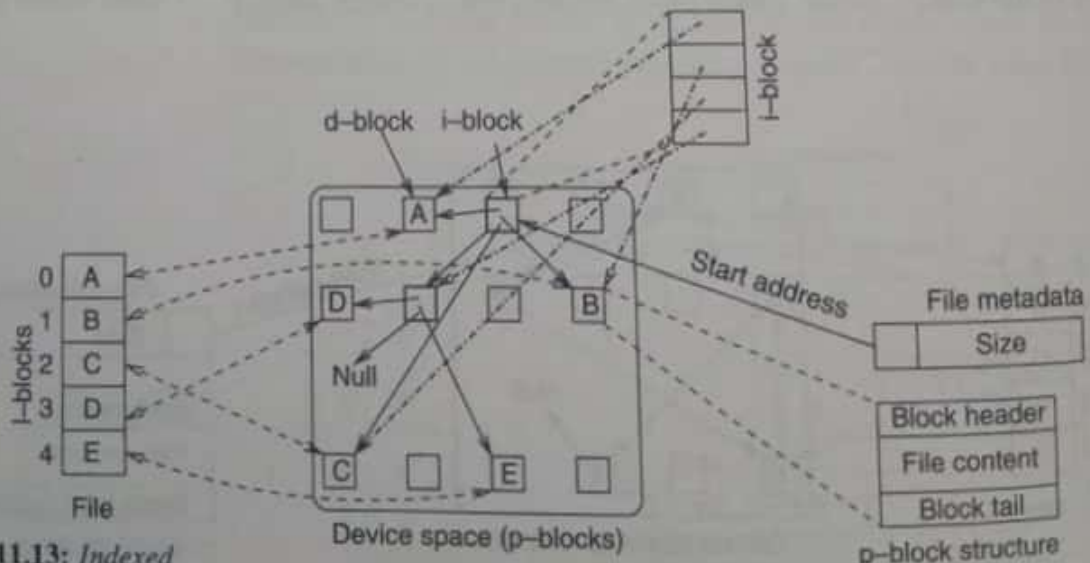

Figure 11.13: *Indexed allocation of space to files.*

blocks. A large file will have many index blocks, and may have considerable overhead in the implementation of direct access files. An indexed allocation scheme has higher space overhead compared to the linked allocation scheme. We can organize all i-blocks of a file in a multilevel index as shown in Fig. 11.14. The multilevel index helps in reducing time overhead in implementing direct access files. The following table compares some properties of contiguous, linked, and indexed space allocation schemes.

	Contiguous	Linked	Indexed
Dynamic access	yes	no	OK, not good
Reliability	yes	no	no
Internal fragmentation	yes	yes	yes
External fragmentation	yes	no	no

UNIX File-space Allocation

UNIX systems use a combination of the linked- and indexed allocation schemes (see Fig. 11.15). They organize block pointers in a hierarchical manner. A file metadata object (called inode) stores 12 direct pointers to the d-blocks that store the first 12 *l*-blocks of the file. One entry in the metadata uses a single indirect index, another entry double indirect index, and another entry triple indirect index. For very large files, we need more indirect i-blocks. The i-blocks and d-blocks are added to and removed from the files on the basis of need.

11.10.3 Free-space Management

For every partition the FMS manages, it maintains information for all the free blocks in the partition. Actually, it maintains a free list from where it allocates blocks to files, and to where it puts the blocks freed from files. The free list originates at a partition superblock. When an allocation request comes, the FMS searches the free list to find suitable block(s) that would satisfy the request.

