MODULE 4

File system - Concept of file and directory - Various file operations - File organization concepts – sequential and indexed.

Different directory structures – single level, two-level, and tree structured directories.

Different allocation methods – contiguous, linked and indexed allocations.

Various disk scheduling algorithms-FCFS, SSTF, Scan, C-Scan, Look & C-Look.

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.

Files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary.
In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.

The information in a file is defined by its creator.
Many different types of information may be stored in a file—source or executable programs, numeric or text data, photos, music, video, and so on.
A file has a certain defined structure, which depends on its type.

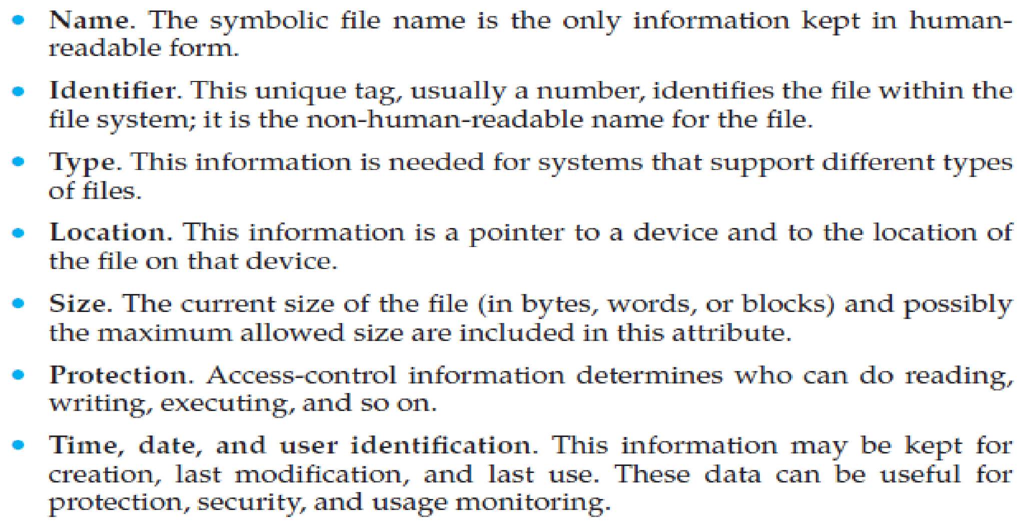
A *text file* is a sequence of characters organized into lines.

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

An *executable file* is a series of code sections that the loader can bring into memory and execute.

File Attributes: -

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.

File Operations: -

Creating a file.

Two steps are necessary to create a file.

First, space in the file system must be found for the file.

Second, an entry for the new file must be made in the directory.

Writing a file.

To write a file, we make a system call specifying both the *name* of the file and the *information* to be written to the file.

Given the name of the file, the system searches the directory to find the file's location.

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

Reading a file.

To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put.

Again, the directory is searched for the associated entry, and the system needs to keep a *read pointer* to the location in the file where the next read is to take place.

Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process *current file- position pointer.*

Repositioning within a file.

The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value.

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.

File Types: -

When we design a file system, always consider whether the operating system should recognize and support file types.

A common technique for implementing file types is to include the type as part of the file name.

The name is split into two parts—

a *name* and an *extension*, usually separated by a period

Most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension made up of additional characters.

Examples include :

resume.docx, server.c, and *ReaderThread.cpp*.

The system uses the extension to indicate the *type* of the file and the type of *operations* that can be done on that file.

Only a file with a *.com*, *.exe*, or *.sh* extension can be executed.

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

File 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

Direct Access

Sequential Access: -

The simplest access method is *sequential access*.

Information in the file is processed in order, one record after the other.

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).

Direct Access: -

Another method is *direct access* (or *relative access*).

Here, a file is made up of *fixed-length logical records* that allow programs to read and write records rapidly in no particular order.

The direct-access method is based on a *disk model* of a file, since disks allow random access to any file block.

