

File System:

For most users, the file system is the most visible aspect of an operating system. It provides the mechanism for on-line storage of and accesses 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.

File Concept:

Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of information storage.

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 through power failures and system reboots.

Definition: 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.

Many different types of information may be stored in a file—source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.

A file has a certain defined structure, which depends on its type. A **text file** is a sequence of characters organized into lines (and possibly pages).

A **source file** is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements.

An **object file** is a sequence of bytes organized into blocks understandable by the system's linker. An **executable file** is a series of code sections that the loader can bring into memory and execute.

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 floppy disk, send it in an e-mail, or copy it across a network, and it could still be called `example.c` on the destination system.

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

- **Name:** The symbolic file name is the only information kept in human readable form.
- **Identifier:** This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type:** This information is needed for systems that support different types of files.
- **Location:** This information is a pointer to a device and to the location of the file on that device.
- **Size:** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection:** Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification:** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

The information about all files is kept in the directory structure, which also resides on secondary storage. 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. Because directories, like files, must be nonvolatile, they must be stored on the device and brought into memory piecemeal, as needed.

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 six basic file operations.

It should then be easy to see how other, similar operations, such as renaming a file, can be implemented.

- ✓ **Creating a file.** Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
- ✓ **Writing a file.** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location.

The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

- ✓ **Reading a file:** To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place.

Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current-file-position pointer. Both the read and write operations use this same pointer, saving space and reducing system complexity.

- ✓ **Repositioning within a file:** The directory is searched for the appropriate entry & the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as files seek.
- ✓ **Deleting a file:** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
- ✓ **Truncating a file:** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length—but lets the file be reset to length zero and its file space released.

Most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an `open()` system call be made before a file is first used actively.

The operating system keeps a small table, called the open-file table, containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required.

When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table, create and delete are system calls that work with closed rather than open files.

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.

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 as shown in figure (Common file types).

In this way, the user and the operating system can tell from the name alone what the type of a file is. For example, most operating systems allow users to specify file names as a sequence of characters followed by a period and terminated by an extension of additional characters. File name examples include `resume.doc`, `Server.java`, and `ReaderThread.c`.

The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a .com, .exe, or .bat extension can be executed, for instance.

The .com and .exe files are two forms of binary executable files, whereas a .bat file is a batch file containing, in ASCII format, commands to the operating system.

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure: Common file types

File Structure:

File types also can be used to indicate the internal structure of the file. Source and Object files have structures that match the expectations of the programs that read them. Further, certain files must conform to a required structure that is understood by the operating system.

For example, the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is. Some operating systems extend this idea into a set of system-supported file structures, with sets of special operations for manipulating files with those structures.

Access Methods:

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

Sequential Access:

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

Reads and writes make up the bulk of the operations on a file. A read operation—read next—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write operation—write next—appends to the end of the file and advances to the end of the newly written material (the new end of file).

Such a file can be reset to the beginning; and on some systems, a program may be able to skip forward or backward n records for some integer n —perhaps only for $n = 1$.

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

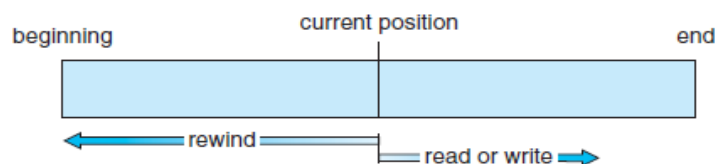


Figure: Sequential-access file

Direct Access:

Another method is direct access (or relative access). A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block.

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

Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.

As a simple example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file.

To store information about a larger set, such as people, we might compute a hash function on the people's names or search a small in-memory index to determine a block to read and search.

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have read n , where n is the block number, rather than read next, and write n rather than write next.

An alternative approach is to retain read next and write next, as with sequential access, and to add an operation position file to n , where n is the block number. Then, to effect a read n , we would, position to n and then read next.

The block number provided by the user to the operating system is normally a relative block number. A **relative block number** is an index relative to the beginning of the file.

Thus, the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block may be 14703 for the first block and 3192 for the second.

The use of relative block numbers allows the operating system to decide where the file should be placed (called the allocation problem) and helps to prevent the user from accessing portions of the file system that may not be part of her file. Some systems start their relative block numbers at 0; others start at 1.

Not all operating systems support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created; such a file can be accessed only in a manner consistent with its declaration.

We can easily simulate sequential access on a direct-access file by simply keeping a variable *cp* that defines our current position, as shown in *figure below*. Simulating a direct-access file on a sequential-access file, however, is extremely inefficient and clumsy.

sequential access	implementation for direct access
reset	<code>cp = 0;</code>
read_next	<code>read cp ; cp = cp + 1;</code>
write_next	<code>write cp ; cp = cp + 1;</code>

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

Other Access Methods:

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. The index, like an index in the back of a book, contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

For example, a retail-price file might list the universal product codes (UPCs) for items, with the associated prices. Each record consists of a 10-digit UPC and a 6-digit price, for a 16-byte record.

If our disk has 1,024 bytes per block, we can store 64 records per block. A file of 120,000 records would occupy about 2,000 blocks (2 million bytes). By keeping the file sorted by UPC, we can define an index consisting of the first UPC in each block. This index would have 2,000 entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory.

To find the price of a particular item, we can make a binary search of the index. From this search, we learn exactly which block contains the desired record and access that block.

This structure allows us to search a large file doing little I/O. With large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file contains pointers to secondary index files, which point to the actual data items.

For example, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index.

The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index.

This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads. Figure below shows a similar situation as implemented by VMS index and relative files.