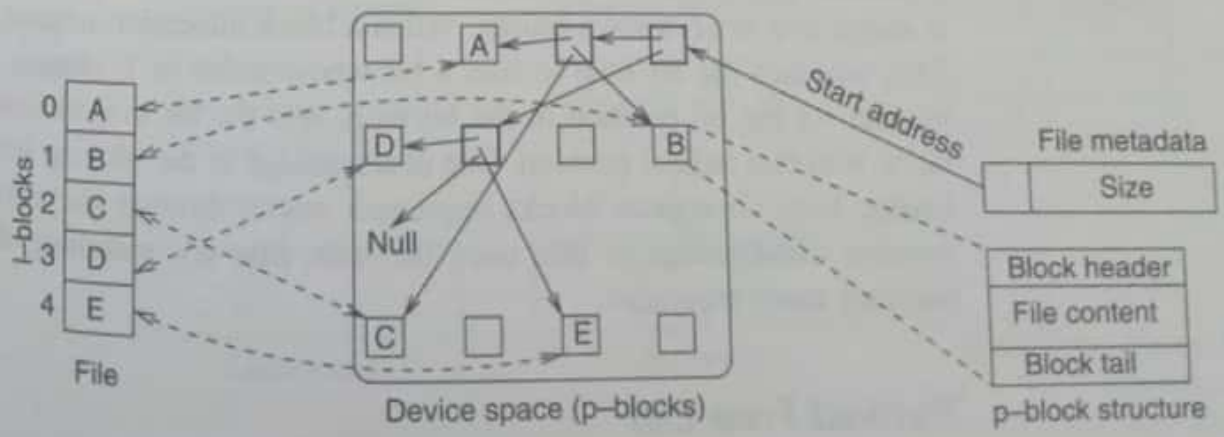

Figure 11.14: *Multilevel indexed allocation of space to files.*

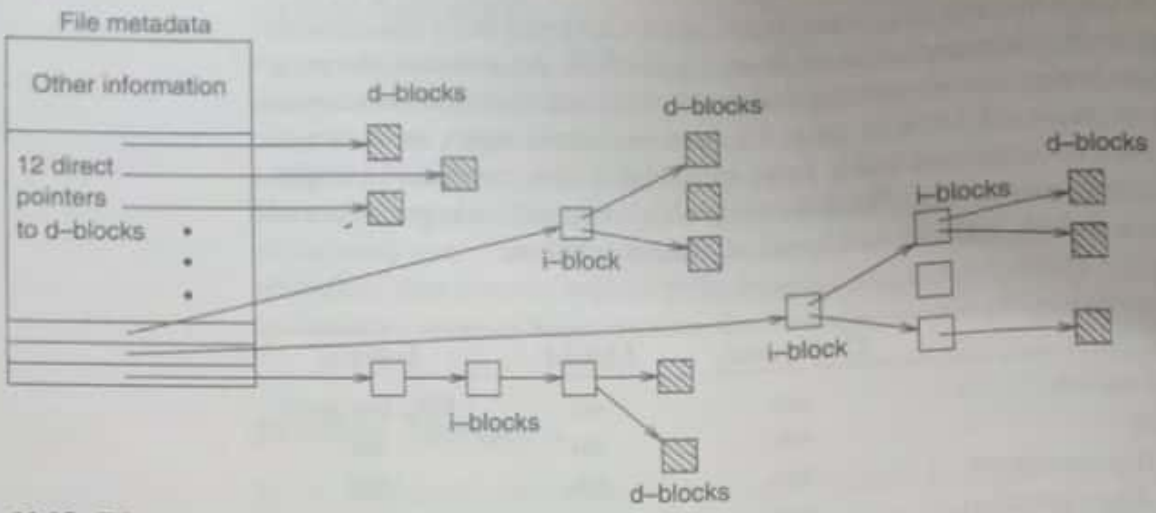

Figure 11.15: *File space allocation in UNIX systems.*

There are many ways to manage the free list. One simple way is to form a link list of free blocks, each block pointing to the next free block. Another approach is where free blocks are organized into chunks of variable length. The first block in each chunk stores the length of the chunk, and points to the next chunk. This scheme is similar to hole management in the segmentation scheme we studied in Section 8.5. We may create an ordered tree by size of the free chunks for faster access to chunks. We may use the buddy system to manage free space. In the following subsection, we discuss a radically different space management scheme implemented in some file systems. It is called the bit map. Following that, we present another simple, free-space tracking scheme.

Bit Map

Suppose there are a total of n blocks in a partition. The file system maintains a single bit of information for each block. If the bit value is 1, the block is free; or else, the block is allocated. The collection of these bits is called a *bit vector* or *bit map*. This map is stored as a linear array of n bits. The value of the bit map represents the current state of all the blocks in the entire file system. The bit map is stored in a priori known blocks. When a block allocation request comes, the FMS searches the bit map to find a bit whose value is 1, obtains the ordinal number for the bit position in the bit map, sets the bit to 0, and allocates the block with that ordinal position. One disadvantage of the bit map scheme is that finding large contiguous blocks requires a search through the whole bit map. Another disadvantage is that once the disk fills up, searching the bit map becomes more expensive.

