**5.**

**Memory Management**

**Memory Management**

Memory management in an operating system involves various mechanisms and techniques to efficiently manage a computer system's memory resources. The memory management module is a critical component within the operating system responsible for controlling and organizing the computer's memory,

#### Memory Allocation and Deallocation:

* + Allocating memory to processes when requested and deallocating it when no longer needed.

#### Memory Organization:

* + Managing the available memory space, dividing it into partitions or segments to accommodate multiple processes.

#### Address Translation:

* + Mapping logical addresses used by processes to physical addresses in the actual hardware memory.

#### Memory Protection:

Implementing measures to prevent unauthorized access to memory regions, ensuring the security and integrity of date

#### Memory Sharing:

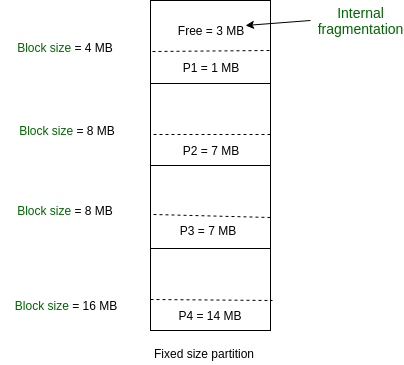
Allowing multiple processes to share certain memory regio

**Memory Partitioning**

Memory partitioning is a technique used in operating systems to manage and allocate memory to multiple processes running concurrently. It involves dividing the physical memory into fixed or variable-sized partitions to accommodate different processes and their memory requirements. There are two main types of memory partitioning: fixed partitioning and dynamic partitioning.

## Fixed Partitioning:

In fixed partitioning, the physical memory is divided into fixed-sized partitions during system initialization. Each partition can accommodate one process. Processes are loaded into these partitions based on their size,



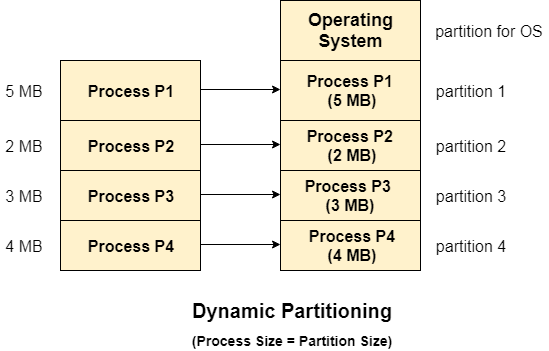
#### Advantages:

* + Simplicity and ease of implementation.
  + Quick and straightforward allocation of memory.

#### Disadvantages:

* + Inefficient memory utilization due to internal fragmentation.
  + Difficulty accommodating processes of varying sizes.

## Dynamic Partitioning:

Dynamic partitioning involves allocating memory dynamically based on the size of the processes. When a new process arrives,

#### Advantages:

* + Efficient memory utilization by reducing internal fragmentation.
  + Ability to accommodate processes of varying sizes.

**Disadvantage:-**

* + Overhead of managing and updating information about partitions and free spaces.
  + Possibility of external fragmentation, where small blocks of free memory become scattered and cannot be used

**Fragmentation**

Fragmentation in the context of operating systems refers to the phenomenon where the total memory space is broken up into smaller, non-contiguous blocks, making it challenging to efficiently allocate these fragmented blocks for new processes or data storage. Fragmentation can occur in both main memory (RAM) and secondary storage (disk).

There are two main types of fragmentation:

1. **Internal Fragmentation:**

Internal fragmentation occurs when memory is allocated in fixed-size blocks or partitions, and the allocated memory may be larger than what the process actually needs. The unused portion within a memory block is wasted, resulting in inefficient memory utilization.

#### Fixed Partitioning:

#### Variable Partitioning:

## External Fragmentation:

External fragmentation occurs when there is enough total free memory to satisfy a memory request, but this free memory is scattered across the system in small, non-contiguous blocks. As a result, a request for a specific amount of memory cannot be fulfilled, even though the aggregate free memory is sufficient.

