UNIT 5

FILE SYSTEM INTERFACE

The concept of a file, Access Methods, Directory structure, File system mounting, files sharing, protection. File System implementation- File system structure, allocation methods, free-space management Mass-storage structure overview of Mass-storage structure, Disk scheduling, Device drivers. Introduction to Dockers.

FILE CONCEPT:

- Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks and 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. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.
- Commonly, files represent programs 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. 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 name is usually a string of characters, such as example.c. When a file is named, it becomes independent of the process, the user, and even the system that created it. 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. 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 as needed.

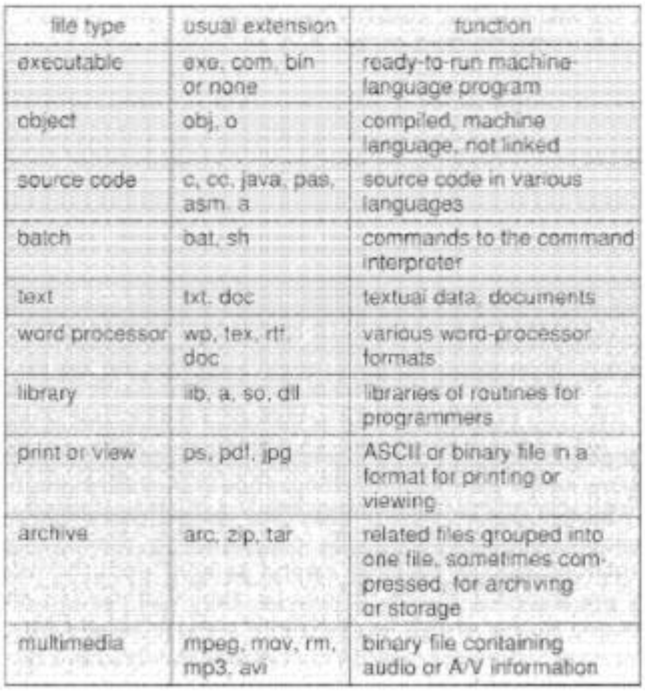
File Operations

A file is an abstract data type. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. The various operations are

***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. Both the read and write operations use this same pointer, saving space and reducing system, complexity.
***Repositioning within a file**:* The directory is searched for the appropriate entry and the current file position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file 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.

File Types



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

Fig: Common File Types

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.

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.

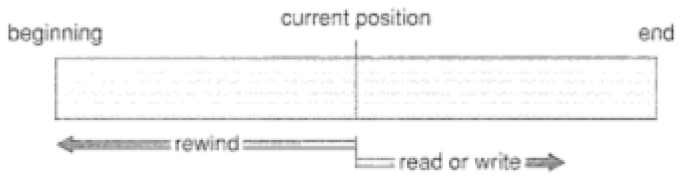
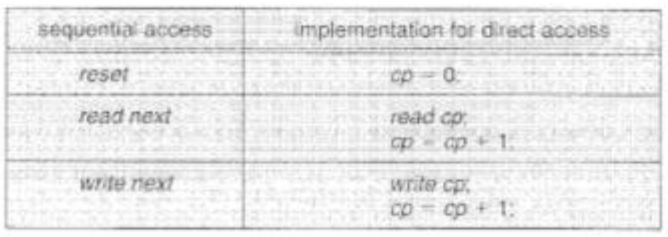


Fig: Sequential-access file.

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 Fig 6.2, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

Direct Access

In direct access a file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order. 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.



sequential access	implementation for direct access
reset	*cp = 0;*

read next	*read cp;*
	cp = cp + 1;

write next	*write cp;*
	cp = cp + 1;

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

Direct-access files are of great use for immediate access to large amounts of information. Databases are often use 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.

For the direct-access method, the file operations must be modified to include the block number as a parameter. 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 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.

Assuming we have a logical record length L, the request for record N is turned into an I/O request for L bytes starting at location L * (N) within the file (assuming the first record is N = 0). 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 Fig.

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.

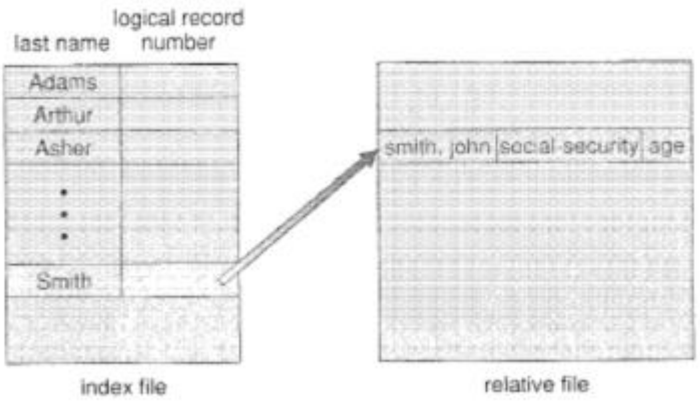


Fig: Examples of Index and Relative files.

With large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files, which would point to the actual data items. Fig shows a similar situation as implemented by VMS index and relative files.

DIRECTORY STRUCTURE

The file systems of computers can be extensive. Some systems store millions of files on terabytes of disk, to manage all these data, we need to organize them. This organization involves the use of directories.

Storage Structure

A disk or any storage device can be used in its entirety for a file system. The disks are divided into various parts known as partitions, slices, or minidisks. A file system can be created on each of these parts of the disk. The parts can also be combined to form larger structures known as volumes.

For now, we simply refer to a chunk of storage that holds a file system as a volume. Volumes can also store multiple operating systems, allowing a system to boot and run more than one. Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a device directory or volume table of contents. The device directory records information such as name, location, size, and type for all files on that volume. Fig 6.5 shows a typical file-system organization.

