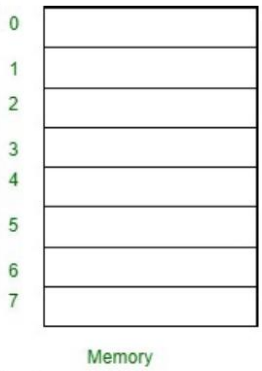
UNIT - 3

What is Address Binding:

The mapping of data and computer instructions to actual memory locations is known as address binding. In computer memory, logical and physical addresses are employed. Additionally, the OS handles this aspect of computer memory management on behalf of programs that need memory access. Let's consider the following example given below for better understanding. Consider a program P1 has a set of instructions such that I1, I2, I3, I4, and program counter values are 10, 20, 30, and 40 respectively.

Program P1
I1 --> 10
I2 --> 20
I3 --> 30
I4 --> 40

Program Counter = 10, 20, 30, 40



Why Do We Need Address Binding?

- Memory Management: Address binding is critical for memory management effectively within a machine.
- Symbol Resolution: Address binding resolves symbolic references in a program to real memory addresses. This method permits the linker/loader to attach program modules, libraries, or functions, and execute correctly.
- Code Relocation: Address binding allows code relocation, allowing executable applications to be loaded into memory at unique addresses whenever they run.
- Dynamic Memory Allocation: Address binding helps dynamic memory allocation, allowing packages to request and release memory dynamically at some stage in runtime. Functions used for dynamic memory allocation are malloc() and free() in programming languages.

Types of Address Binding

The mapping of data and computer instructions to actual memory locations is known as address binding. In computer memory, logical and physical addresses are employed. Address Binding is divided into three types as follows:

- Compile-time Address Binding
- Load time Address Binding
- Execution time Address Binding

Compile-Time Address Binding
If the compiler is responsible for performing address binding then it is called compile-time address binding. During the compilation stage of a program, compile-time binding sometimes referred to as static binding, links symbolic addresses with physical memory addresses. Before the program runs, the addresses are found and fixed. For functions and global variables that have a fixed memory location during program execution, this kind of binding is frequently utilized. Compile-time binding has the benefit of being straightforward and effective because the addresses are known ahead of time. Its inability to adjust to runtime changes stems from the addresses staying the same throughout program execution. The compiler requires interaction with an OS memory manager to perform compile-time address binding.

Load Time Address Binding
It will be done after loading the program into memory. This type of address binding will be done by the OS memory manager i.e. loader. The address binding procedure is postponed by load-time binding until the program is loaded into memory to be executed. The linker and loader assign memory addresses to variables and functions during the loading phase by the memory capacity and the functions' needs. External references are resolved by the linker, which also replaces symbolic locations with their actual physical addresses. Compared to compile-time binding, load-time binding offers greater flexibility because the addresses can be changed by particular runtime circumstances. It provides dynamic libraries and lets the application adjust to variations in memory available.

Execution Time or Dynamic Address Binding
It will be postponed even after loading the program into memory. The program will be kept on changing the locations in memory until the time of program execution. During program execution, address binding is handled by runtime binding, sometimes referred to as dynamic binding. More flexibility is possible with this kind of binding since memory addresses can be dynamically allocated and deallocated as needed. In dynamic and object-oriented programming languages, where the memory layout might change while the program is running, runtime binding is frequently employed. When a program uses runtime binding, it resolves symbolic addresses based on the program's present state. This improves the program's flexibility and adaptability by enabling late binding of functions, polymorphism, and dynamic memory allocation.

Memory Management:

Logical and Physical Address in Operating System
A logical address is generated by the CPU while a program is running. The logical address is a virtual address as it does not exist physically, therefore, it is also known as a Virtual Address. The physical address describes the precise position of necessary data in a memory. Before they are used, the MMU must map the logical address to the physical address. In operating systems, logical and physical addresses are used to manage and access memory.

What is a Logical Address?
A logical address, also known as a virtual address, is an address generated by the CPU during program execution. It is the address seen by the process and is relative to the program's address space. The process accesses memory using logical addresses, which are translated by the operating system into physical addresses. An address that is created by the CPU while a program is running is known as a logical address. Because the logical address is virtual—that

is, it doesn't exist physically—it is also referred to as such. The CPU uses this address as a reference to go to the actual memory location. All logical addresses created from a program's perspective are referred to as being in the "logical address space". This address is used as a reference to access the physical memory location by CPU. The term Logical Address Space is used for the set of all logical addresses generated by a program's perspective.

What is a Physical Address?

A physical address is the actual address in the main memory where data is stored. It is a location in physical memory, as opposed to a virtual address. Physical addresses are used by the Memory Management Unit (MMU) to translate logical addresses into physical addresses. The user must use the corresponding logical address to go to the physical address rather than directly accessing the physical address. For a computer program to function, physical memory space is required. Therefore, the logical address and physical address need to be mapped before the program is run.