#### Example:- Imagine you have 100 units of free memory, divided into blocks of 30, 10, 25, and 35 units. If a process requests 50 units of memory, it cannot be accommodated even though there is enough free memory available

**Paging**

**Paging** is a memory management scheme used by modern operating systems to efficiently manage and utilize the computer's physical memory (RAM). It's a technique that allows the operating system to store and retrieve data from the main memory in fixed-size blocks called "pages."

#### Here's a breakdown of how paging works:

1. **Page:** A fixed-size block of memory, typically 4 KB in size, used by the operating system to manage memory. The size of a page is defined by the hardware and is a power of 2.
2. **Frame:** A fixed-size block of physical memory corresponding to the size of a page. Frames are also a power of 2 and match the size of pages.
3. **Page Table:** A data structure maintained by the operating system that maps virtual addresses (used by the processes) to physical addresses (locations in RAM). Each entry in the page table corresponds to a page and stores the frame number where the page is located in physical memory.
4. **Virtual Address Space:** The range of addresses that a process can use, as seen from the perspective of the process. This address space is divided into fixed-size pages.
5. **Physical Address Space:** The range of actual memory addresses in the physical RAM.

**Page Replacement Algorithms**

**Page replacement** algorithms are techniques used by the operating system to manage the limited amount of physical memory (RAM) effectively when there is a need to swap pages in and out of memory due to memory demand. When a page fault occurs and a new page needs to be brought into memory

#### FIFO (First-In-First-Out):

This algorithm removes the page that has been in memory the longest. It's like a queue where the page that came in first is the first to be replaced. However, FIFO does not consider how frequently a page is accessed or how recently it was used.

FIFO replacement algoritham work, Initialization Page Request,

Page Replacement, Updating the Queue:

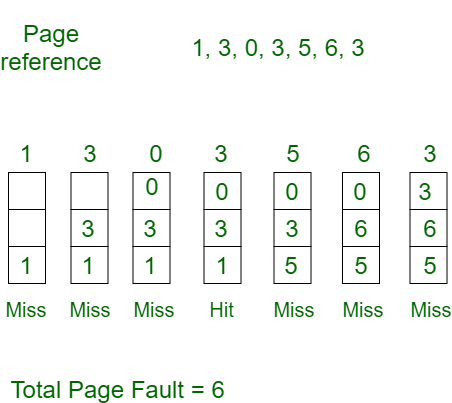
#### LRU (Least Recently Used):

#### The LRU stands for the Least Recently Used. It keeps track of page usage in the memory over a short period of time. It works on the concept that pages that have been highly used in the past are likely to be significantly used again in the future. It removes the page that has not been utilized in the memory for the longest time. LRU is the most widely used algorithm because it provides fewer page faults than the other methods

#### Optimal Page Replacement:

This is an idealized algorithm that replaces the page that will not be used for the longest time in the future. It's used as a reference to measure the efficiency of other page replacement algorithms, although it's not practically implementable because it requires knowledge of future memory accesses.

#### Example -:



**LRU ( Least Recently Used**

.



LRU, which stands for "Least Recently Used," is a page replacement algorithm used in computer operating systems to manage and optimize the use of physical memory (RAM) when executing multiple processes or programs. The primary goal of the LRU algorithm is to minimize page faults by evicting the page that has not been used for the longest period of time.

#### Here's how the LRU page replacement algorithm works:

1. **Maintaining a Page Reference List:** LRU keeps track of the order in which pages are accessed by creating a list or queue, which is sometimes called a "page reference list." This list is usually implemented as a data structure that records the recent page accesses.
2. **Page Access:** Whenever a page is accessed, it is moved to the front of the page reference list to indicate that it's the most recently used page. If a page is already in the list, it is promoted to the front. If a page is not in the list, it is added to the front.
3. **Page Replacement Decision:** When the operating system needs to bring in a new page (due to a page fault) and there's no free space in RAM, the LRU algorithm selects the page at the end of the list (the one that hasn't been used for the longest time) for replacement.
4. **Maintaining List Size:** To ensure that the page reference list doesn't grow infinitely, it is usually limited to a fixed size, which means that when a new page is added to the front of the list, the page at the end of the list is removed.

