MODULE 4

FILE SYSTEM & I/O SYSTEM

FILE MANAGEMENT

- A file is a collection of related information defined by its creator.
- Files are managed by the OS. How they are
  there structured, named, accessed, used, protected, and implemented are major topics in OS design.
- The file system consists of two distinct parts:
  1. **a collection of files**; each storing related data,
  2. **a directory structure**; which organizes and provides information about all the files in the system.

- Objectives for a file management system;
  - Provide a convenient naming system for files.
  - Provide a standardized set of I/O interface routines and provide access control for multiple users.
  - Guarantee that the data in the file are valid. Minimize or eliminate the potential for lost or destroyed data.
  - Optimize performance.

FILE ATTRIBUTES

- 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 OS to another but typically consist of these:
  - **Name**. 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**. needed for 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.

- ○
- The information about all files is kept in the directory structure, which also resides on secondary storage. Typically, a directory entry consists of the file's name and its unique identifier.

FILE OPERATIONS

- A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.

Six basic file operations.

The OS can provide system calls to C**reate, write, read, reposition, delete, and truncate files.**

1. Creating a file. Two steps are necessary to create a file.

a. Space in the file system must be found for the file.
b. An entry for the new file must be made in the directory.

2. 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. 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.

3.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. The system needs to keep a read pointer to the location in the file where the next read is to take place.

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

4.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***.

5.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.

6.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.

- ❖ These six basic operations comprise the minimal set of required file operations
- ❖ These primitive operations can then be combined to perform other file operations (i.e., copying).
- ❖ The OS keeps a small table, called the open-file table, containing information about all open files.
    - When a file operation is requested, the file is specified via an index into this table, so no searching is required.
    - When the file is no longer being actively used, it is closed by the process, and the OS removes its entry from the open-file table
- ❖ Typically, the OS uses two levels of internal tables
    1. **A per-process table**. The per-process table tracks all files that a process has open. For instance, the current file pointer for each file is found here. Access rights to the file and accounting information can also be included.
    2. **A system-wide table**. Each entry in the per-process table in turn points to a system-wide open-file table. The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file.
- ❖ several pieces of information are associated with an open file.

    File pointer.

    File-open count.

    Disk location of the file. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.

    Access rights. Each process opens a file in an access mode. This information is stored on the per-process table so the OS can allow or deny subsequent I/O requests.

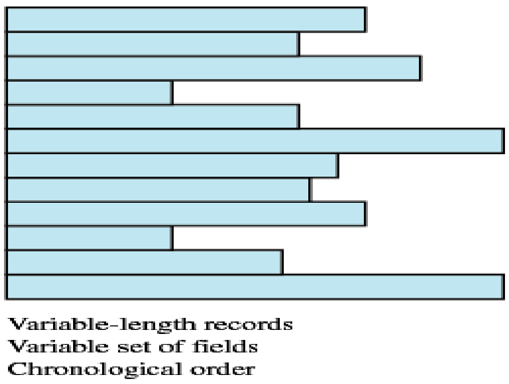
- ❖ Some OSs provide facilities for locking an open file (or sections of a file). File locks allow one process to lock a file and prevent other processes from gaining access to it. File locks are useful for files that are shared by several processes -for example, a system log file that can be accessed and modified by a number of processes in the system.

FILE ORGANISATION

- File Organization refers to the logical structuring of the records as determined by the way in which they are accessed
- Criteria for File Organization :
    - Short access time
    - Ease of update
    - Economy of storage
    - Simple maintenance
    - Reliability
- Five different types of File Organization available:
    - The pile
    - The Sequential file
    - The Indexed Sequential file
    - The Indexed file
    - The Direct or hashed file

The Pile

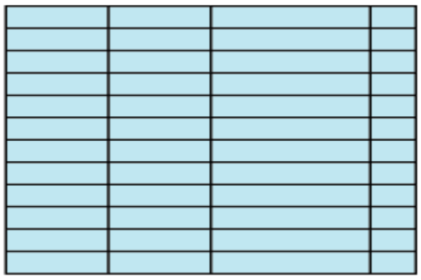
- Data are collected in the order they arrive
- Purpose is to accumulate a mass of data and save it
- Records may have different fields
- No structure Record access is by exhaustive search



(a) Pile File

The Sequential File

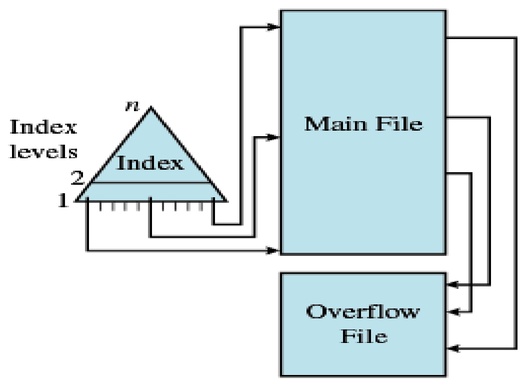
- Fixed format used for records
- Records are the same length
- All fields the same (order and length)
- Field names and lengths are attributes of the file
- One field is the key filed (Primary key)
- New records are placed in a log file or transaction file
- Batch update is performed to merge the log file with the master file.


