MODULE 4. FILE SYSTEM AND SECONDARY STORAGE

File system provides the mechanism for storage of data and access to data & programs. The file system consists of two distinct parts:

- collection of files : each storing related data
- directory structure: organizes and provides information about all the files in the system.

4.1 FILE CONCEPT

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. The information in a file is defined by its creator. Many different types of information may be stored in a file like source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings etc. A file has a certain defined structure according to 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.

4.1.1 File Attributes

A file has certain attributes, which vary from one OS to another, but typically consist of the following:

- **Name**: The symbolic file name is the only information kept in human readable form.
- **Identifier**: This unique number identifies the file within the file system; it is the non-human-readable name for the file.
- **Type**: This information is needed for those systems that support different types.
- **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.

4.1.2 File Operations

File is an abstract data type. To define it properly, we need to define certain operations on it:

- **Creating a file:** This includes two steps: find the space in file system and make an entry 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. Using the name of the file, the system searches the directory to find the location of the file. 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.
- **Repositioning within a file:** The directory is searched for the appropriate entry, and the current-file-position is set to a given value. This file operation is also known as ***file seek***.
- **Deleting a file:** To delete a file, search the directory. Then, release all file space 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, the truncation allows all attributes to remain unchanged-except for file length. The file –length is reset to zero and its file space released.

Other common operations include ***appending*** new information to the end of an existing file and ***renaming*** an existing file.

Most of the file operations involve searching the directory for the entry associated with the named file. To avoid this constant searching, the OS keeps small table (known as ***open –file table***) containing information about all open files. When a file operation is requested, this table is checked. When the file is closed, the OS removes its entry in the open-file table.

Every file which is open has certain information associated with it:

- **File pointer:** Used to track the last read-write location. This pointer is unique to each process.
- **File open count:** When a file is closed, its entry position (the space) in the open-file table must be reused. Hence, we need to track the number of opens and closes using the file open count.
- **Disk location of the file:** Most file operations require the system to modify data within the file. So, location of the file on disk is essential.
- **Access rights:** Each process opens a file in an access mode (read, write, append etc). This information is by the OS to allow or deny subsequent I/O requests.

4.1.3 File Types

An OS can operate the file in a required manner only if it recognizes the type of that file. A file name is split into two parts – name and extension. The file extension normally indicates the type of a file. Table 4.1 indicates various file types.

Table 4.1 Common File Types

file type	usual extension	function
executable	exe, com, bin or none	read 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, rrf, doc	various word-processor formats
library	lib, a, so, dll, mpeg, mov, rm	libraries of routines for programmers
print or view	arc, zip, tar	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	binary file containing audio or A/V information

4.2 ACCESS METHODS

There are several methods to access the information stored in the file. Some techniques are discussed here.

4.2.1 Sequential Access

It is the simplest method of file access. Here, information in the file are accessed one record after the other in an order. It works on the logic: *read next, write next, reset.* It is shown in Figure 4.1. (Note that, in programming languages like C, the functions like *fseek(), rewind()* etc can be used).

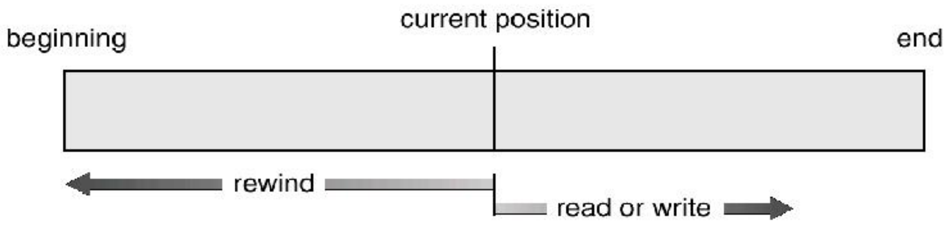


Figure 4.1 Sequential Access

4.2.2 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. A direct-access file allows arbitrary blocks to be read or written. 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.

4.2.3 Other Access Methods

There are several access methods based on direct access method. One of such methods uses an index for a file. Index contains pointers to 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. The working of indexed file is shown in Figure 4.2.

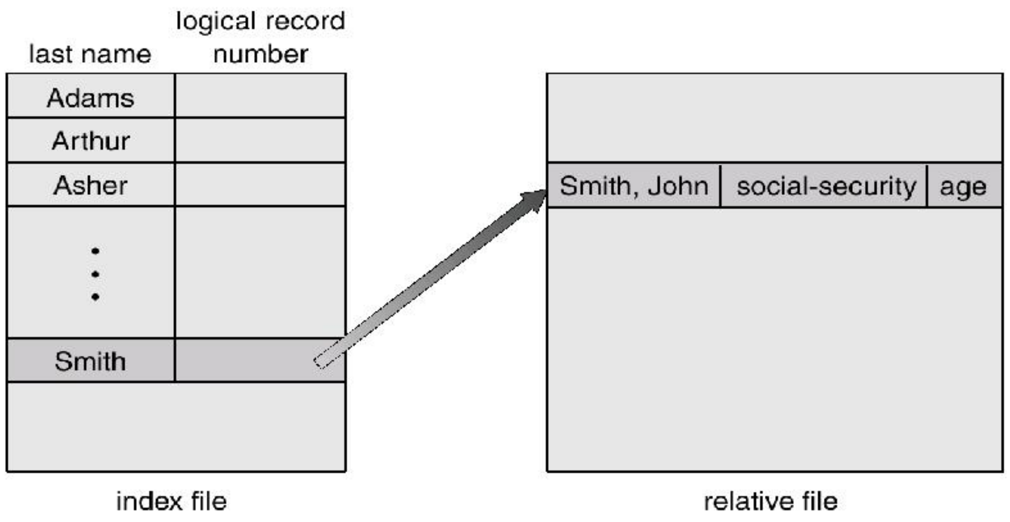


Figure 4.2 Example of index and relative files

4.3 DIRECTORY STRUCTURE

The file systems of computers can be very large. To manage all these data, we need to organize them. The disks are split into one or more partitions. Typically, each disk on a system contains at least one partition, which is a low-level structure in which files and directories reside. Sometimes, partitions are used to provide several separate areas within one disk, each treated as a separate storage device, whereas other systems allow partitions to be larger than a disk to group disks into one logical structure.