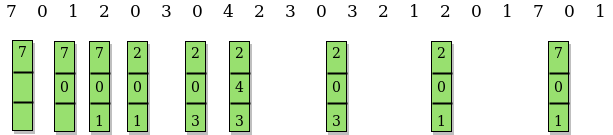
**Optimal Page Replacement**

The Optimal Page Replacement algorithm, also known as the Belady's Algorithm, is a theoretical memory management algorithm used to evaluate the performance of other page replacement algorithms.

* + It is not a practical algorithm for real-world systems due to its reliance on future knowledge, which is impossible to obtain in most situations. Instead, it helps us establish an upper bound on the performance of other page replacement algorithms.
  + The goal of the Optimal Page Replacement algorithm is to minimize the number of page faults.
  + To achieve this, it assumes perfect knowledge of future memory access patterns, meaning it knows which pages will be accessed and when.
  + This idealized knowledge allows the algorithm to make the best possible decisions.

Here's how the Optimal Page Replacement algorithm works:

1. At each memory access request, the algorithm examines the future references to determine which page will be used furthest in the future.
2. It selects the page that will not be used for the longest time (the page that has the farthest future reference).
3. If the memory is full, and a page replacement is required, the algorithm replaces the page that will not be needed for the longest time, according to its future knowledge.



**6.**

**File Management**

**Concept of File**

In an operating system, a "file" is a fundamental unit of data storage that contains information, data, or instructions. Files are organized into directories or folders, and they serve as a means to store, retrieve, and manage data. Each file is characterized by several attributes that provide information about the file and its properties. These attributes can include:

1. **File Name:** This is the human-readable name of the file, which is used to identify and reference it.
2. **File Extension:** The file extension is typically a part of the file name and indicates the file type or format. For example, ".txt" is often used for plain text files, ".jpg" for image files, and ".exe" for executable programs.
3. **File Size:** This attribute specifies the size of the file in bytes or another appropriate unit of measurement, indicating how much storage space the file occupies.
4. **File Location:** The file's path or directory location specifies where the file is stored within the file system's hierarchy. It includes information about the folder(s) in which the file is contained.
5. **Date Created:** This attribute indicates the date and time when the file was initially created.
6. **Date Modified:** This attribute indicates the date and time when the file was last modified or updated.
7. **Date Accessed:** Some operating systems track the date and time when the file was last accessed, although this attribute is often disabled by default due to performance considerations.
8. **File Permissions:** File permissions define who can access, read, write, or execute the file. These permissions are usually categorized into read, write, and execute permissions for the owner, group, and others.
9. **File Type:** This attribute provides information about the type or format of the file, which can be used by the operating system and associated applications to determine how to handle the file.
10. **File Owner:** Every file is associated with an owner, typically the user account that created the file. The owner has special privileges and control over the file's permissions.

**File Operations**

In an operating system, file operations are a set of actions that can be performed on files to create, read, write, update, delete, and manage them. These operations are essential for managing and manipulating data within the file system. Here are some of the most common file operations:

1. **File Creation:** Creating a file involves specifying a file name and optionally choosing a location within the file system. The operating system reserves space for the file and assigns initial attributes such as creation date and permissions.
2. **File Reading:** Reading from a file allows you to retrieve data from an existing file. Reading can be done sequentially or randomly, depending on the file access method. The operating system provides system calls and APIs for reading data from files.
3. **File Writing:** Writing to a file involves adding or modifying data within an existing file. You can append data to the end of the file or overwrite existing content. File writing is often subject to file permissions, which control who can write to a file.
4. **File Updating:** Updating a file means modifying specific parts of its content, typically within the file's structure. For instance, updating a database file might involve changing a record's data without affecting the rest of the file.
5. **File Deletion:** Deleting a file removes it from the file system, freeing up storage space. Care should be taken when deleting files, as they may not always be recoverable from the recycling bin or trash.