Trunked Free List

The free blocks are organized into a rooted trunk (see Fig. 11.16). There are two types of free blocks: trunk block and leaf block. The trunk blocks form a linked

list. There is pointer in the file system superblock, which points to the first block on the trunk. Each trunk block contains references to multiple leaf blocks.

11.10.4 File-system Layout

So far we discussed many file system concepts in piecemeal. Figure 11.17 displays an integrated view of a typical file system. The figure is an integration of Figs. 11.13, 11.9, 11.10, 11.12, and 11.16. The figure depicts an end-to-end layout of a typical file system. The first block is the boot block, the next one is the file system metadata (superblock), and the next to that is the metadata of the root directory of the file system. Once the root is located, we can navigate through the file system easily by chasing the directory pointers and their metadata. In the figure, the root metadata points to the root content block. The root directory contains Dir1 and a pointer to Dir1's metadata. This metadata points to the Dir1 content block. The Dir1 directory contains file File1 and a pointer to File1's metadata. The File1 metadata contains a pointer to the first p-block. From there we can chase p-block pointers to read the file content proper. All free blocks are linked together to form a free-list that is referenced from the superblock.

11.10.5 File-system Journaling

Although data ultimately resides on a device recording medium, the CPU can only manipulate data after reading it in the main memory. When the CPU has finished using the data, it writes back the (new) data to a device in the form of p-blocks. The writing is done in-place, that is, the old content is overwritten.

We note that writing a p-block in a device is not an atomic action as it takes a non-zero amount of time to overwrite the previous block content. If the device or the entire system were to lose power in the middle of a p-block write, the block content will be corrupted. Also, many operations on files involve changing more than one p-block. The modified blocks are written back into the device in two- or more independent write operations, either

