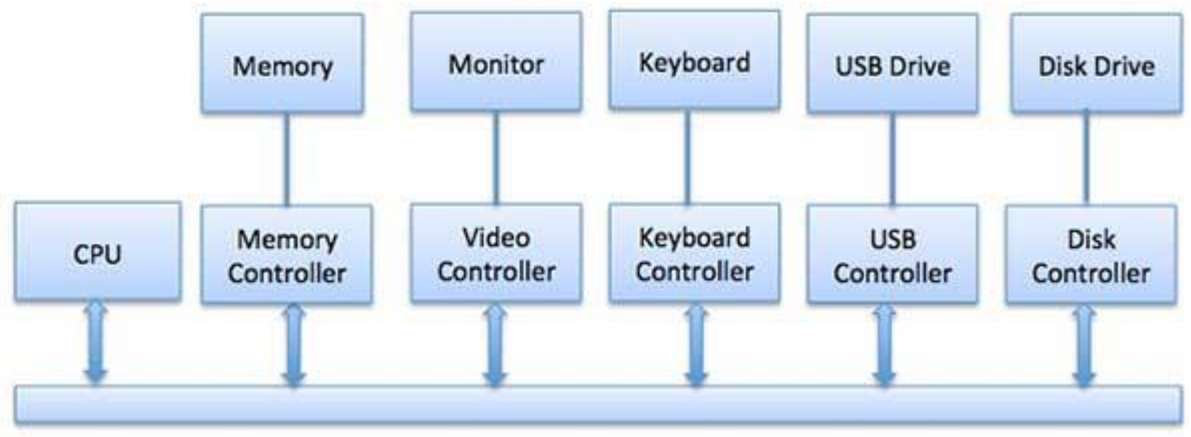
Device Controllers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices.

The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller.

There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as necessary.

Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.



Synchronous vs asynchronous I/O

- **Synchronous I/O** − In this scheme CPU execution waits while I/O proceeds
- **Asynchronous I/O** − I/O proceeds concurrently with CPU execution

<u>**Communication to I/O Devices**</u>

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

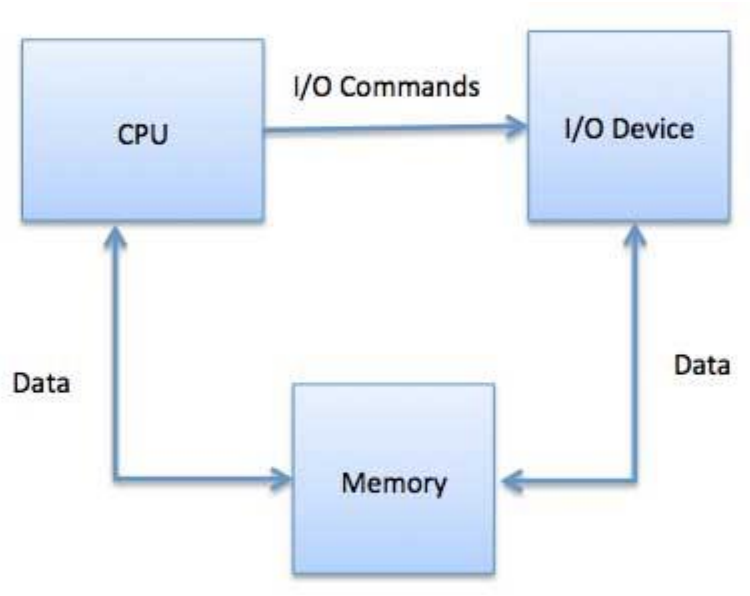
- Special Instruction I/O
- Memory-mapped I/O
- Direct memory access (DMA)

Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

Memory-mapped I/O

When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU.



While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished.

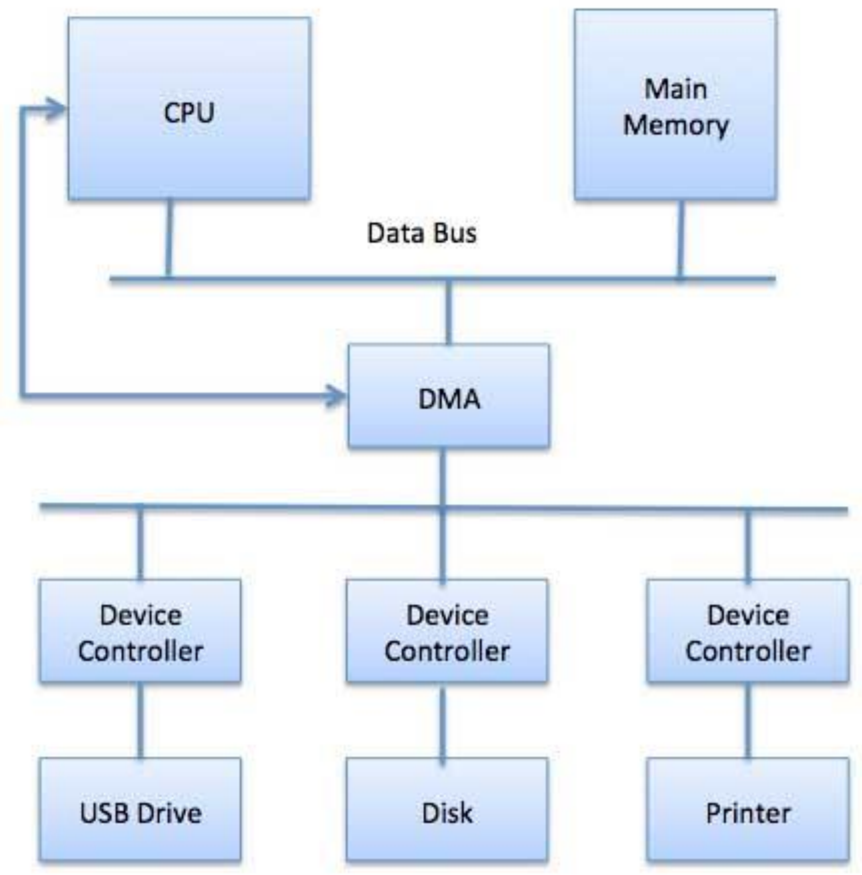
The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

Direct Memory Access (DMA)

Direct memory access (DMA) is a technology that allows hardware devices to transfer data between themselves and memory without involving the central processing unit (CPU).