Fixed-length records
Fixed set of fields in fixed order
Sequential order based on key field

(b) Sequential File

Indexed Sequential File

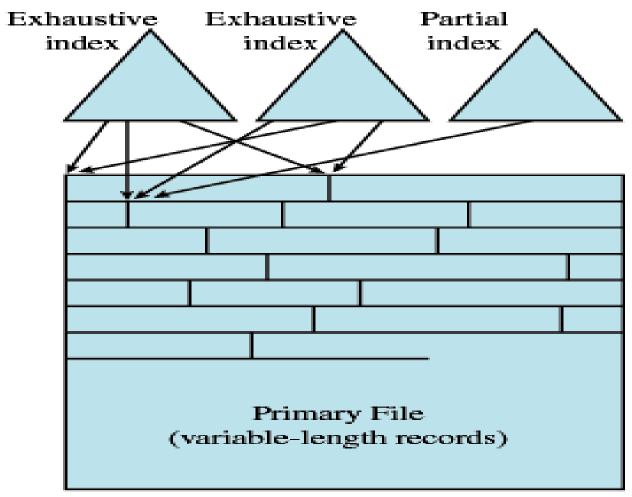
- New records are added to an overflow file.Record in main file that precedes it is updated to contain a pointer to the new record
- The overflow is merged with the main file
- Multiple indexes for the same key field can be set up to increase efficency



(c) Indexed Sequential File

Indexed File

- Uses multiple indexes for different key fields
- May contain an exhaustive index that contains one entry for every record in the main file
- May contain a partial index



(d) Indexed File

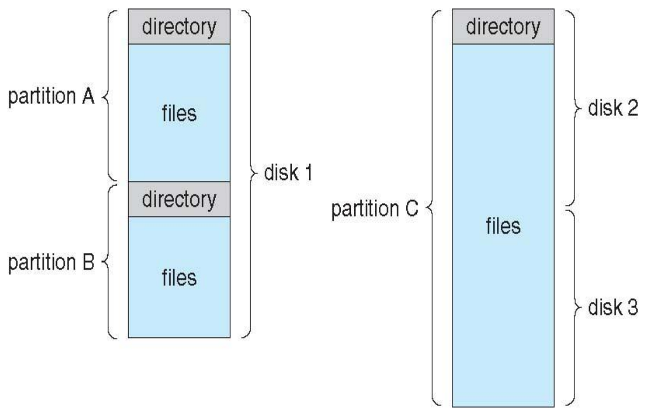
The Direct or Hashed File

- Directly access a block at a known address
- Key field required for each record

Directory

➢ A collection of nodes containing information about all files

A Typical File-system Organization



- Sometimes, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things, such as swap space or unformatted (raw) disk space.
- These parts are known variously as **partitions**, **slices**, or (in the IBM world) **minidisks**.
- A file system can be created on each of these parts of the disk. We simply refer to a chunk of storage that holds a file system as a **volume**.
- Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents**.
- The device directory (more commonly known simply as a **directory**) records information-such as name, location, size, and type-for all files on that volume.

Operations Performed on Directory

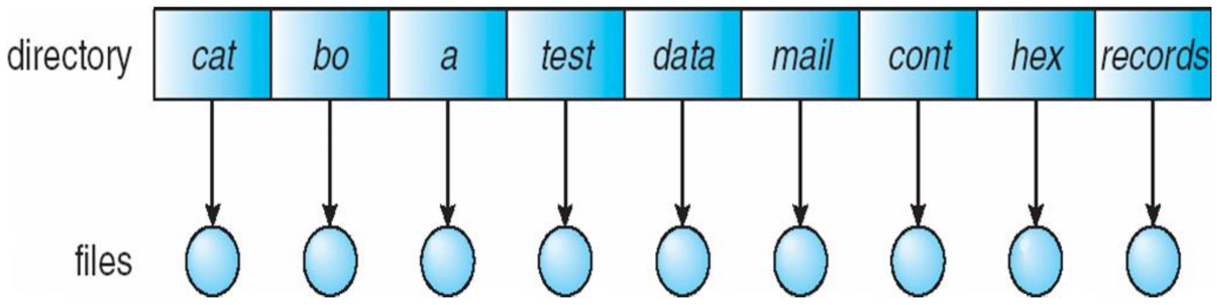
- **Search for a file**. We need to be able to search a directory structure to find the entry for a particular file.
- **Create a file**. New files need to be created and added to the directory.
- **Delete a file**. When a file is no longer needed, we want to be able to remove it from the directory.
- **List a directory**. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Rename a file**. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes.
- **Traverse the file system**. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals (backup copy).