Each partition contains information about files within it. This information is kept 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 partition. Figure 4.3 shows file-system organization.

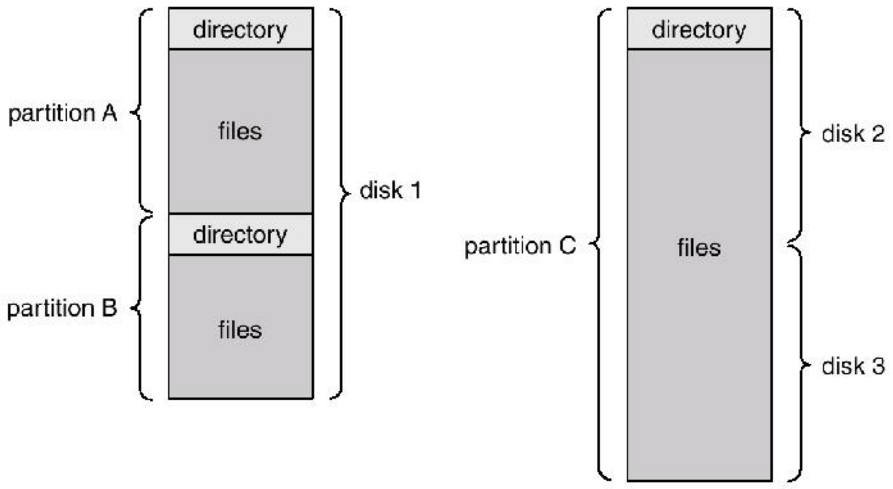


Figure 4.3 A typical file-system organization

Various 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.
- **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 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:** As the name of a file represents its contents to its users, the name must be changeable 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:** Files may need to be copied into another storage device like tape, pen-drive etc. and the disk space of that file released for reuse by another file.

4.3.1 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. It is shown in Figure 4.4. There are certain limitations for single-level directory:

- As all files will be in one directory, each file should have a unique name.
- If there are more users on the same system, each one of them should remember their file names – which is difficult.
- It is not possible to group the related files together.

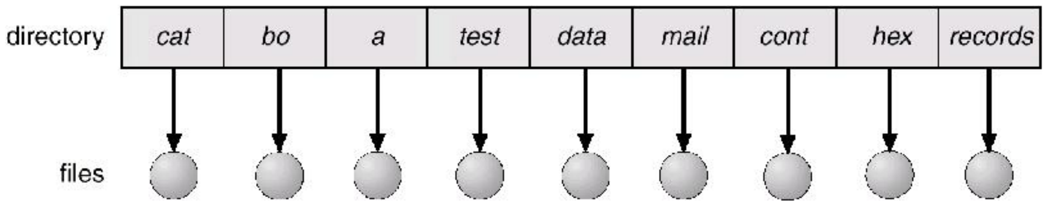


Figure 4.4 Single – level directory structure

4.3.2 Two – Level Directory

A solution to the problems faced with single-level directory is to create a separate directory for each user. In the two-level directory structure, each user has his/her own user file directory (UFD). Each UFD has a similar structure, but 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. Two – level directory is shown in Figure 4.5.

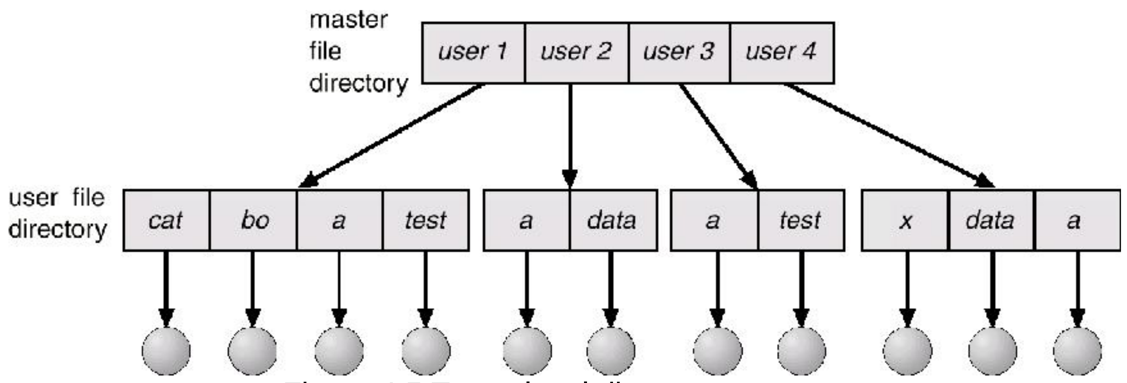


Figure 4.5 Two – level directory structure

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.

4.3.3 Tree – Structured Directories

The generalization of two-level directory is a tree structure. Here, users can have any number of directories and sub-directories and files inside each directory as shown in Figure 4.6. Tree – structure is a most common directory structure applied in almost all OS.

Here, every file has a path name. Path names can be of two types: ***absolute*** and ***relative.*** An absolute path name is a sequence of directory names starting from root to file name. A relative path name defines a path from current directory. For example, in the tree-structured file system of Figure 4.6, 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.*

4.3.4 Acyclic – Graph Directories

A tree structure prohibits the sharing of files or directories. An acyclic graph allows directories to have shared subdirectories and files (Figure 4.7). The same file or subdirectory may be in two different directories. This is useful when more than one programmers are working on a same project and need to share the files.