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

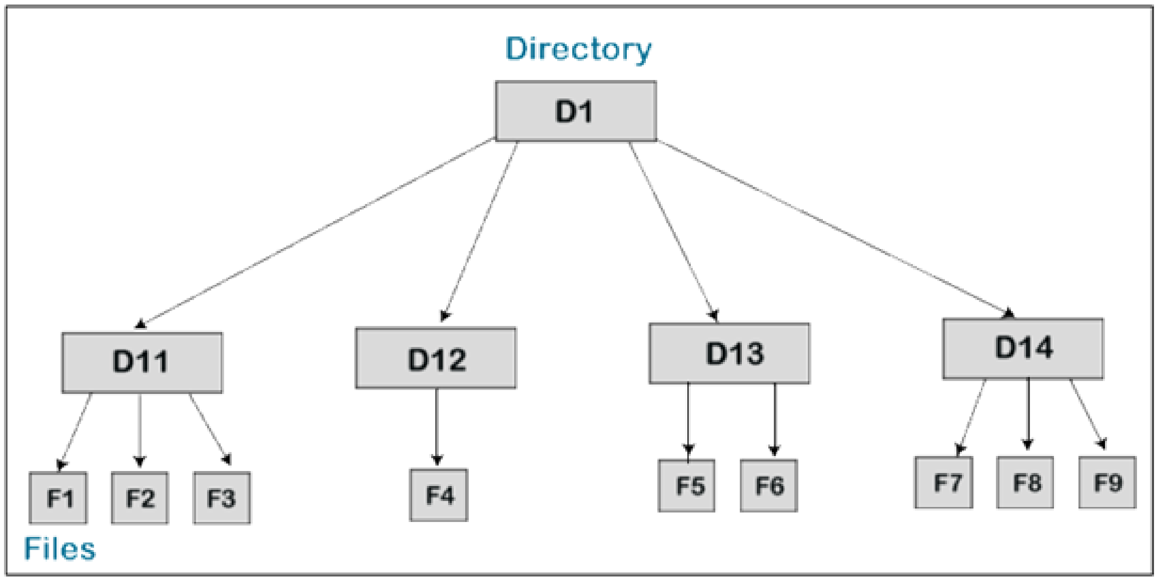
Direct-access files are of great use for immediate access to large amounts of information. *Databases* are often of this type.

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

Thus, we have *read(n),* where n is the block number, rather than *read next(),* and *write(n)* rather than *write next().*

Directory Structure: -

A Directory is the collection of the correlated files on the disk. In simple words, a directory is like a container which contains file and folder. In a directory, we can store the complete file attributes or some attributes of the file. A directory can be comprised of various files. With the help of the directory, we can maintain the information related to the files.



There are various types of information which are stored in a directory:

Name

Type

Location

Size

Position

Protection

Usage

Mounting

1. **Name: -** Name is the name of the directory, which is visible to the user.
2. **Type: -** Type of a directory means what type of directory is present such as single-level directory, two-level directory, tree-structured directory, and Acyclic graph directory.
3. **Location: -** Location is the location of the device where the header of a file is located.
4. **Size: -** Size means number of words/blocks/bytes in the file.
5. **Position: -** Position means the position of the next-read pointer and the next-write pointer.
6. **Protection: -** Protection means access control on the read/write/delete/execute.
7. **Usage: -** Usage means the time of creation, modification, and access, etc.
8. **Mounting: -** Mounting means if the root of a file system is grafted into the existing tree of other file systems.

Operations on Directory: -

The various types of operations on the directory are:

1. **Creating**
2. **Deleting**
3. **Searching**
4. **List a directory**
5. **Renaming**

1. Creating: - In this operation, a directory is created. The name of the directory should be unique.
2. Deleting: - If there is a file that we don't need, then we can delete that file from the directory. We can also remove the whole directory if the directory is not required. An

empty directory can also be deleted. An empty directory is a directory that only consists of dot and dot-dot.

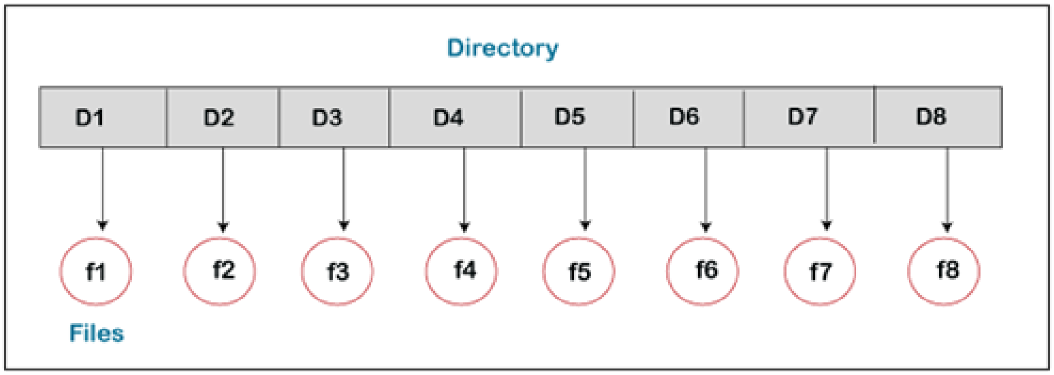
3. Searching: - Searching operation means, for a specific file or another directory, we can search a directory.
4. List a directory: - In this operation, we can retrieve all the files list in the directory. And we can also retrieve the content of the directory entry for every file present in the list.

Types of Directory Structure

There are various types of directory structure:

1. Single-Level Directory
2. Two-Level Directory
3. Tree-Structured Directory

Single-Level Directory: - Single-Level Directory is the easiest directory structure. There is only one directory in a single-level directory, and that directory is called a root directory. In a single-level directory, all the files are present in one directory that makes it easy to understand. In this, under the root directory, the user cannot create the subdirectories.



Advantages of Single-Level Directory

The advantages of the single-level directory are:

1. The implementation of a single-level directory is so easy.
2. In a single-level directory, if all the files have a small size, then due to this, the searching of the files will be easy.