**File System Structure**

In operating systems, file systems can be structured in different ways to organize and store data. Three common structures are byte sequence, tree sequence, and record sequence. Each of these structures has its own characteristics and use cases:

**1. Byte Sequence File System:**

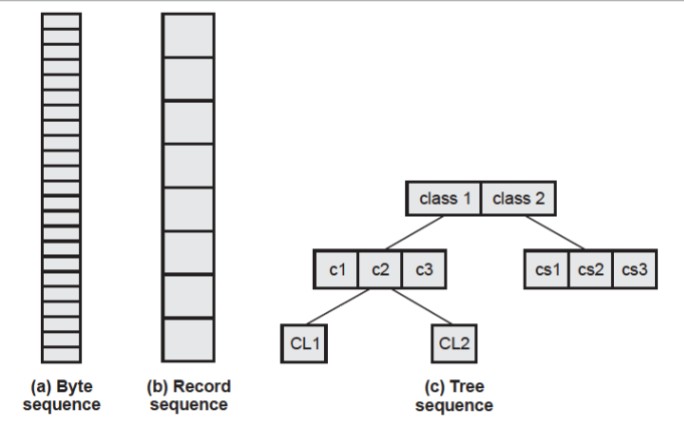
* + In a byte sequence file system, data is organized as a continuous stream of bytes.
  + This structure is the simplest and most straightforward, as it treats files as unstructured sequences of bytes with no inherent hierarchy.
  + There are no directories or folders in a byte sequence file system, only files.
  + Files are identified by their position within the sequence of bytes, usually starting from the beginning of the storage medium.
  + This structure is often used in embedded systems or where simplicity and minimal overhead are required. However, it lacks the ability to efficiently organize and categorize data.

**2. Tree Sequence File System:**

* + In a tree sequence file system, data is organized in a hierarchical tree-like structure.
  + Files and directories (folders) are represented as nodes in the tree, with the root node at the top and subdirectories branching off from parent directories.
  + This structure allows for a logical organization of files and makes it easy to navigate and locate specific data.
  + Each file or directory has a unique path that specifies its location within the tree (e.g., "C:\Users\John\Documents\file.txt" in Windows or "/home/jane/documents/file.txt" in Unix-like systems).
  + Tree sequence file systems are commonly used in desktop and server operating systems, providing efficient data organization and retrieval.

**3. Record Sequence File System:**

* + A record sequence file system stores data as records, where each record consists of a fixed-size block or entry.
  + This structure is well-suited for applications where data is organized into records, such as databases.
  + Records are typically identified by a record number or key and can be read, written, or updated individually.
  + Record sequence file systems provide efficient access to specific data within a file without the need to read or write the entire file.
  + This structure is commonly used in database management systems (DBMS) and other data-centric applications.



**File Access Methods**

File access methods in operating systems refer to the techniques or approaches used to read and write data within files. These methods determine how data is accessed and retrieved from storage media, such as hard drives or solid-state drives. There are primarily two common file access methods: sequential access and direct (random) access.

**1. Sequential Access:**

In sequential access, data is read or written one item at a time in a linear or sequential manner. To access a specific piece of data, you must start at the beginning of the file and read or write sequentially until you reach the desired location.

* + - Sequential access is similar to reading a book page by page, starting from the first page and moving forward until you reach the desired page.
    - This method is suitable for tasks that involve reading or writing data in a specific order, such as scanning a text file line by line or processing a data file sequentially.
    - Example: Reading a text file line by line in a sequential manner. You start at the beginning of the file and read each line until you find the one you're looking for.

**File Contents:**

Line 1: This is the first line. Line 2: This is the second line. Line 3: This is the third line.

Line 4: This is the fourth line.

**2. Direct (Random) Access:**

Direct access, also known as random access, allows you to read or write data at a specific location within the file without the need to traverse the entire file sequentially.

* + Each data item within the file is associated with an address or index, allowing you to directly access the desired item.
  + This method is suitable for tasks that require quick access to specific data within a file, such as database systems or indexed data structures.
  + Example: Accessing a specific record in a database file by its record number. You don't need to read through all the records to find the one you're interested in.

Record 1: User: Alice, Age: 28, Email: [alice@example.com](mailto:alice@example.com) Record 2: User: Bob, Age: 35, Email: [bob@example.com](mailto:bob@example.com) Record 3: User: Carol, Age: 42, Email: [carol@example.com](mailto:carol@example.com)

**File Allocation**

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.