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.



The operating system uses the DMA hardware as follows −

Step	Description
1	Device driver is instructed to transfer disk data to a buffer address X.
2	Device driver then instruct disk controller to transfer data to buffer.
3	Disk controller starts DMA transfer.
4	Disk controller sends each byte to DMA controller.
5	DMA controller transfers bytes to buffer, increases the memory address, decreases the counter C until C becomes zero.
6	When C becomes zero, DMA interrupts CPU to signal transfer completion.

DMA Principles

DMA principles govern efficient and reliable data transfer between devices and memory without CPU intervention. These principles include:

- **Independent data transfer:** DMA allows devices to transfer data directly to and from memory without CPU intervention. This independence reduces CPU overhead and allows the CPU to focus on executing other tasks while data transfer occurs.
- **Bus arbitration:** DMA controllers must arbitrate for access to the system bus to ensure that data transfers do not interfere with other bus transactions. They typically prioritize DMA transfers appropriately to ensure the system's smooth operation.
- **Memory access control:** DMA controllers must have mechanisms to access system memory safely and efficiently. They adhere to memory access protocols and coordinate with memory controllers to read from and write to memory locations without causing conflicts with CPU operations.
- **Transfer modes:** DMA controllers support various transfer modes, such as single transfer, block transfer, and demand-based transfer. These modes accommodate different data transfer scenarios and optimize data movement between devices and memory.
- **Data buffering:** DMA controllers often incorporate data buffering mechanisms to store data temporarily during transfers. Buffers help smooth out differences in transfer rates between devices and memory and mitigate potential data loss or corruption.
- **Error handling:** DMA implements error detection and handling mechanisms to ensure the integrity of data transfers. They can detect transmission errors, address conflicts, and retry or report errors as needed to maintain data integrity and system reliability.
- **Interrupt handling:** The controllers generate interrupts to inform the CPU of transfer completion or errors. Interrupt handling mechanisms allow the CPU to respond promptly to DMA events, enabling efficient coordination between DMA transfers and CPU operations.

Steps of working of DMA

- Step 1. Initiation
- Initiation is the first step in the DMA process. It kicks off the data transfer operation between devices without involving the CPU constantly. When a device needs to send or receive data from memory, it initiates a DMA request.
- During initiation, the DMA controller identifies and prioritizes incoming requests based on predefined criteria. This ensures efficient utilization of system resources and minimizes delays in data transfer. Once a request is accepted, the DMA controller temporarily takes control of the bus to facilitate direct communication between devices and memory.

- Step 2. Request
- When a peripheral device needs to transfer data to or from memory, it sends a DMA request signal to the DMA controller. This signal indicates that the CPU is not required for this particular data transfer operation.
- Upon receiving the DMA request, the DMA controller checks if the bus is available and then initiates access to memory. By handling these requests independently of the CPU, DMA significantly reduces processor overhead and speeds up data transfers between devices and memory.
- The DMA controller manages the timing and prioritization of these requests through efficient arbitration techniques. This ensures that multiple devices can communicate with memory seamlessly without causing conflicts or bottlenecks in data flow.

- Step 3: Arbitration
- When multiple devices need to access the memory simultaneously, the DMA controller arbitrates between these requests to ensure efficient utilization of the system's resources.
- Through arbitration, the DMA controller prioritizes and schedules data transfer tasks based on predefined rules or algorithms. This process helps prevent conflicts and ensures that each device gets fair access to the memory bus without causing bottlenecks or delays in data transfers.
- By intelligently managing contention for memory access, the DMA controller optimizes the system's overall performance by minimizing idle time and maximizing throughput. It coordinates communication between different components seamlessly, allowing for smoother operation and improved efficiency in handling large volumes of data.

- Step 4: Bus mastering
- In this process, a DMA-capable device takes control of the system bus to manage data transfers independently from the CPU. The DMA controller coordinates with other devices on the bus for efficient data movement, ensuring smooth communication flow within the system.

- Step 5: Memory access

- Once the DMA controller gains control of the system bus, it can directly access the memory without involving the CPU. This direct interaction allows efficient and speedy data transfers between peripherals and memory locations.
- During memory access, data is read from or written to specific memory addresses as instructed by the DMA controller. The controller ensures that data is transferred accurately and promptly without requiring constant intervention from the CPU.

- Step 6: Data transfer
- Once the DMA controller has control of the bus, it initiates the actual data movement between devices and memory. The DMA controller coordinates with the source and destination devices to efficiently transfer data without involving the CPU. During data transfer, information flows directly from one device to another through DMA channels without CPU intervention.

- Step 7: Completion and interrupt
- Once the data transfer is completed, the DMA controller triggers an interrupt to notify the CPU. This interrupt signals that the DMA operation has finished successfully. The CPU can then resume its tasks or handle any necessary follow-up actions based on the completion of the data transfer.
- Interrupts are crucial as they allow efficient communication between the DMA controller and the CPU without constant polling. By using interrupts, system resources are utilized more effectively since the CPU can attend to other processes while waiting for DMA operations to finish.
- Upon receiving an interrupt from the DMA controller, the CPU may execute specific routines or procedures set up to handle post-DMA tasks, such as updating status flags, notifying software components about data availability, or initiating further processing steps based on the transferred data.

- Step 8: Release of bus control
- Once the data transfer is complete, the DMA controller releases control of the system bus. This step is crucial as it allows other devices to access the bus for their own operations without any interference from the DMA process.

- This final step paves the way for ongoing processes within the computer system to continue smoothly without any hindrance caused by the exclusive use of resources during data transfers.