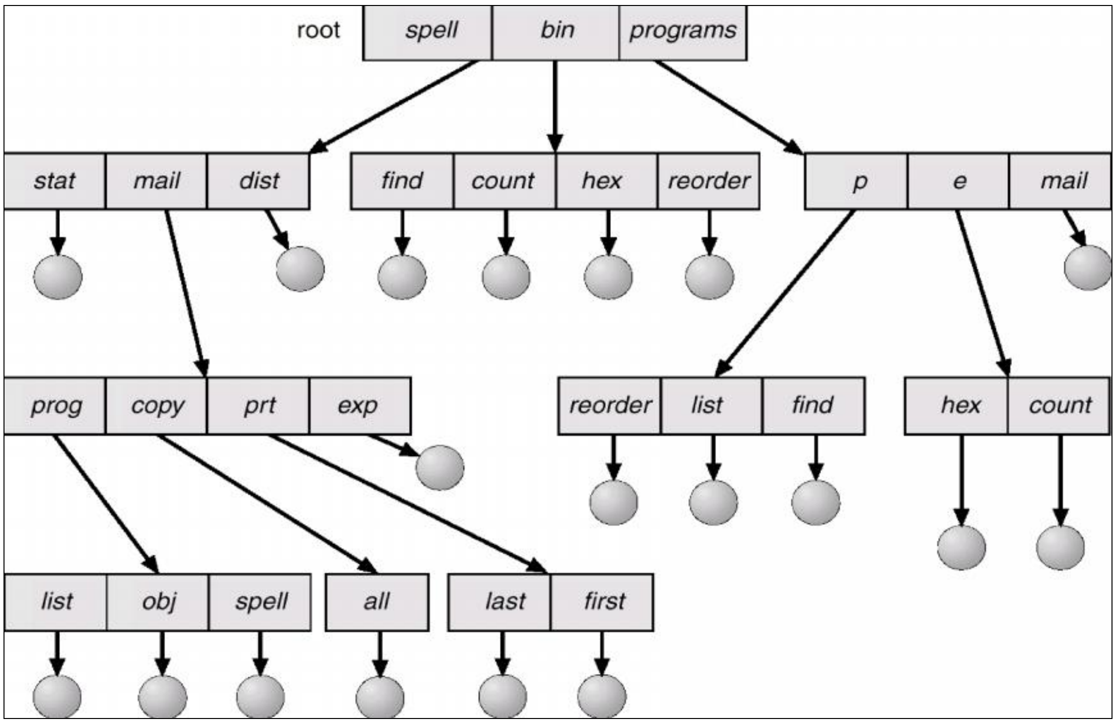


Figure 4.6 Tree – structure directory system

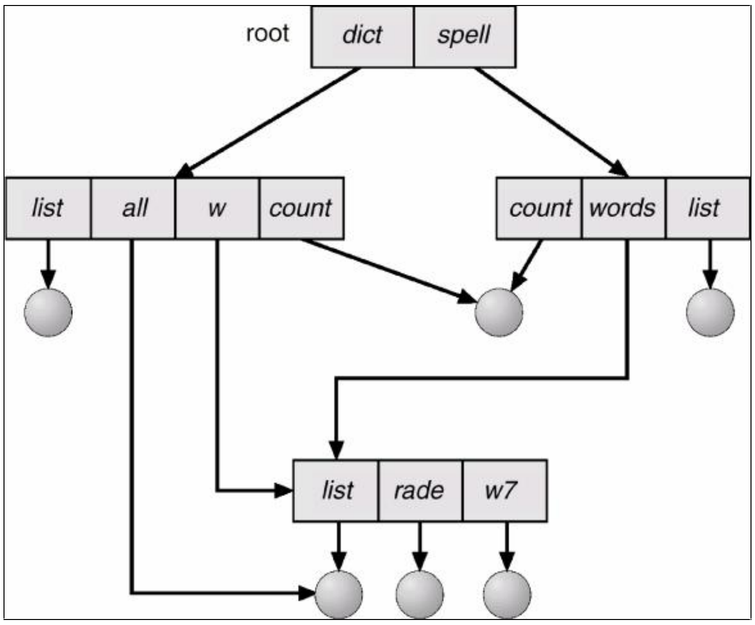


Figure 4.7 Acyclic graph directory structure

4.4 FILE – SYSTEM MOUNTING

Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system. More specifically, the directory structure can be built out of multiple partitions, which must be mounted to make them available within the file system name space.

The mount procedure is as follows: The OS is given the name of the device, and the location within the file structure at which to attach the file system (or mount point). Typically, a mount point is an empty directory at which the mounted file system will be attached. For instance, on a UNIX system, a file system containing user's home directories might be mounted as home; then, to access the directory structure within that file system, one could precede the directory names with */home*, as in */home/jane*. Mounting that file system under */users* would result in the path name */users/jane* to reach the same directory.

Next, the OS 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. Finally, the OS notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems as appropriate.

4.5 FILE SHARING

File sharing is very essential for users who want to collaborate and to reduce the effort required to achieve a computing goal. Therefore, user-oriented OS must accommodate the need to share files in spite of the difficulties.

4.5.1 Multiple Users

When an OS accommodates multiple users, the issues of file sharing, file naming, and file protection become important. The system either can allow a user to access the files of other users by default, or it may require that a user specifically grant access to the files.

To implement sharing and protection, the system must maintain more file and directory attributes than on a single-user system. Mainly, the attributes file/directory ***owner*** (or user) and ***group*** are included. The owner is the user who may change attributes, grant access, and has the most control over the file or directory. The group attribute of a file is used to define a subset of users who may share access to the file. Exactly which operations can be executed by group members and other users is definable by the file's owner.

4.5.2 Remote File Systems

Networking allows the sharing of resources spread within a campus or even around the world. One obvious resource to share is data, in the form of files. Through the evolution of network and file technology, file-sharing methods have changed. In the first implemented method, users manually transfer files between machines via programs like ***FTP***. The second major method is a distributed file system (DFS) in which remote directories are visible from the local machine. In some ways, the third method, 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. The World Wide Web uses anonymous file exchange almost exclusively. DFS involve a much tighter integration between the machine that is accessing the remote files and the machine providing the files.

4.6 PROTECTION

When information is kept in a computer system, we want to keep it safe from physical damage (reliability) and improper access (protection).

Reliability is generally provided by keeping duplicate copies of files. Many computers have systems programs that automatically copy disk files to tape at regular intervals. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software 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.