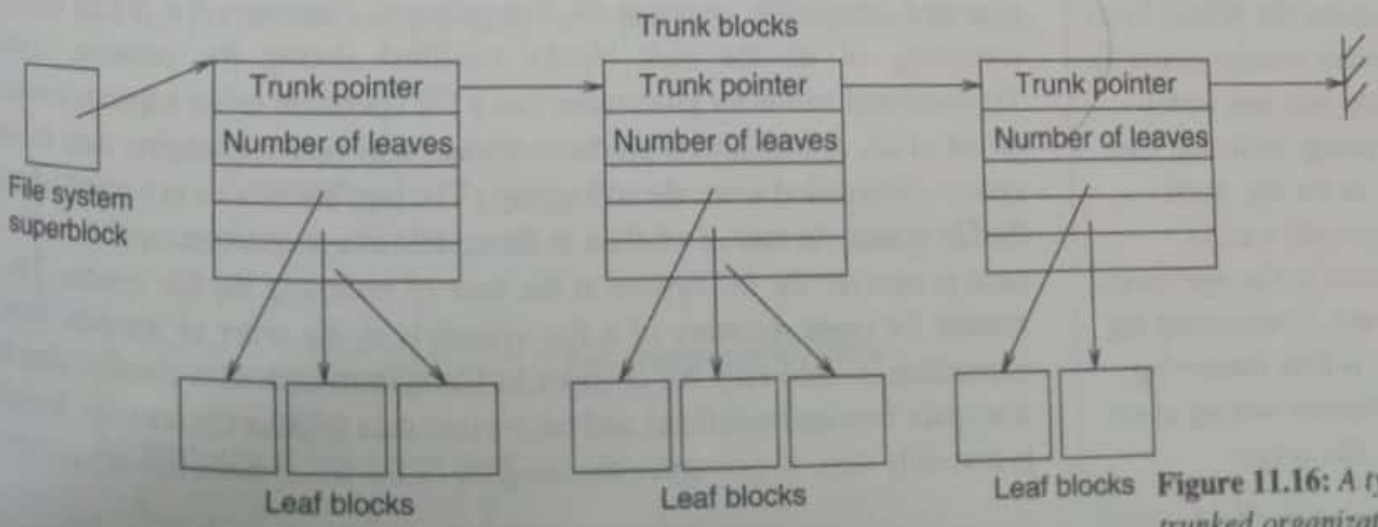

Figure 11.16: *A typical trunked organization of free blocks.*

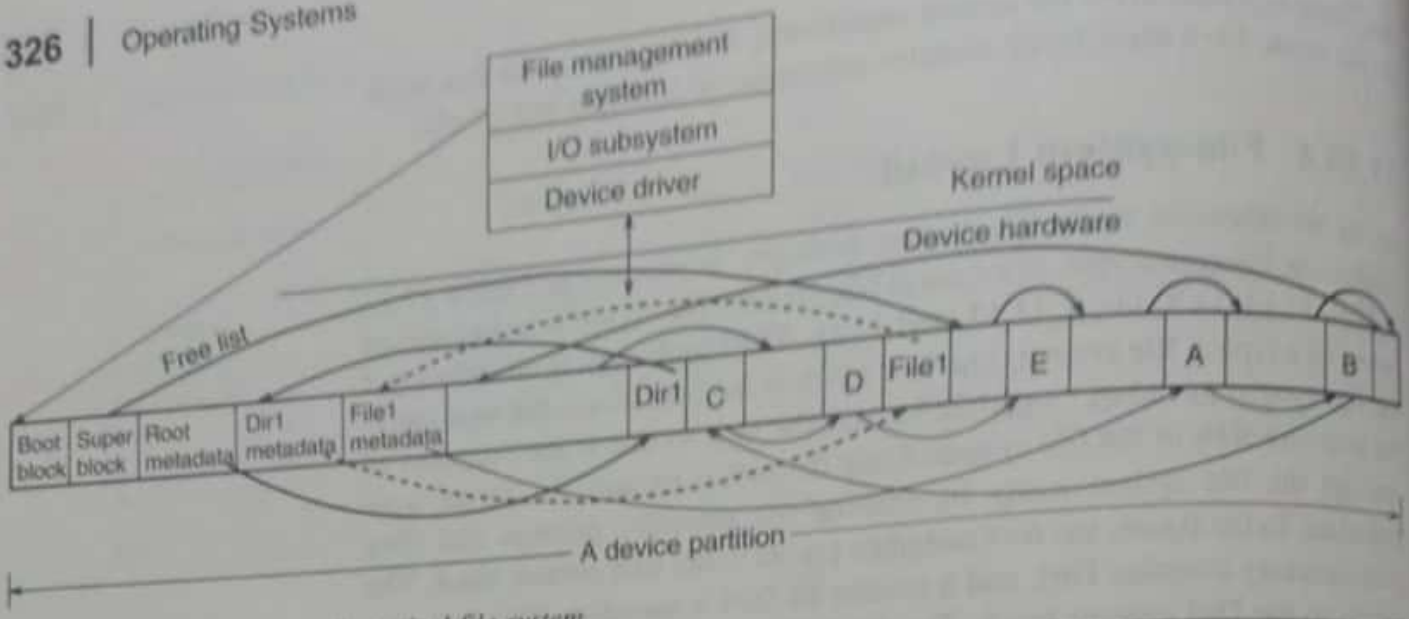

Figure 11.17: *Layout of a typical file system.*

sequentially or concurrently. The FMS also keeps some information (especially about directories) in the main memory to speed up executions of file operations. The in-memory information is more up-to-date than the device copy. A simple power failure or a failure after partial p-block write may leave the file system in an inconsistent state. The FMS must have means to recover a file system into a consistent state after failures. There must not be any loss or corruption of data due to failures. Most operating systems run special programs to check for and correct inconsistencies in file systems—the system root file system at the bootstrap time and others at mount time. Such recovery may be a lengthy operation in the order of tens of minutes.