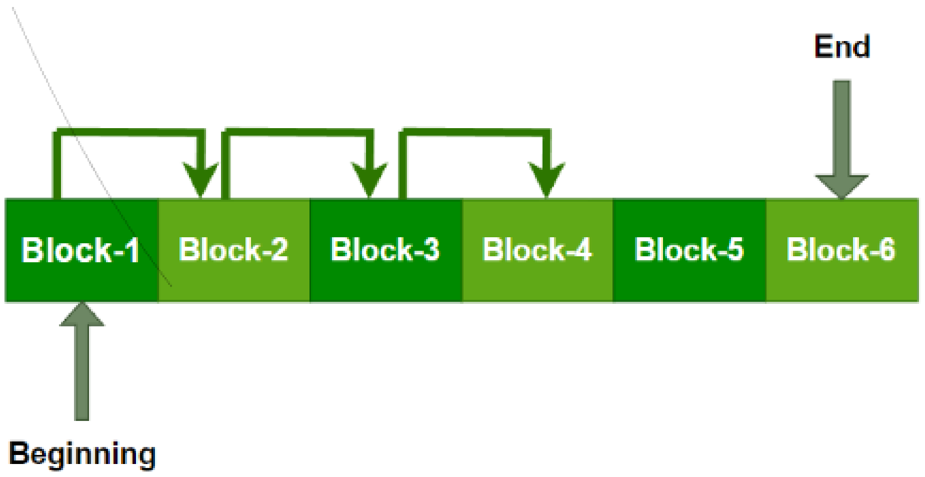
File Access Methods

There are three ways to access a file in a computer system:

- Sequential-Access
- Direct Access
- Index sequential Method

Sequential Access

It is the simplest access method. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, the editor and compiler usually access the file in this fashion.
Read and write make up the bulk of the operation on a file. A read operation -*read next*- reads the next position of the file and automatically advances a file pointer, which keeps track of the I/O location. Similarly, for the -write *next*- append to the end of the file and advance to the newly written material.



Key Points related to Sequential Access

- Data is accessed from one record right after another record in an order.
- When we use the read command, it moves ahead pointer by one.
- When we use the write command, it will allocate memory and move the pointer to the end of the file.
- Such a method is reasonable for tape.

Advantages of Sequential Access Method

- It is simple to implement this file access mechanism.

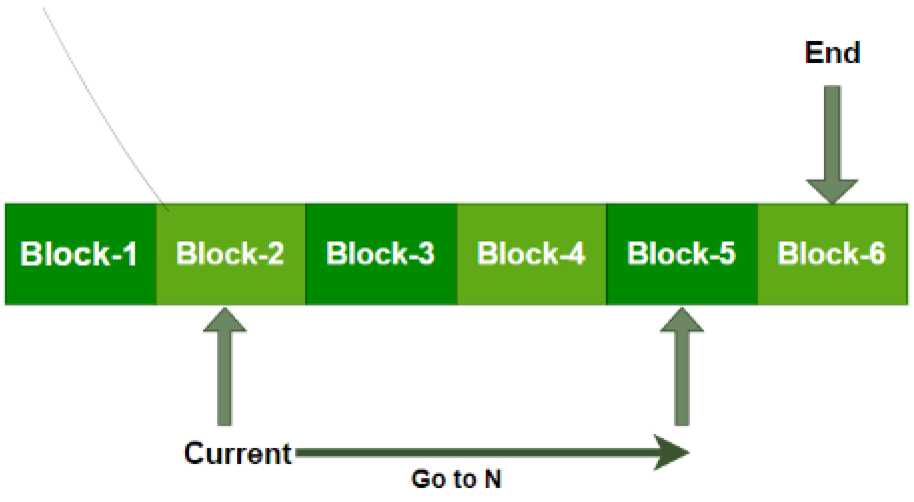
- It is suitable for applications that require access to all records in a file, in a specific order.
- It is less prone to data corruption as the data is written sequentially and not randomly.
- It is a more efficient method for reading large files, as it only reads the required data and does not waste time reading unnecessary data.
- It is a reliable method for backup and restore operations, as the data is stored sequentially and can be easily restored if required.

Disadvantages of Sequential Access Method

- If the file record that needs to be accessed next is not present next to the current record, this type of file access method is slow.
- Moving a sizable chunk of the file may be necessary to insert a new record.
- It does not allow for quick access to specific records in the file. The entire file must be searched sequentially to find a specific record, which can be time-consuming.
- It is not well-suited for applications that require frequent updates or modifications to the file. Updating or inserting a record in the middle of a large file can be a slow and cumbersome process.
- Sequential access can also result in wasted storage space if records are of varying lengths. The space between records cannot be used by other records, which can result in inefficient use of storage.

Direct Access Method

Another method is *direct access method* also known as *relative access method*. A fixed-length logical record that allows the program to read and write record rapidly in no particular order. The direct access is based on the disk model of a file since disk allows random access to any file block. For direct access, the file is viewed as a numbered sequence of block or record. Thus, we may read block 14 then block 59, and then we can write block 17. There is no restriction on the order of reading and writing for a direct access file.
A block number provided by the user to the operating system is normally a *relative block number*, the first relative block of the file is 0 and then 1 and so on.



Advantages of Direct Access Method

- The files can be immediately accessed decreasing the average access time.
- In the direct access method, in order to access a block, there is no need of traversing all the blocks present before it.

Disadvantages of Direct Access Method

- Complex Implementation: Implementing direct access can be complex, requiring sophisticated algorithms and data structures to manage and locate records efficiently.
- Higher Storage Overhead: Direct access methods often require additional storage for maintaining data location information (such as pointers or address tables), which can increase the overall storage requirements.

Index Sequential method

It is the other method of accessing a file that is built on the top of the sequential access method. These methods construct an index for the file. The index, like an index in the back of a book, contains the pointer to the various blocks. To find a record in the file, we first search the index, and then by the help of pointer we access the file directly.

Key Points Related to Index Sequential Method

- It is built on top of Sequential access.
- It control the pointer by using index.

Advantages of Index Sequential Method

- **Efficient Searching**: Index sequential method allows for quick searches through the index.
- **Balanced Performance**: It combines the simplicity of sequential access with the speed of direct access, offering a balanced approach that can handle various types of data access needs efficiently.
- **Flexibility**: This method allows both sequential and random access to data, making it versatile for different types of applications, such as batch processing and real-time querying.
- **Improved Data Management**: Indexing helps in better organization and management of data. It makes data retrieval faster and more efficient, especially in large databases.
- **Reduced Access Time**: By using an index to directly locate data blocks, the time spent searching for data within large datasets is significantly reduced.

Disadvantages of Index Sequential Method

- **Complex Implementation**: The index sequential method is more complex to implement and maintain compared to simple sequential access methods.
- **Additional Storage**: Indexes require additional storage space, which can be significant for large datasets. This extra space can sometimes offset the benefits of faster access.
- **Update Overhead**: Updating the data can be more time-consuming because both the data and the indexes need to be updated. This can lead to increased processing time for insertions, deletions, and modifications.