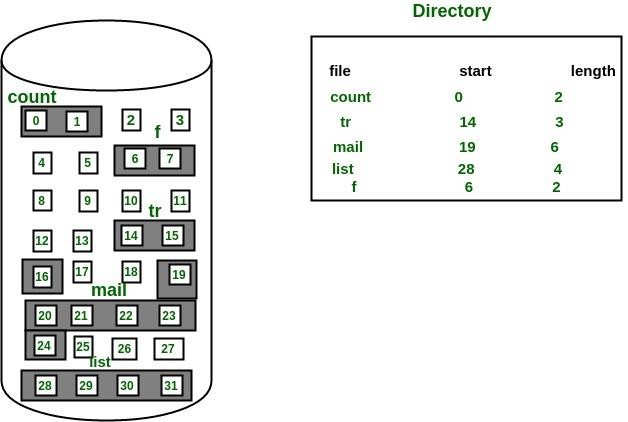
**1. Contiguous Allocation**

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.

#### Advantages:



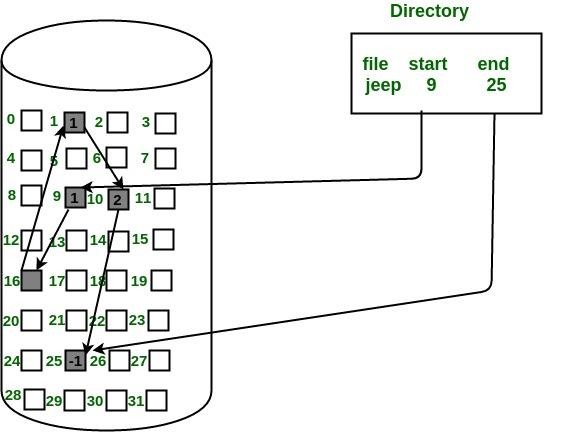
* Both the Sequential and Direct Accesses are supported by this. For direct access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k).
* This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

#### 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*

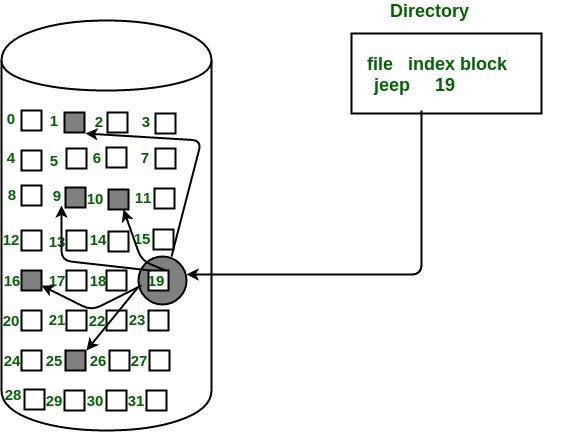
*block.*

#### 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:

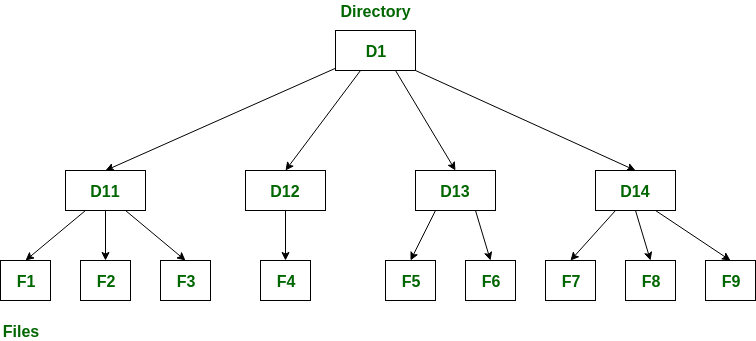
#### Advantages:

* 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.

**Directory Structure**

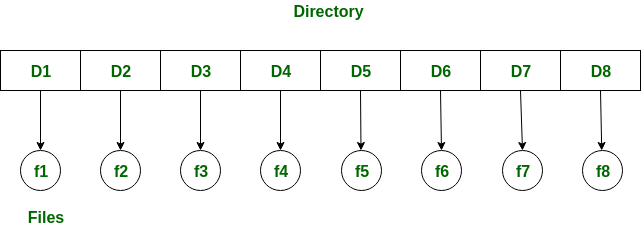


A **directory** is a container that is used to contain folders and files. It organizes files and folders in a hierarchical manner.

Following are the logical structures of a directory, each providing a solution to the problem faced in previous type of directory structure.