DIRECTORY STRUCTURE

- Five types of directory structures are available
    1. Single Level Directory
    2. Two- Level Directory
    3. Tree Structured Directory
    4. Acyclic Graph Directory
    5. General Graph Directory

SINGLE LEVEL DIRECTORY

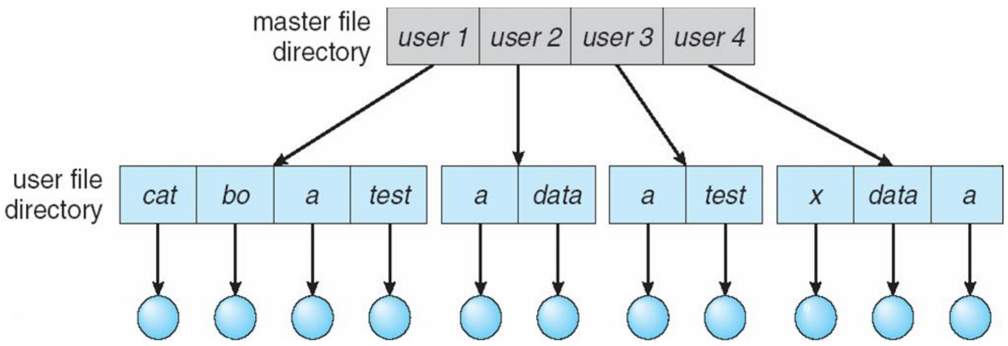
- ➢ The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand
- ➢ It is represented as a simple sequential file with the name of the file as a key



- ➢ On early personal computers, this system was common, in part because there was only one user. The world's first supercomputer, the CDC 6600, also had only a single directory for all files, even though it was used by many users at once.
- ➢ A single-level directory has significant limitations, when the number of files increases or when the system has more than one user.
- ➢ Since all files are in the same directory, they must have unique names. If two users call their data file ***test***, then the unique-name rule is violated.
- ➢ 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.

Two-Level Directory

- The standard solution to limitations of single-level directory is to create a separate directory for each user.
- In the two-level directory structure, each user has his own **user file directory** (**UFD**). The UFDs have similar structures, but each lists only the files of a single user.
- When a user job starts or a user logs in, the system's **master file directory** (MFD) is searched.
- The MFD is indexed by user name or account number, and each entry points to the UFD for that user
- when a user refers to a particular file, only his own UFD is searched (create a file, delete a file)
- Although the two-level directory structure solves the name-collision problem, it still has disadvantages



This structure effectively isolates one user from another

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. To name a file uniquely, a user must know the path name of the file desired.

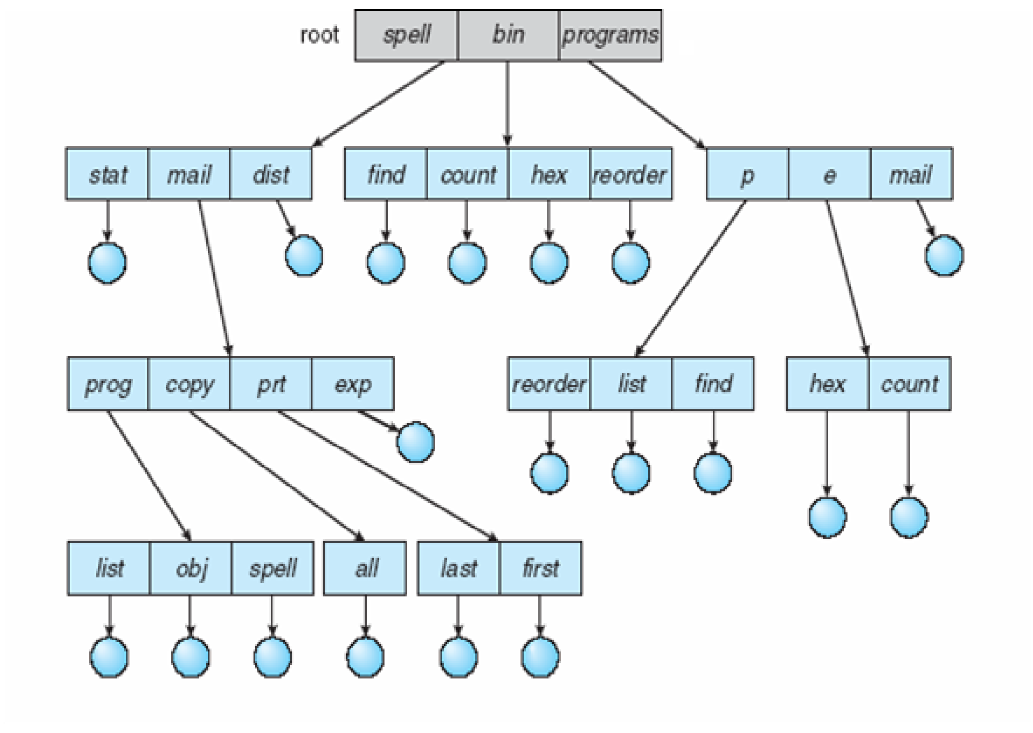
Additional syntax is needed to specify the volume of a file. A file specification might be

```
C:\userb\test
```

Tree-Structured Directories

- A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.
- A directory 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),
  - as a subdirectory (1).
- 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.

- With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users.
  - For example, user B can access a file of user A by specifying its path names.
  - User B can specify either an absolute or a relative path name.
  - Alternatively, user B can change her current directory to be user A 's directory and access the file by its file names.