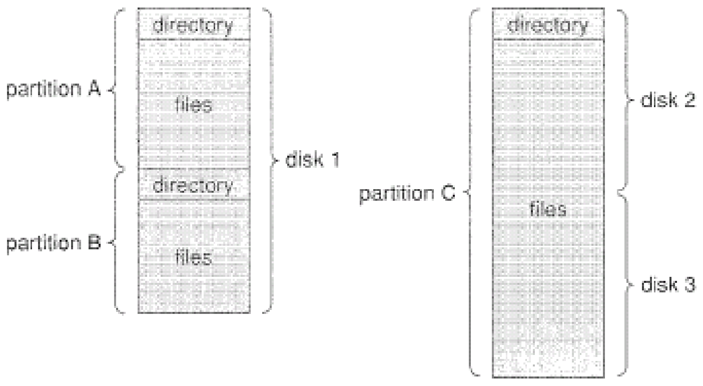


Fig: A typical file-system organization.

Directory Overview

The operations that are to be performed on a directory with a particular structure are:

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 between 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: 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.

Single-Level Directory

In single-level directory all files are contained in the same directory with unique names, which is easy to support and understand (Fig 6.6) however, the limitation is when the number of files increases or when the system has more than one user. 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.

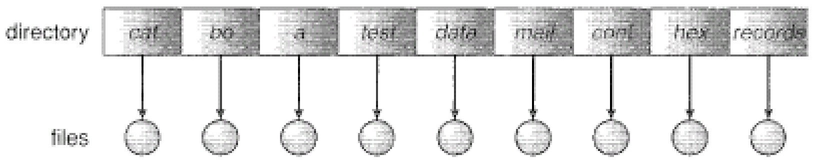


Fig: Single-level directory structure.

Two-Level Directory

A single-level directory often leads to confusion of file names among different users. In the two-level directory structure, each user has its own user file directory (UFD). The UFDs have similar structures, but each lists only the files of a single user. When 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 (Fig).

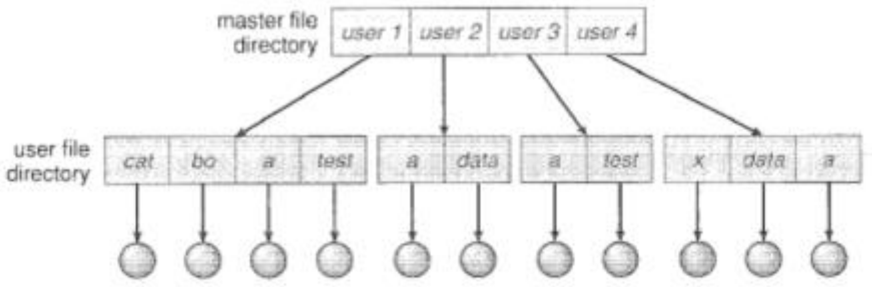


Fig: Two-level directory structure.

When a user refers to a particular file, only his own UFD is searched. To create a file for a user, the operating system searches only that user's UFD to check whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot 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 by the administrators with the appropriate user name and account information. The program creates a new UFD and adds an entry for it to the MFD. Isolation of users 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. 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.

Tree-Structured Directories

The next generalization is to extend the directory structure to a tree of arbitrary height (Fig 6.8). This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a. unique path name.

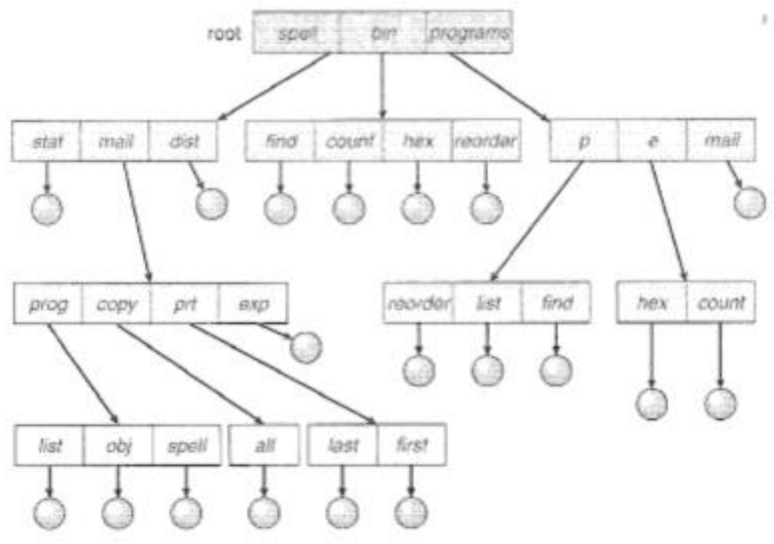


Fig: Tree-structured directory structure.

A directory 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 his current directory whenever he desires.

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 deletion of a directory, if a directory is empty, its entry in the directory that contains it can simply be deleted. Suppose the directory to be deleted is not empty, the user must first delete all the files in that directory, if any subdirectories exist, this procedure must be applied recursively to them, so that they can be deleted also.

Acyclic-Graph Directories

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 (Fig 6.9). The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

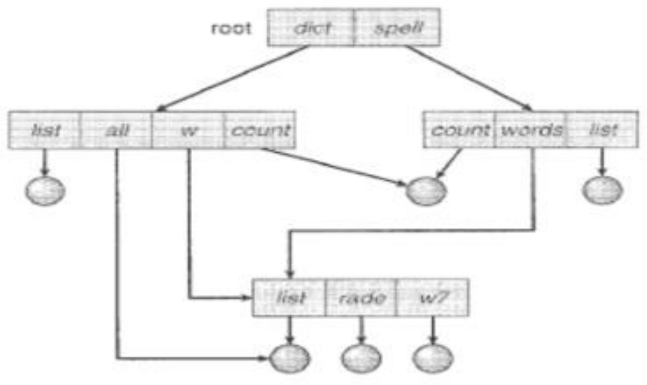


Fig: Acyclic-graph directory structure.

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.

A 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. A 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.

General Graph Directory

When we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure (Fig 6.10). 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.

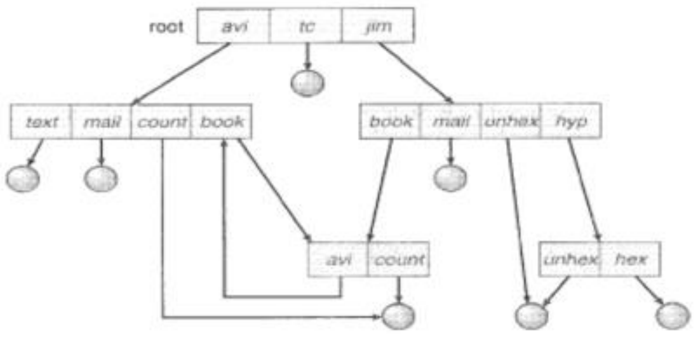


Fig: General graph directory structure.