- **Index Maintenance**: Keeping the index up to date requires regular maintenance, especially in dynamic environments where data changes frequently. This can add to the system's overhead.

Disk scheduling

A hard disk is a secondary storage that stores a large amount of data. The hard disk drive contains dozens of disks. These disks are also known as platters. These platters are mounted over the spindle, which rotates in any direction, i.e., clockwise or anti-clockwise.

- One surface of the Platter requires one Read/Write head, and a second R/W head is used for the other surface to store information.
- Every Platter holds the same no of tracks.
- Multiple platters increase the storage capacity

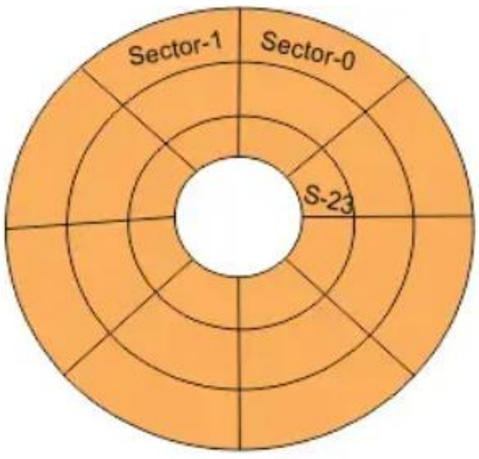
Tracks
Circular areas of the disk are known as tracks.

- Track Numbering starts with **zero** from the outermost track.

Sectors
We further divide tracks into several small units, and these units are known as sectors.

- Sectors are the most minor physical storage units on disk.
- The size of each sector is almost always 512 Bytes.
- Sector Numbering starts from number 1, in the outermost tracks.

Below is a descriptive diagram of a single platter.



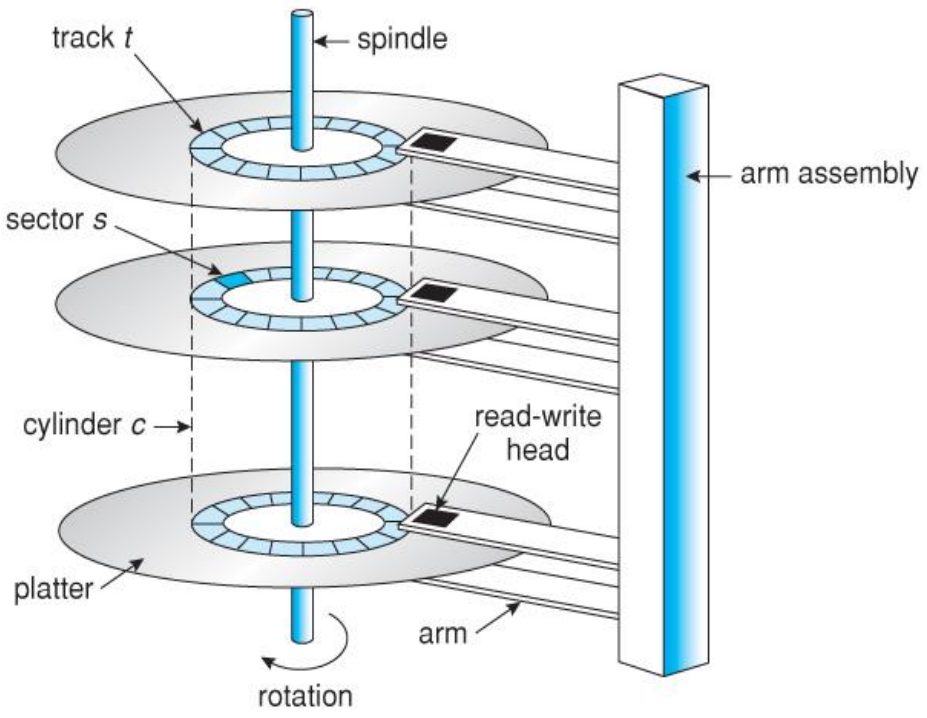
Single Platter with 3 tracks
Where each track contains 8-sectors

R/W Head
R/W Heads move forth and back over the Platter surfaces to Read or Write the data on sectors. Read/Write heads do not touch the platter surface.

- The magnetic field writes data onto the platter surface.

- When the R/W head contacts the platter surface, it may create bad sectors.
- Had disk may damage due to these bad sectors.

Cylinder
All Corresponding tracks with the same radius of all platters in the Hard disk are known as cylinders. In simple words, we say each track of all platters with the same radius is called cylinder.



Seek Time
Seek time is the time taken in locating the disk arm to a specified track where the read/write request will be satisfied.

Rotational Latency
It is the time taken by the desired sector to rotate itself to the position from where it can access the R/W heads.

Transfer Time
It is the time taken to transfer the data.

Disk Access Time
Disk access time is given as,
Disk Access Time = Rotational Latency + Seek Time + Transfer Time