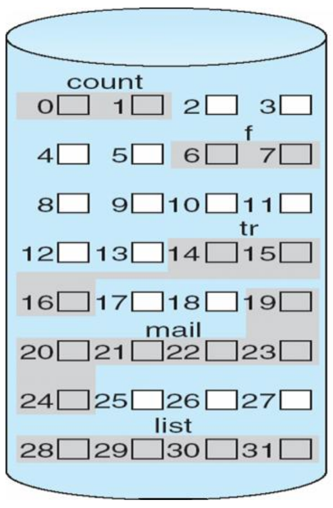


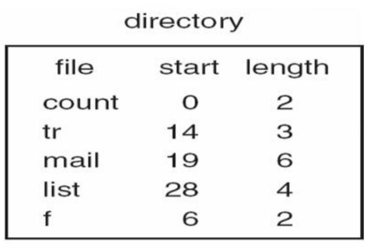
ALLOCATION METHODS

- An allocation method refers to how disk blocks are allocated for files:
- Three major methods of allocating disk space are in wide use: **contiguous, linked,** and **indexed**

Contiguous Allocation

- **Contiguous allocation** requires that each file occupy a set of contiguous blocks 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 $\boldsymbol{n}$ blocks long and starts at location $\boldsymbol{b}$, then it occupies blocks $b, b+1, b+2, \ldots b+n-1$.





directory

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

- Contiguous allocation is widely used on CD-ROMs. Here all the file sizes are known in advance and will never change during subsequent use of the CD-ROM file system.
- Accessing a file that has been allocated contiguously is easy.

  1. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block.

  2. For direct access to block i of a file that starts at block $\boldsymbol{b}$ we can immediately access block $\boldsymbol{b+i}$.

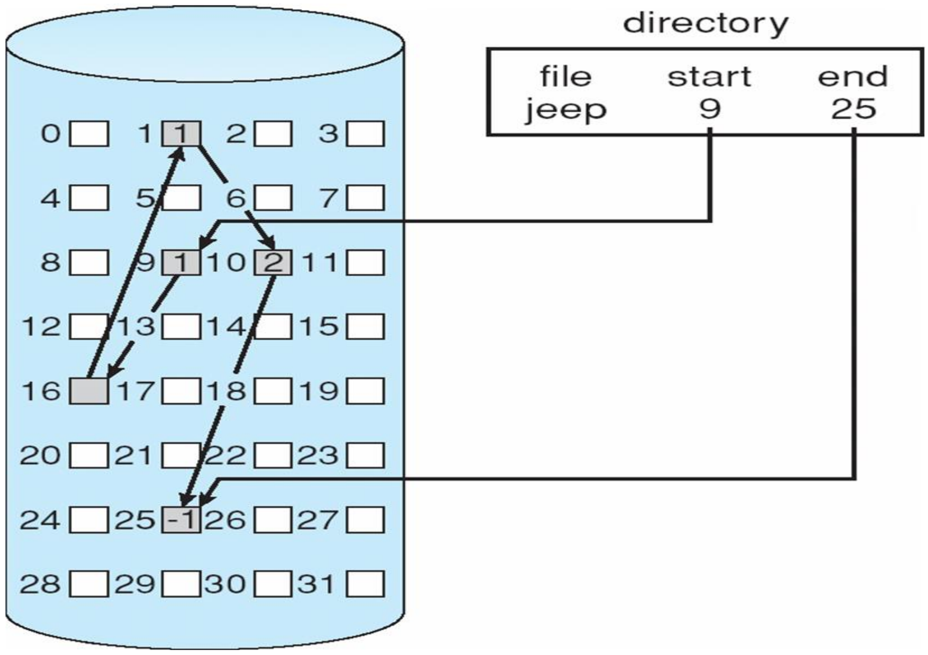
- Thus, both sequential and direct access can be supported by contiguous allocation.

- As files are allocated and deleted, the free disk space is broken into little pieces.
- **External fragmentation** exists whenever free space is broken into chunks.
  - It becomes a problem when the largest contiguous chunk is insufficient for a request;
  - Storage is fragmented into a number of holes, no one of which is large enough to store the data.
- Compacting all free space into one contiguous space, solves the fragmentation problem.
  - The cost of this compaction is time.
  - The time cost is particularly severe for large hard disks that use contiguous allocation, where compacting all the space may take hours and may be necessary on a weekly basis.
- Another problem with contiguous allocation is determining how much space is needed for a file.
- When the file is created, the total amount of space it will need must be found and allocated.
- If we allocate too little space to a file, we may find that the file cannot be **extended**.
- Two possibilities then exist.
- First, the user program can be terminated with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may be costly. To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space.
- The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space.
- Even if the total amount of space needed for a file is known in advance, pre allocation may be inefficient. The file therefore has a lager amount of internal fragmentation.