The term "physical address" describes the precise position of necessary data in a memory. Before they are used, the MMU must map the logical address to the physical address. This is because the user program creates the logical address and believes that the program is operating in this logical address. However, the program requires physical memory to execute. All physical addresses that match the logical addresses in a logical address space are collectively referred to as the "physical address space"

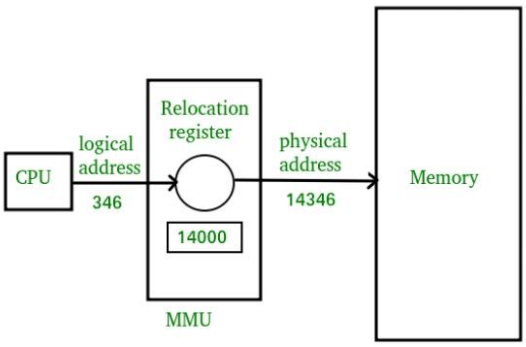
The translation from logical to physical addresses is performed by the operating system's memory management unit (MMU) within the computer's hardware architecture. The MMU uses a page table to translate logical addresses into physical addresses. The page table maps each logical page number to a physical frame number. While the operating system plans this process, it's important to note that the MMU itself is a hardware component separate from the software-based elements of the operating system.

Similarities Between Logical and Physical Addresses in the Operating System

- Both logical and physical addresses are used to identify a specific location in memory.
- Both logical and physical addresses can be represented in different formats, such as binary, hexadecimal, or decimal.
- Both logical and physical addresses have a finite range, which is determined by the number of bits used to represent them.

Important Points about Logical and Physical Addresses in Operating Systems

- The use of logical addresses provides a layer of abstraction that allows processes to access memory without knowing the physical memory location.
- Logical addresses are mapped to physical addresses using a page table. The page table contains information about the mapping between logical and physical addresses.
- The MMU translates logical addresses into physical addresses using the page table. This translation is transparent to the process and is performed by hardware.
- The use of logical and physical addresses allows the operating system to manage memory more efficiently by using techniques such as paging and segmentation.



What is Memory Management Unit?

The physical hardware of a computer that manages its virtual memory and caching functions is called the memory management unit (MMU). The MMU is sometimes housed in a separate Integrated Chip (IC), but it is typically found inside the central processing unit (CPU) of the computer. The MMU receives all inputs for data requests and decides whether to retrieve the data from ROM or RAM storage.

Difference Between Logical address and Physical Address:

Parameter	LOGICAL ADDRESS	PHYSICAL ADDRESS
Basic	generated by CPU	location in a memory unit
Address Space	Logical Address Space is set of all logical addresses generated by CPU in reference to a program.	Physical Address is set of all physical addresses mapped to the corresponding logical addresses.
Visibility	User can view the logical address of a program.	User can never view physical address of program.
Generation	generated by the CPU	Computed by MMU
Access	The user can use the logical address to access the physical address.	The user can indirectly access physical address but not directly.
Editable	Logical address can be change.	Physical address will not change.
Also called	virtual address.	real address.

Memory Allocation Methods:
In operating systems, memory allocation refers to the process of assigning memory to different processes or programs running on a computer system. There are two types of memory allocation techniques that operating systems use: contiguous and non-contiguous memory allocation. In contiguous memory allocation, memory is assigned to a process in a contiguous block. In non-contiguous memory allocation, memory is assigned to a process in non-adjacent blocks.

Contiguous Memory Allocation
Contiguous memory allocation is a technique where the operating system allocates a contiguous block of memory to a process. This memory is allocated in a single, continuous chunk, making it easy for the operating system to manage and for the process to access the memory. Contiguous memory allocation is suitable for systems with limited memory sizes and where fast access to memory is important.

Contiguous memory allocation can be done in two ways

- **Fixed Partitioning** − In fixed partitioning, the memory is divided into fixed-size partitions, and each partition is assigned to a process. This technique is easy to implement but can result in wasted memory if a process does not fit perfectly into a partition.
- **Dynamic Partitioning** − In dynamic partitioning, the memory is divided into variable size partitions, and each partition is assigned to a process. This technique is more efficient as it allows the allocation of only the required memory to the process, but it requires more overhead to keep track of the available memory.

Advantages of Contiguous Memory Allocation

- **Simplicity** − Contiguous memory allocation is a relatively simple and straightforward technique for memory management. It requires less overhead and is easy to implement.
- **Efficiency** − Contiguous memory allocation is an efficient technique for memory management. Once a process is allocated contiguous memory, it can access the entire memory block without any interruption.
- **Low fragmentation** − Since the memory is allocated in contiguous blocks, there is a lower risk of memory fragmentation. This can result in better memory utilization, as there is less memory wastage.

Disadvantages of Contiguous Memory Allocation

- **Limited flexibility** − Contiguous memory allocation is not very flexible as it requires memory to be allocated in a contiguous block. This can limit the amount of memory that can be allocated to a process.
- **Memory wastage** − If a process requires a memory size that is smaller than the contiguous block allocated to it, there may be unused memory, resulting in memory wastage.
- **Difficulty in managing larger memory sizes** − As the size of memory increases, managing contiguous memory allocation becomes more difficult. This is because finding a contiguous block of memory that is large enough to allocate to a process becomes challenging.
- **External Fragmentation** − Over time, external fragmentation may occur as a result of memory allocation and deallocation, which may result in non − contiguous blocks of free memory scattered throughout the system.

Overall, contiguous memory allocation is a useful technique for memory management in certain circumstances, but it may not be the best solution in all situations, particularly when working with larger amounts of memory or if flexibility is a priority.