4.6.1 Types of Access

Protection of files can be achieved by restricting the users for accessing those files. Protection mechanisms provide controlled access by limiting the types of file access that can be made. 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.

4.6.2 Access Control

Usually, protection is achieved by controlling file-access depending on the identity of the user. Various users may need different types of access to a file or directory. Hence, an ***access-control list*** (ACL) is attached with every file and directory. It specifies user name and types of access allowed for each user. When a user requests access to a particular file, the OS 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.

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.

In most of the OS, the access is denoted by *r, w, x* indicating read, write and execute. A sequence of these characters is used to understand the type of access provided to each user.

4.7 FILE – SYSTEM STRUCTURE

Disks provide the bulk of secondary storage on which a file system is maintained. They have two characteristics which will help for convenient storage of multiple files:

- They can be rewritten in place; it is possible to read a block from the disk, to modify the block, and to write it back into the same place.
- They can access directly any given block of information on the disk.

To provide an efficient and convenient access to the disk, the OS imposes one or more file systems to allow the data to be stored, located, and retrieved easily. The file system is composed of many different levels. The Figure 4.8 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.

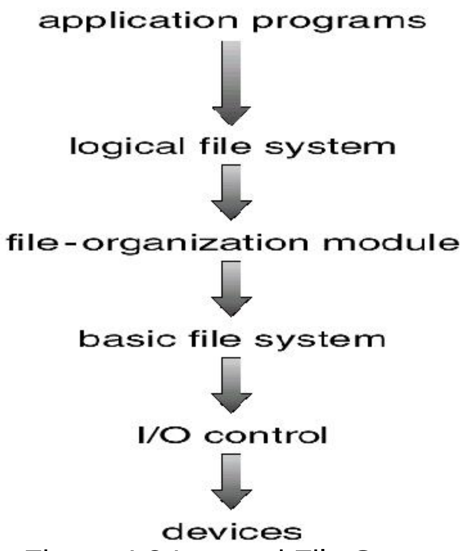


Figure 4.8 Layered File System

The lowest level 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. The file-organization module knows about files and their logical blocks, as well as physical blocks. The logical file system manages metadata information

4.8 FILE – SYSTEM IMPLEMENTATION

Several ***on-disk*** and ***in-memory*** structures are used to implement a file system. These vary depending on the OS and the file system, but some general principles are common for all OS.

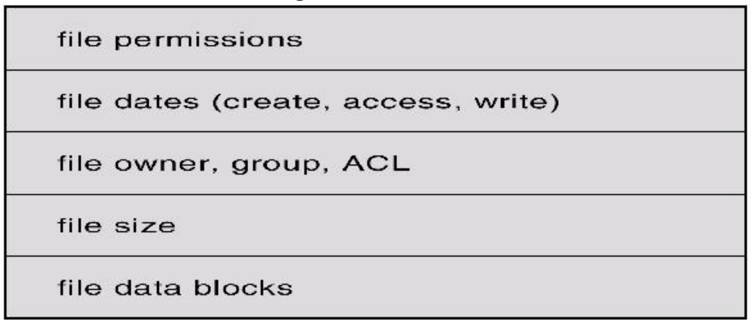
On-disk, the file system may contain information about how to boot an OS stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files. The on-disk structures include:

- A **boot control block** can contain information needed by the OS to boot from that partition. If the disk does not contain an OS, this block can be empty. It is typically the first block of a partition.
- A **partition control block** contains partition details, such as the number of blocks in the partition, size of the blocks, free-block count and free-block pointers, and free FCB count and FCB pointers.
- A directory structure is used to organize the files
- An FCB contains many of the file's details, including file permissions, ownership, size, and location of the data blocks.

The in-memory information is used for both file-system management and performance improvement via caching. The structures can include:

- An in-memory partition table containing information about each mounted partition.
- An in-memory directory structure that holds the directory information of recently accessed directories.
- 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.

A typical file control block is shown in Figure 4.9.



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

Figure 4.9 A typical File Control Block (FCB)

The operating structures of a file – system implementations for *file open* and *file read* are shown in Figure 4.10.

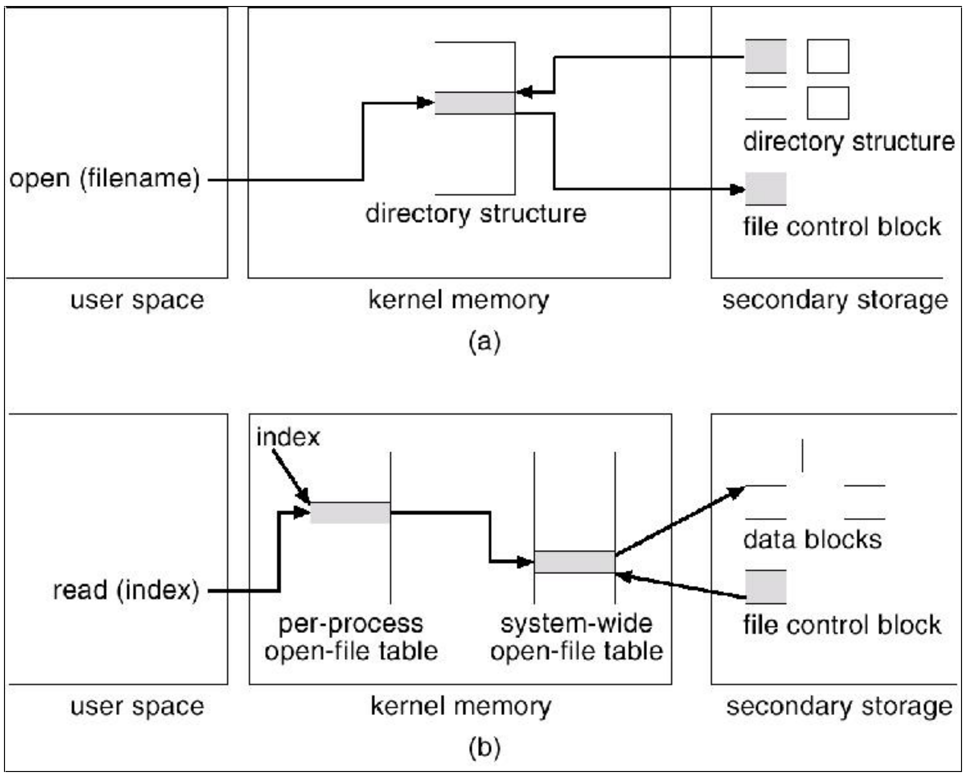


Figure 4.10 In-memory file-system structures: (a) File Open (b) File Read

4.9 DIRECTORY IMPLEMENTATION

The selection of directory-allocation and directory-management algorithms has a large effect on the efficiency, performance, and reliability of the file system.