<u>Linked Allocation</u>

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

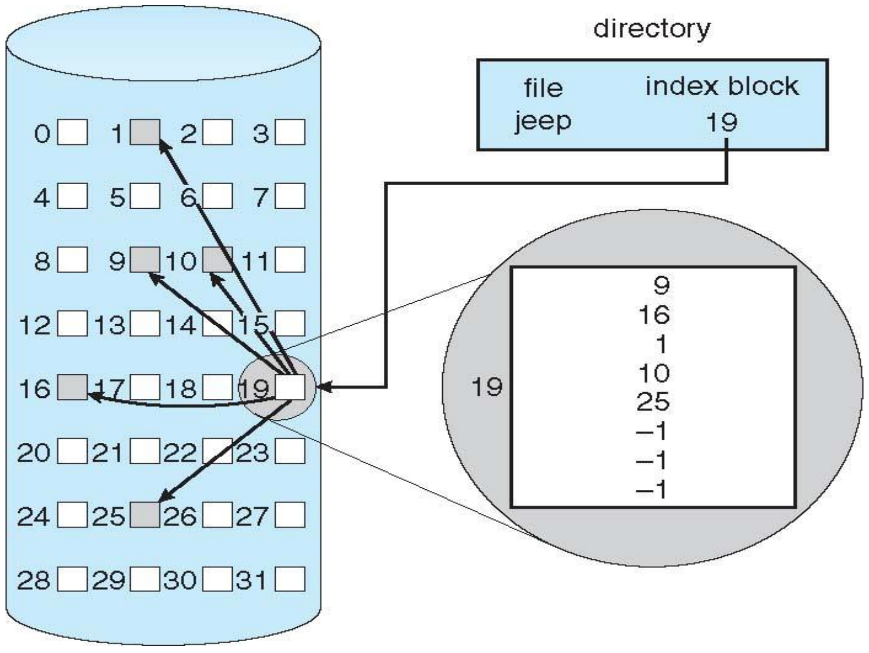


- To create a new file, we simply create a new entry in the directory. The pointer is initialized to nil (the end-of-list pointer value) to signify an empty file. The size field is also set to O.
- A write to the file causes the free-space management system to find a free block, and this new block is 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. No space is lost to disk fragmentation (except for internal fragmentation in the last block).
- 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.
- The major problem is that it can be used effectively only for sequential-access files.
    - To find the i^{th} block of a file, we must start at the beginning of that file and follow the pointers until we get to the i^{th} block.
    - Each access to a pointer requires a disk read, and some require a disk seek.
- Consequently, it is inefficient to support a direct-access capability for linked-allocation files.
- Another disadvantage is the space required for the pointers.
    - If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.

- The usual solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate clusters rather than blocks.
  - For instance, the file system may define a cluster as four blocks and operate on the disk only in cluster units.
  - Pointers then use a much smaller percentage of the file's disk space.
  - Improves disk throughput (because fewer disk-head seeks are required) and decreases the space needed for block allocation and free-list management.
  - The cost of this approach is an increase in internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full.
- Yet another problem of linked allocation is reliability.
  - Recall that the files are linked together by pointers scattered all over the disk, and consider what would happen if a pointer were lost or damaged.
  - A bug in the OS software or a disk hardware failure might result in picking up the wrong pointer.
- An important variation on linked allocation is the use of a file-allocation table (FAT).

INDEXED ALLOCATION

- Each file has its own index block, which is an array of disk-block addresses.
- Files are stored in randomly spreaded blocks of secondary memory.one block is designated as index block and it contains the address of all other blocks
- 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



- To find and read the i^{th} block, we use the pointer in the i^{th} index-block entry. This scheme is similar to the paging scheme.
- Given the i-node, it is then possible to find all the blocks of the file. The big advantage of this scheme over linked files using an in-memory table is that the i-node need only be in memory when the corresponding file is open.
- When the file is created, all pointers in the index block are set to nil. When the i^{th} block is first written, a block is obtained from the free-space manager, and its address is put in the i^{th} index-block entry.
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.
- The main advantage is that there is no external fragmentation and it support sequential and direct access of files

<u>I/O SYSTEM</u>

One of the main functions of the operating system is to manage the input/ output devices connected with the computer efficiently

That is it must resolve the problems arising from getting the devices among various users

The following are the important I/O management functions of OS

1. Keep track of available devices
2. Decide the efficient way of allocation
3. Allocate the device
4. Get back the device once job is done

<u>Applications of I/O Interface</u>

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- Devices vary in many dimensions

Character-stream or block

Sequential or random-access

Sharable or dedicated

Speed of operation

Read-write, read only, or write only

KERNEL I/O SUBSYSTEM

Kernel provided many services related to I/O.

The several services.

(1) Scheduling

(2) Buffering

(3) Caching

(4) Spooling

(5) Device reservation and

(6) Error handling

1*)* **I/O scheduling:** To schedule a set of-I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes and can reduce the average waiting time for I/O to complete.

Operating system developers implement scheduling by maintaining a queue of requests for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The I/O sub system improves the efficiency of the computer is by scheduling I/O operations.

2) **Buffering:** A Buffer is a memory area that stores data while they are transferred between two devices (or) between a device and an applications. Buffering is done fore three reasons.

One reason is to cope with a speed mismatch between the producer and consumer of a data stream. For example that a file is being received via modem for storage on the hard disk. The modem is about a thousand times slower than the hard disk. So a buffer is created in main memory to accumulate the bytes received from the modem. When an entire buffer ot data has arrived, the buffer can be written to disk in a single operation.