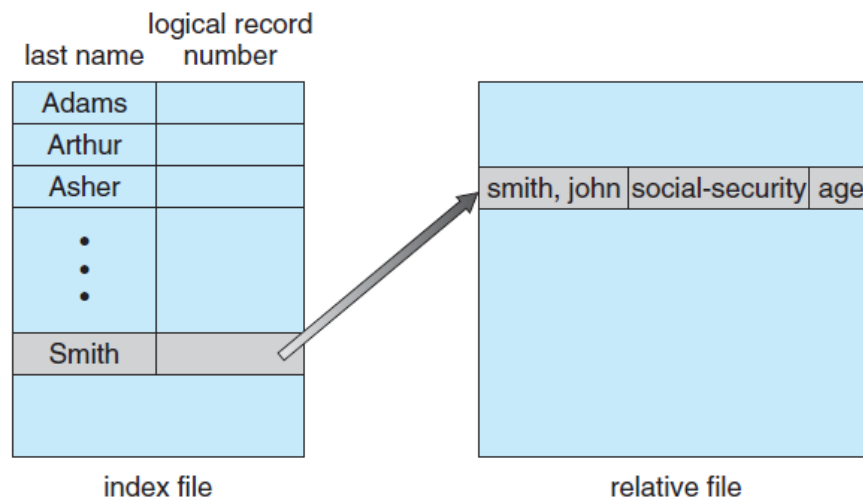


Figure: Example of index and relative files

Directory Structure:

Next, we consider how to store files. Certainly, no general-purpose computer stores just one file. There are typically thousands, millions, even billions of files within a computer. Files are stored on random-access storage devices, including hard disks, optical disks, and solid-state (memory-based) disks.

A storage device can be used in its entirety for a file system. It can also be subdivided for finer-grained control. For example, a disk can be partitioned into quarters, and each quarter can hold a separate file system.

Storage devices can also be collected together into RAID sets that provide protection from the failure of a single disk. Sometimes, disks are subdivided and also collected into RAID sets.

Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device, or leaving part of the device available for other uses, such as swap space or unformatted (raw) disk space.

A file system can be created on each of these parts of the disk. Any entity containing a file system is generally known as a volume. The volume may be a subset of a device, a whole device, or multiple devices linked together into a RAID set. Each volume can be thought of as a virtual disk.

Volumes can also store multiple operating systems, allowing a system to boot and run more than one operating system. Each volume that contains a file system must also contain information about the files in the system.

This information is kept in entries in a device directory or volume table of contents. The device directory (more commonly known simply as the directory) records information like: name, location, size and type for all files on that volume. Figure below shows a typical file-system organization.

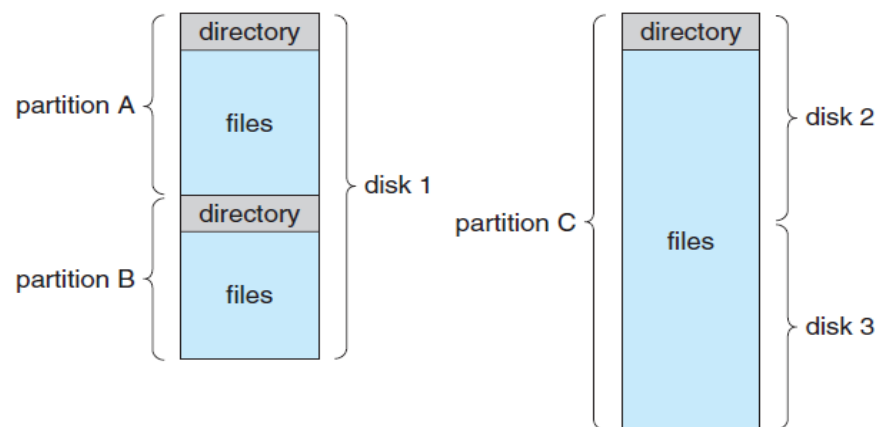


Figure: A typical file-system organization

Storage Structure:

As we have just seen, a general-purpose computer system has multiple storage devices, and those devices can be sliced up into volumes that hold file systems.

Computer systems may have zero or more file systems, and the file systems may be of varying types. For example, a typical Solaris system may have dozens of file systems of a dozen different types, as shown in the file system list in Figure beside:

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	proc
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fd
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs

Consider the types of file systems in the Solaris example mentioned above:

- ✓ **tmpfs**—a “temporary” file system that is created in volatile main memory and has its contents erased if the system reboots or crashes
- ✓ **objfs**—a “virtual” file system (essentially an interface to the kernel that looks like a file system) that gives debuggers access to kernel symbols
- ✓ **ctfs**—a virtual file system that maintains “contract” information to manage which processes start when the system boots and must continue to run during operation
- ✓ **lofs**—a “loop back” file system that allows one file system to be accessed in place of another one
- ✓ **procfs**—a virtual file system that presents information on all processes as a file system
- ✓ **ufs, zfs**—general-purpose file systems

The file systems of computers, then, can be extensive. Even within a file system, it is useful to segregate files into groups and manage and act on those groups. This organization involves the use of directories. In the remainder of this section, we explore the topic of *directory structure*.

Directory Overview:

The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, we see that the directory itself can be organized in many ways. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. In this section, we examine several schemes for defining the logical structure of the directory system.

When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

- ✓ **Search for a file.** We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar names may indicate a relationship among files, we may want to be able to find all files whose names match a particular pattern.
- ✓ **Create a file.** New files need to be created and added to the directory.

- ✓ **Delete a file.** When a file is no longer needed, we want to be able to remove it from the directory.
- ✓ **List a directory.** We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- ✓ **Rename a file.** Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- ✓ **Traverse the file system.** We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied to tape and the disk space of that file released for reuse by another file.

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

Single-Level Directory:

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand (as shown in figure below).