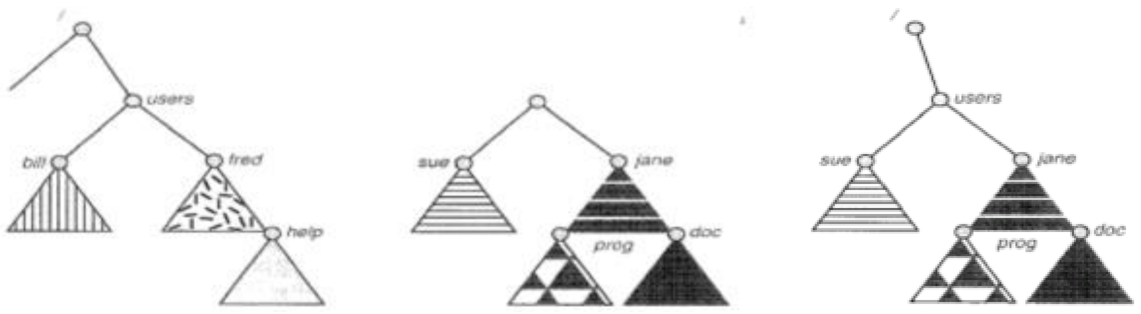
If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution is to limit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

A similar problem exists when we are trying to determine when a file can be deleted. With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted.

However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a. directory or file. This anomaly results from the possibility of self-referencing (or a cycle) in the directory structure.

In this case, we generally need to use a garbage-collection scheme to determine when the last reference has been deleted and the disk space can be reallocated. Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. Garbage collection for a disk-based file system, however, is extremely time consuming and is thus seldom attempted. Garbage collection is necessary only because of possible cycles in the graph. Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles as new links are added to the structure.

FILE-SYSTEM MOUNTING



(a) (b) (c)

Fig: File system, (a) Existing system. (b) Unmounted volume. (c) Mount point.

To illustrate file mounting, consider the file system depicted in Fig, where the triangles represent sub-trees of directories that are of interest. Fig (a) shows an existing file system, while Fig (b) shows an unmounted volume residing on device/disk. At this point, only the files on the existing file system can be accessed. Fig (c) shows the effects of mounting the volume residing on device/disk over users. If the volume is unmounted, the file system is restored to the situation, depicted in Fig. Systems impose semantics to clarify functionality. For example, a system may disallow mount over a directory that contains files; or it may make the mounted file system available at that directory and obscure the directory's existing files until the file system is unmounted, terminating the use of the file system and allowing access to the original files in that directory. As another example, a system may allow the same file system to be mounted repeatedly, at different mount points; or it may only allow one mount per file system.

FILE SHARING

Multiple Users

By multiple users the issues of file sharing, file naming, and file protection become preeminent. Given a directory structure that allows files to be shared by users, the system must referee the file sharing. The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files. These are the issues of access control and protection. Most systems have evolved to use the concepts of file (or directory) owner (or user) and group. The owner is the user who can change attributes and grant access and who has the most control over the file. The group attribute defines a subset of users who can share access to the file. The owner and group IDs of a given file (or directory) are stored with the other file attributes. When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file.
Likewise, the group IDs can be compared. The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it.

Remote File Systems

In remote file-sharing methods the first implemented method involves manually transferring files between machines via programs like ftp. The second major method uses a distributed file system (DFS) in which remote directories is visible from a local machine. In some ways, the World Wide Web, is a reversion to the first. A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for ftp) are used to transfer files. ftp is used for both anonymous and authenticated access. Anonymous access allows a user to transfer files without having an account on the remote system. DFS involves a much tighter integration between the machine that is accessing the remote files and the machine providing the files. This integration adds complexity, which we describe in this section.

The Client-Server Model

Remote file systems allow a computer to mount one or more file systems from one or more remote machines. In this case with networked machines, the machine containing the files is the server, and the machine seeking access to the files is the client. Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients. A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client-server facility. The server usually specifies the available files on a volume or directory level. Client identification is more difficult. A client can be specified by a network name or other identifier, such as an IP address, but these can be spoofed, or imitated. As a result of spoofing, an unauthorized client could be allowed access to the server. More secure solutions include secure authentication of the client via encrypted keys.

Distributed Information Systems

To make client-server systems easier to manage, distributed information systems, also known as distributed naming services, provide unified access to the information needed for remote computing. The domain name system (DNS) provides host-name-to-network-address translations for the entire Internet (including the World Wide Web). Before DNS became widespread, files containing the same information were sent via e-mail or ftp between all networked hosts.

Failure Modes

Local file systems can fail for a variety of reasons, including failure of the disk containing the file system, corruption of the directory structure or other disk-management information, disk-controller failure, cable failure, and host-adapter failure. Remote file systems have even more failure modes. Because of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems. Consider a client in the midst of using a remote file system.

It has files open from the remote host; among other activities, it may be performing directory lookups to open files, reading or writing data to files, and closing files. Now consider a partitioning of the network, a crash of the server, or even a scheduled shutdown of the server. Suddenly, the remote file system is no longer reachable. This scenario is rather common, so it would not be appropriate for the client system to act as it would if a local file system were lost. Rather, the system can either terminate all operations to the lost server or delay operations until the server is again reachable. To implement this kind of recovery from failure, some kind of state information may be maintained on both the client and the server. If both and client maintain knowledge of their current activities and open files, then they can seamlessly recover from a failure.

Consistency Semantics

Consistency semantics represent an important criterion for evaluating any file system that supports file sharing. These semantics specify how multiple users of a system are to access a shared file simultaneously.

In. particular, they specify when modifications of data by one user will be observable by other users. These semantics are typically implemented as code with the file system.

PROTECTION

When data is stored in a computer system, we want to keep it safe from physical damage (reliability) and improper access (protection). Reliability is generally provided by duplicate copies of files. File systems can be damaged by hardware problems, power surges or failures, head crashes, dirt, temper-ature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system soft ware can also cause file contents to be lost. Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multiuser system, however, other mechanisms are needed.