A second use of buffering is to adapt between devices that have different data transfer sizes. Such disparities are especially common in computer networking. Where buffers are used widely for fragmentation and reassembly of messages. At the sending side, a large message is fragmented into small network packets. The packets are sent over the network, and the receiving side places them in a reassembly buffer to form an image of the source data.

A third use of buffering is to support copy semantics for application I/O.

3) **Caching:** A cache is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. For instance the instructions of the currently running process are stored on disk, cached in physical memory and copied again in the CPU's secondary and primary caches. The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, where as a cache, by definition, just holds a copy an faster storage of an item that resides else where.

4) **Spooling and Device reservation:** A spool is a buffer that holds output for a device, such as a printer that cannot accept inter level data streams. Although a printer can serve only one job at a time, several applications may wish to print their output mixed together. The operating system solves this problem by intercepting all output to the printer. Each application's output is spooled to a separate disk file. When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer. The spooling system copies the queued spool flies to the printer one at a time.

5)**Error Handling:** An operating system that uses protected memory can guard against many kinds of hardware and application errors. So that a complete system failure is not the usual result of each minor mechanical glitch. Devices and I/O transfers can fail in many ways, either for transient reasons, such a network becoming overloaded (or) for 'permanent' reasons such as a disk controller becoming detective. Operating systems can often compensate effectively for transient failures. For instance, a disk read ( ) failure results in a read ( ) retry, and a network send ( ) error results in a resend ( ), if the protocol so specifies, unfortunately, if an important component experiences a permanent failure, the operating system is unlikely to recover.

DISK STRUCTURE

- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer.
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
    - Sector 0 is the first sector of the first track on the outermost cylinder.
    - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

DISK SCHEDULING

- One of the responsibilities of the OS is to use the hardware efficiently. For the disk drives, meeting this responsibility entails 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.
- For a multiprogramming system with many processes, the disk queue may often have several pending requests.
  - Thus, when one request is completed, the OS chooses which pending request to service next.
  - OS make this choice by Disk-scheduling algorithms.

Several algorithms exist to schedule the servicing of disk I/O requests.

1. FCFS scheduling
2. SSTF scheduling
3. SCAN scheduling
4. C- SCAN scheduling

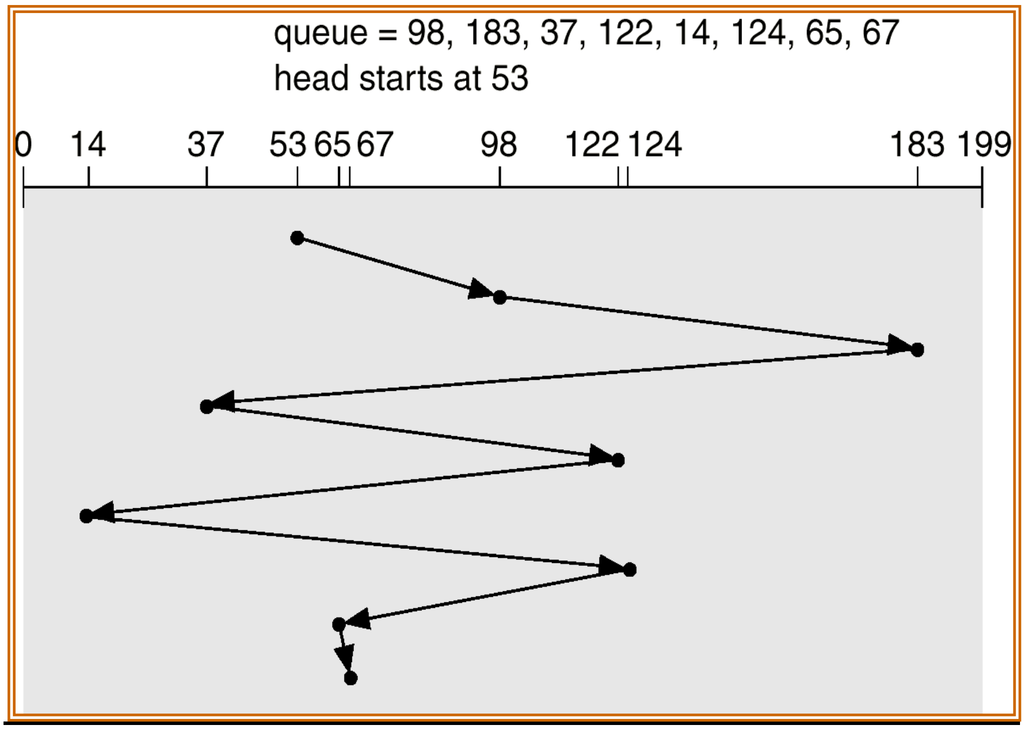
FCFS SCHEDULING

- 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 in that order;