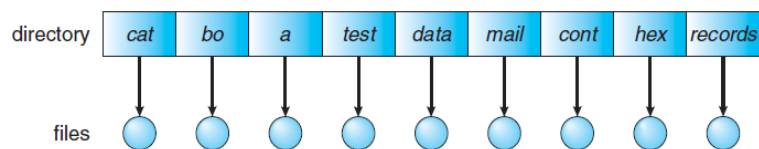


Figure: Single-level directory

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

For example, in one programming class, 23 students called the program for their second assignment prog2.c; another 11 called it assign2.c. Fortunately, most file systems support file names of up to 255 characters, so it is relatively easy to select unique file names.

Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

Two-Level Directory:

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

In the two-level directory structure, each user has his own **user file directory (UFD)**. The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user (Figure below).

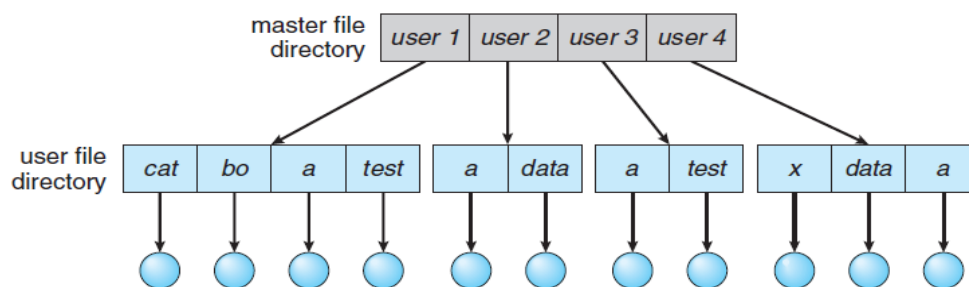


Figure: Two-level directory structure

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.

To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information.

The program creates a new UFD and adds an entry for it to the MFD. The execution of this program might be restricted to system administrators. Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another.

Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.

If access is to be permitted, one user must have the ability to name a file in another user's directory. To name a particular file uniquely in a two-level directory, we must give both the user name and the file name. A two-level directory can be thought of as a tree, or an inverted tree, of height 2. The root of the tree is the MFD. Its direct descendants are the UFDs.

The descendants of the UFDs are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file). Thus, a user name and a file name define a **path name**. Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired.

Tree-Structured Directories:

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

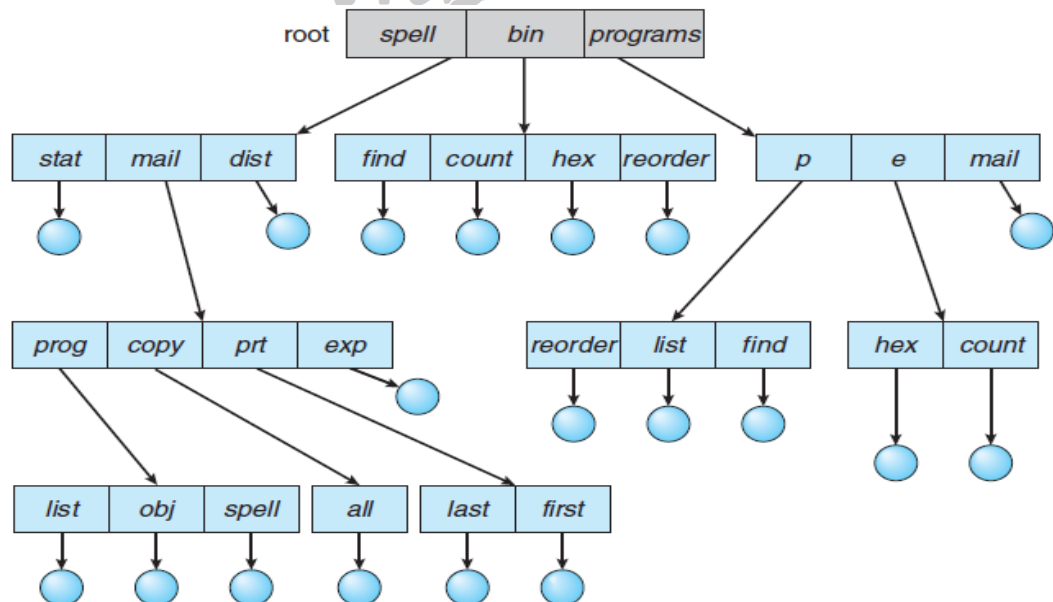


Figure: Tree-structured directory structure

This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.

A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.

In normal use, each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process.

When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file.

To change directories, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory. Thus, the user can change her current directory whenever she wants.

From one change directory() system call to the next, all open() system calls search the current directory for the specified file. Note that the search path may or may not contain a special entry that stands for “the current directory.”

Path names can be of two types: absolute and relative. An **absolute path name** begins at the root and follows a path down to the specified file, giving the directory names on the path.

A **relative path name** defines a path from the current directory. For example, in the tree-structured file system (Figure: tree-structured directory structure), if the current directory is root/spell/mail, then the relative path name prt/first refers to the same file as does the absolute path name root/spell/mail/prt/first.

Acyclic-Graph Directories:

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. In this situation, the common subdirectory should be *shared*.

A shared directory or file exists in the file system in two (or more) places at once. A tree structure prohibits the sharing of files or directories. An **acyclic graph** — that is, a graph with no cycles—allows directories to share subdirectories and files as shown in figure (Acyclic-graph directory structure).

The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

It is important to note that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy.

With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

When people are working as a team, all the files they want to share can be put into one directory. The UFD of each team member will contain this directory of shared files as a subdirectory. Even in the case of a single user, the user's file organization may require that some file be placed in different subdirectories.

For example, a program written for a particular project should be both in the directory of all programs and in the directory for that project. Shared files and subdirectories can be implemented in several ways. A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a link.