Non-contiguous Memory Allocation

Non-contiguous memory allocation, on the other hand, is a technique where the operating system allocates memory to a process in non-contiguous blocks. The blocks of memory allocated to the process need not be contiguous, and the operating system keeps track of the various blocks allocated to the process. Non-contiguous memory allocation is suitable for larger memory sizes and where efficient use of memory is important.

Non-contiguous memory allocation can be done in two ways

- **Paging** − In paging, the memory is divided into fixed-size pages, and each page is assigned to a process. This technique is more efficient as it allows the allocation of only the required memory to the process.
- **Segmentation** − In segmentation, the memory is divided into variable-sized segments, and each segment is assigned to a process. This technique is more flexible than paging but requires more overhead to keep track of the allocated segments.

Non-contiguous memory allocation is a memory management technique that divides memory into non-contiguous blocks, allowing processes to be allocated memory that is not necessarily contiguous. Here are some of the advantages and disadvantages of non-contiguous memory allocation −

Advantages of Non-Contiguous Memory Allocation

- **Reduced External Fragmentation** − One of the main advantages of non-contiguous memory allocation is that it can reduce external fragmentation, as memory can be allocated in small, non-contiguous blocks.
- **Increased Memory Utilization** − Non-contiguous memory allocation allows for more efficient use of memory, as small gaps in memory can be filled with processes that need less memory.
- **Flexibility** − This technique allows for more flexibility in allocating and deallocating memory, as processes can be allocated memory that is not necessarily contiguous.
- **Memory Sharing** − Non-contiguous memory allocation makes it easier to share memory between multiple processes, as memory can be allocated in non-contiguous blocks that can be shared between multiple processes.

Disadvantages of Non-Contiguous Memory Allocation

- **Internal Fragmentation** − One of the main disadvantages of non-contiguous memory allocation is that it can lead to internal fragmentation, as memory can be allocated in small, non-contiguous blocks that are not fully utilized.
- **Increased Overhead** − This technique requires more overhead than contiguous memory allocation, as the operating system needs to maintain data structures to track memory allocation.
- **Slower Access** − Access to memory can be slower than contiguous memory allocation, as memory can be allocated in non-contiguous blocks that may require additional steps to access.

In summary, non-contiguous memory allocation has advantages such as reduced external fragmentation, increased memory utilization, flexibility, and memory sharing. However, it also has disadvantages such as internal fragmentation, increased overhead, and slower access to memory. Operating systems must carefully consider the trade-offs between these advantages and disadvantages when selecting memory management techniques.

Difference between contagious and non-contagious memory allocation in operating system

Aspect	Contiguous Memory Allocation	Non-Contiguous Memory Allocation
Method	Allocates memory in a contiguous block to a process	Allocates memory to a process in non-contiguous blocks
Block Size	Memory allocated in a single, continuous chunk	Memory allocated in noncontiguous blocks of varying sizes
Management	Easy to manage by the operating system	Requires additional overhead and can be more complicated to manage
Memory Usage	May result in memory wastage and external fragmentation	Efficient use of memory and reduces fragmentation within memory blocks
Suitable For	Systems with limited amounts of memory and fast access to memory is important	Larger memory sizes and systems that require more efficient use of memory
Advantages	Simple and efficient technique for memory management	More flexible and efficient technique for larger memory sizes and systems that require more efficient use of memory
Disadvantages	Can be inflexible and result in memory wastage and fragmentation	Requires additional overhead and can be more complicated to manage

What is Virtual Memory:

Virtual Memory is a storage scheme that provides user an illusion of having a very big main memory. This is done by treating a part of secondary memory as the main memory.
In this scheme, User can load the bigger size processes than the available main memory by having the illusion that the memory is available to load the process.
Instead of loading one big process in the main memory, the Operating System loads the different parts of more than one process in the main memory.
By doing this, the degree of multiprogramming will be increased and therefore, the CPU utilization will also be increased.

How Virtual Memory Works?

In modern word, virtual memory has become quite common these days. In this scheme, whenever some pages needs to be loaded in the main memory for the execution and the memory is not available for those many pages, then in that case, instead of stopping the pages from entering in the main memory, the OS search for the RAM area that are least used in the recent times or that are not referenced and copy that into the secondary memory to make the space for the new pages in the main memory.
Since all this procedure happens automatically, therefore it makes the computer feel like it is having the unlimited RAM.

Demand Paging

Demand Paging is a popular method of virtual memory management. In demand paging, the pages of a process which are least used, get stored in the secondary memory.