1. **Single-level directory:**

* If the files are smaller in size, searching will become faster.



The single-level directory is the **simplest directory structure**. In it, all files are contained in the same directory which makes it easy to support and understand.

A single level directory has a significant limitation, however, when the number of files increases or when the system has more than one user. Since all the files are in the same directory, they must have a **unique name**. If two users

call their dataset test, then the unique name rule violated.

**Advantages:**

* Since it is a single directory, so its implementation is very easy.
* The operations like file creation, searching, deletion, updating are very easy in such a directory structure.
* **Logical Organization**: Directory structures help to logically organize files and directories in a hierarchical structure. This provides an easy way to

navigate and manage files, making it easier for users to access the data they need.

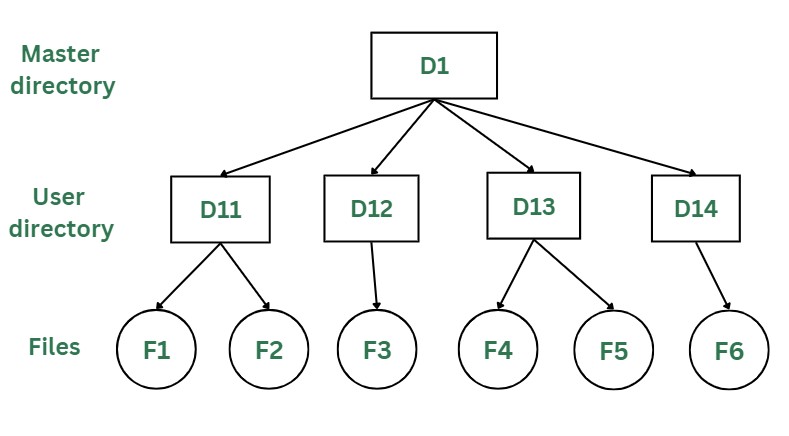
* **Increased Efficiency:** Directory structures can increase the efficiency of the file system by reducing the time required to search for files. This is because directory structures are optimized for fast file access, allowing users to quickly locate the file they need.
* **Improved Security**: Directory structures can provide better security for files by allowing access to be restricted at the directory level. This helps to prevent unauthorized access to sensitive data and ensures that important files are protected.
* **Facilitates Backup and Recovery**: Directory structures make it easier to backup and recover files in the event of a system failure or data loss. By storing related files in the same directory, it is easier to locate and backup all the files that need to be protected.
* **Scalability:** Directory structures are scalable, making it easy to add new directories and files as needed. This helps to accommodate growth in the system and makes it easier to manage large amounts of data.

#### Disadvantages:

* There may chance of name collision because two files can have the same name.
* Searching will become time taking if the directory is large.
* This can not group the same type of files together.

1. **Two-level directory:**

As we have seen, a single level directory often leads to confusion of files names among different users. The solution to this problem is to create a **separate directory for each user**.



In the two-level directory structure, each user has their own **user files directory (UFD).** The UFDs have similar structures, but each lists only the files of a single user. System’s **master file directory (MFD*)*** is searched

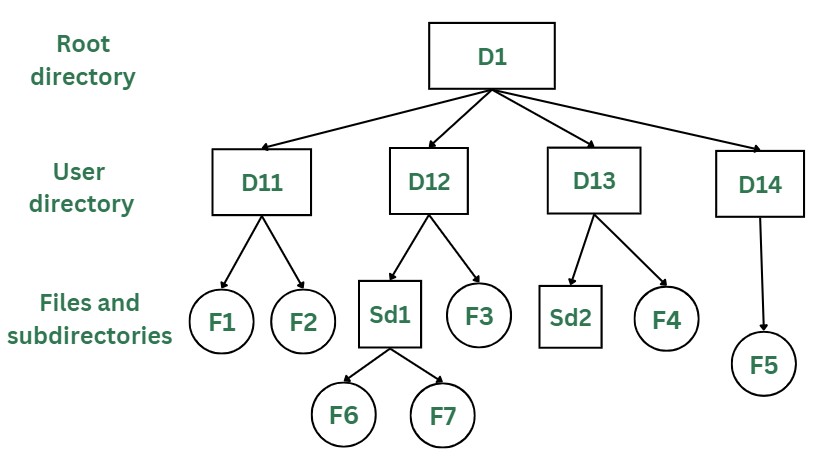
whenever a new user id is created.