A **link** is effectively a pointer to another file or subdirectory. For example, a link may be implemented as an absolute or a relative path name. When a reference to a file is made, we search the directory. If the directory entry is marked as a link, then the name of the real file is included in the link information.

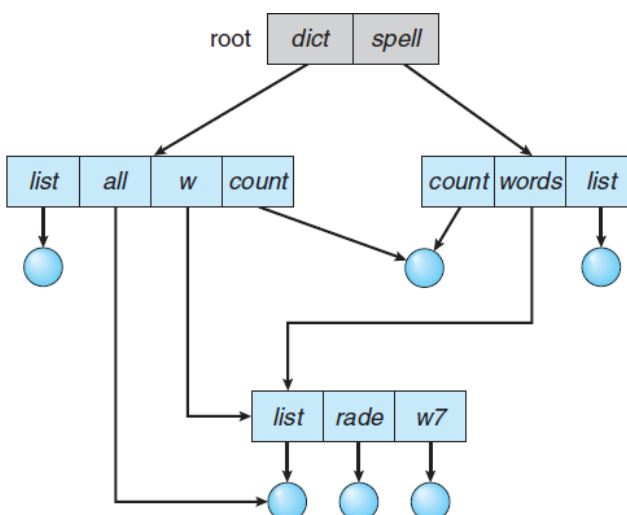


Figure: Acyclic-graph directory structure

We **resolve** the link by using that path name to locate the real file. Links are easily identified by their format in the directory entry (or by having a special type on systems that support types) and are effectively indirect pointers.

The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system. Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal. Consider the difference between this approach and the creation of a link.

The link is clearly different from the original directory entry; thus, the two are not equal. Duplicate directory entries, however, make the original and the copy indistinguishable.

A major problem with duplicate directory entries is maintaining consistency when a file is modified. An acyclic-graph directory structure is more flexible than a simple tree structure, but it is also more complex. Several problems must be considered carefully. A file may now have multiple absolute path names.

Consequently, distinct file names may refer to the same file. This situation is similar to the aliasing problem for programming languages. If we are trying to traverse the entire file system – to find a file, to accumulate statistics on all files, or to copy all files to backup storage – this problem becomes significant, since we do not want to traverse shared structures more than once.

Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused? One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file. Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.

In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link need not affect the original file; only the link is removed.

If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them as well, but unless a list of the associated links is kept with each file, this search can be expensive.

Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist and can fail to resolve the link name; the access is treated just as with any other illegal file name.

General Graph Directory:

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results.

It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links, the tree structure is destroyed, resulting in a simple graph structure as shown in figure (General graph directory).

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. We want to avoid traversing shared sections of an acyclic graph twice, mainly for performance reasons.

If we have just searched a major shared subdirectory for a particular file without finding it, we want to avoid searching that subdirectory again; the second search would be a waste of time.

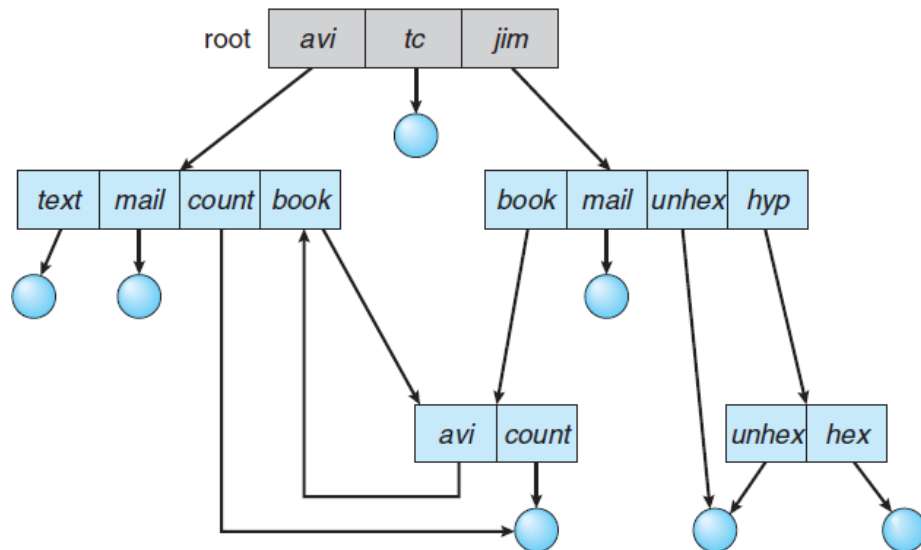


Figure: General graph directory

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution is to limit arbitrarily the number of directories that will be accessed during a search.

File-System Structure:

Disks provide most of the secondary storage on which file systems are maintained. Two characteristics make them convenient for this purpose:

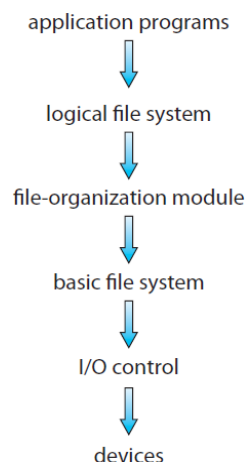
1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read–write heads and waiting for the disk to rotate.

To improve I/O efficiency, I/O transfers between memory and disk are performed in units of blocks. Each block has one or more sectors. Depending on the disk drive, sector size varies from 32 bytes to 4,096 bytes; the usual size is 512 bytes.

File systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user.

This task involves defining a file and its attributes, the operations allowed on a file and the directory structure for organizing files. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in *Figure: Layered file system* (shown below) is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.



The **I/O control** levels consist of device drivers and interrupt handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high-level commands such as “retrieve block 123.”

Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system. The device driver usually writes specific bit patterns to special locations in the I/O controller’s memory to tell the controller which device location to act on and what actions to take.

The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address (for example, drive 1, cylinder 73, track 2, and sector 10). This layer also manages the memory buffers and caches that hold various file-system, directory, and data blocks.

A block in the buffer is allocated before the transfer of a disk block can occur. When the buffer is full, the buffer manager must find more buffer memory or free up buffer space to allow a requested I/O to complete.

Caches are used to hold frequently used file-system metadata to improve performance, so managing their contents is critical for optimum system performance.

The **file-organization module** knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.

Each file’s logical blocks are numbered from 0 (or 1) through N. Since the physical blocks containing the data usually do not match the logical numbers, a translation is needed to locate each block.

The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Finally, the **logical file system** manages metadata information. Metadata includes all of the file-system structure except the actual data (or contents of the files). The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. It maintains file structure via file-control blocks.

A file control block (FCB) (an inode in UNIX file systems) contains information about the file, including ownership, permissions, and location of the file contents. The logical file system is also responsible for protection.

When a layered structure is used for file-system implementation, duplication of code is minimized. The I/O control and sometimes the basic file-system code can be used by multiple file systems. Each file system can then have its own logical file-system and file-organization modules.

Unfortunately, layering can introduce more operating-system overhead, which may result in decreased performance. The use of layering, including the decision about how many layers to use and what each layer should do, is a major challenge in designing new systems.

Many file systems are in use today, and most operating systems support more than one. For example, most CD-ROMs are written in the ISO 9660 format, a standard format agreed on by CD-ROM manufacturers. In addition to removable-media file systems, each operating system has one or more disk based file systems.

UNIX uses the **UNIX file system (UFS)**, which is based on the Berkeley Fast File System (FFS). Windows supports disk file-system formats of FAT, FAT32, and NTFS (or Windows NT File System), as well as CD-ROM and DVD file-system formats.

Although Linux supports over forty different file systems, the standard Linux file system is known as the **extended file system**, with the most common versions being ext3 and ext4. There are also distributed file systems in which a file system on a server is mounted by one or more client computers across a network.

```
owner mjb
group os
type regular file
perms rwxr-xr-x
accessed Oct 23 1984 1:45 P.M.
modified Oct 22 1984 10:30 AM.
inode Oct 23 1984 1:30 P.M.
size 6030 bytes
disk addresses
```

I-node:

The inode (index node) is a data structure in a Unix-style file system that describes a file-system object such as a file or a directory. Each inode stores the attributes and disk block location(s) of the object's data. File-system object attributes may include metadata (times of last change, access, modification), as well as owner and permission data.

Directories are lists of names assigned to inodes. A directory contains an entry for itself, its parent, and each of its children. It exists in a static form on disk. A directory entry contains only a name for the associated file and a pointer to the file itself.

This pointer is an integer called the i-number (for index number) of the file. When the file is accessed, its i-number is used as an index into a system table (the i-list) stored in a known part of the device on which the directory resides. The entry found thereby (the file's i-node) contains the description of the file.

A file system relies on data structures about the files, beside the file content. The former are called metadata—data that describes data. Each file is associated with an inode, which is identified by an integer number, often referred to as an i-number or inode number.

Inodes store information about files and directories (folders), such as file ownership, access mode (read, write, execute permissions), and file type. On many types of file system implementations, the maximum number of inodes is fixed at file system creation, limiting the maximum number of files the file system can hold. A typical allocation heuristic for inodes in a file system is one percent of total size.

The inode number indexes a table of inodes in a known location on the device. From the inode number, the kernel's file system driver can access the inode contents, including the location of the file - thus allowing access to the file.

File Descriptors:

In Unix and related computer operating systems, a file descriptor (FD, less frequently fildes) is an abstract indicator (handle) used to access a file or other input/output resource, such as a pipe or network socket.

A file descriptor is a non-negative integer, generally represented in the C programming language as the type `int` (negative values being reserved to indicate "no value" or an error condition).

Each Unix process (except perhaps a daemon) should expect to have three standard POSIX file descriptors, corresponding to the three standard streams:

Integer value	Name	<unistd.h> symbolic constant	<stdio.h> file stream
0	Standard input	STDIN_FILENO	stdin
1	Standard output	STDOUT_FILENO	stdout
2	Standard error	STDERR_FILENO	stderr

In the traditional implementation of UNIX, file descriptors index into a per-process file descriptor table maintained by the kernel, that in turn indexes into a system-wide table of files opened by all processes, called the file table.

This table records the mode with which the file (or other resource) has been opened: for reading, writing, appending, and possibly other modes. It also indexes into a third table called the inode table that describes the actual underlying files.

To perform input or output, the process passes the file descriptor to the kernel through a system call, and the kernel will access the file on behalf of the process. The process does not have direct access to the file or inode tables.

The following lists typical operations on file descriptors on modern Unix-like systems. Most of these functions are declared in the <unistd.h> header, but some are in the <fcntl.h> header instead.

File-System Implementation:

Overview:

Several on-disk and in-memory structures are used to implement a file system. These structures vary depending on the operating system and the file system, but some general principles apply.

On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.

A **boot control block** (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a volume. In UFS, it is called the **boot block**. In NTFS, it is the **partition boot sector**.

A **volume control block** (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers. In UFS, this is called a **superblock**. In NTFS, it is stored in the **master file table**.

A directory structure (per file system) is used to organize the files. In UFS, this includes file names and associated inode numbers. In NTFS, it is stored in the master file table. A per-file FCB contains many details about the file. It has a unique identifier number to allow association with a directory entry.

In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file. The in-memory information is used for both file-system management and performance improvement via caching.