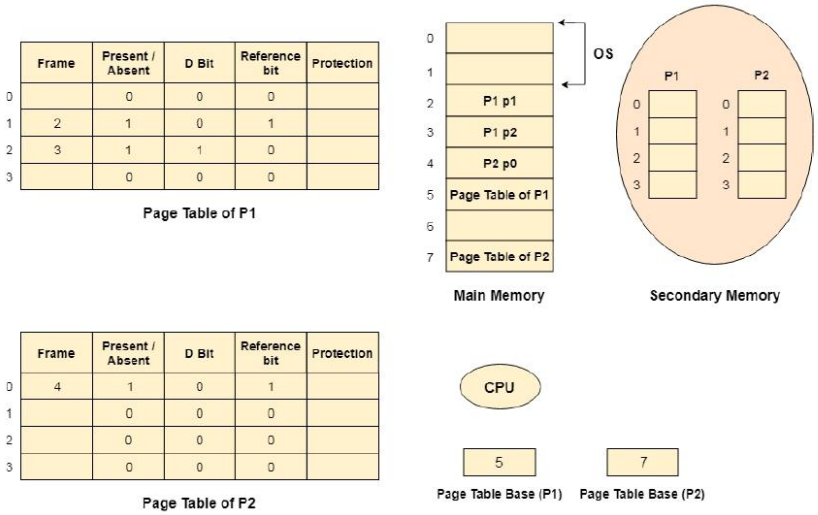
A page is copied to the main memory when its demand is made or page fault occurs. There are various page replacement algorithms which are used to determine the pages which will be replaced. We will discuss each one of them later in detail.

Snapshot of a virtual memory management system

Let us assume 2 processes, P1 and P2, contains 4 pages each. Each page size is 1 KB. The main memory contains 8 frame of 1 KB each. The OS resides in the first two partitions. In the third partition, 1st page of P1 is stored and the other frames are also shown as filled with the different pages of processes in the main memory.

The page tables of both the pages are 1 KB size each and therefore they can be fit in one frame each. The page tables of both the processes contain various information that is also shown in the image.

The CPU contains a register which contains the base address of page table that is 5 in the case of P1 and 7 in the case of P2. This page table base address will be added to the page number of the Logical address when it comes to accessing the actual corresponding entry.



Advantages of Virtual Memory

1. The degree of Multiprogramming will be increased.
2. User can run large application with less real RAM.
3. There is no need to buy more memory RAMs.

Disadvantages of Virtual Memory

1. The system becomes slower since swapping takes time.
2. It takes more time in switching between applications.
3. The user will have the lesser hard disk space for its use.

What is Demand Paging:

According to the concept of Virtual Memory, in order to execute some process, only a part of the process needs to be present in the main memory which means that only a few pages will only be present in the main memory at any time.

However, deciding, which pages need to be kept in the main memory and which need to be kept in the secondary memory, is going to be difficult because we cannot say in advance that a process will require a particular page at particular time.
Therefore, to overcome this problem, there is a concept called Demand Paging is introduced. It suggests keeping all pages of the frames in the secondary memory until they are required. In other words, it says that do not load any page in the main memory until it is required.
Whenever any page is referred for the first time in the main memory, then that page will be found in the secondary memory.

What is a Page Fault?
If the referred page is not present in the main memory, then there will be a miss and the concept is called Page miss or page fault.
The CPU has to access the missed page from the secondary memory. If the number of page fault is very high then the effective access time of the system will become very high.

What is Thrashing?
If the number of page faults is equal to the number of referred pages or the number of page faults are so high so that the CPU remains busy in just reading the pages from the secondary memory then the effective access time will be the time taken by the CPU to read one word from the secondary memory and it will be so high. The concept is called thrashing.
If the page fault rate is PF %, the time taken in getting a page from the secondary memory and again restarting is S (service time) and the memory access time is ma then the effective access time can be given as;

$$EAT = PF \times S + (1 - PF) \times (ma)$$

Page Replacement Algorithms in Operating Systems

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when a new page comes in. Page replacement becomes necessary when a page fault occurs and no free page frames are in memory. in this article, we will discuss different types of page replacement algorithms.

Page Replacement Algorithms

Page replacement algorithms are techniques used in operating systems to manage memory efficiently when the virtual memory is full. When a new page needs to be loaded into physical memory, and there is no free space, these algorithms determine which existing page to replace.

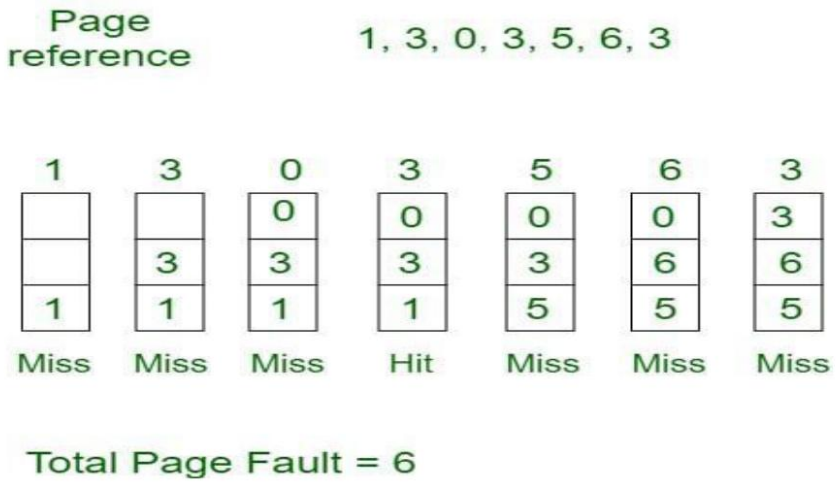
If no page frame is free, the virtual memory manager performs a page replacement operation to replace one of the pages existing in memory with the page whose reference caused the page fault. It is performed as follows: The virtual memory manager uses a page replacement algorithm to select one of the pages currently in memory for replacement, accesses the page table entry of the selected page to mark it as “not present” in memory, and initiates a page-out operation for it if the modified bit of its page table entry indicates that it is a dirty page.