Types of Access

Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

• Read: Read from the file.
• Write: Write or rewrite the file.
• Execute: Load the file into memory and execute it.
• Append: Write new information at the end of the file.
• Delete: Delete the file and free its space for possible reuse.
• List: List the name and attributes of the file.

Other operations, such as renaming, copying, and editing the file, may also be controlled. For many systems, however, these higher-level functions may be implemented by a system program that makes lower-level system calls. Protection is provided at only the lower level.

Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. The most general scheme to implement identity dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated, with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access.
This technique has two undesirable consequences:
• Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
• The directory entry, previously of fixed size, now needs to be of variable size, resulting in more complicated space management.
These problems can be resolved by use of a condensed version of the access list.

To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

• Owner: The user who created the file is the owner.
• Group: A set of users who are sharing the file and need similar access is a group or work group.
• Universe: All other users in the system constitute the universe.

For this scheme to work properly, permissions and access lists must be controlled tightly. This control can be accomplished in several ways.

Other Protection Approaches

Another approach to the protection problem is to associate a password with each file. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages; however first, the number of passwords that a user needs to remember may become large, making the scheme impractical. Second, if only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis.

In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories; that is, we need to provide a mechanism for directory protection. The directory operations that must be protected are different from the file operations to control the creation and deletion of files in a directory.

In addition, we want to control whether a user can determine the existence of a file in. a directory. Sometimes, knowledge of the existence and name of a file is significant in itself. Thus, listing the contents of a directory must be a protected operation.

FILE-SYSTEM STRUCTURE

Disks provide the bulk of secondary storage on which a file system is maintained and the two characteristics that make them a convenient medium for storing multiple files:

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 given 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.

Rather than transferring a byte at a time, 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, sectors vary from 32 bytes to 4,096 bytes; usually, they are 512 bytes. To provide efficient and convenient access to the disk, the operating system imposes one or more file systems to allow the data to be stored, located, and retrieved easily. The structure shown in Fig 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.

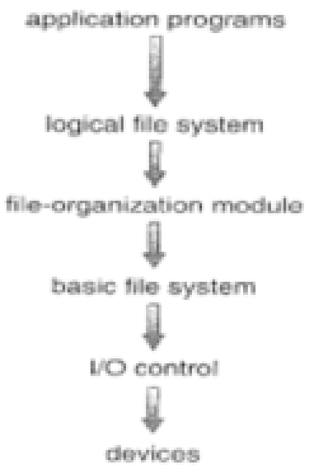


Fig: Layered file system.

The lowest level i.e., I/O control, consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system. 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).

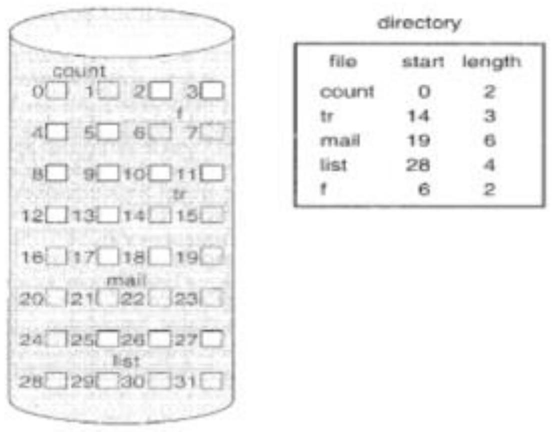
The file-organization module knows about files, their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.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

ALLOCATION METHODS ,

Three major methods of allocating disk space are contiguous, linked, and indexed with advantages and disadvantages. More commonly, a system uses 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 with a linear ordering on the disk. 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, ..., 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 (Fig).With this ordering, assuming that only one job is accessing the disk, accessing block *b + 1* after block *b* normally requires no head movement. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time when finally needed.



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Fig: Contiguous allocation of disk space.

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 is easy and can be supported by contiguous allocation.

Contiguous allocation has some problems like finding space for a new file. Another 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 suffering from the problem of external fragmentation. If we allocate too little space to a file, we may find that the file cannot be extended. Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Even if the total amount of space needed for a file is known in advance, pre-allocation may be inefficient. A file that will grow slowly over a long period must be allocated enough space for its final size, even though much of that space will be unused for a long time. The file therefore has a large amount of internal fragmentation.

To minimize these drawbacks, some operating systems use a modified contiguous-allocation scheme. Here, a contiguous chunk of space is allocated initially; and then, if that amount proves not to be large enough, another chunk of contiguous space, known as an extent, is added. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent.

Linked Allocation

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 (Fig). 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 nil (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. 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 that 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.

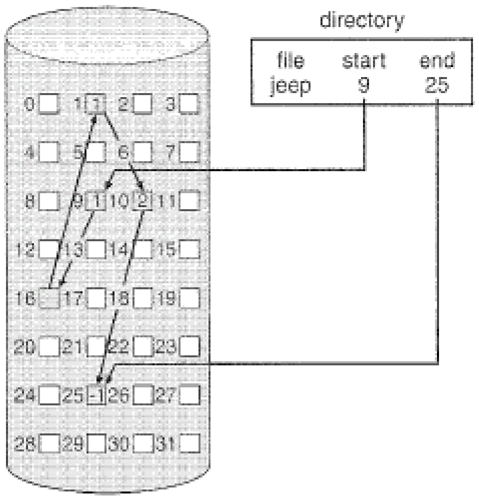


Fig: Linked allocation of disk space.

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

Yet another problem of linked allocation is reliability. 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. One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block; however, these schemes require even more overhead for each file.

A variation on linked allocation is the use of a file-allocation table (FAT). 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 the last block, which has a special end-of-file value as the table entry. Unused blocks are indicated by a 0 table value.
Allocating a new block to a file is done by 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 Fig for a file consisting of disk blocks 217, 618, and 339.

The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. The disk head must move to the start of the volume to read the FAT and find the location, of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. A benefit is that random-access time is improved, because the disk head can find the location of any block by reading the information in the fat.