4.9.1 Linear List

The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. A linear list of directory entries requires a linear search to find a particular entry. 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, then release the space allocated to it.

To reuse the directory entry, we can do one of several things.

- Mark the entry as unused (by assigning it a special name, such as an all-blank name, or with a used-unused bit in each entry)
- Attach it to a list of free directory entries
- 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 to delete a file.

The real disadvantage of a linear list of directory entries is the linear search to find a file. Directory information is used frequently, and users would notice a slow implementation of access to it.

4.9.2 Hash Table

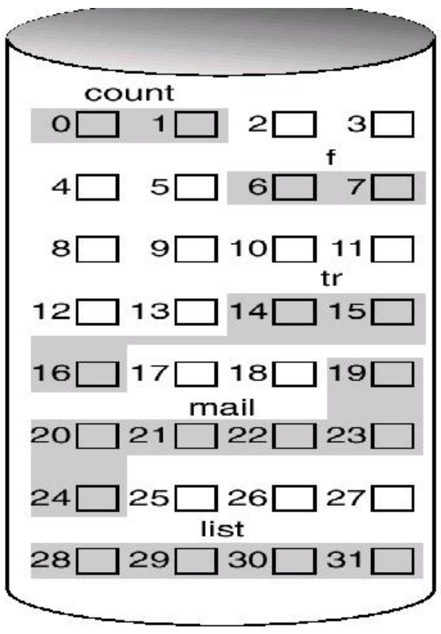
Another data structure that has been used for a file directory is a hash table. In this method, a linear list stores the directory entries, but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time. Insertion and deletion are also straightforward, although some provision must be made for collisions- situations where two file names hash to the same location. The major difficulties with a hash table are its generally fixed size (if collision is resolved using linear probing) and the dependence of the hash function on that size.

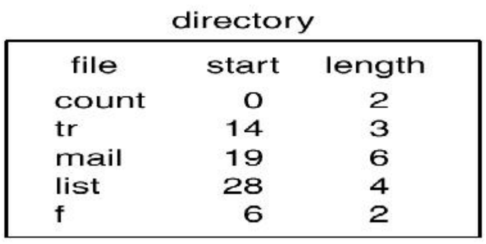
4.10 ALLOCATION METHODS

The direct-access nature of disks allows us flexibility in the implementation of files. In almost every case, many files will be 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: contiguous, linked, and indexed.

4.10.1 Contiguous Allocation

In contiguous allocation, files are assigned to contiguous areas of secondary storage. A user specifies in advance the size of the area needed to hold a file to be created. If the desired amount of contiguous space is not available, the file cannot be created. A contiguous allocation of disk space is shown in Figure 4.11.




directory

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

Figure 4.11 Contiguous allocation of disk space

One advantage of contiguous allocation is that all successive records of a file are normally physically adjacent to each other. This increases the accessing speed of records. It means that if records are scattered through the disk it is accessing will be slower. For sequential access the file system remembers the disk address of the last block and when necessary reads the next block. For direct access to block B of a file that starts at location L, we can immediately access block L+B. Thus contiguous allocation supports both sequential and direct accessing.

The disadvantage of contiguous allocation algorithm is, it suffers from external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces. Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.

4.10.2 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 as shown in Figure 4.12.

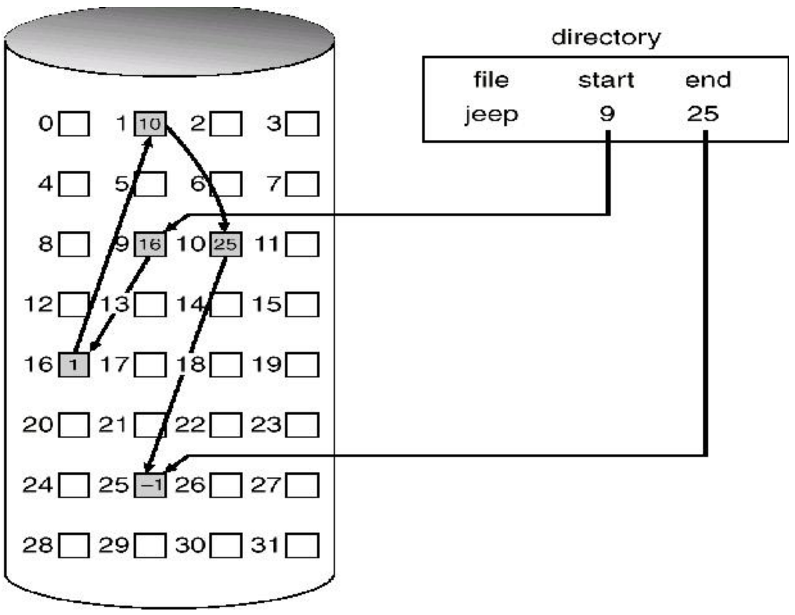


Figure 4.12 Linked Allocation of disk space

Linked allocation solves the problem of external fragmentation, which was present in contiguous allocation. But, still it has a disadvantage: Though it can be effectively used for sequential-access files, to find ith file, we need to start from the first location. That is, random-access is not possible.

4.10.3 Indexed Allocation

This method allows direct access of files and hence solves the problem faced in linked allocation. 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. The directory contains the address of the index block as shown in Figure 4.13.