The data are loaded at mount time, updated during file-system operations, and discarded at dismount. Several types of structures may be included:

- ✓ An in-memory **mount table** contains information about each mounted volume.
- ✓ An **in-memory directory-structure cache** holds the directory information of recently accessed directories. (For directories at which volumes are mounted, it can contain a pointer to the volume table.)
- ✓ The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
- ✓ The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.

Buffers hold file-system blocks when they are being read from disk or written to disk. To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures.

To create a new file, it allocates a new FCB. (Alternatively, if the file-system implementation creates all FCBs at file-system creation time, an FCB is allocated from the set of free FCBs.)

The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk. A typical FCB is shown in figure (A typical file-control block). Some operating systems, including UNIX, treat a directory exactly the same as a file—one with a “type” field indicating that it is a directory.

Other operating systems, including Windows, implement separate system calls for files and directories and treat directories as entities separate from files. Whatever the larger structural issues, the logical file system can call the file-organization module to map the directory I/O into disk-block numbers, which are passed on to the basic file system and I/O control system.

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Figure: A typical file-control block

Partitions and Mounting:

The layout of a disk can have many variations, depending on the operating system. A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks.

Each partition can be either “raw,” containing no file system, or “cooked,” containing a file system. **Raw disk** is used where no file system is appropriate. UNIX swap space can use a raw partition, for example, since it uses its own format on disk and does not use a file system.

Boot information can be stored in a separate partition. Again, it has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format. Rather, boot information is usually a sequential series of blocks, loaded as an image into memory.

Execution of the image starts at a predefined location, such as the first byte. This **boot loader** in turn knows enough about the file-system structure to be able to find and load the kernel and start it executing. It can contain more than the instructions for how to boot a specific operating system.

The **root partition**, which contains the operating-system kernel and sometimes other system files, is mounted at boot time. Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system.

As part of a successful mount operation, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format.

If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention. Finally, the operating system notes in its in-memory mount table that a file system is mounted, along with the type of the file system. The details of this function depend on the operating system.

Microsoft Windows-based systems mount each volume in a separate name space, denoted by a letter and a colon. To record that a file system is mounted at F:, for example, the operating system places a pointer to the file system in a field of the device structure corresponding to F:.

When a process specifies the driver letter, the operating system finds the appropriate file-system pointer and traverses the directory structures on that device to find the specified file or directory.

On UNIX, file systems can be mounted at any directory. Mounting is implemented by setting a flag in the in-memory copy of the inode for that directory. The flag indicates that the directory is a mount point. A field then points to an entry in the mount table, indicating which device is mounted there.

The mount table entry contains a pointer to the superblock of the file system on that device. This scheme enables the operating system to traverse its directory structure, switching seamlessly among file systems of varying types.

Directory Implementation:

The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system. In this section, we discuss the trade-offs involved in choosing one of these algorithms.

Linear List:

The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. This method is simple to program but time-consuming to execute.

To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory. To delete a file, we search the directory for the named file and then release the space allocated to it.

To reuse the directory entry, we can do one of several things. We can mark the entry as unused (by assigning it a special name, such as an all-blank name, or by including a used– unused bit in each entry), or we can attach it to a list of free directory entries.

A third alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory. A linked list can also be used to decrease the time required to delete a file.

The real disadvantage of a linear list of directory entries is that finding a file requires a linear search. Directory information is used frequently, and users will notice if access to it is slow. In fact, many operating systems implement a software cache to store the most recently used directory information.

A cache hit avoids the need to constantly reread the information from disk. A sorted list allows a binary search and decreases the average search time. However, the requirement that the list be kept sorted may complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory.

A more sophisticated tree data structure, such as a balanced tree, might help here. An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

Hash Table:

Another data structure used for a file directory is a hash table. Here, a linear list stores the directory entries, but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.

Therefore, it can greatly decrease the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for collisions—situations in which two file names hash to the same location.

The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size. For example, assume that we make a linear-probing hash table that holds 64 entries.

The hash function converts file names into integers from 0 to 63 (for instance, by using the remainder of a division by 64). If we later try to create a 65th file, we must enlarge the directory hash table—say, to 128 entries.

As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values.

Alternatively, we can use a chained-overflow hash table. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.

Lookups may be somewhat slowed, because searching for a name might require stepping through a linked list of colliding table entries. Still, this method is likely to be much faster than a linear search through the entire directory.

Allocation Methods:

The direct-access nature of disks gives us flexibility in the implementation of files. In almost every case, many files are stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.

Three major methods of allocating disk space are in wide use: *contiguous*, *linked*, and *indexed*. Each method has advantages and disadvantages. Although some systems support all three, it is more common for a system to use one method for all files within a file-system type.

Contiguous Allocation:

Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block $b + 1$ after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head need only move from one track to the next.

Thus, the number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time when seek is finally needed. Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is n blocks long and starts at location b , then it occupies blocks $b, b + 1, b + 2, \dots, b + n - 1$.

The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file as shown in figure as follows.

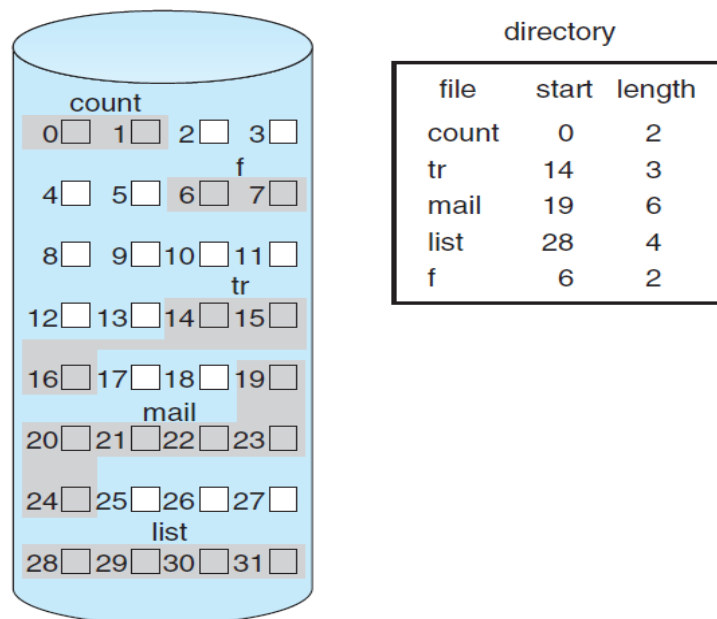


Figure: Contiguous allocation of disk space

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block.