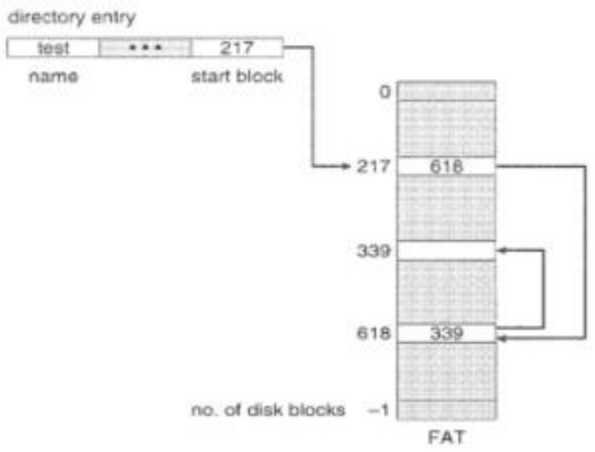


Fig: File-allocation table.

Indexed Allocation

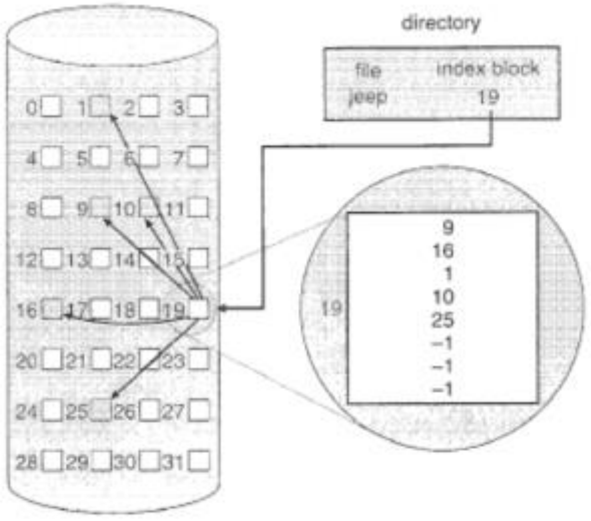


Fig: Indexed allocation of disk space.

Each file has its own index block, which is an array of disk-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block (Fig). To find and read the ith block, we use the pointer in the ith index-block entry. When the file is created, all pointers in the index block are set to nil. When the ith block is first written, a block is obtained from the free-space manager, and its address is put in the ith 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. 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 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 nil (for a small file) or is a pointer to another index block (for a large file).

***Multilevel index*:** A variant of the linked representation is to use 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.

***Combined scheme**:* Another alternative, used in the UFS, is to keep the first, say, 15 pointers of the index block in the file's i node. 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 do not need a separate index block. If the block size is 4 KB, then upto 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*.

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 2^{32} bytes, or 4 GB.

FREE-SPACE MANAGEMENT

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. To keep track of free disk space, the system, maintains a free-space list. The free-space list records all free disk blocks those not allocated to some file or directory.
To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. The free-space list, despite its name, might not be implemented as a list, as we discuss next.

Bit Vector
Frequently, the free-space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.
For example, consider a disk where blocks 2, 3, 4, 5, 8, 9,10,11,12,13,17, 18,25,26, and 27 are free and the rest are allocated. The free-space bit map would be 001111001111110001100000111….

The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk. Indeed, many computers supply bit-manipulation instructions that can be used effectively for that purpose. One technique for finding the first free block on a system that uses a bit-vector to allocate disk space is to sequentially check each word in the bit map to see whether that value is not 0, since a 0-valued word has all 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block.

The calculation of the block number is
(number of bits per word) x (number of 0-value words) + offset of first 1 bit.
Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory. Keeping it in main memory is possible for smaller disks but not necessarily for larger ones.

Linked List
In this all the free disk blocks are linked together, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on. For example, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on (Fig).

However, this scheme is not efficient to traverse the list; we must read each block, which requires substantial I/O time. Fortunately, traversing the free list is not a frequent action. Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used. The FAT method incorporates free-block accounting into the allocation data structure. No separate method is needed.

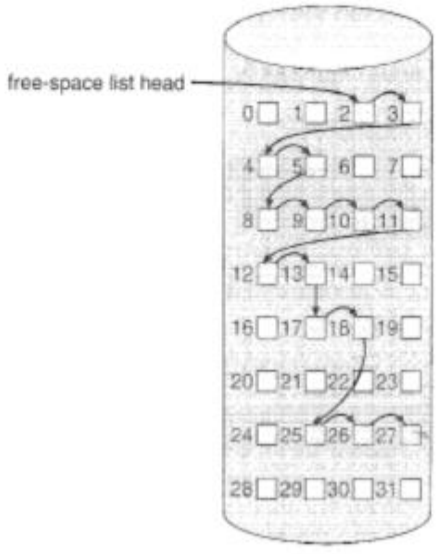


Fig: Linked free-space list on disk.

Grouping
A modification of the free-list approach is to store the addresses of n free blocks in the first free block. The first *n-1* of these blocks is actually free. The last block contains the addresses of other *n* free blocks, and so on. The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.

Counting
Generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering. Thus, rather than keeping a list of *n* free disk addresses, we can keep the address of the first free block and the number *n* of free contiguous blocks that follow the first block.

Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

OVERVIEW OF MASS-STORAGE STRUCTURE

Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Conceptually, disks are relatively simple (Fig). Each disk platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 5.25 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.

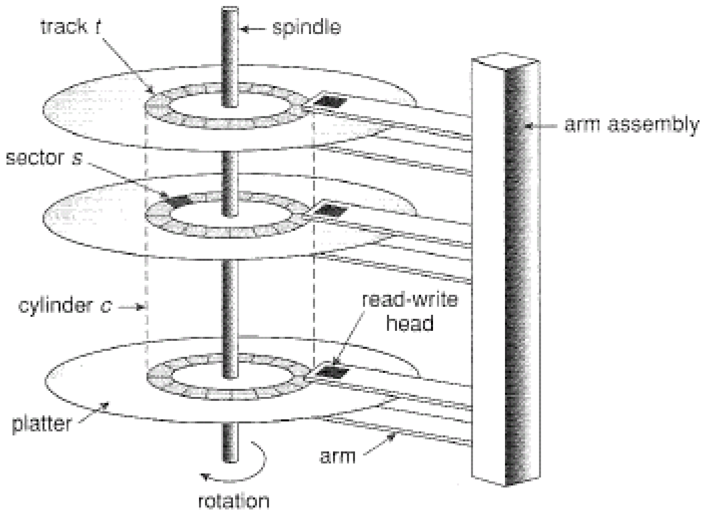


Fig: Moving-head disk mechanism.