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

  - 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 as below

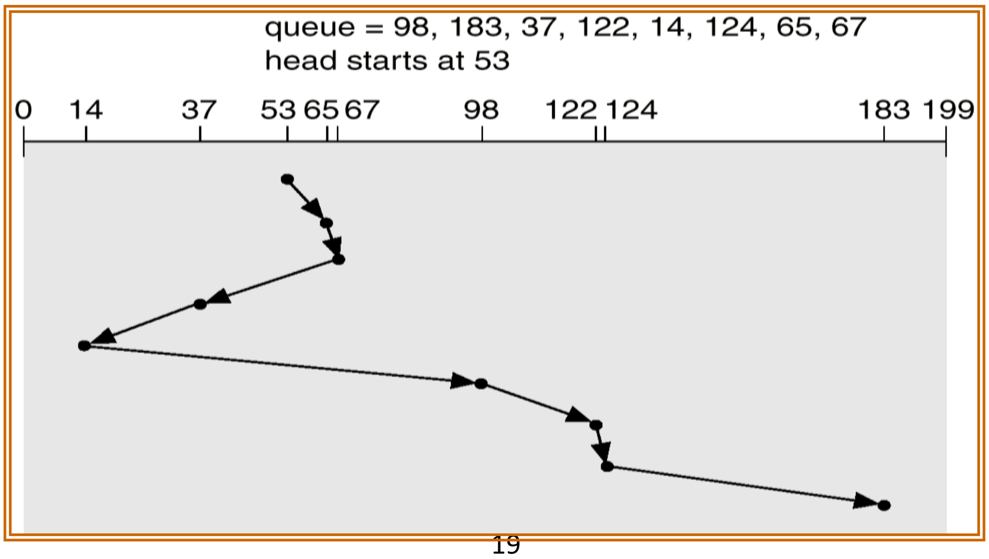


SSTF Scheduling

- It seems reasonable to 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 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

```
53, 65, 67, 37, 14, 98, 122, 124, 183
```



- 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; and like SJF scheduling, it may cause starvation of some requests (steady supply of shorter seek time requests).
- Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal

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.

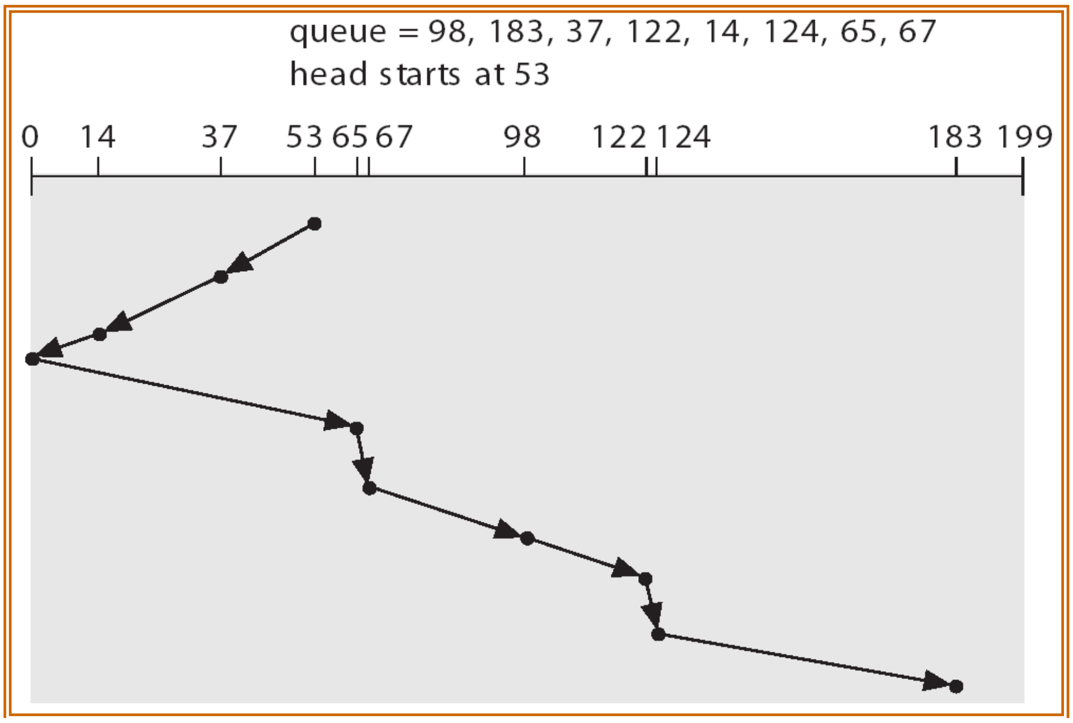
At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.

Sometimes called as ***elevator algorithm*.**

The SCAN algorithm is sometimes called the elevator algorithm, since the disk arms behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

For our example request queue, we need to know the direction of head movement in addition to the head's current position, 53