Common Page Replacement Techniques:

First In First Out (FIFO)
Optimal Page replacement
Least Recently Used (LRU)
Most Recently Used (MRU)

First In First Out (FIFO)
This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example 1: Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3-page frames. Find the number of page faults using FIFO Page Replacement Algorithm.



FIFO – Page Replacement

Initially, all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 Page Faults.
when 3 comes, it is already in memory so —> 0 Page Faults. Then 5 comes, it is not available in memory, so it replaces the oldest page slot i.e 1. —> 1 Page Fault. 6 comes, it is also not available in memory, so it replaces the oldest page slot i.e 3 —> 1 Page Fault. Finally, when 3 come it is not available, so it replaces 0 1-page fault.

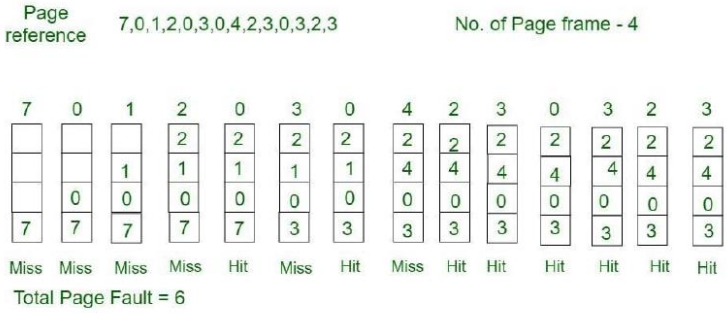
Implementation of FIFO Page Replacement Algorithm

Program for Page Replacement Algorithm (FIFO)

Optimal Page Replacement

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

Example: Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4-page frame. Find number of page fault using Optimal Page Replacement Algorithm.



Optimal Page Replacement

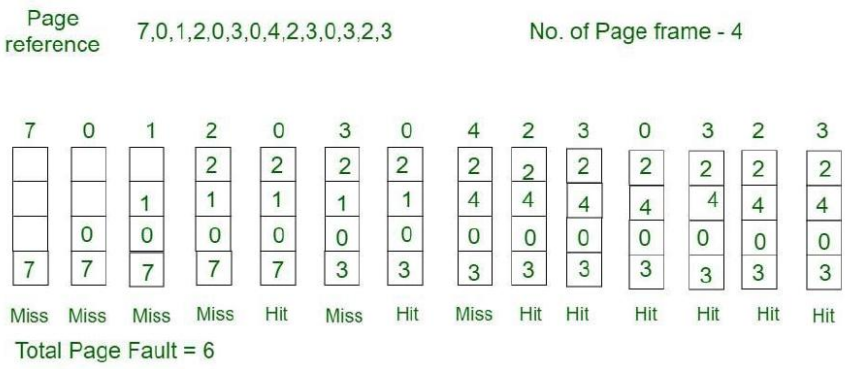
Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults 0 is already there so —> 0 Page fault. when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future—> 1 Page fault. 0 is already there so —> 0 Page fault. 4 will takes place of 1 —> 1 Page Fault.

Now for the further page reference string —> 0 Page fault because they are already available in the memory. Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

Least Recently Used

In this algorithm, page will be replaced which is least recently used.

Example Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4-page frames. Find number of page faults using LRU Page Replacement Algorithm.



Here LRU has same number of page fault as optimal but it may differ according to question.

Least Recently Used – Page Replacement

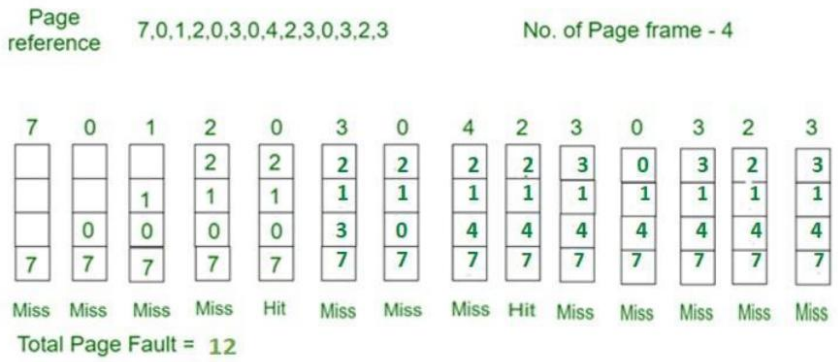
Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults
0 is already there so —> 0 Page fault. when 3 came it will take the place of 7 because it is least recently used —> 1 Page fault
0 is already in memory so —> 0 Page fault.
4 will takes place of 1 —> 1 Page Fault
Now for the further page reference string —> 0 Page fault because they are already available in the memory.

Implementation of LRU Page Replacement Algorithm
Program for Least Recently Used (LRU) Page Replacement algorithm

Most Recently Used (MRU)

In this algorithm, page will be replaced which has been used recently. Belady's anomaly can occur in this algorithm.

Example 4: Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4-page frames. Find number of page faults using MRU Page Replacement Algorithm.