A recent trend is to incorporate the journalling concept in file systems to avoid or minimize corruption of management information, and to avoid time consuming recovery at bootstrapping or mount time. Journalling is a mechanism for ensuring the correctness of physical structures stored in disk blocks. The journalling system incorporates the concept of "atomic" transaction, originally practiced in databases. All updates to management information are done in transactions. The transaction is the unit of atomic operation in journalling systems. Each transaction is either executed completely or not at all. For example, creating a file is one transaction consisting of all the disk blocks modified during the creation operation. Transactional atomicity guarantees that a file operation either happens completely or not at all. A transaction produces update logs before changing data in the file system. (It is called write-ahead logging.) The logs are written in a reserved area for the file system. In case of a failure in the middle of a transaction execution, the log is used to recover the file system at the time of restarting the file system. The time needed for crash recovery of a file system is in the order of seconds. Note that journalling is used only for changes in file system structures (inodes, directories, free space management data), and not for user data (regular file content). Journalling is normally done to guarantee the integrity of the file system, and not of user data.

≫ Ext3 file system (in Linux) and NTFS (in Windows) are journalling systems.

≫ Some file systems go beyond ordinary journalling, and they can recover from corruption of user data too. A log structured file system treats the entire storage device as the log area and writes everything, including user data, in the log. Such systems add a lot of overhead to file operations. The task of reclaiming log space is time consuming, and requires locking of the entire file system.

11.11 Virtual File Systems

Many modern operating systems allow more than one FMS to coexist in the same computer system. Different FMSes may offer different interface operations. Even though there are many FMSes, the users definitely prefer to see only one unified FMS. This is achieved through a new interface called *virtual file-system switch* (VFS, in short). It is a thin software layer that resides on the top of the FMSes proper. The VFS provides a unified common interface to several real file systems, see Fig. 11.18. It is responsible for management of files stored in different file systems. The VFS defines a common file system interface, and applications see this interface. The VFS implements a common "logical" file system model that is capable of unifying many known file systems. All file related system calls are channelled through the VFS. The VFS routes the calls to appropriate (real) FMSes that know how to handle the calls on the individual real file system.

The VFS specifies generic interface functions for the individual file system and individual files. Any software package that implements those functions can be plugged-in under the VFS as a separate file system. For example, the main memory based /proc file system in Linux is such a kernel module. Therefore, sometimes VFS may be deceptive. Although VFS makes all real file systems appear to be the same to computer users, some real file systems may not support some expected features. For example, the virtual file allocation table (VFAT) system does not support symbolic links.

11.12 Runtime Structures

In the previous sections we first discussed files, directories, and file systems as abstract entities, and later we showed how a file system can be constructed in a disk partition (see Fig. 11.17). The disk is handled by a device driver and the I/O subsystem. The FMS that manages the file system sits on the top of the I/O subsystem. There are still some missing links in connecting processes to the FMS. These links are shown in Fig. 11.19. The figure is full of many handles, aka descriptors.

The VFS maintains many runtime data structures to represent open file sessions. In Fig. 11.19, we have a table of open file descriptors for each

≫ Different kinds of handles are used at different levels to manage files. The operating system uses the superblock as the handle for file system and the FCB as the handle for an individual file. Programs use file descriptor as the handle to manipulate the file and users make use of files and directory as the handles to manage the stored information.

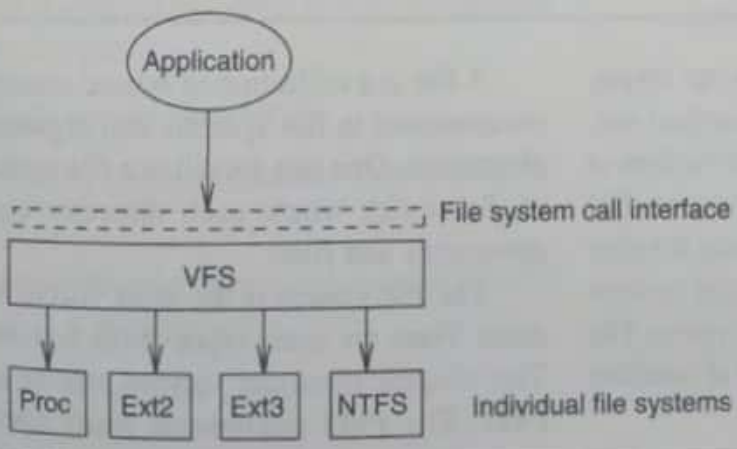

Figure 11.18: *Structure of Virtual File System.*