***Advantages:***

* The main advantage is there can be more than two files with same name, and would be very helpful if there are multiple users.
* A security would be there which would prevent user to access other user’s files.
* Searching of the files becomes very easy in this directory structure.

##### Disadvantages:

* As there is advantage of security, there is also disadvantage that the user cannot share the file with the other users.
* Unlike the advantage users can create their own files, users don’t have the ability to create subdirectories.
* Scalability is not possible because one use can’t group the same types of files together.



### Tree Structure/ Hierarchical Structure:

Tree directory structure of operating system is most commonly used in our **personal computers**. User can create files and subdirectories too, which was a disadvantage in the previous directory structures.

* + This directory structure resembles a real tree upside down, where the **root directory** is at the peak. This root contains all the directories for each user. The users can create subdirectories and even store files in their directory.
  + A user do not have access to the root directory data and cannot modify it. And, even in this directory the user do not have access to other user’s directories. The structure of tree directory is given below which shows how there are files and subdirectories in each user’s directory.

##### Advantages:

* This directory structure allows subdirectories inside a directory.
* The searching is easier.
* File sorting of important and unimportant becomes easier.
* This directory is more scalable than the other two directory structures explained.

##### Disadvantages:

* As the user isn’t allowed to access other user’s directory, this prevents the file sharing among users.
* As the user has the capability to make subdirectories, if the number of subdirectories increase the searching may become complicated.
* Users cannot modify the root directory data.
* If files do not fit in one, they might have to be fit into other directories.

**Raid Levels -:**

RAID, or “Redundant Arrays of Independent Disks” is a technique which makes use of a combination of multiple disks instead of using a single disk for increased performance, data redundancy or both. The term was coined by David Patterson, Garth A. Gibson, and Randy Katz at the University of California, Berkeley in 1987.

It is a way of storing the same data in different places on multiple hard disks or solid-state drives to protect data in the case of a drive failure. A RAID system consists of two or more drives working in parallel. These can be hard discs, but there is a trend to use SSD technology (Solid State Drives).

RAID combines several independent and relatively small disks into single storage of a large size. The disks included in the array are called **array members**. The disks can combine into the array in different ways, which are known as **RAID levels**.

**How RAID Works**

RAID works by placing data on multiple disks and allowing input/output operations to overlap in a balanced way, improving performance. Because various disks increase the mean time between failures (MTBF), storing data redundantly also increases fault tolerance.

RAID arrays appear to the operating system as a single logical drive. RAID employs the techniques of disk mirroring or disk striping.

* Disk Mirroring will copy identical data onto more than one drive.
* Disk Striping partitions help spread data over multiple disk drives. Each drive's storage space is divided into units ranging from 512 bytes up to several megabytes. The stripes of all the disks are interleaved and addressed in order.
* Disk mirroring and disk striping can also be combined in a RAID array.

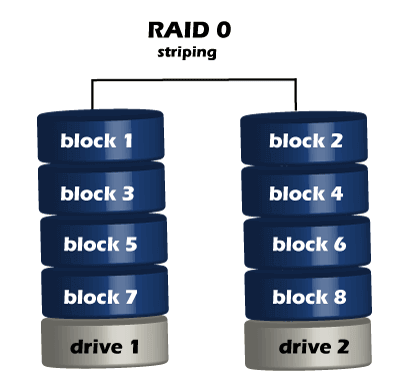
**Levels of RAID**

Many different ways of distributing data have been standardized into various RAID levels. Each RAID level is offering a trade-off of data protection, system

performance, and storage space. The number of levels has been broken into three categories, standard, nested, and non-standard RAID levels.

## RAID 0 (striped disks)

RAID 0 is taking any number of disks and merging them into one large volume. It will increase speeds as you're reading and writing from multiple disks at a time. But all data on all disks is lost if any one disk fails. An individual file can then use the speed and capacity of all the drives of the array. The downside to RAID 0, though, is that it is NOT redundant. The loss of any individual disk will cause complete data loss. This RAID type is very much less reliable than having a single disk.