Most Recently Used – Page Replacement

Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults

0 is already their so–> 0 page fault

when 3 comes it will take place of 0 because it is most recently used —> 1 Page fault

when 0 comes it will take place of 3 —> 1 Page fault

when 4 comes it will take place of 0 —> 1 Page fault

2 is already in memory so —> 0 Page fault

when 3 comes it will take place of 2 —> 1 Page fault

when 0 comes it will take place of 3 —> 1 Page fault

when 3 comes it will take place of 0 —> 1 Page fault

when 2 comes it will take place of 3 —> 1 Page fault

when 3 comes it will take place of 2 —> 1 Page fault

File System Implementation in Operating System

A file is a collection of related information. The file system resides on secondary storage and provides efficient and convenient access to the disk by allowing data to be stored, located, and retrieved. File system implementation in an operating system refers to how the file system manages the storage and retrieval of data on a physical storage device such as a hard drive, solid-state drive, or flash drive.
File system implementation is a critical aspect of an operating system as it directly impacts the performance, reliability, and security of the system. Different operating systems use different file system implementations based on the specific needs of the system and the intended use cases. Some common file systems used in operating systems include NTFS and FAT in Windows, and ext4 and XFS in Linux.

Components of File System Implementation
The file system implementation includes several components, including:

- **File System Structure:** The file system structure refers to how the files and directories are organized and stored on the physical storage device. This includes the layout of file systems data structures such as the directory structure, file allocation table, and inodes.
- **File Allocation:** The file allocation mechanism determines how files are allocated on the storage device. This can include allocation techniques such as contiguous allocation, linked allocation, indexed allocation, or a combination of these techniques.
- **Data Retrieval:** The file system implementation determines how the data is read from and written to the physical storage device. This includes strategies such as buffering and caching to optimize file I/O performance.
- **Security and Permissions:** The file system implementation includes features for managing file security and permissions. This includes access control lists (ACLs), file permissions, and ownership management.
- **Recovery and Fault Tolerance:** The file system implementation includes features for recovering from system failures and maintaining data integrity. This includes techniques such as journaling and file system snapshots.

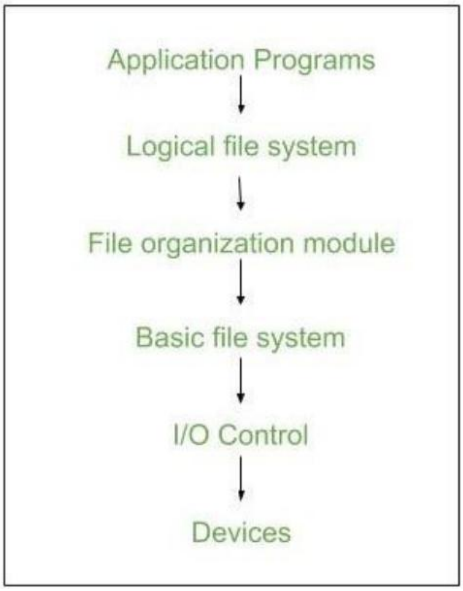
Different Types of File Systems
There are several types of file systems, each designed for specific purposes and compatible with different operating systems. Some common file system types include:

- **FAT32 (File Allocation Table 32):** Commonly used in older versions of Windows and compatible with various operating systems.
- **NTFS (New Technology File System):** Used in modern Windows operating systems, offering improved performance, reliability, and security features.
- **ext4 (Fourth Extended File System):** Used in Linux distributions, providing features such as journaling, large file support, and extended file attributes.

- **HFS+ (Hierarchical File System Plus):** Used in macOS systems prior to macOS High Sierra, offering support for journaling and case-insensitive file names.
- **APFS (Apple File System):** Introduced in macOS High Sierra and the default file system for macOS and iOS devices, featuring enhanced performance, security, and snapshot capabilities.
- **ZFS (Zettabyte File System):** A high-performance file system known for its advanced features, including data integrity, volume management, and efficient snapshots.

Layers in File System

A file system in an operating system is organized into multiple layers, each responsible for different aspects of file management and storage. Here are the key layers in a typical file system:



Layers in File System

- **Application Programs:** This is the topmost layer where users interact with files through applications. It provides the user interface for file operations like creating, deleting, reading, writing, and modifying files. Examples include text editors, file browsers, and command-line interfaces.
- **Logical File system –** It manages metadata information about a file i.e includes all details about a file except the actual contents of the file. It also maintains via file control blocks. File control block (FCB) has information about a file – owner, size, permissions, and location of file contents.
- **File Organization Module –** It has information about files, the location of files and their logical and physical blocks. Physical blocks do not match with logical numbers of logical blocks numbered from 0 to N. It also has a free space that tracks unallocated blocks.

- **Basic File system –** It Issues general commands to the device driver to read and write physical blocks on disk. It manages the memory buffers and caches. A block in the buffer can hold the contents of the disk block and the cache stores frequently used file system metadata.
- **I/O Control level –** Device drivers act as an interface between devices and OS, they help to transfer data between disk and main memory. It takes block number as input and as output, it gives low-level hardware-specific instruction.
- **Devices Layer:** The bottommost layer, consisting of the actual hardware devices. It performs the actual reading and writing of data to the physical storage medium. This includes hard drives, SSDs, optical disks, and other storage devices.