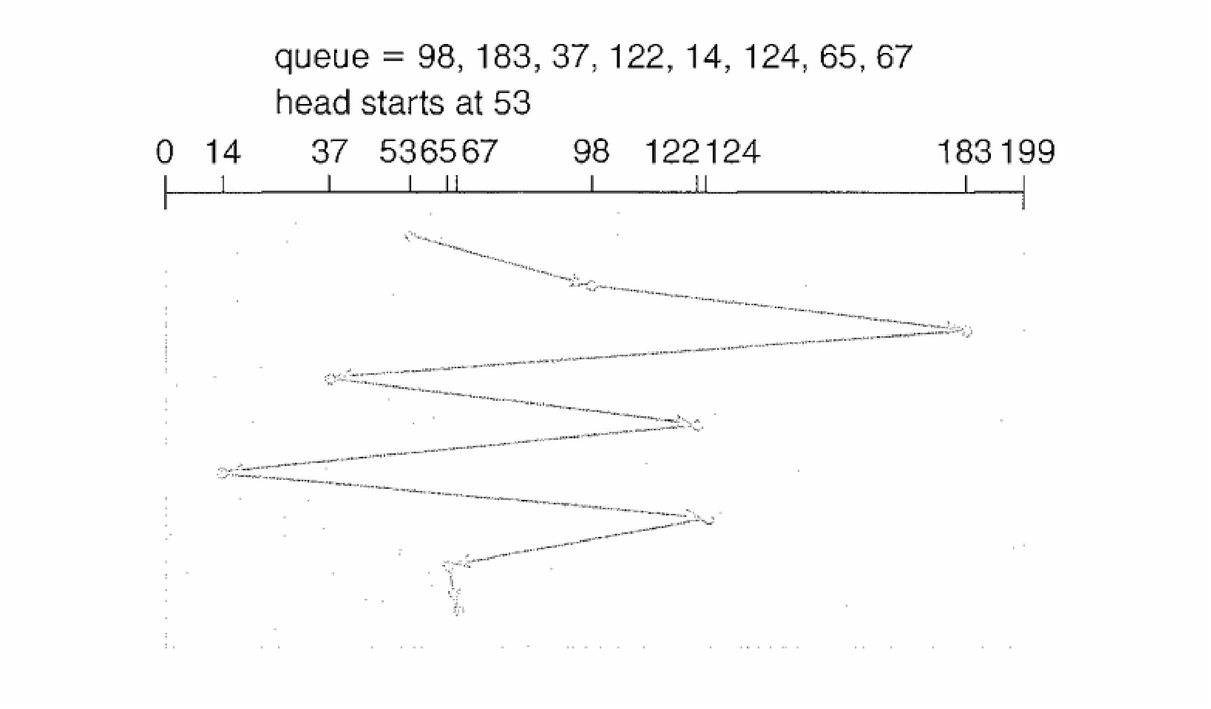
Disk Response Time
It is the average of time spent by each request waiting for the IO operation.

Disk scheduling algorithms

Disk scheduling algorithms are an essential component of modern operating systems and are responsible for determining the order in which disk access requests are serviced. The primary goal of these algorithms is to minimize disk access time and improve overall system performance.

1. **FCFS disk scheduling –**
   It is the simplest Disk Scheduling algorithm. It services the IO requests in the order in which they arrive. It stands for 'first-come-first-serve'. As the name suggests, the request that comes first will be processed first and so on. The requests coming to the disk are arranged in a proper sequence as they arrive. Since every request is processed in this algorithm, so there is no chance of 'starvation'.

Consider the following disk request sequence for a disk with 100
Head pointer starting at 50 and moving in left direction.
98, 183, 37, 122, 14, 124, 65, 67



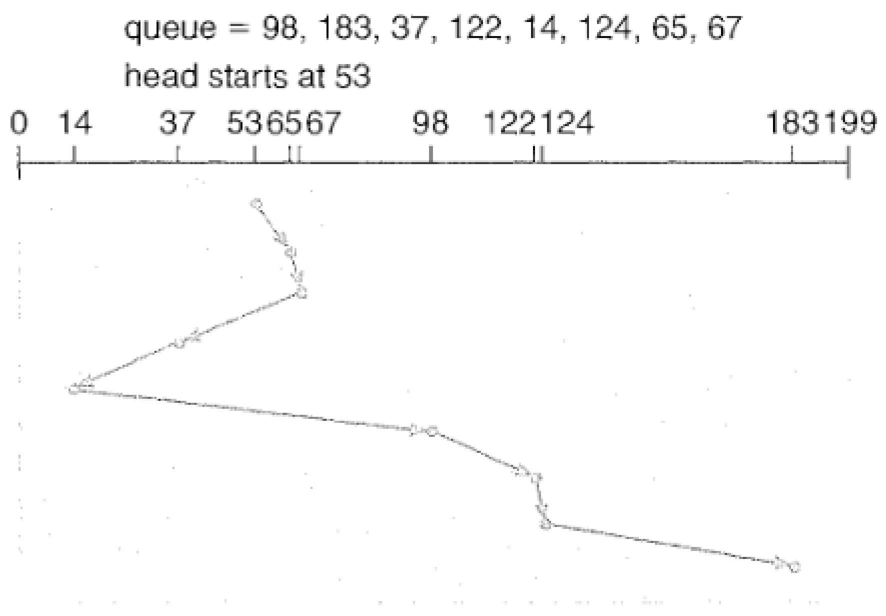
2. **SSTF disk scheduling –**

Shortest seek time first (SSTF) algorithm selects the disk I/O request which requires the least disk arm movement from its current position regardless of the direction. It reduces the total seek time as compared to FCFS.

It allows the head to move to the closest track in the service queue.

The SSTF algorithm selects the request with the least 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.



Total distance the read/write head will traverse

= (65-53)+(67-65)+(67-37)+(37-14)+(98-14)+(122-98)+(124-122)+(183-124)

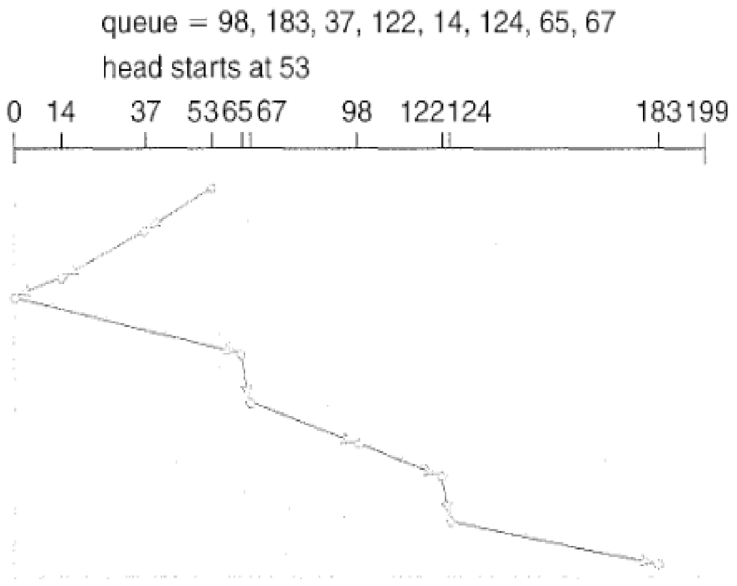
= 12+2+30+23+84+24+2+59

=236

3. **SCAN Scheduling**

In this method, here the disk arm starts at one end of the disk and moves toward the 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.