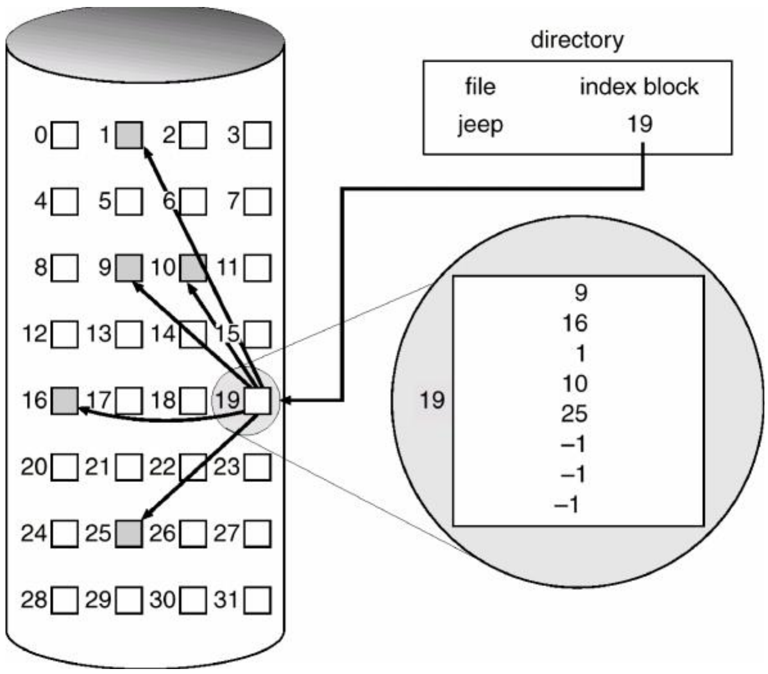


Figure 4.13 Indexed allocation of disk space

4.11 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. Free-space management is done using different techniques as explained hereunder.

4.11.1 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 of the blocks are allocated. The free-space bit map would be –

001111001111110001100000011100000

This technique is simple and efficient in finding the first free block, or *n* consecutive free blocks on the disk. But, bit vectors are inefficient unless the entire vector is kept in main memory. Keeping it in main memory is possible for smaller disks, such as on microcomputers, but not for larger ones.

4.11.2 Linked List

Another approach to free-space management is to link together all the free disk blocks, 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.

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 OS 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.

4.11.3 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 are actually free. The last block contains the addresses of another *n* free blocks, and so on. The importance of this implementation is that the addresses of a large number of free blocks can be found quickly, unlike in the standard linked-list approach.

4.11.4 Counting

Another approach is to 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.

4.12 DISK STRUCTURE

Disks provide the bulk of secondary storage for modern computer systems. Magnetic tape was used as an early secondary-storage medium, but the access time is much slower than for disks. 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 choose 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.

4.13 DISK SCHEDULING

One of the responsibilities of the OS is to use the hardware efficiently. This will help in achieving fast access time and 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 waiting 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. The request specifies several pieces of information:

- 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 bytes 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 on the queue of pending requests for that drive. For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed, the operating system chooses which pending request to service next.

4.13.1 FCFS Scheduling

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service.

Example: Consider a disk queue with requests for I/O to blocks on cylinders:
98, 183, 37, 122, 14, 124, 65, 67
in that order. There are 200 cylinders numbered from 0 to 199. The disk head is initially at the cylinder 53. Compute total head movements.

Solution: Starting with cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67 as shown in Figure 4.14.

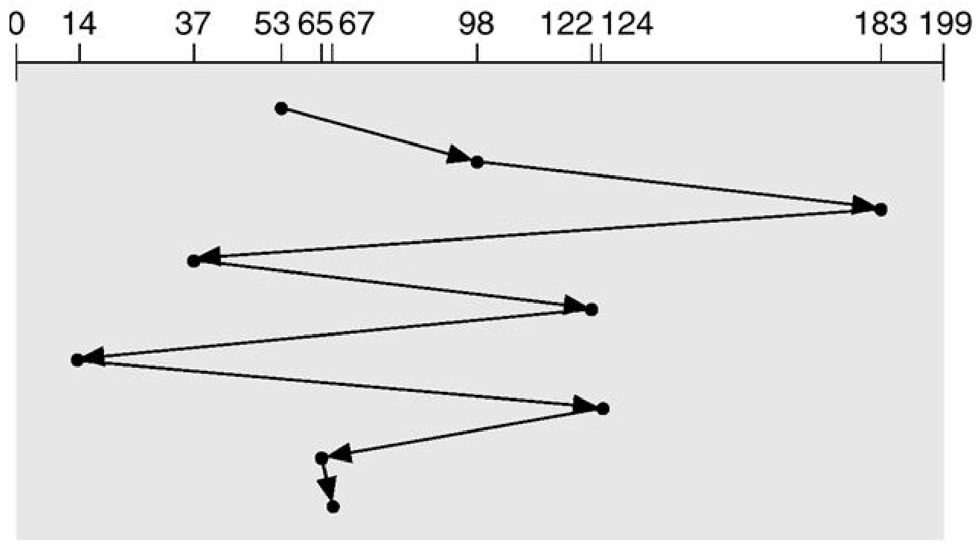


Figure 4.14 FCFS Scheduling

Head movement from 53 to 98 = 45
Head movement from 98 to 183 = 85
Head movement from 183 to 37 = 146
Head movement from 37 to 122 =85
Head movement from 122 to 14 =108
Head movement from 14 to 124 =110
Head movement from 124 to 65 =59
Head movement from 65 to 67 = 2
Total head movement = 640

4.13.2 SSTF Scheduling

The SSTF (shortest-seek-time-first) 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.

Example: Consider a disk queue with requests for I/O to blocks on cylinders:
98, 183, 37, 122, 14, 124, 65, 67
in that order. There are 200 cylinders numbered from 0 to 199. The disk head is initially at the cylinder 53. Compute total head movements.

Solution: 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 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183. It is shown in Figure 4.15.

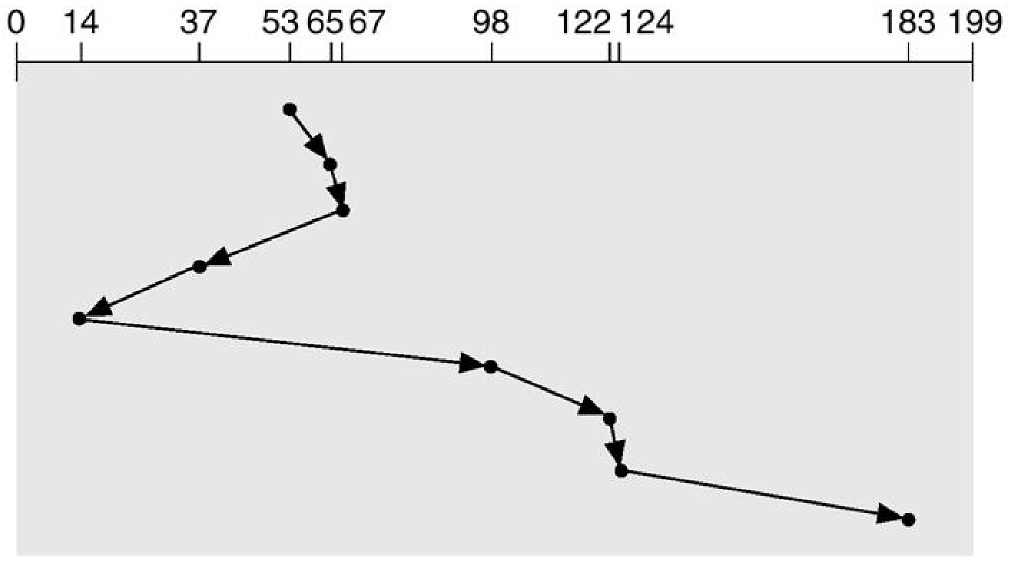


Figure 4.15 SSTF Scheduling

Head movement from 53 to 65 = 12
Head movement from 65 to 67 = 2

Head movement from 67 to 37 = 30
Head movement from 37 to 14 =23
Head movement from 14 to 98 =84
Head movement from 98 to 122 =24
Head movement from 122 to 124 =2
Head movement from 124 to 183 = 59
Total head movement = 236

4.13.3 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.

Example: Consider a disk queue with requests for I/O to blocks on cylinders:
98, 183, 37, 122, 14, 124, 65, 67
in that order. There are 200 cylinders numbered from 0 to 199. The disk head is initially at the cylinder 53. Compute total head movements.

Solution: Before applying SCAN algorithm, we need to know the current direction of head movement. Assume that 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 It is shown in Figure 4.16.

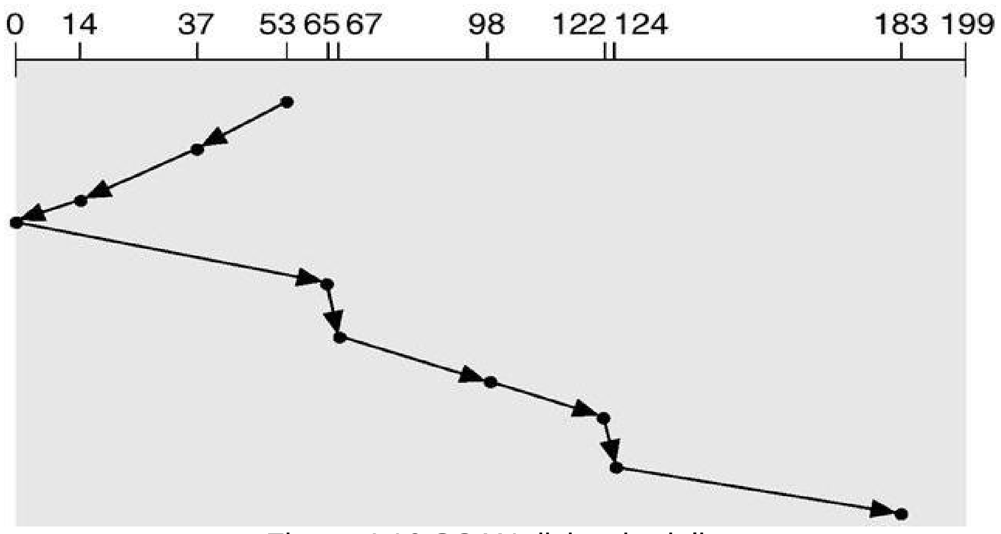


Figure 4.16 SCAN disk scheduling

Head movement from 53 to 37 = 16
Head movement from 37 to 14 = 23
Head movement from 14 to 0 = 14
Head movement from 0 to 65 =65
Head movement from 65 to 67 =2

Head movement from 67 to 98 =31
Head movement from 98 to 122 =24
Head movement from 122 to 124 = 2
Head movement from 124 to 183 = 59

Total head movement = 236

4.13.4 C- SCAN Scheduling

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time. 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 (Figure 8.4). The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

Example: Consider a disk queue with requests for I/O to blocks on cylinders:
98, 183, 37, 122, 14, 124, 65, 67
in that order. There are 200 cylinders numbered from 0 to 199. The disk head is initially at the cylinder 53. Compute total head movements.

Solution: Before applying C - SCAN algorithm, we need to know the current direction of head movement. Assume that disk arm is moving toward 199, the head will service 65, 67, 98, 122, 124, 183. Then it will move to 199 and the arm will reverse and move towards 0. While moving towards 0, it will not serve. But, after reaching 0, it will reverse again and then serve 14 and 37. It is shown in Figure 4.17.

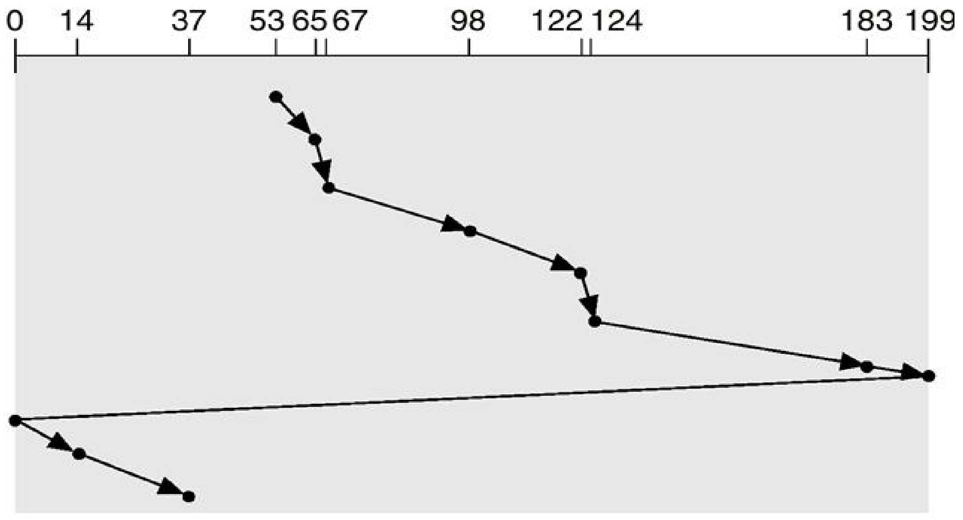


Figure 4.17 C-SCAM Disk Scheduling

Head movement from 53 to 65 = 12
Head movement from 65 to 67 = 2
Head movement from 67 to 98 = 31
Head movement from 98 to 122 =24

Head movement from 122 to 124 =2
Head movement from 124 to 183 =59
Head movement from 183 to 199 =16
Head movement from 199 to 0 = 199
Head movement from 0 to 14 = 14
Head movement from 14 to 37 = 23

Total head movement = 382

4.13.5 Look Scheduling

As we described them, both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice, neither algorithm is implemented this way. 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. These versions of SCAN and C-SCAN are called LOOK and C-LOOK scheduling, because they look for a request before continuing to move in a given direction.

Example: Consider a disk queue with requests for I/O to blocks on cylinders:
98, 183, 37, 122, 14, 124, 65, 67
in that order. There are 200 cylinders numbered from 0 to 199. The disk head is initially at the cylinder 53. Compute total head movements.
Solution: Assume that disk arm is moving toward 199, the head will service 65, 67, 98, 122, 124, 183. Then the arm will reverse and move towards 14. Then it will serve 37. It is shown in Figure 4.18.

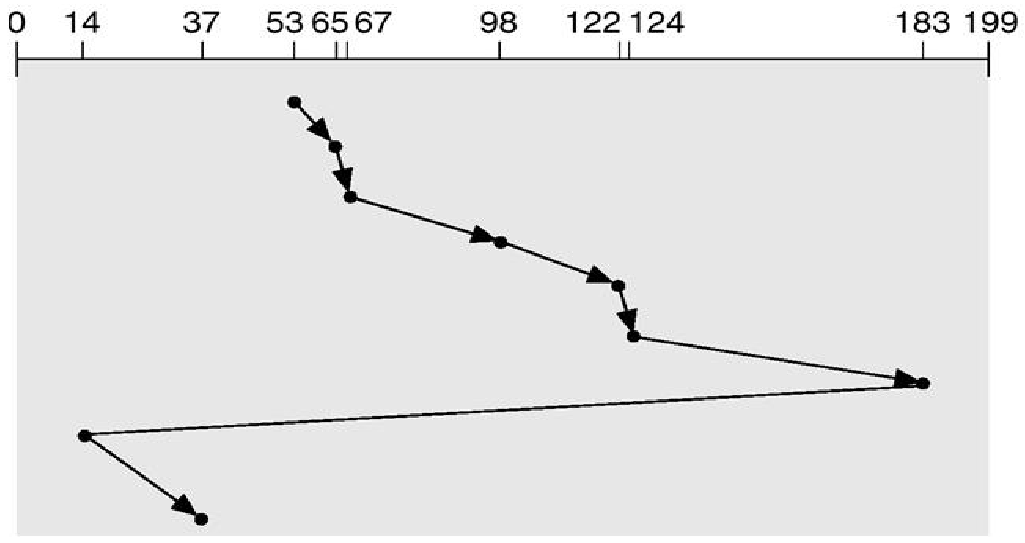


Figure 4.18 Look Disk Scheduling

Head movement from 53 to 65 = 12
Head movement from 65 to 67 = 2
Head movement from 67 to 98 = 31
Head movement from 98 to 122 =24

Head movement from 122 to 124 =2
Head movement from 124 to 183 =59
Head movement from 183 to 14 = 169
Head movement from 14 to 37 = 23

Total head movement = 322

4.14 DISK MANAGEMENT

The OS is responsible for several other aspects of disk management as well.

4.14.1 Disk Formatting

A new magnetic disk is a blank slate: It is just platters of a magnetic recording material. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting (or physical formatting). Low-level formatting fills the disk with a special data structure for each sector.

To use a disk to hold files, the OS needs to record its own data structures on the disk. It does so in two steps. The first step is to **partition** the disk into one or more groups of cylinders. The OS treats each partition as though it were a separate disk. After partitioning, the second step is **logical formatting** (or creation of a file system). In this step, the OS stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space and an initial empty directory.

4.14.2 Boot Blocks

For a computer to start running, it needs to have an initial program to run. This bootstrap program initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the OS. To do its job, the bootstrap program finds the OS kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the operating-system execution.

4.14.3 Bad Blocks

Because disks have moving parts and small tolerances, they are prone to failure. Sometimes the failure is complete, and the disk needs to be replaced, and its contents restored from backup media to the new disk. More frequently, one or more sectors become defective. Most disks even come from the factory with bad blocks. Depending on the disk and controller in use, these blocks are handled in a variety of ways:

- On simple disks with IDE, bad blocks are handled manually (like formatting the disk)
- The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level format at the factory, and is updated over the life of the disk.

****