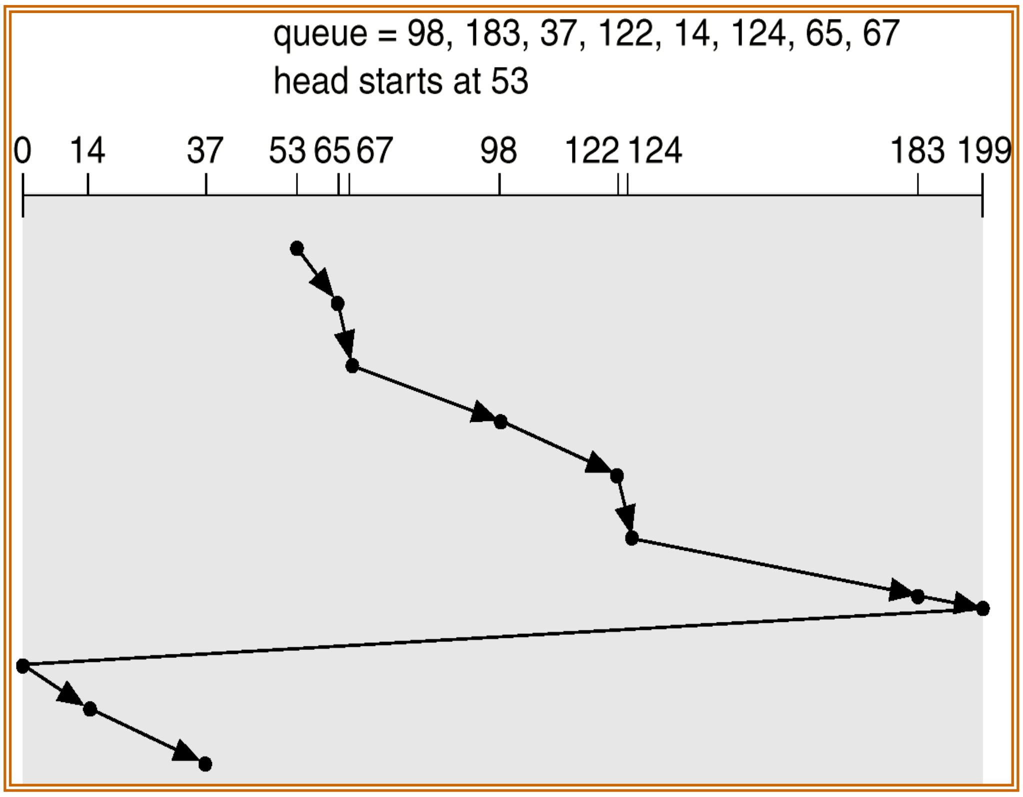
```
37, 14, 65, 67, 98, 122, 124, 183
```



- If a request arrives in the queue just in front of the head, it will be serviced almost immediately
- If 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.

C -SCAN Scheduling

- Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time.
- Like SCAN, CSCAN 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.



DISK MANAGEMENT

- Os is responsible for several aspects of Disk management such as Disk initialization, booting from disk, and bad- block recovery

DISK FORMATTING

- A new magnetic disk is a blank state
- 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 (physical Formatting)***
- It fills the disk with special data structure for each sector
- The data structure for a sector consists of a header, a data area and a trailer
- Header and trailer contains informations used by the disk controller such as a sector number and an error correcting code(ECC)
- When a controller writes a sector of data during normal I/O , ECC is updated with a value calculated from all the bytes in the data area
- When the sector is read the ECC is recalculated and is compared with the stored value
- If the stored value and calculated value are different, the mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad
- Most hard disks are low level formatted at the factory as a part of manufacturing process

- To use a disk to hold files, The OS still needs to record its own data structures on the disk. It does so in 2 steps
- The first step is to **partition** the disk into one or more group of cylinders
- After partitioning , the second step is **logical formatting.** In this step the OS stores the initial file system data structure onto disk

BOOT BLOCK

- The initial bootstrap program initializes all the aspects of the system from CPU registers to device controllers and contents of main memory and then starts the OS
- Bootstrap is stored in ROM(read only memory), since it doesn't needs the initalisation
- A disk that that has boot partition is called as **boot disk or system disk**
- The code in the ROM instructs the disk controller to read the boot blocks into memory and then starts executing that code

BAD BLOCKS

- Depending upon the disk and the controller in use, bad blocks are handled ina many ways
- Datas residing in the bad blocks are usually lost
- The controller can told to replace each bad sector logically with one of the spare sectors. This scheme is known as ***sector sparing or forwarding***
- Some controllers can be instructed to replace a bad block by ***sector slipping***

SWAP SPACE MANAGEMENT

- The main goal of design and implementation of swap space is to provide the best throughput for the virtual memory system

Swap Space Use

- Swap space is used in various ways by different operating system, depending on the implemented memory management algorithm
- Systems that implement swapping may use swap space to hold the entire process image, including the code and data segments
- Paging system may simply store pages that have pushed out of main memory
- The amount of swap space needed on a system can therefore vary depending on the amount of physical memory, amount of virtual memory it is backing and the way in which the virtual memory is used
- Some operating systems use multiple swap spaces

Swap Space Locations

- A swap space can reside in two places: Swap space can be carved out of the normal file system or it can be in a separate disk partition.
- If the swap space is simply a large file within a file system, normal file system routines can be used to create it, name it and allocate its space.
- This is easy to implement but inefficient
- While creating swap space in a separate disk partition, no file system or directory structure is placed on this space
- Rather a separate swap space storage manager is used to allocate and de allocate the blocks
- This manager uses algorithms optimized for speed , rather than for storage efficiency
- Some Operating systems are flexible and can swap both in raw partitions and in file system space