A disk is having 200 cylinders numbered from 0 to 199. The disk is currently servicing at cylinder 53 and previous request was at 60.



Total distance the read/write head will traverse

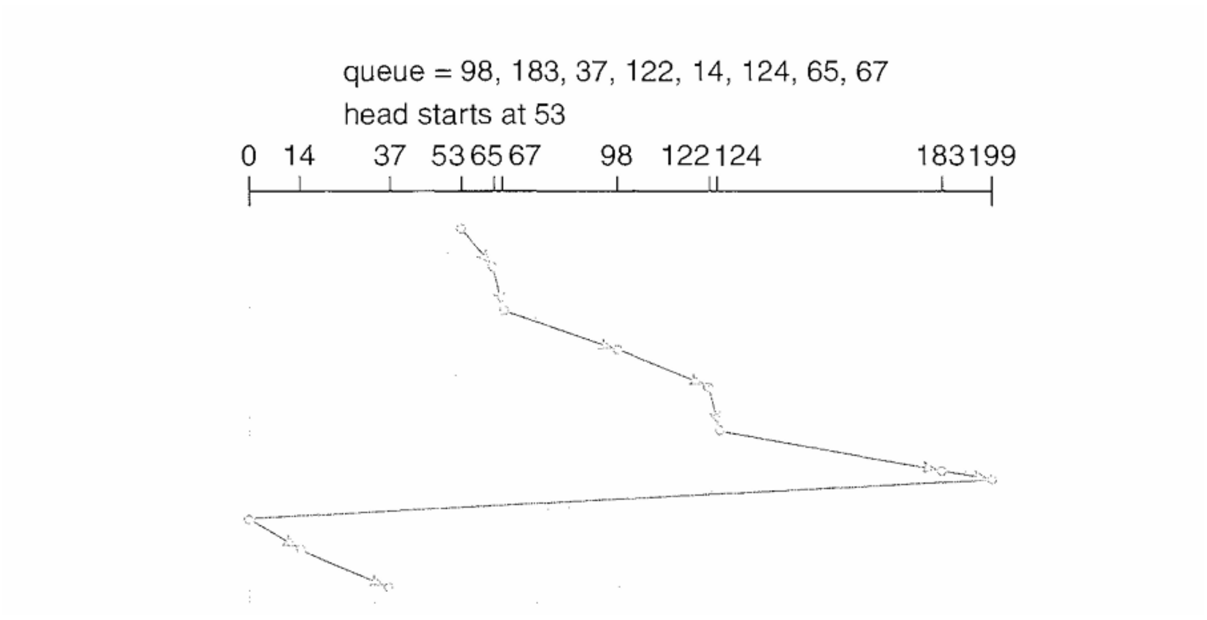
= (53-37) + (37-14) + (14- 0) + (65-0) + (67-65) + (98-67) + (122-98) + (124-122) + (183-124)

= 236

4. **C-SCAN Scheduling**

It 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. The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

- If the cylinder is having maximum number n- 1 (0 to n-1) then the next cylinder number will be 0, as it is circular
- It will move in increasing order of cylinder number.



Total distance the read/write head will traverse

= (65-53) + (67-65) + (98-67) + (122-98) + (124-122) + (183-124) + (198-183) + (14-0)+ (37-14)

= 183

Given so many disk-scheduling algorithms how do we choose the best one?

- SSTF is common and has a natural appeal because it increases performance over FCFS. S
- CAN and C-SCAN perform better for systems that place a heavy load on the disk because they are less likely to cause a starvation problem.
  For any particular list of requests we can define an optimal order of retrieval but the computation needed to find an optimal schedule may not justify the savings over SSTF or SCAN.
    - With any scheduling algorithm, however performance depends heavily on the number and types of requests. For instance, suppose that the queue usually has just one outstanding request. Then all scheduling algorithms behave the same because they have only one choice of where to move the disk head: they all behave like FCFS scheduling. Requests for disk service can be greatly influenced by the file-allocation method. A program reading a contiguously allocated file will generate several requests that are close together on the disk, resulting in limited head movement. A linked or indexed file in contrast may include blocks that are widely scattered on the disk resulting in greater head movement.

Disk Formatting

A new magnetic disk is a blank slate: it is just a platter 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. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an error correcting code (ECC) . When the controller writes a sector of data during normal I/0, the 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 compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad.

The ECC is an error-correcting code because it contains enough information, if only a few bits of data have been corrupted, to enable the controller to identify which bits have changed and calculate what their correct values should be. It then reports a recoverable soft error. The controller automatically does the ECC processing whenever a sector is read or written. Most hard disks are low-level-formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the disk and to initialize the mapping from logical block numbers to defect-free sectors on the disk. For many hard disks, when the disk controller is instructed to low-level-format the disk, it can also be told how many bytes of data space to leave between the header and trailer of all sectors. It is usually possible to choose among a few sizes, such as 256,512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on each track; but it also means that fewer headers and trailers are written on each track and more space is available for user data. Some operating systems can handle only a sector size of 512 bytes.

Before it can use a disk to hold files, the operating system still 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 operating system can treat each partition as though it were a separate disk. For instance, one partition can hold a copy of the operating system's executable code, while another holds user files. The second step is logical formatting or creation of a file system. In this step, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or inodes) and an initial empty directory. To increase efficiency, most file systems group blocks together into larger chunks, frequently called clusters. Disk I/0 is done via blocks, but file system II 0 is done via clusters, effectively assuring that II 0 has more sequential-access and fewer random-access characteristics. Some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the raw disk, and II 0 to this array is termed raw l/0. For example, some database systems prefer raw IIO because it enables them to control the exact disk location where each database record is stored. Raw l/0 bypasses all the file-system services, such as the buffer cache, file locking, prefetching, space allocation, file names, and directories. We can make certain applications more efficient by allowing them to implement their own special-purpose storage services on a raw partition, but most applications perform better when they use the regular file-system services.

Boot Block

For a computer to start running-for instance, when it is powered up or rebooted -it must have an initial program to run. This initial bootstrap program tends to be simple. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system. To do its job, the bootstrap program finds the operating-system kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the operating-system execution. For most computers, the bootstrap is stored in ROM. This location is convenient, because ROM needs no initialization and is at a fixed location that the processor can start executing when powered up or reset. And, since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM hardware chips. For this reason, most systems store a tiny bootstrap loader program in the boot ROM whose only job is to bring in a full bootstrap program from disk. The full bootstrap program can be changed easily: a new version is simply written onto the disk. The full bootstrap program is stored in the "boot blocks" at a fixed location on the disk. A disk that has a boot partition is called a boot disk. The code in the boot ROM instructs the disk controller to read the boot blocks into memory (no device drivers are loaded at this point) and then starts executing that code. The full bootstrap program is more sophisticated than the bootstrap loader in the boot ROM; it is able to load the entire operating system from a non-fixed location on disk and to start the operating system ruru1ing. Even so, the full bootstrap code may be small.

Once the system identifies the boot partition, it reads the first sector from that partition (which is called the boot sector and continues with the remainder of the boot process, which includes loading the various subsystems and system services.

Bad Blocks

Because disks have moving parts and small tolerances they are prone to failure. Sometimes the failure is complete; in this case, 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, such as some disks with IDE controllers, bad blocks are handled manually. For instance, the MS-DOS format command performs logical formatting and, as a part of the process, scans the disk to find bad blocks. If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block. If blocks go bad during normal operation, a special program (such as chkdsk) must be run manually to search for the bad blocks and to lock them away. Data that resided on the bad blocks usually are lost.

More sophisticated disks, such as the SCSI disks used in high-end PCs and most workstations and servers, are smarter about bad-block recovery. The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level formatting at the factory and is updated over the life of the disk. Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as sector sparing or forwarding.

As an alternative to sector sparing some controllers can be instructed to replace a bad block by *sector slipping.*

Here is an example: Suppose that logical block 17 becomes defective and the first available spare follows sector 202. Then, sector slipping remaps all the sectors front 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.

The replacement of a bad block generally is not totally automatic because the data in the bad block are usually lost.

<u>File System</u>

File System provide efficient access to the disk by allowing data to be stored, located and retrieved in a convenient way. A file System must be able to store the file, locate the file and retrieve the file.

Most of the Operating Systems use layering approach for every task including file systems. Every layer of the file system is responsible for some activities.

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

1. Contiguous File Allocation

In this method, files are stored in a continuous block of free space on the disk meaning that all the data for a particular file is stored in one continuous section of the disk. Each file occupies a contiguous set of blocks on the disk.

When a file is created, the operating system searches for a contiguous block of free space large enough to accommodate the file. If such a block is found, the file is stored in that block, and the operating system keeps track of the starting address and the size of the block.

The advantage of contiguous file allocation is that it provides fast access to files, as the operating system only needs to remember the starting address of the file. When a user requests access to a file, the operating system can quickly locate the file's starting address and read the entire file sequentially.

However, contiguous file allocation has some limitations. One significant disadvantage is that it can lead to fragmentation when files are deleted or when new files are created. If a file is deleted, the space it occupied becomes free, but that space may not be contiguous with the remaining free space on the disk. This can result in gaps or fragments of free space scattered throughout the disk, making it difficult for the operating system to find contiguous blocks of free space for new files.

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

2. Linked File Allocation

- Each file is a linked list of disk blocks **which need not be continuous.** The disk block can be scattered anywhere in the disk.
- The directory entry contains a pointer to the starting and ending file block. Each block contains a pointer to the next block occupied by the file allowing the operating system to access the entire file by following the chain of pointers
- The advantage of linked file allocation is that it can accommodate files of any size, as the file can be stored in multiple non-contiguous blocks. It also avoids fragmentation, as files can be stored in any available free space on the disk, without the need to find a contiguous block of free space
- One significant disadvantage is that it can result in slower access times to files, as the operating system needs to follow the chain of pointers to access the entire file. This method may also require more disk space, as each block contains a pointer to the next block in the file. Additionally, if a pointer becomes damaged or lost, it can result in the loss of the entire file, as the operating system cannot access the entire chain of blocks

3. Indexed File Allocation

- In this method, files are stored in non-contiguous blocks, but instead of linking each block together, a special block called index block that contains a list of pointers to each block in the file
- When a file is created, the operating system searches for a series of free blocks that are large enough to store the file and creates an index block that contains pointers to each of those blocks.
- Each block of the file is then stored in a separate block on the disk.
- The advantage of indexed file allocation is that it provides fast access to files, as the operating system only needs to read the index block to locate the file's blocks. This method also avoids fragmentation, as files can be stored in any available free space on the disk, without the need to find a contiguous block of free space. Indexed file allocation also reduces

the risk of data loss, as the index block can be duplicated to provide redundancy.

- One significant disadvantage is that it can result in wasted disk space, as the index block can take up a significant amount of space on the disk. This method also requires more disk space than linked file allocation, as each block of the file is stored separately on the disk.

MAKAUT question

2019

1. *Explain what is indexed allocation of file space on disk. What are the advantages and disadvantages of contiguous allocation*

Free space management

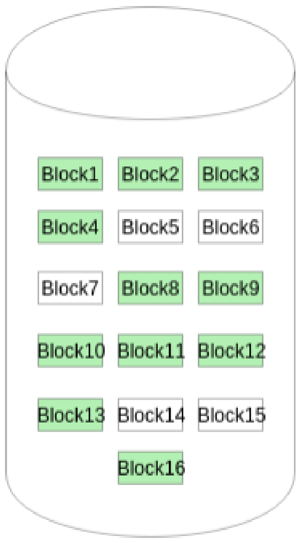
Free space management is a critical aspect of operating systems as it involves managing the available storage space on the hard disk or other secondary storage devices. The operating system uses various techniques to manage free space and optimize the use of storage devices.

The system keeps tracks of the free disk blocks for allocating space to files when they are created. Also, to reuse the space released from deleting the files, free space management becomes crucial. The system maintains a free space list which keeps track of the disk blocks that are not allocated to some file or directory. The free space list can be implemented mainly as:

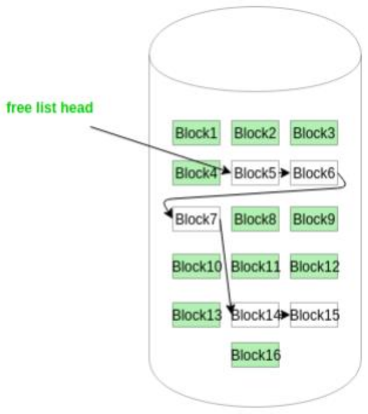
1. **Bitmap or Bit vector -**

A bitmap or bit vector is series or collection of bits where each bit corresponds to a disk block. The bit can take two values: 0 and 1: *0 indicates that the block is free* and 1 indicates an allocated block.

The given instance of disk blocks on the disk in *the figure below* (where green blocks are allocated) can be represented by a bitmap of 16 bits as: **1111000111111001**.



2. **Linked list –**
   This is another technique for free space management. In this technique, linked list of all the free block is maintained. There is a head pointer which points the first free block of the list which is kept in a special location on the disk. This block contains the pointer to the next block and the next block contain the pointer of another next and this process is repeated. By using this disk it is not easy to search the free list. This technique is not sufficient to traverse the list because we have to read each disk block that requires I/O time. So traversing in the free list is not a frequent action.
   In this approach, the free disk blocks are linked together i.e. a free block contains a pointer to the next free block. The block number of the very first disk block is stored at a separate location on disk and is also cached in memory.
   In the figure below, the free space list head points to Block 5 which points to Block 6, the next free block and so on. The last free block would contain a null pointer indicating the end of free list.

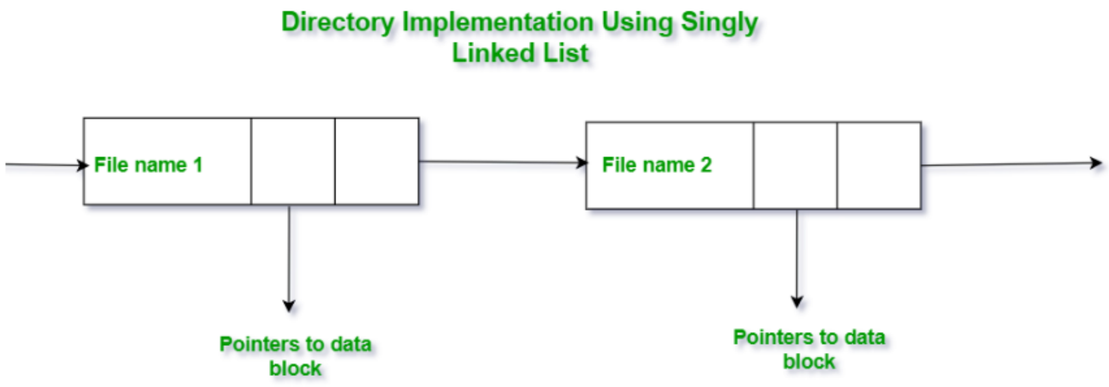


3. **Grouping -**
   This approach stores the address of the free blocks in the first free block. The first free block stores the address of some, say n free blocks. Out of these n blocks, the first n-1 blocks are actually free and the last block contains the address of next free n blocks. An advantage of this approach is that the addresses of a group of free disk blocks can be found easily.

Directory Implementation in Operating System

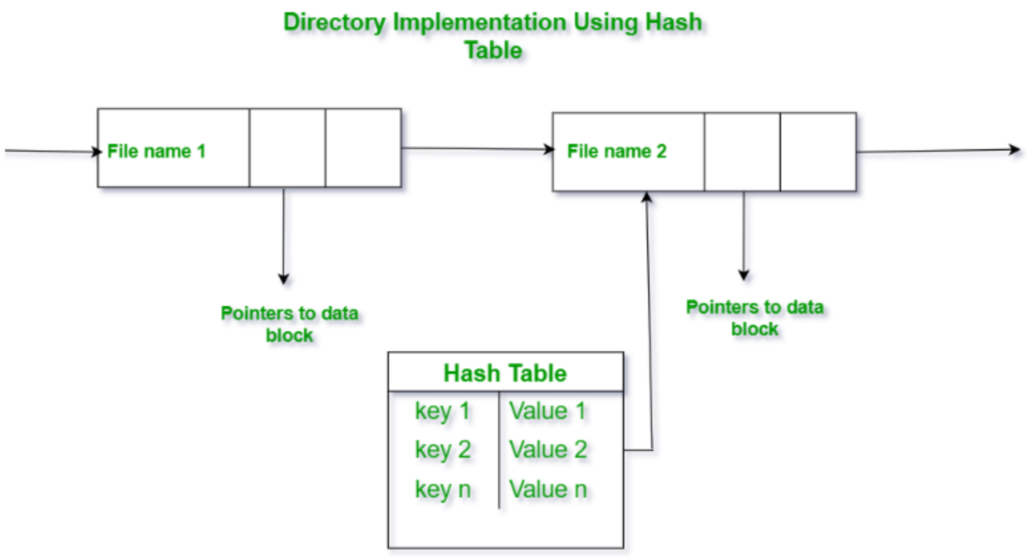
Directory implementation in the operating system can be done using Singly Linked List and Hash table. The efficiency, reliability, and performance of a file system are greatly affected by the selection of directory-allocation and directory-management algorithms. There are numerous ways in which the directories can be implemented.

I. ***Directory Implementation using Singly Linked List***



- All the files in the directory are maintained as singly linked list
- Each file contains the pointers to the data blocks which are assigned to it and the next file in the directory
- When a new file is created the entire list has to be checked such that the new directory does not exist previously, if it does not exist the new directory then can be added to the end of the list or at the beginning of the list
- In order to delete a file, we first search the directory with the name of the file to be deleted. After searching we can delete that file by releasing the space allocated to it.
- To reuse the directory entry we can mark that entry as unused or we can append it to the list of free directories.
- To delete a file linked list is the best choice as it takes less time.

II. Directory Implementation using Hash Table



- It overcome the drawbacks of singly linked list implementation
- This approach suggest to use hash table along with linked list
- In the hash table for each pair in the directory key-value pair is generated. The hash function on the file name determines the key

and this key points to the corresponding file stored in the directory. This method efficiently decreases the directory search time as the entire list will not be searched on every operation. Using the keys the hash table entries are checked and when the file is found it is fetched.

- Now, searching becomes efficient due to the fact that now, entire list will not be searched on every operating. Only hash table entries are checked using the key and if an entry found then the corresponding file will be fetched using the value.