3. In a single-Level directory, the operations such as searching, creation, deletion, and updating can be performed.

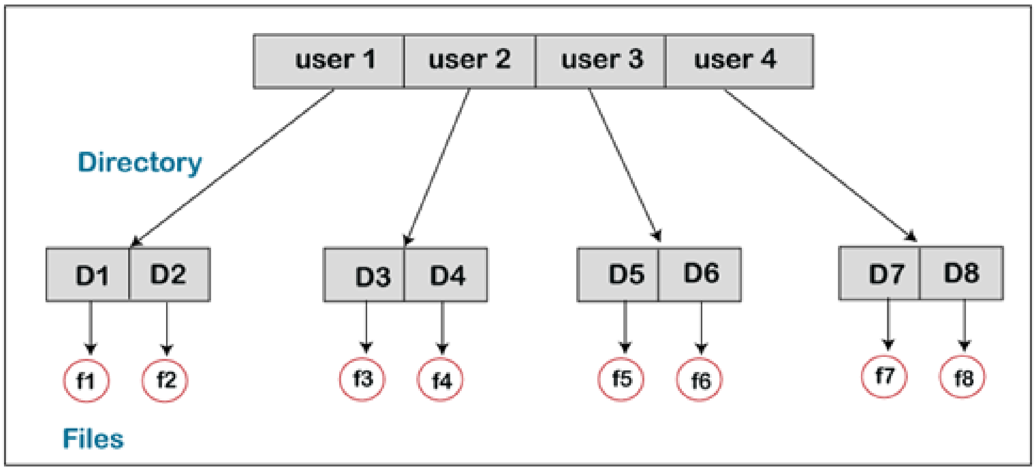
Disadvantages of Single-Level Directory

The disadvantages of Single-Level Directory are:

1. If the size of the directory is large in Single-Level Directory, then the searching will be tough.
2. In a single-level directory, we cannot group the similar type of files.
3. Another disadvantage of a single-level directory is that there is a possibility of collision because the two files cannot have the same name.
4. The task of choosing the unique file name is a little bit complex.

Two-Level Directory: -

Two-Level Directory is another type of directory structure. In this, it is possible to create an individual directory for each of the users. There is one master node in the two-level directory that include an individual directory for every user. At the second level of the directory, there is a different directory present for each of the users. Without permission, no user can enter into the other user's directory.



Advantages of Two-Level Directory

The advantages of the two-level directory are:

1. In the two-level directory, various users have the same file name and also directory name.
2. Because of using the user-grouping and pathname, searching of files are quite easy.

Disadvantages of Two-Level Directory

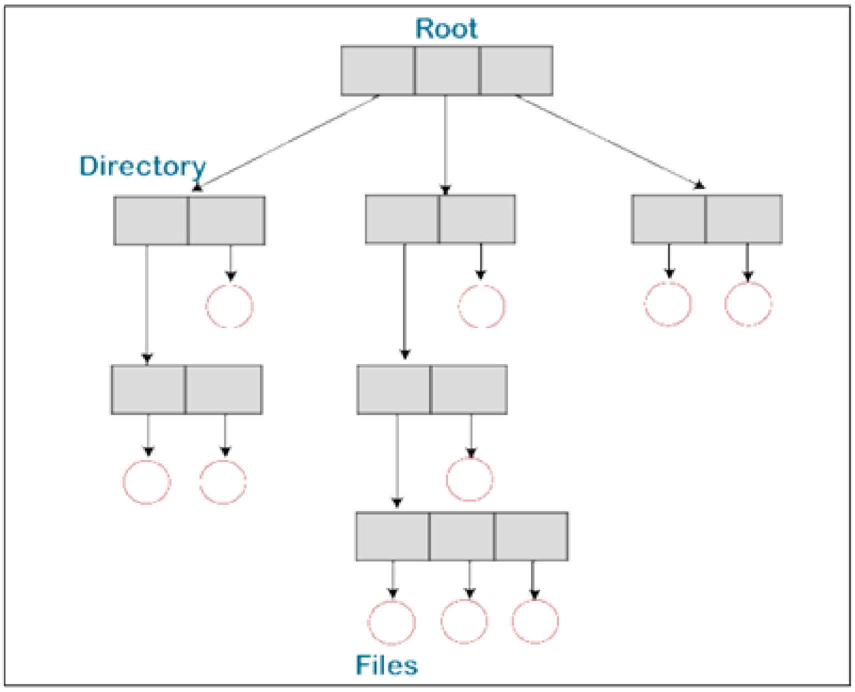
The disadvantages of the two-level directory are:

1. In a two-level directory, one user cannot share the file with another user.
2. Another disadvantage with the two-level directory is it is not scalable.

Tree-Structured Directory: -

A Tree-structured directory is another type of directory structure in which the directory entry may be a sub-directory or a file. The tree-structured directory reduces the limitations of the two-level directory. We can group the same type of files into one directory.

In a tree-structured directory, there is an own directory of each user, and any user is not allowed to enter into the directory of another user. Although the user can read the data of root, the user cannot modify or write it. The system administrator only has full access to the root directory. In this, searching is quite effective and we use the current working concept. We can access the file by using two kinds of paths, either absolute or relative.



Advantages of tree-structured directory

The advantages of the tree-structured directory are:

1. The tree-structured directory is very scalable.
2. In the tree-structures directory, the chances of collision are less.
3. In the tree-structure directory, the searching is quite easy because, in this, we can use both types of paths, which are the absolute path and relative path.

Disadvantages of Tree-Structure Directory

The disadvantages of tree-structure directory are:

1. In the tree-structure directory, the files cannot be shared.
2. Tree-structure directory is not efficient because, in this, if we want to access a file, then it may go under multiple directories.
3. Another disadvantage of the tree-structure directory is that each file does not fit into the hierarchal model. We have to save the files into various directories.

Allocation Methods: -

The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

The main idea behind these methods is to provide:

- Efficient disk space utilization.
- Fast access to the file blocks.

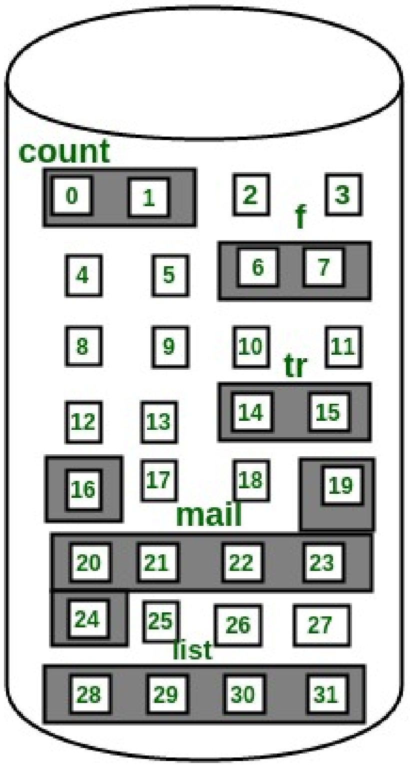
All the three methods have their own advantages and disadvantages as discussed below:

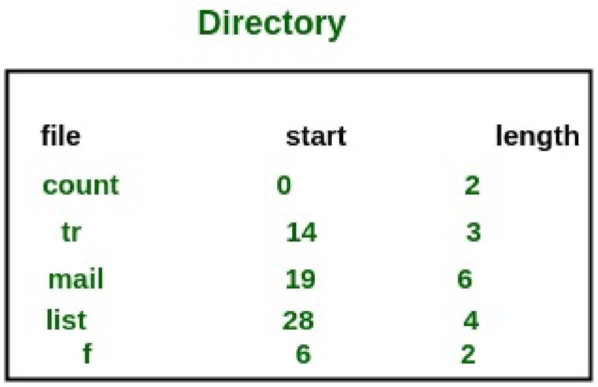
1. Contiguous Allocation

In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: *b, b+1, b+2,……b+n-1*. This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file. The directory entry for a file with contiguous allocation contains

- Address of starting block
- Length of the allocated portion.

The *file 'mail'* in the following figure starts from the block 19 with length = 6 blocks. Therefore, it occupies *19, 20, 21, 22, 23, 24* blocks.





Directory

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

Advantages:

- Both the Sequential and Direct Accesses are supported by this. For direct access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k).
- This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

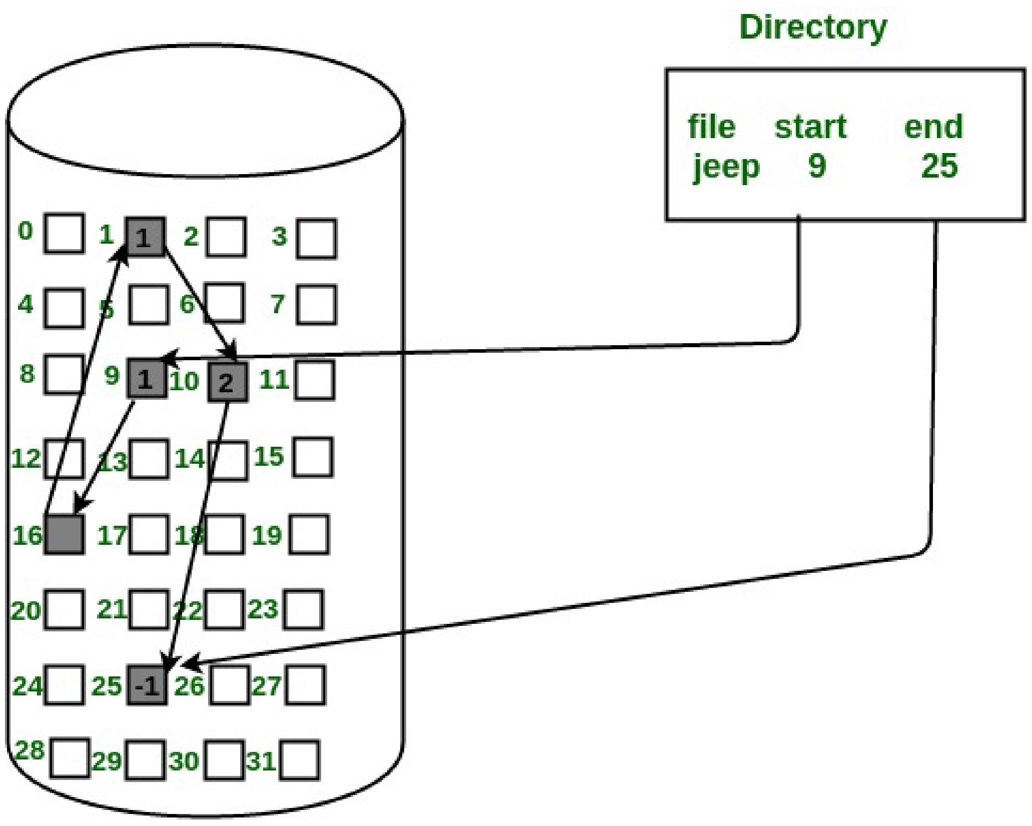
Disadvantages:

- This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.
- Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

2. Linked List Allocation

In this scheme, each file is a linked list of disk blocks which **need not be** contiguous. The disk blocks can be scattered anywhere on the disk.
The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

The file 'jeep' in following image shows how the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.



Advantages:

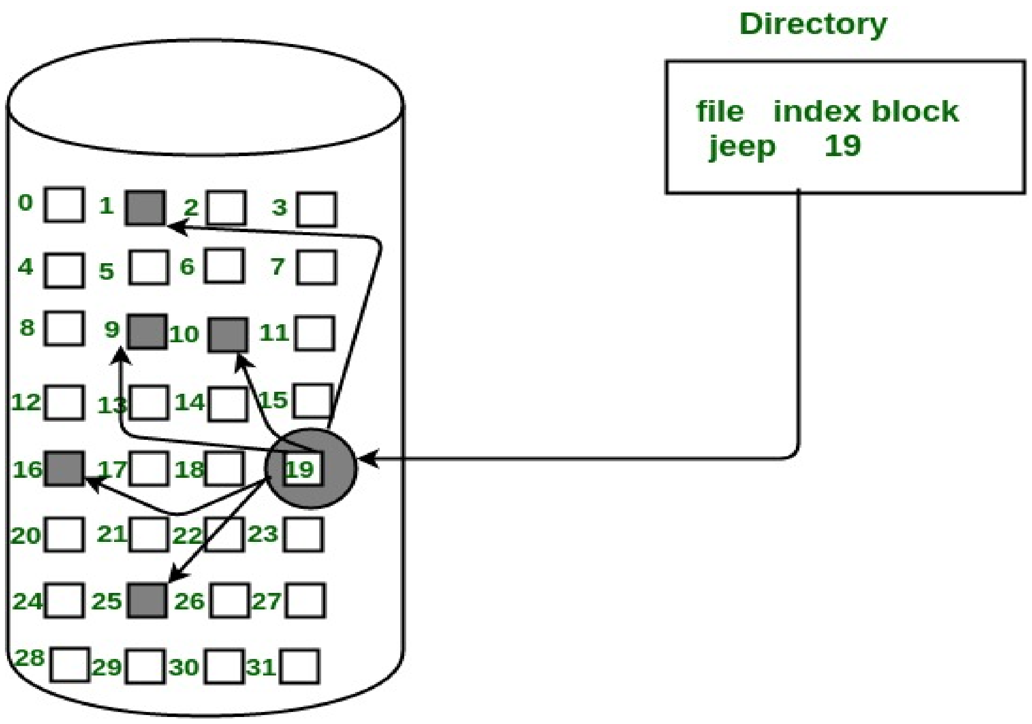
- This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.
- This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.

Disadvantages:

- Because the file blocks are distributed randomly on the disk, a large number of seeks are needed to access every block individually. This makes linked allocation slower.
- It does not support random or direct access. We can not directly access the blocks of a file. A block k of a file can be accessed by traversing k blocks sequentially (sequential access) from the starting block of the file via block pointers.
- Pointers required in the linked allocation incur some extra overhead.

3. Indexed Allocation

In this scheme, a special block known as the **Index block** contains the pointers to all the blocks occupied by a file. Each file has its own index block. The ith entry in the index block contains the disk address of the ith file block. The directory entry contains the address of the index block as shown in the image:



Advantages:

- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

Disadvantages:

- The pointer overhead for indexed allocation is greater than linked allocation.
- For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

Disk Scheduling Algorithms: -

For magnetic disks, 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.

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. For this purpose the *disk-scheduling algorithms can be used.*

- **FCFS Scheduling: -**
  The simplest form of disk scheduling is the *first-come, first-served (FCFS)* algorithm.
  This algorithm is intrinsically fair, but it generally *does not provide the fastest service.*
  Consider, for example, a disk queue with requests for I/O to blocks on cylinders

98, 183, 37, 122, 14, 124, 65, 67,

in that order.

The disk head is initially at cylinder 53.

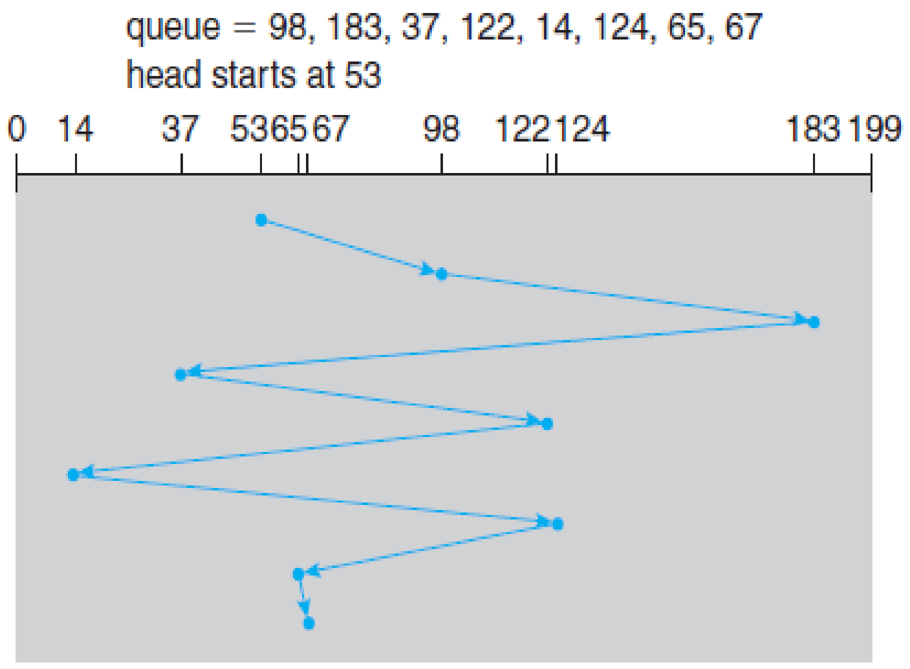


Figure 10.4 FCFS disk scheduling.

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*.

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 for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

- **SSTF Scheduling**

  Service all the requests *close to the current head position* before moving the head far away to service other requests.

  This assumption is the basis for the *shortest-seek-time-first (SSTF)* algorithm.

The SSTF algorithm selects the request with the *least seek time from the current head position*.

In other words, SSTF chooses the pending request closest to the current head positioned.

Consider, for example, a disk queue with requests for I/O to blocks on cylinders

98, 183, 37, 122, 14, 124, 65, 67,

in that order.

The disk head is initially at cylinder 53.

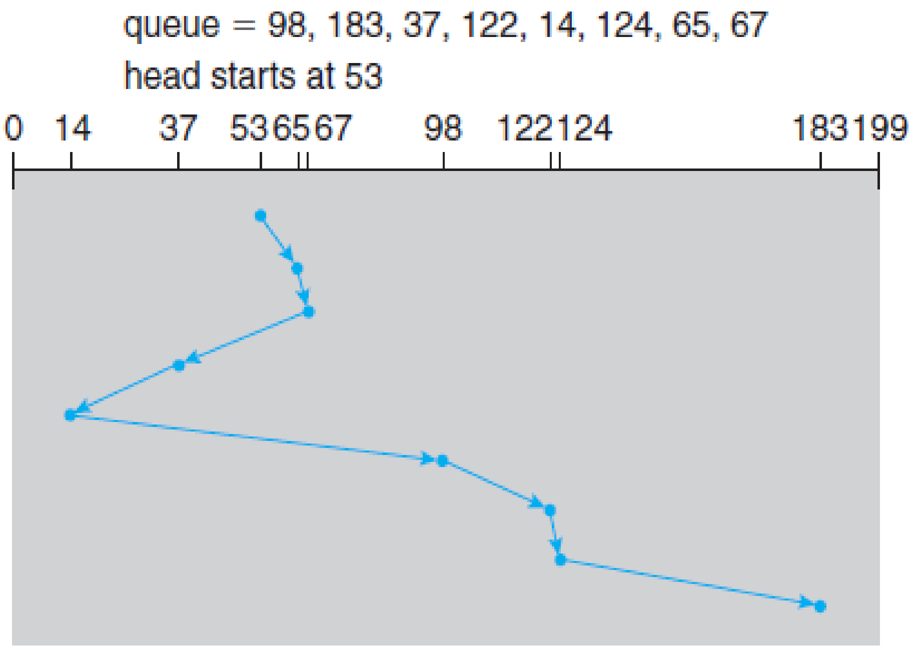


Figure 10.5 SSTF disk scheduling.

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 (Figure 10.5).

This scheduling method results in a total head movement of only *236 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.

  The SCAN algorithm is sometimes called the *elevator algorithm*

  Consider, for example, a disk queue with requests for I/O to blocks on cylinders

  98, 183, 37, 122, 14, 124, 65, 67,

  in that order

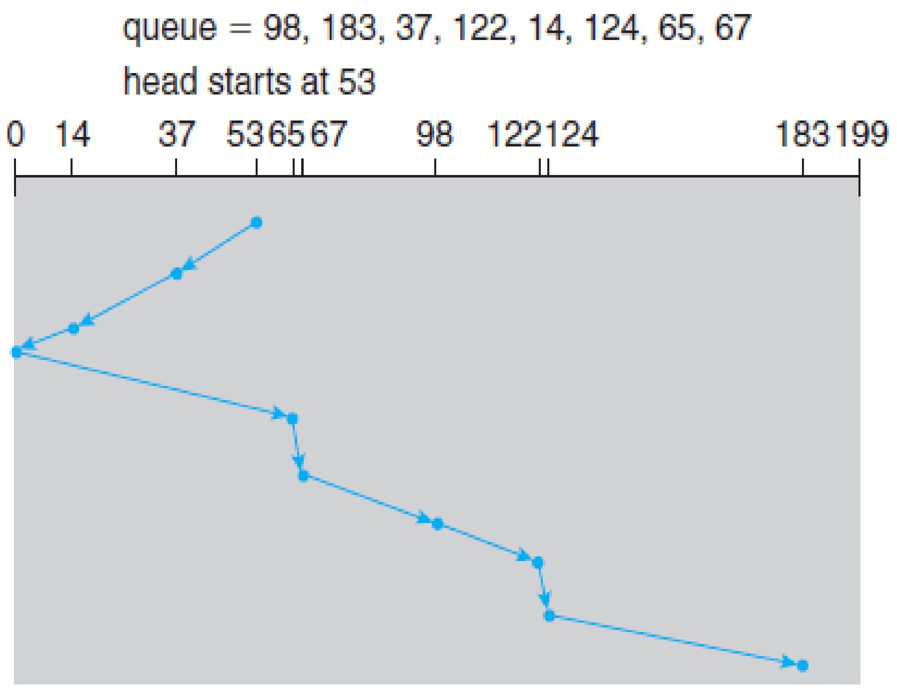


Figure 10.6 SCAN disk scheduling.

Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67. We need to know the *direction of head movement* in addition to the *head's current position.*

Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next 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 (Figure 10.6).

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.

Assuming a *uniform distribution of requests for cylinders*, consider the density of requests when the head reaches one end and reverses direction. At this point, relatively few requests are immediately in front of the head, since these cylinders have recently been serviced. The heaviest density of requests is at the other end of the disk. These requests have also *waited the longest.*

- **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*

  Consider, for example, a disk queue with requests for I/O to blocks on cylinders

  98, 183, 37, 122, 14, 124, 65, 67,

  in that order.
  The disk head is initially at cylinder 53.

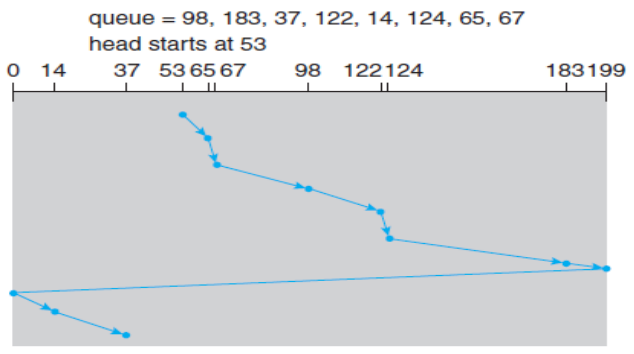


Figure 10.7 C-SCAN disk scheduling.

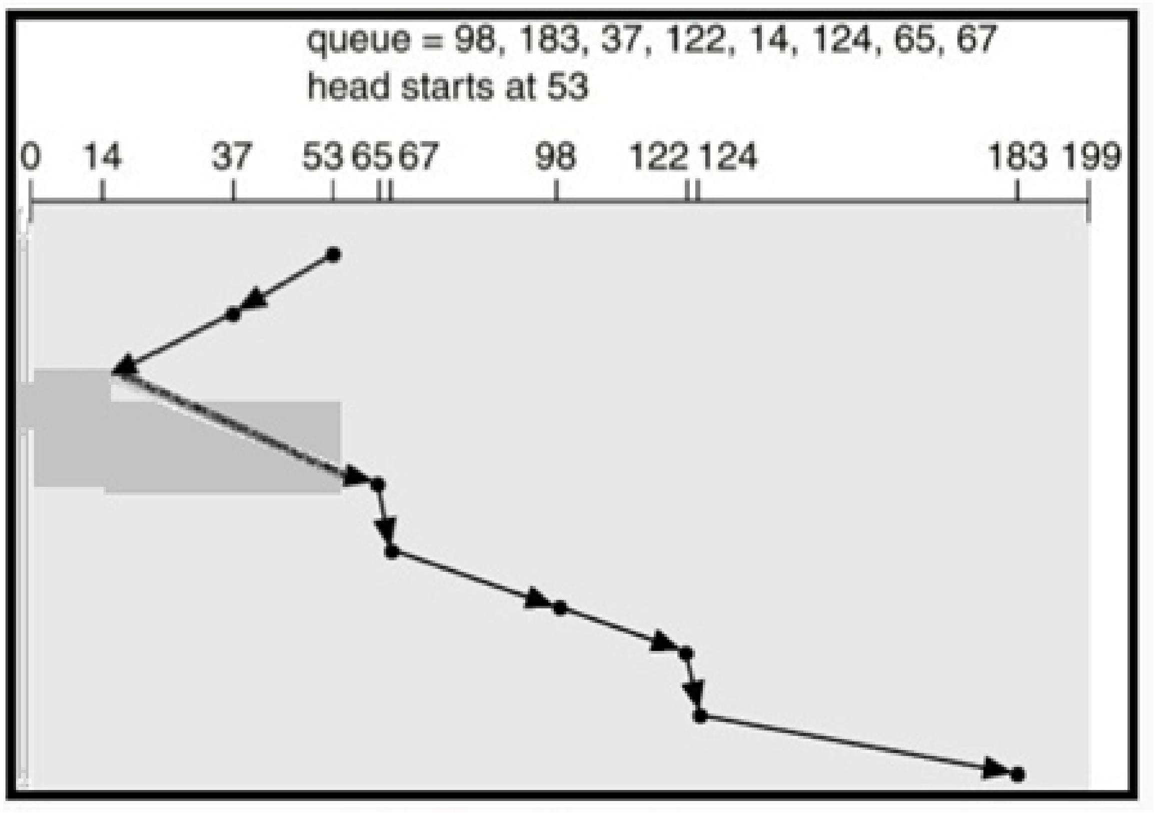
- **LOOK Scheduling: -**

  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.

  Consider, for example, a disk queue with requests for I/O to blocks on cylinders

  98, 183, 37, 122, 14, 124, 65, 67,

  in that order.
  The disk head is initially at cylinder 53.



- **C-LOOK (Circular LOOK) Scheduling: -**

  C-LOOK is an enhanced version of both SCAN as well as LOOK disk scheduling algorithms.

  This algorithm also uses the idea of wrapping the tracks as a circular cylinder as C-SCAN algorithm but the seek time is better than C-SCAN algorithm.

  In C-look scheduling, the disk arm moves and service each request till the head reaches its highest request, and after that, the disk arm jumps to the lowest cylinder *without servicing any request*, and the disk arm moves further and service those requests which are remaining.

  Consider, for example, a disk queue with requests for I/O to blocks on cylinders

  98, 183, 37, 122, 14, 124, 65, 67, in that order.

  The disk head is initially at cylinder 53.

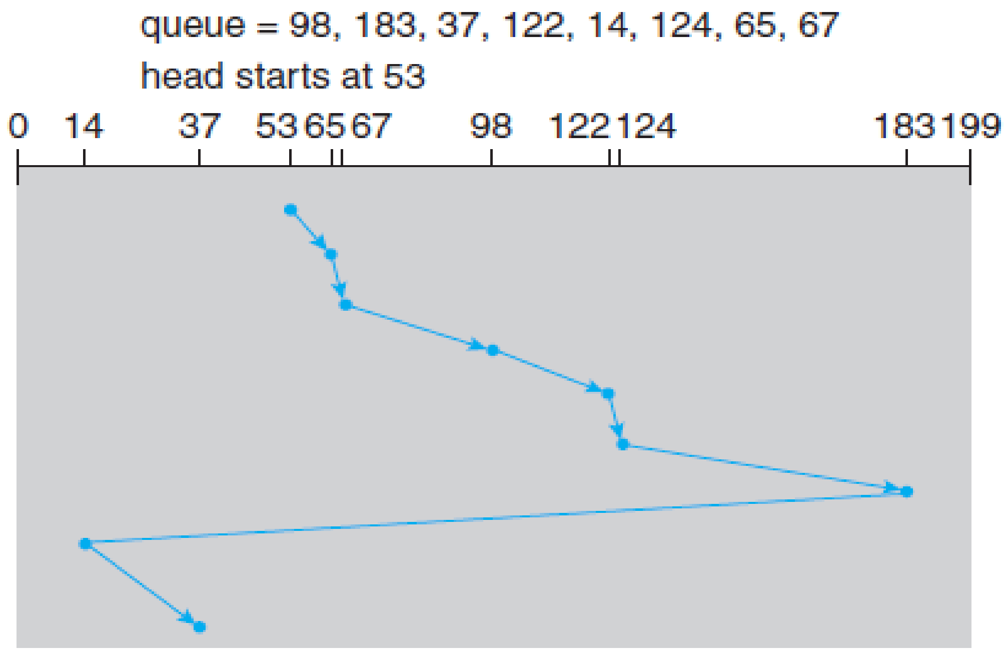


Figure 10.8 C-LOOK disk scheduling.