Implementation Issues

- **Management of Disc pace:** To prevent space wastage and to guarantee that files can always be stored in contiguous blocks, file systems must manage disc space effectively. Free space management, fragmentation prevention, and garbage collection are methods for managing disc space.
- **Checking for Consistency and Repairing Errors:** The consistency and error-free operation of files and directories must be guaranteed by file systems. Journaling, check summing, and redundancy are methods for consistency checking and error recovery. File systems may need to perform recovery operations if errors happen in order to restore lost or damaged data.
- **Locking Files and Managing Concurrency:** To prevent conflicts and guarantee data integrity, file systems must control how many processes or users can access a file at once. File locking, semaphore, and other concurrency-controlling methods are available.
- **Performance Optimization:** File systems need to optimize performance by reducing file access times, increasing throughput, and minimizing system overhead. Caching, buffering, prefetching, and parallel processing are methods for improving performance.

Key Steps Involved in File System Implementation

File system implementation is a crucial component of an operating system, as it provides an interface between the user and the physical storage device. Here are the key steps involved in file system implementation:

- **Partitioning The Storage Device:** The first step in file system implementation is to partition the physical storage device into one or more logical partitions. Each partition is formatted with a specific file system that defines the way files and directories are organized and stored.
- **File System Structures:** File system structures are the data structures used by the operating system to manage files and directories. Some of the key file system structures include the superblock, inode table, directory structure, and file allocation table.
- **Allocation of Storage Space:** The file system must allocate storage space for each file and directory on the storage device. There are several methods for allocating storage space, including contiguous, linked, and indexed allocation.
- **File Operations:** The file system provides a set of operations that can be performed on files and directories, including create, delete, read, write, open, close, and seek. These

operations are implemented using the file system structures and the storage allocation methods.

- **File System Security:** The file system must provide security mechanisms to protect files and directories from unauthorized access or modification. This can be done by setting file permissions, access control lists, or encryption.
- **File System Maintenance:** The file system must be maintained to ensure efficient and reliable operation. This includes tasks such as disk defragmentation, disk checking, and backup and recovery.

Overall, file system implementation is a complex and critical component of an operating system. The efficiency and reliability of the file system have a significant impact on the performance and stability of the entire system.

Advanced Topics

Systems for Journaling Files

Journaling file systems are intended to enhance data integrity and shorten the amount of time it takes to recover from a system crash or power outage. They achieve this by keeping track of changes to the file system metadata before they are written to disc in a log, or journal. The journal can be used to quickly restore the file system to a consistent state in the event of a crash or failure.

File Systems on A Network

Multiple computers connected by a network can access and share files thanks to network file systems. Users can access files and directories through a transparent interface they offer, just as if the files were locally stored. instances of networks.

Advantages

- Duplication of code is minimized.
- Each file system can have its own logical file system.
- File system implementation in an operating system provides several advantages, including:
- **Efficient Data Storage:** File system implementation ensures efficient data storage on a physical storage device. It provides a structured way of organizing files and directories, which makes it easy to find and access files.
- **Data Security:** File system implementation includes features for managing file security and permissions. This ensures that sensitive data is protected from unauthorized access.
- **Data Recovery:** The file system implementation includes features for recovering from system failures and maintaining data integrity. This helps to prevent data loss and ensures that data can be recovered in the event of a system failure.
- **Improved Performance:** File system implementation includes techniques such as buffering and caching to optimize file I/O performance. This results in faster access to data and improved overall system performance.
- **Scalability:** File system implementation can be designed to be scalable, making it possible to store and retrieve large amounts of data efficiently.
- **Flexibility:** Different file system implementations can be designed to meet specific needs and use cases. This allows developers to choose the best file system implementation for their specific requirements.

- **Cross-Platform Compatibility:** Many file system implementations are cross-platform compatible, which means they can be used on different operating systems. This makes it easy to transfer files between different systems.

In summary, file system implementation in an operating system provides several advantages, including efficient data storage, data security, data recovery, improved performance, scalability, flexibility, and cross-platform compatibility. These advantages make file system implementation a critical aspect of any operating system.

Disadvantages

If we access many files at the same time then it results in low performance. We can implement a file system by using two types of data structures:

- **Boot Control Block –** It is usually the first block of volume and it contains information needed to boot an operating system. In UNIX it is called the boot block and in NTFS it is called the partition boot sector.
- **Volume Control Block –** It has information about a particular partition ex: - free block count, block size and block pointers, etc. In UNIX it is called superblock and in NTFS it is stored in the master file table.
- **Directory Structure –** They store file names and associated inode numbers. In UNIX, includes file names and associated file names and in NTFS, it is stored in the master file table.
- **Per-File FCB –** It contains details about files and it has a unique identifier number to allow association with the directory entry. In NTFS it is stored in the master file table.
- **Mount Table –** It contains information about each mounted volume.
- **Directory-Structure Cache –** This cache holds the directory information of recently accessed directories.
- **System-Wide Open-File Table –** It contains the copy of the FCB of each open file.
- **Per-Process Open-File Table –** It contains information opened by that particular process and it maps with the appropriate system-wide open-file.
- **Linear List –** It maintains a linear list of filenames with pointers to the data blocks. It is time-consuming also. To create a new file, we must first search the directory to be sure that no existing file has the same name then we add a file at the end of the directory. To delete a file, we search the directory for the named file and release the space. To reuse the directory entry either we can mark the entry as unused or we can attach it to a list of free directories.
- **Hash Table –** The hash table takes a value computed from the file name and returns a pointer to the file. It decreases the directory search time. The insertion and deletion process of files is easy. The major difficulty is hash tables are its generally fixed size and hash tables are dependent on the hash function of that size.