For direct access to block i of a file that starts at block b , we can immediately access block $b + i$. Thus, both sequential and direct access can be supported by contiguous allocation.

Contiguous allocation has some problems, however. One difficulty is finding space for a new file. The system chosen to manage free space determines how this task is accomplished.

The contiguous-allocation problem can be seen as a particular application of the general **dynamic storage-allocation** problem, which involves how to satisfy a request of size n from a list of free holes. First fit and best fit are the most common strategies used to select a free hole from the set of available holes.

Simulations have shown that both first fit and best fit are more efficient than worst fit in terms of both time and storage utilization. Neither first fit nor best fit is clearly best in terms of storage utilization, but first fit is generally faster.

All these algorithms suffer from the problem of **external fragmentation**. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks.

It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data. Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.

One strategy for preventing loss of significant amounts of disk space to external fragmentation is to copy an entire file system onto another disk. The original disk is then freed completely, creating one large contiguous free space.

We then copy the files back onto the original disk by allocating contiguous space from this one large hole. This scheme effectively **compacts** all free space into one contiguous space, solving the fragmentation problem.

Another problem with contiguous allocation is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated.

How does the creator (program or person) know the size of the file to be created? In some cases, this determination may be fairly simple (copying an existing file, for example). In general, however, the size of an output file may be difficult to estimate.

Linked Allocation:

JKD

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk.

The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25 as shown in figure as follows (Linked allocation of disk space).

Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to null (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file.

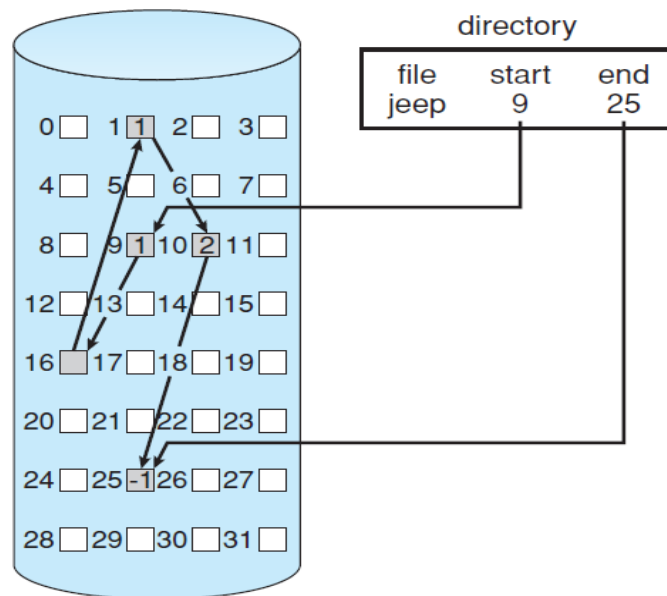


Figure: Linked allocation of disk space

To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request.

The size of a file need not be declared when the file is created. A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files. To find the i th block of a file, we must start at the beginning of that file and follow the pointers until we get to the i th block.

Each access to a pointer requires a disk read, and some require a disk seek. Consequently, it is inefficient to support a direct-access capability for linked-allocation files.

Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information. Each file requires slightly more space than it would otherwise.

The usual solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate clusters rather than blocks. For instance, the file system may define a cluster as four blocks and operate on the disk only in cluster units. Pointers then use a much smaller percentage of the file's disk space.

This method allows the logical-to-physical block mapping to remain simple but improves disk throughput (because fewer disk-head seeks are required) and decreases the space needed for block allocation and free-list management.

The cost of this approach is an increase in internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full. Clusters can be used to improve the disk-access time for many other algorithms as well, so they are used in most file systems.

Yet another problem of linked allocation is reliability. Recall that the files are linked together by pointers scattered all over the disk, and consider what would happen if a pointer were lost or damaged.

A bug in the operating-system software or a disk hardware failure might result in picking up the wrong pointer. This error could in turn result in linking into the free-space list or into another file.

An important variation on linked allocation is the use of a **file-allocation table (FAT)**. This simple but efficient method of disk-space allocation was used by the MS-DOS operating system. A section of disk at the beginning of each volume is set aside to contain the table.

The table has one entry for each disk block and is indexed by block number. The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file.

This chain continues until it reaches the last block, which has a special end-of-file value as the table entry. An unused block is indicated by a table value of 0. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block.

The 0 is then replaced with the end-of-file value. An illustrative example is the FAT structure shown in Figure (File-allocation table) as follows for a file consisting of disk blocks 217, 618, and 339.

Indexed Allocation:

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order.

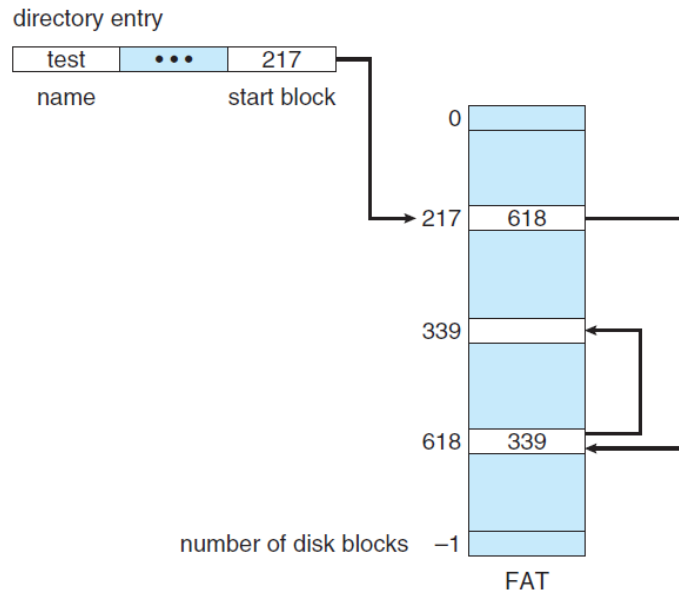


Figure: File-allocation table

Indexed allocation solves this problem by bringing all the pointers together into one location: the **index block**. Each file has its own index block, which is an array of disk-block addresses. The *i*th entry in the index block points to the *i*th block of the file.

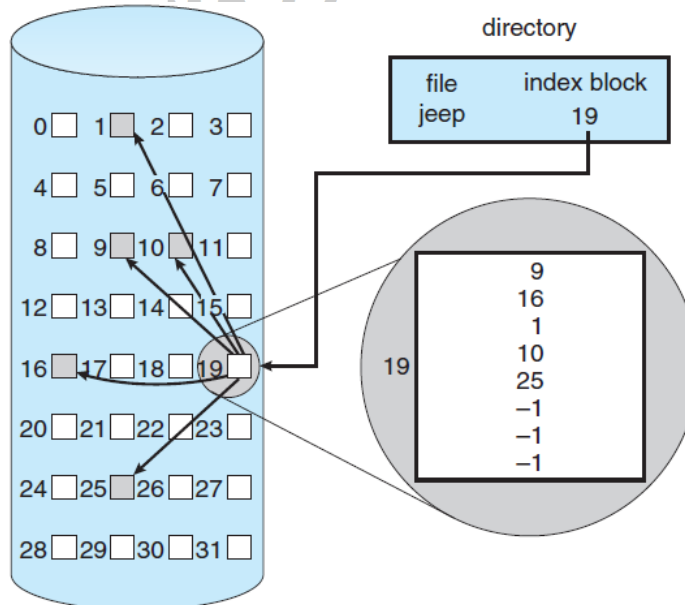


Figure: Indexed allocation of disk space

The directory contains the address of the index block as shown in figure above. To find and read the *i*th block, we use the pointer in the *i*th index-block entry. This scheme is similar to the paging scheme.

When the file is created, all pointers in the index block are set to null. When the i^{th} block is first written, a block is obtained from the free-space manager, and its address is put in the i^{th} index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space. Indexed allocation does suffer from wasted space, however. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.

Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block. With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-null.

This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue. Mechanisms for this purpose include the following:

Linked scheme: An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks. For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is null (for a small file) or is a pointer to another index block (for a large file).

Multilevel index: A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block.

This approach could be continued to a third or fourth level, depending on the desired maximum file size. With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block. Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.

Combined scheme: Another alternative, used in UNIX-based file systems, is to keep the first, say, 15 pointers of the index block in the file's inode. The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file.

Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then, up to 48 KB of data can be accessed directly. The next three pointers point to **indirect blocks**. The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data.

The second, points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a **triple indirect block**. A UNIX inode is shown in Figure below:

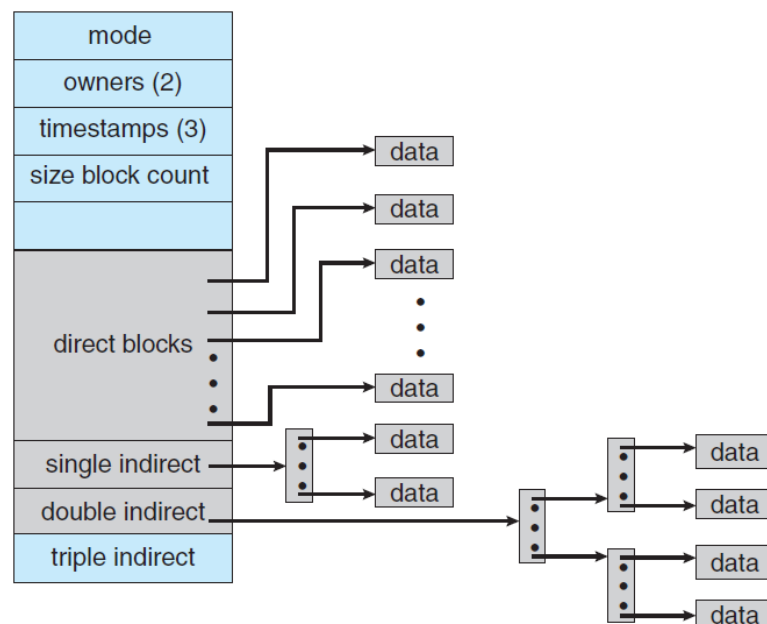


Figure: The UNIX Inode

Under this method, the number of blocks that can be allocated to a file exceeds the amount of space addressable by the 4-byte file pointers used by many operating systems. A 32-bit file pointer reaches only 2³² bytes, or 4 GB. Many UNIX and Linux implementations now support 64-bit file pointers, which allows files and file systems to be several exbibytes in size. The ZFS file system supports 128-bit file pointers.

Indexed-allocation schemes suffer from some of the same performance problems as does linked allocation. Specifically, the index blocks can be cached in memory, but the data blocks may be spread all over a volume.