A read-write head "flies" just above each surface of every platter. The heads are attached to a disk arm that moves all the heads as a unit. The surface of a platter is logically divided into circular tracks, which are subdivided into sectors. The set of tracks that are at one arm position makes up a cylinder. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes.

When the disk is in use, a drive motor spins it at high speed. Disk speed has two parts. The transfer rate is the rate at which data flow between the drive and the computer. The positioning time, sometimes called the random-access time, consists of the time to move the disk arm to the desired cylinder, called the seek time, and the time for the desired sector to rotate to the disk head, called the rotational latency.

A disk drive is attached to a computer by a set of wires called an I/O bus. The data transfers on a bus are carried out by electronic processors called controllers. The host controller is the controller at the computer end of the bus. A disk controller is built into each disk drive. The host controller then sends the command via messages to the disk controller, and the disk controller operates the disk-drive hardware to carry out the command. Disk controllers usually have a built-in cache. Data transfer at the disk drive happens between the cache and the disk surface, and data transfer to the host, at fast electronic speeds, occurs between the cache and the host controller.

Magnetic Tapes

Magnetic tape was used as an early secondary-storage medium. Although it is relatively permanent and can hold large quantities of data, its access time is slow compared with that of main memory and magnetic disk. Tapes are used primarily for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

A tape is kept in a spool and is wound or rewound past a read-write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives. Tape capacities vary greatly, depending on the particular kind of tape drive. Typically, they store from 20 GB to 200 GB. Some have built-in compressions that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch.

DISK STRUCTURE

Modern disk drives are addressed as large one-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be low-level formatted to have a different logical block size, such as 1,024 bytes. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.
By using this mapping, we can at least in theory convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track.

In practice, it is difficult to perform, this translation, for two reasons.
First, most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk.
Second, the number of sectors per track is not a constant on some drives.
On media that use constant linear velocity (CLV), the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head.

This method is used in CD-ROM and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant, and the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as constant angular velocity (CAV).
The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders

DISK ATTACHMENT

Computers access disk storage in two ways. One way is via I/O ports common on small systems. The other way is via a remote host in a distributed file system; this is referred to as network-attached storage.
Host-Attached Storage
Host-attached storage is storage accessed through local I/O ports. The typical desktop PC uses an I/O bus architecture called IDE or ATA. This architecture supports a maximum of two drives per I/O bus. A similar protocol that has simplified cabling is SATA. High-end workstations and servers generally use more sophisticated I/O architectures, such as SCSI and fiber channel (FC).

SCSI is a bus architecture. Its physical medium is usually a ribbon cable having a large number of conductors (typically 50 or 68). The SCSI protocol supports a maximum of 16 devices on the bus.

Generally, the devices include one controller card in the host (the SCSI initiator) and up to 15 storage devices (the SCSI targets). A SCSI disk is a common SCSI target, but the protocol provides the ability to address up to 8 logical units in each SCSI target. A typical use of logical unit addressing is to direct commands to components of a RAID array or components of a removable media library.
FC is a high-speed serial architecture that can operate over optical fiber or over a four-conductor copper cable. It has two variants. One is a large switched fabric having a 24-bit address space. This variant is expected to dominate in the future and is the basis of storage-area networks (SANs). Because of the large address space and the switched nature of the communication, multiple hosts and storage devices can attach to the fabric, allowing great flexibility in I/O communication. The other FC variant is an arbitrated loop (FC-AL) that can address 126 devices (drives and controllers).
A wide variety of storage devices are suitable for use as host-attached storage. Among these are hard disk drives, RAID arrays, and CD, DVD, and tape drives. The I/O commands that initiate data transfers to a host-attached storage device are reads and writes of logical data blocks directed to specifically identified storage units (such as bus ID, SCSI ID, and target logical unit).

Network-Attached Storage
A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a data network (Fig). Clients access network-attached storage via a remote-procedure-call interface such as NFS for UNIX systems or CIFS for Windows machines. The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network usually the samelocal-area network (LAN) that carries all data traffic to the clients.

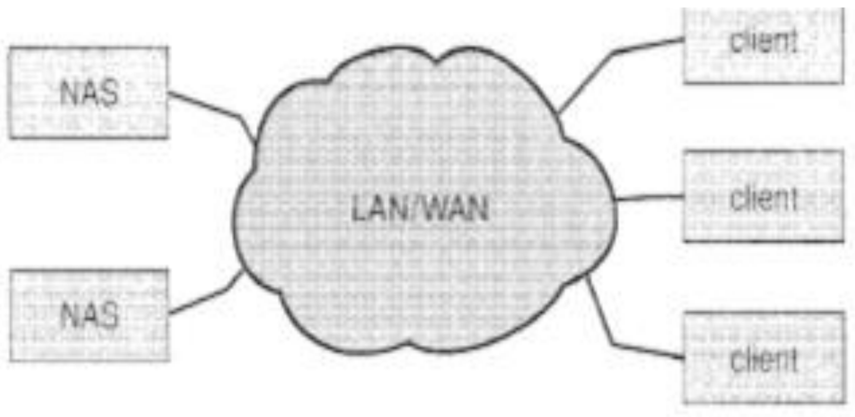


Fig: Network-attached storage.

Network-attached storage provides a convenient way for all the computers on a LAN to share a pool of storage with the same ease of naming and access enjoyed with local host-attached storage. However, it tends to be less efficient and have lower performance than some direct-attached storage options.

Storage-Area Network
One drawback of network-attached storage systems is that the storage I/O operations consume bandwidth on the data network, thereby increasing the latency of network communication. This problem can be particularly acute in large client-server installations, the communication between servers and clients competes for bandwidth with the communication among servers and storage devices.