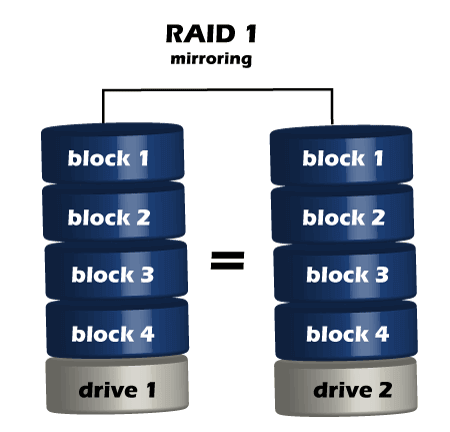


There is rarely a situation where you should use RAID 0 in a server environment. You can use it for cache or other purposes where speed is essential, and reliability or data loss does not matter at all.

## RAID 1 (mirrored disks)

It duplicates data across two disks in the array, providing full redundancy. Both disks are store exactly the same data, at the same time, and at all times. Data is not lost as long as one disk survives. The total capacity of the array equals the capacity of the smallest disk in the array. At any given instant, the contents of both disks in the array are identical.

RAID 1 is capable of a much more complicated configuration. The point of RAID 1 is primarily for redundancy. If you completely lose a drive, you can still stay up and running off the other drive.

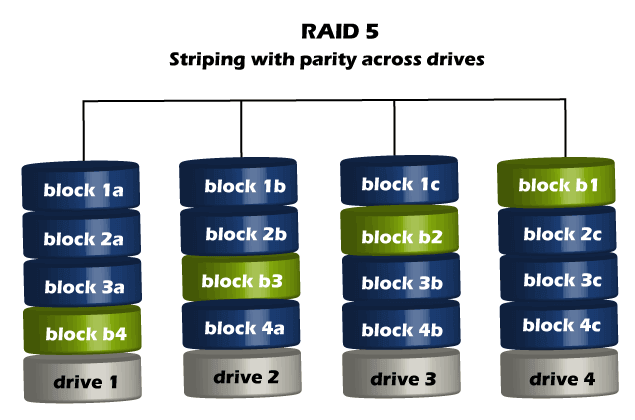


If either drive fails, you can then replace the broken drive with little to no downtime. RAID 1 also gives you the additional benefit of increased read performance, as data can read off any of the drives in the array. The downsides

are that you will have slightly higher write latency. Since the data needs to be written to both drives in the array, you'll only have a single drive's available capacity while needing two drives.

## RAID 5(striped disks with single parity)

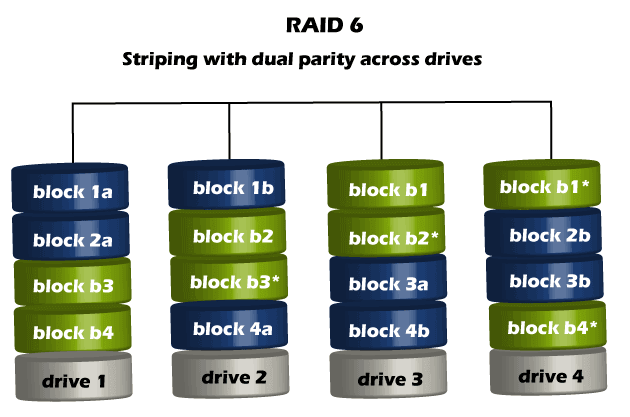
RAID 5 requires the use of at least three drives. It combines these disks to protect data against loss of any one disk; the array's storage capacity is reduced by one disk. It strips data across multiple drives to increase performance. But, it also adds the aspect of redundancy by distributing parity information across the disks.



#### RAID 6 (Striped disks with double parity)

RAID 6 is similar to RAID 5, but the parity data are written to two drives. The use of additional parity enables the array to continue to function even if two

disks fail simultaneously. However, this extra protection comes at a cost. RAID 6 has a slower write performance than RAID 5.



The chances that two drives break down at the same moment are minimal. However, if a drive in a RAID 5 system died and was replaced by a new drive, it takes a lot of time to rebuild the swapped drive. If another drive dies during that time, you still lose all of your data. With RAID 6, the RAID array will even survive tha