File Allocation Methods

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

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

The main idea behind these methods is to provide:

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

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

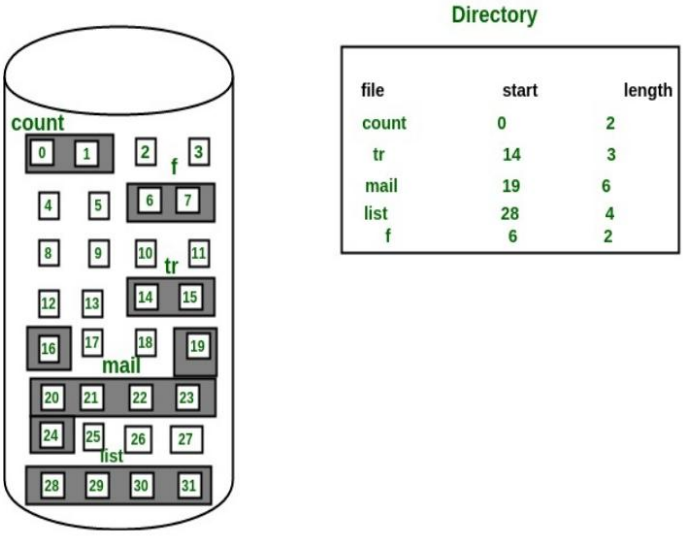
1. Contiguous Allocation

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

The directory entry for a file with contiguous allocation contains

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

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



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:

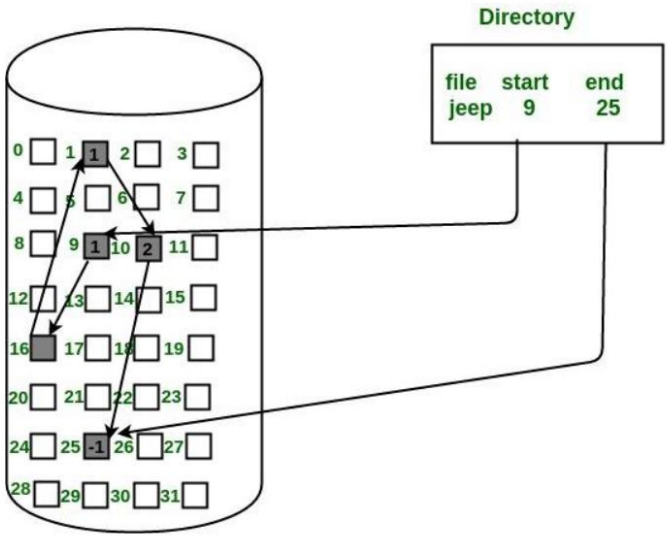
- This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.
- Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

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:

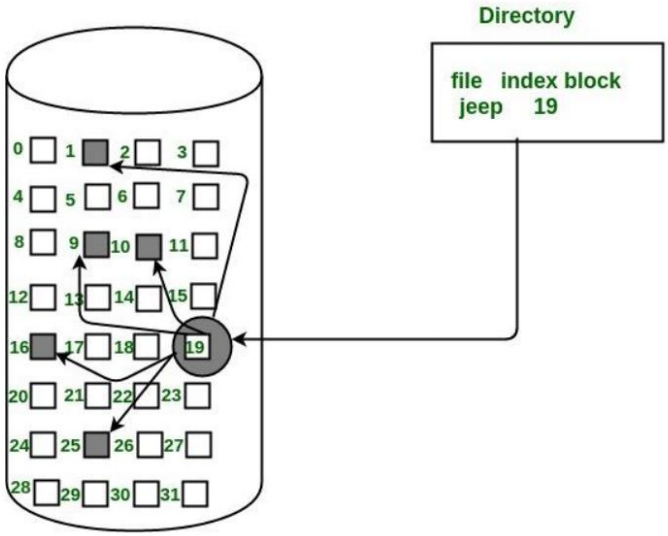
- This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.
- This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.

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.

For files that are very large, single index block may not be able to hold all the pointers. Following mechanisms can be used to resolve this:

1. **Linked scheme:** This scheme links two or more index blocks together for holding the pointers. Every index block would then contain a pointer or the address to the next index block.
2. **Multilevel index:** In this policy, a first level index block is used to point to the second level index blocks which inturn points to the disk blocks occupied by the file. This can be extended to 3 or more levels depending on the maximum file size.
3. **Combined Scheme:** In this scheme, a special block called the **Inode (information Node)** contains all the information about the file such as the name, size, authority, etc and the remaining space of Inode is used to store the Disk Block addresses which contain the actual file *as shown in the image below.* The first few of these pointers in Inode point to the **direct blocks** i.e the pointers contain the addresses of the disk blocks that contain data of the file. The next few pointers point to indirect blocks. Indirect blocks may be single indirect, double indirect or triple indirect. **Single Indirect block** is the disk block that does not contain the file data but the disk address of the blocks that contain the file data. Similarly, **double indirect blocks** do not contain the file data but

the disk address of the blocks that contain the address of the blocks containing the file data.