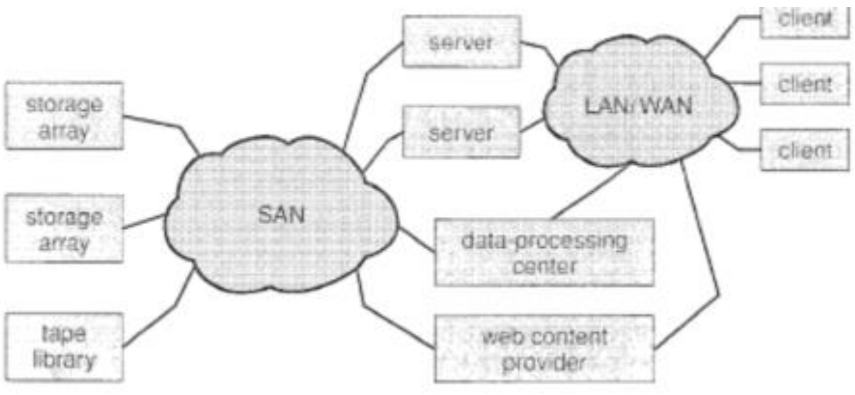


Fig: Storage-area network.

A storage-area network (SAN) is a private network connecting servers and storage units, as shown in Fig. The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN, and storage can be dynamically allocated to hosts. A SAN switch allows or prohibits access between the hosts and the storage. As one example, if a host is running low on disk space, the SAN can be configured to allocate more storage to that host. SANs make it possible for clusters of servers to share the same storage and for storage arrays to include multiple direct host connections. SANs typically have more ports, and less expensive ports, than storage arrays. FC is the most common SAN interconnect.

DISK SCHEDULING

One of the responsibilities of the operating system is to use the hardware efficiently having fast access time and large disk bandwidth. The access time has two major components. The seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector. The rotational latency is the additional time for the disk to rotate the desired sector to the disk head.

The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order. Whenever a process needs I/O to or from the disk, it issues a system call to the operating system requesting several pieces of information like whether this operation is input or output ,what the disk address for the transfer is, what the memory address for the transfer is , what the number of sectors to be transferred is. If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive. Several disk-scheduling algorithms can be used like

FCFS Scheduling

The simplest form of disk scheduling is the first-come, first-served (FCFS) algorithm. This algorithm is basically fair but does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in Fig.

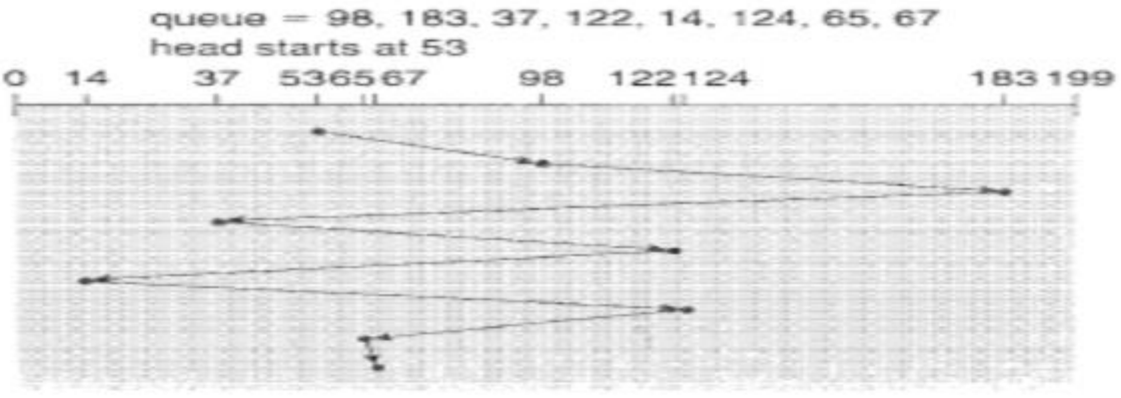


Fig: FCFS disk scheduling.

The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests at 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

SSTF Scheduling

To provide service all the requests close to the current head position before moving the head far away to service other requests is the basis for the shortest-seek-time-first (SSTF) algorithm. The SSTF algorithm selects the request with the minimum seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head, position.
For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67.
From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98,122,124, and finally 183 (Fig). This scheduling method results in a total head movement of only 236 cylinders—little more than one-third of the distance needed for FCFS scheduling of this request queue. This algorithm gives a substantial improvement in performance.
SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling it may cause starvation of some requests.

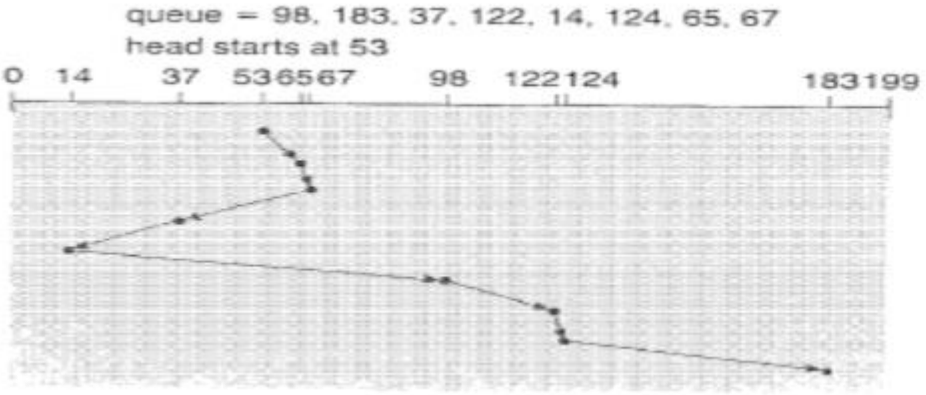


Fig: SSTF disk scheduling.

Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal. In the example, we can do better by moving the head from 53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183. This strategy reduces the total head movement to 208 cylinders.

SCAN Scheduling

In the SCAN algorithm, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.

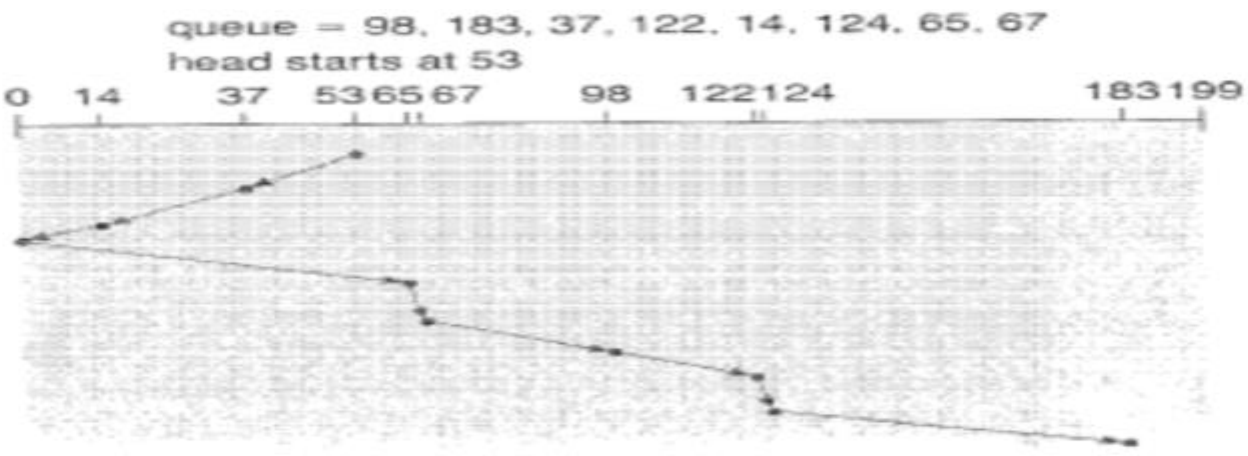


Fig: SCAN disk scheduling.

In the example to illustrate, before applying SCAN to schedule the requests on cylinders 98,183, 37,1.22,14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position (53).

If the disk arm is moving toward 0, the head will service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Fig).

If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

C-SCAN Scheduling

Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip (Fig). The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

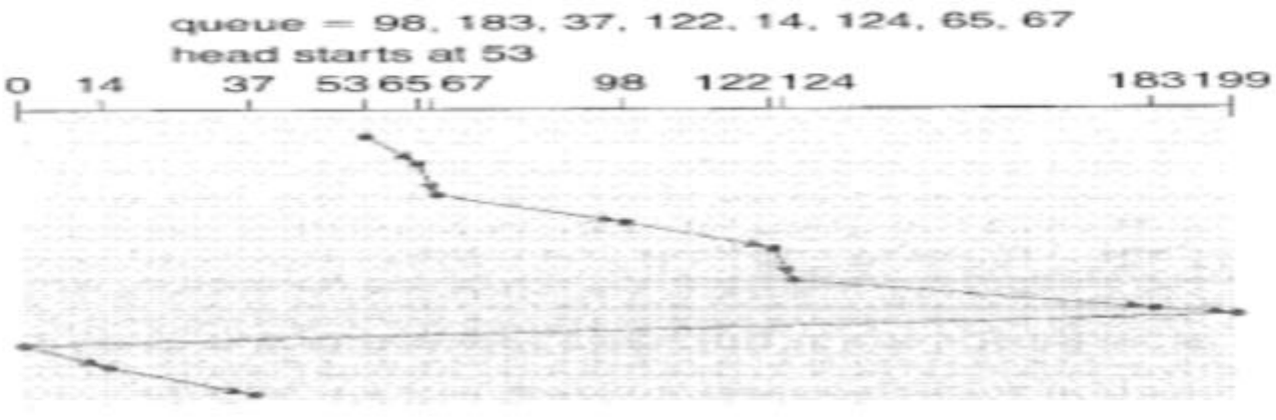


Fig: C-SCAN disk scheduling.

LOOK Scheduling

Both SCAN and C-SCAN move the disk arm across the full width of the disk. More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk. Versions of SCAN and C-SCAN that follow this pattern are called LOOK and C-LOOK scheduling, because they look for a request before continuing to move in a given direction (Fig).

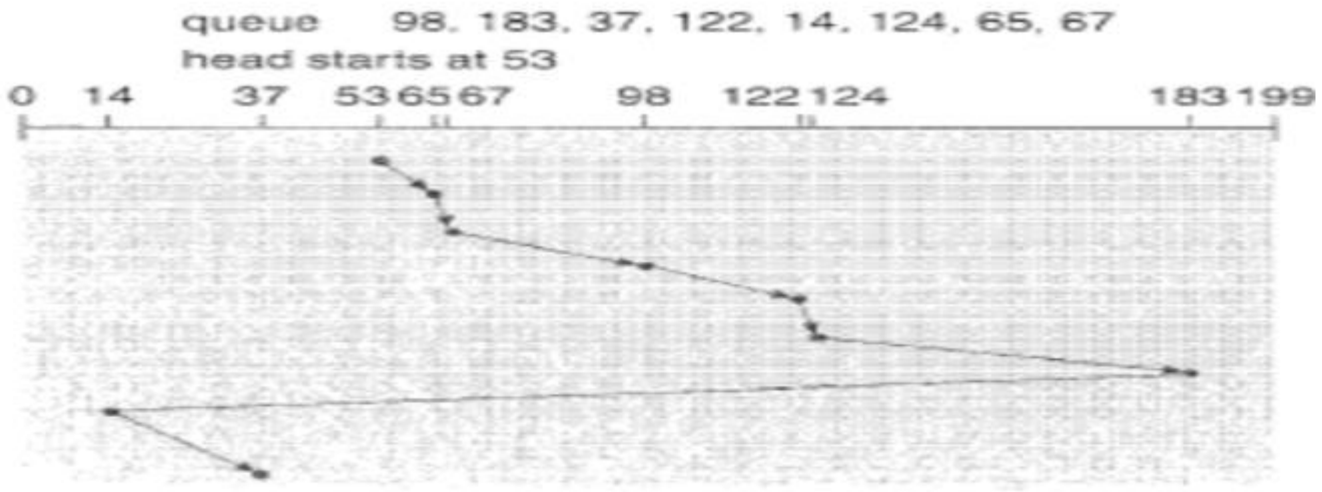


Fig: LOOK disk scheduling.

Introduction to Dockers:

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications.