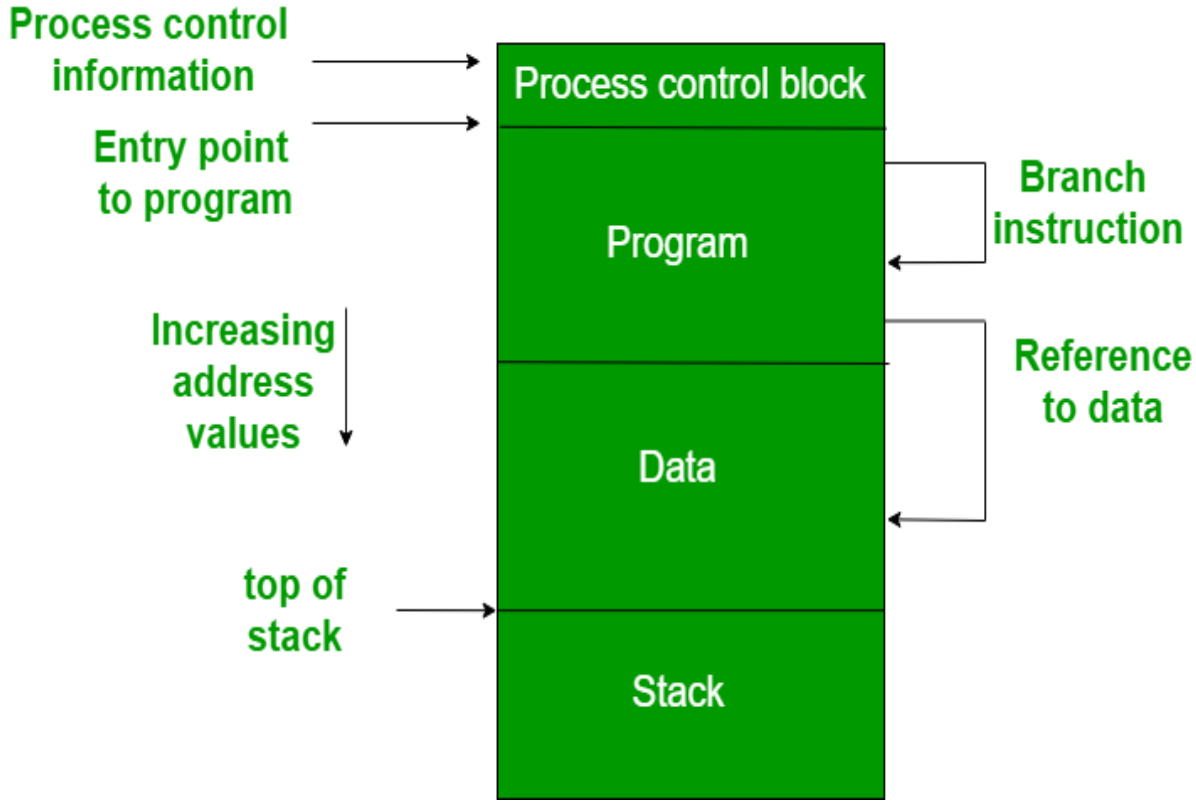
OS Module 4 THEORY

Memory Management

REQUIREMENTS OF MEMORY MANAGEMENT

1. **Relocation –** The available memory is generally shared among a number of processes in a multiprogramming system, so it is not possible to know in advance which other programs will be resident in main memory at the time of execution of this program. Swapping the active processes in and out of the main memory enables the operating system to have a larger pool of ready-to-execute process.

   When a program gets swapped out to a disk memory, then it is not always possible that when it is swapped back into main memory then it occupies the previous memory location, since the location may still be occupied by another process. We may need to **relocate** the process to a different area of memory. Thus there is a possibility that program may be moved in main memory due to swapping.



The figure depicts a process image. The process image is occupying a continuous region of main memory. The operating system will need to know many things including the location of process control information, the execution stack, and the code entry. Within a program, there are memory references in various instructions and these are called logical addresses.

After loading of the program into main memory, the processor and the operating system must be able to translate logical addresses into physical addresses. Branch instructions

contain the address of the next instruction to be executed. Data reference instructions contain the address of byte or word of data referenced.

2. **Protection –** There is always a danger when we have multiple programs at the same time as one program may write to the address space of another program. So every process must be protected against unwanted interference when other process tries to write in a process whether accidental or incidental. Between relocation and protection requirement a trade-off occurs as the satisfaction of relocation requirement increases the difficulty of satisfying the protection requirement.

   Prediction of the location of a program in main memory is not possible, that's why it is impossible to check the absolute address at compile time to assure protection. Most of the programming language allows the dynamic calculation of address at run time. The memory protection requirement must be satisfied by the processor rather than the operating system because the operating system can hardly control a process when it occupies the processor. Thus it is possible to check the validity of memory references.

3. **Sharing –** A protection mechanism must have to allow several processes to access the same portion of main memory. Allowing each processes access to the same copy of the program rather than have their own separate copy has an advantage.

   For example, multiple processes may use the same system file and it is natural to load one copy of the file in main memory and let it shared by those processes. It is the task of Memory management to allow controlled access to the shared areas of memory without compromising the protection. Mechanisms are used to support relocation supported sharing capabilities.

4. **Logical organization –** Main memory is organized as linear or it can be a one-dimensional address space which consists of a sequence of bytes or words. Most of the programs can be organized into modules, some of those are unmodifiable (read-only, execute only) and some of those contain data that can be modified. To effectively deal with a user program, the operating system and computer hardware must support a basic module to provide the required protection and sharing. It has the following advantages:

   - Modules are written and compiled independently and all the references from one module to another module are resolved by `the system at run time.
   - Different modules are provided with different degrees of protection.
   - There are mechanisms by which modules can be shared among processes. Sharing can be provided on a module level that lets the user specify the sharing that is desired.

5. **Physical organization –** The structure of computer memory has two levels referred to as main memory and secondary memory. Main memory is relatively very fast and costly as compared to the secondary memory. Main memory is volatile. Thus secondary memory is provided for storage of data on a long-term basis while the main memory holds currently used programs. The major system concern between main memory and secondary memory is the flow of information and it is impractical for programmers to understand this for two reasons:

    - The programmer may engage in a practice known as overlaying when the main memory available for a program and its data may be insufficient. It allows different modules to be assigned to the same region of memory. One disadvantage is that it is time-consuming for the programmer.
    - In a multiprogramming environment, the programmer does not know how much space will be available at the time of coding and where that space will be located inside the memory.

What is Fixed (or static) Partitioning in the Operating System?

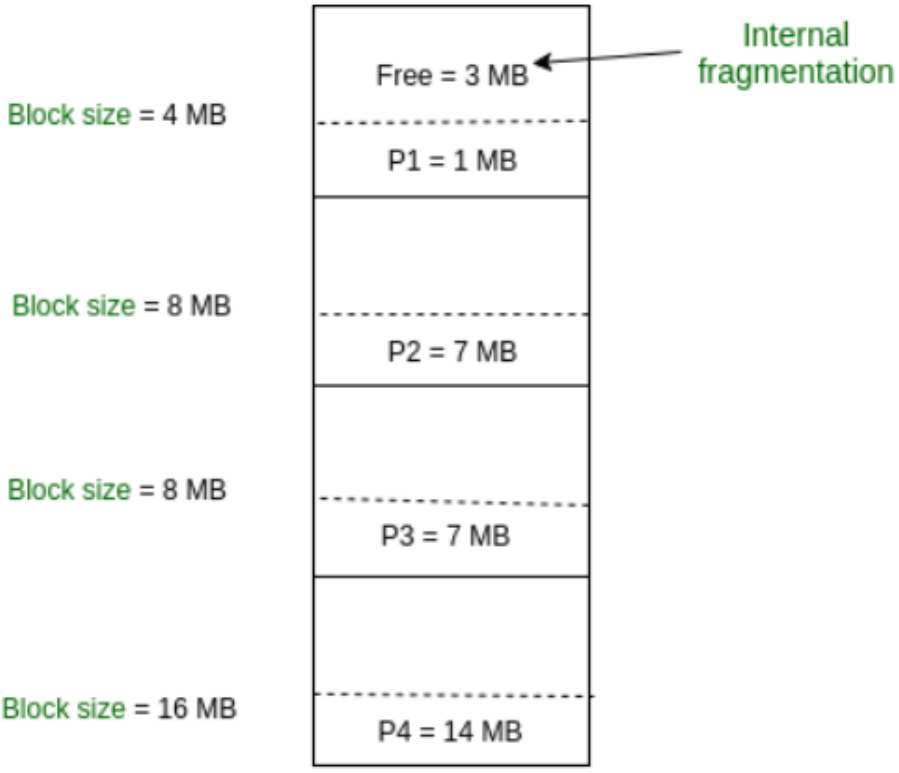
Fixed (or static) partitioning is one of the earliest and simplest memory management techniques used in operating systems. It involves dividing the main memory into a fixed number of partitions at system startup, with each partition being assigned to a process. These partitions remain unchanged throughout the system's operation, providing each process with a designated memory space. This method was widely used in early operating systems and remains relevant in specific contexts like embedded systems and real-time applications. However, while fixed partitioning is simple to implement, it has significant limitations, including inefficiencies caused by internal fragmentation.

1. In fixed partitioning, the memory is divided into fixed-size chunks, with each chunk being reserved for a specific process. When a process requests memory, the operating system assigns it to the appropriate partition. Each partition is of the same size, and the memory allocation is done at system boot time.
2. Fixed partitioning has several advantages over other memory allocation techniques. First, it is simple and easy to implement. Second, it is predictable, meaning the operating system can ensure a minimum amount of memory for each process. Third, it can prevent processes from interfering with each other's memory space, improving the security and stability of the system.
3. However, fixed partitioning also has some disadvantages. It can lead to internal fragmentation, where memory in a partition remains unused. This can happen when the process's memory requirements are smaller than the partition size, leaving some memory unused. Additionally, fixed partitioning limits the number of processes that can run concurrently, as each process requires a dedicated partition.

Overall, fixed partitioning is a useful memory allocation technique in situations where the number of processes is fixed, and the memory requirements for each process are known in

advance. It is commonly used in embedded systems, real-time systems, and systems with limited memory resources.

In operating systems, Memory Management is the function responsible for allocating and managing a computer's main memory. Memory Management function keeps track of the status of each memory location, either allocated or free to ensure effective and efficient use of Primary Memory.

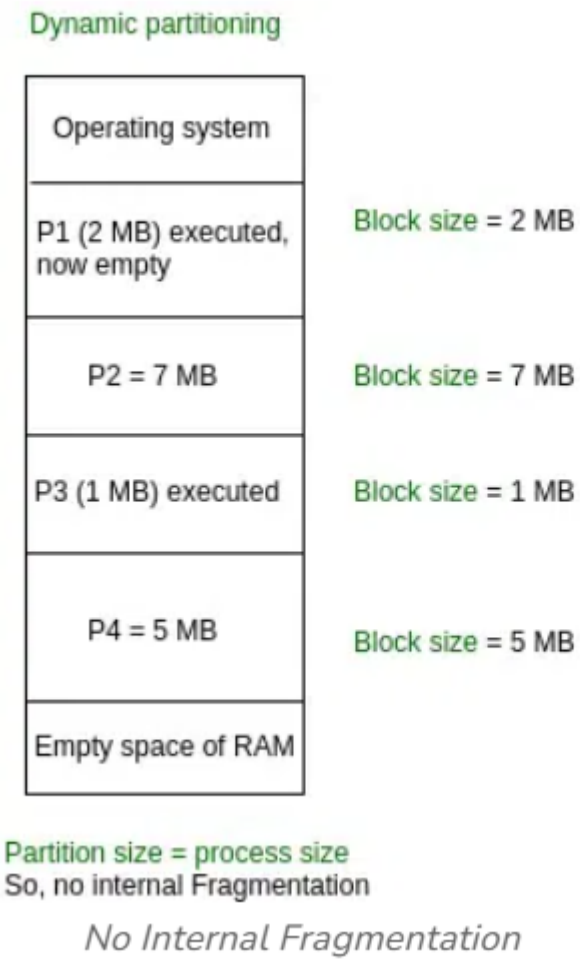


Fixed size partition

What is Variable (Dynamic) Partitioning?

It is a part of the Contiguous allocation technique. It is used to alleviate the problem faced by Fixed Partitioning. In contrast with fixed partitioning, partitions are not made before the execution or during system configuration. Various **features** associated with variable Partitioning-

- Initially, RAM is empty and partitions are made during the run-time according to the process's need instead of partitioning during system configuration.
- The size of the partition will be equal to the incoming process.
- The partition size varies according to the need of the process so that internal fragmentation can be avoided to ensure efficient utilization of RAM.
- The number of partitions in RAM is not fixed and depends on the number of incoming processes and the Main Memory's size.



No Internal Fragmentation

PARTITION ALLOCATION

In **Partition Allocation**, when there is more than one partition freely available to accommodate a process's request, a partition must be selected. To choose a particular partition, a partition allocation method is needed. A partition allocation method is considered better if it avoids internal fragmentation.

When it is time to load a process into the main memory and if there is more than one free block of memory of sufficient size then the OS decides which free block to allocate.

There are different Placement Algorithm:

A. First Fit

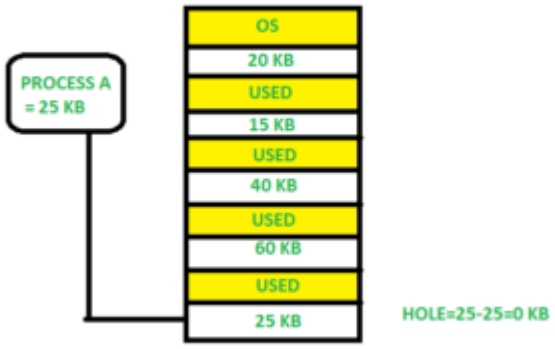
B. Best Fit

C. Worst Fit

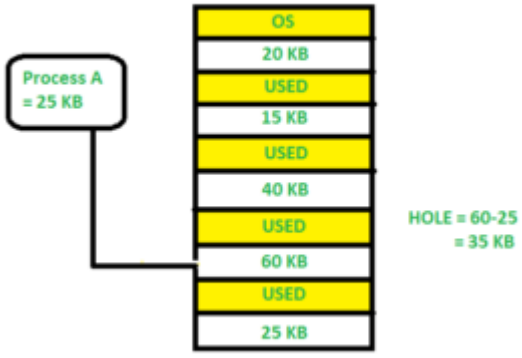
D. Next Fit

1. First Fit: In the first fit, the partition is allocated which is the first sufficient block from the top of Main Memory. It scans memory from the beginning and chooses the first available block that is large enough. Thus it allocates the first hole that is large enough.

2. Best Fit Allocate the process to the partition which is the first smallest sufficient partition among the free available partition. It searches the entire list of holes to find the smallest hole whose size is greater than or equal to the size of the process.



3. Worst Fit Allocate the process to the partition which is the largest sufficient among the freely available partitions available in the main memory. It is opposite to the best-fit algorithm. It searches the entire list of holes to find the largest hole and allocate it to process.



4. Next Fit: Next fit is similar to the first fit but it will search for the first sufficient partition from the last allocation point.

Is Best-Fit really best?
Although best fit minimizes the wastage space, it consumes a lot of processor time for searching the block which is close to the required size. Also, Best-fit may perform poorer than other algorithms in some cases. For example, see the below exercise.

Comparison of Partition Allocation Methods:

Sl.No.	Partition Allocation Method	Advantages	Disadvantages
1.	Fixed Partition	Simple, easy to use, no complex algorithms needed	Memory waste, inefficient use of memory resources
2.	Dynamic Partition	Flexible, more efficient, partitions allocated as required	Requires complex algorithms for memory allocation
3.	Best-fit Allocation	Minimizes memory waste, allocates smallest suitable partition	More computational overhead to find smallest split
4.	Worst-fit Allocation	Ensures larger processes have sufficient memory	May result in substantial memory waste
5.	First-fit Allocation	Quick, efficient, less computational work	Risk of memory fragmentation

Fixed Partitioning	**Variable Partitioning**
Memory is divided into fixed-sized partitions.	Memory is allocated dynamically in varying sizes.
Only one process can be placed in each partition.	A process is allocated a chunk of free memory as needed.

Fixed Partitioning	Variable Partitioning
Inefficient memory utilization due to internal fragmentation.	More efficient memory utilization with less internal fragmentation.
Internal and external fragmentation occur.	Only external fragmentation occurs.
Limited degree of multi-programming.	Higher degree of multi-programming due to flexible allocation.
Easier to implement.	More complex to implement.
Restricts process size based on partition size.	No size limitation on processes.

What is the Buddy System?

Buddy System is a memory allocation technique used in computer OS to allocate and manage memory efficiently. This technique by dividing the memory into fixed-size blocks, and whenever a process requests memory, the system finds the smallest available block that can accommodate the requested memory size.

It splits memory blocks, called “buddies,” to minimize fragmentation and ensure efficient allocation. When a process is deallocated, its buddy can be merged back into a larger block, reducing wasted space

Algorithm of Buddy System Memory Allocation Technique

Below are the steps involved in the Buddy System Memory Allocation Technique:

- The first step includes the division of memory into fixed-sized blocks that have a power of 2 in size (such as 2, 4, 8, 16, 32, 64, 128, etc.).
- Each block is labeled with its size and unique identification.
- Initially, all the memory blocks are free and are linked together in a binary tree structure, with each node representing a block and the tree’s leaves representing the smallest available blocks.

- When a process requests memory, the system finds the smallest available block that can accommodate the requested size. If the block is larger than the requested size, the system splits the block into two equal-sized “buddy” blocks.
- The system marks one of the buddy blocks as allocated and adds it to the process’s memory allocation table, while the other buddy block is returned to the free memory pool and linked back into the binary tree structure.
- When a process releases memory, the system marks the corresponding block as free and looks for its buddy block. If the buddy block is also free, the system merges the two blocks into a larger block and links it back into the binary tree structure.

The Buddy System technique has several advantages, including efficient use of memory, reduced fragmentation, and fast allocation and deallocation of memory blocks. However, it also has some drawbacks, such as internal fragmentation, where a block may be larger than what the process requires, leading to a waste of memory. Overall, the Buddy System is a useful memory allocation technique in operating systems, particularly for embedded systems with limited memory.

- **If 2 U-1 <S<=2 U :** Allocate the whole block
- **Else:** Recursively divide the block equally and test the condition at each time, when it satisfies, allocate the block and get out of the loop.

The system also keeps a record of all the unallocated blocks and can merge these different-sized blocks to make one big chunk.

Types of Buddy System

The Buddy System in memory control generally refers to a selected method used to allocate and deallocate memory blocks. However, within this framework, versions or adaptations may additionally exist relying on specific necessities or optimizations wished for distinctive structures. Here are a few types of Buddy Systems:

- **Fibonacci Buddy System** : A Fibonacci buddy system has block sizes of 16, 32, 48, 80, 128, and 208 bytes, with each block size equal to the total of the two blocks previous it. When a block is divided from one free list, its two portions are added to the two free lists that came before it. It can be minimized by bringing the allowable block sizes close together. Now, let’s look at an example to better grasp the Fibonacci buddy system. Assume the memory size is 377 kb. Then, 377 kb will be partitioned into 144 kb and 233 kb. Following that, 144 will be divided into (55 + 89), whereas 233 will be divided into (89 + 144). This separation will continue based on memory requirements.
- **Binary Buddy System** : The buddy system keeps track of the free blocks of each size (known as a free list) so that you can easily discover a block of the necessary size if one is available. If no blocks of the requested size are available, Allocate examines the first non-empty list for blocks of at least the requested size. In both cases, a block is deleted from the free list. For ex: The 512 KB memory size is initially partitioned

into two active partitions of 256 KB each, with additional subdivisions based on a capacity of 2 to handle memory requests.

- **Weighted Buddy System:** In a weighted peer system, each memory block is associated with a weight, which represents its size relative to other blocks. When a memory allocation request occurs, the system searches for the appropriate block considering the size of the requested memory and the weight of the available blocks.
- **Tertiary Buddy System** : In a traditional buddy system, memory is divided into blocks of fixed size, usually a power of 2, and allocated to these blocks but the tertiary buddy system introduces a third memory structure, which allows flexibility large in memory allocation.

What is Fragmentation in an Operating System?

An unwanted problem with operating systems is fragmentation, which occurs when processes load and unload from memory and divide available memory. Because memory blocks are so small, they cannot be assigned to processes, and thus remain idle. It's also important to realize that programs create free space or holes in memory when they are loaded and unloaded. Because additional processes cannot be assigned to these little pieces, memory is used inefficiently.

The memory allocation scheme determines the fragmentation circumstances. These regions of memory become fragmented when the process loads and unloads from it, making it unusable for incoming processes. We refer to it as fragmentation.

Cause of Fragmentation

This can happen when a file is too large to fit into a single contiguous block of free space on the storage medium, or when the blocks of free space on the medium are insufficient to hold the file. Because the system must search for and retrieve individual fragments from different locations in order to open the file, fragmentation can cause problems when reading or accessing the file.

Effect of Fragmentation

This can reduce system performance and make it more difficult to access the file. It is generally best to defragment your hard disc on a regular basis to avoid fragmentation, which is a process that rearranges the blocks of data on the disc so that files are stored in contiguous blocks and can be accessed more quickly.

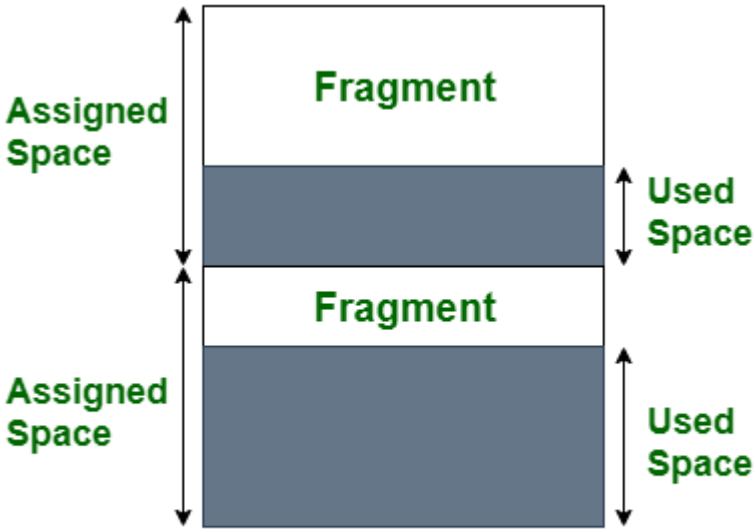
Types of Fragmentation

There are two main types of fragmentation:

- Internal Fragmentation
- External Fragmentation

1. Internal Fragmentation

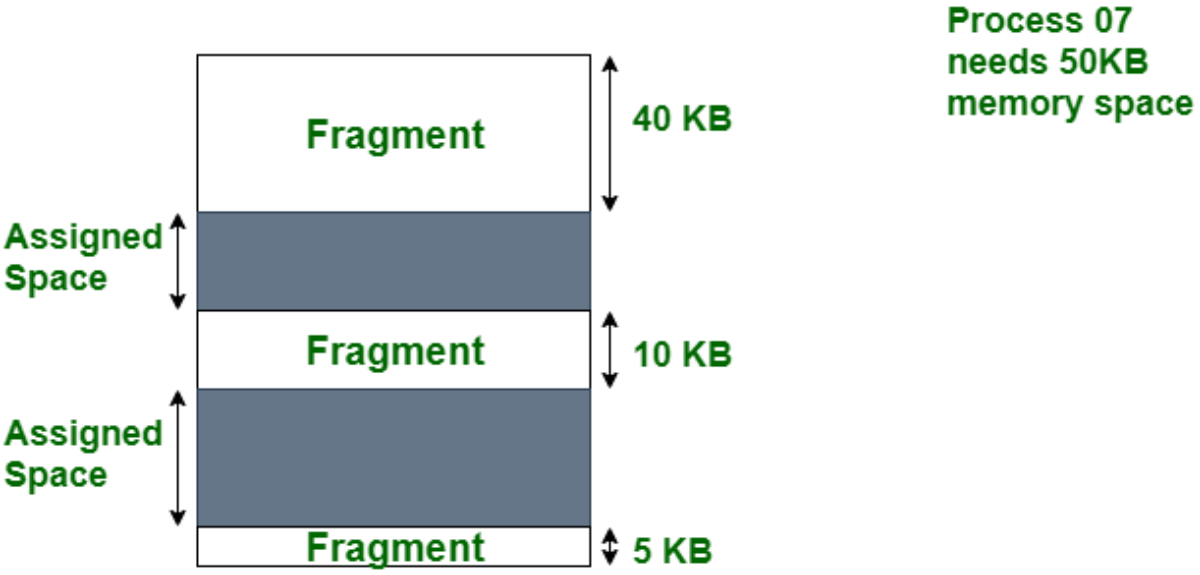
Internal fragmentation occurs when there is unused space within a memory block. For example, if a system allocates a 64KB block of memory to store a file that is only 40KB in size, that block will contain 24KB of internal fragmentation. When the system employs a fixed-size block allocation method, such as a memory allocator with a fixed block size, this can occur.



Internal Fragmentation

2. External Fragmentation

External fragmentation occurs when a storage medium, such as a hard disc or solid-state drive, has many small blocks of free space scattered throughout it. This can happen when a system creates and deletes files frequently, leaving many small blocks of free space on the medium. When a system needs to store a new file, it may be unable to find a single contiguous block of free space large enough to store the file and must instead store the file in multiple smaller blocks. This can cause external fragmentation and performance problems when accessing the file.



Fragmentation can also occur at various levels within a system. File fragmentation, for example, can occur at the file system level, in which a file is divided into multiple non-contiguous blocks and stored on a storage medium. Memory fragmentation can occur at the memory management level, where the system allocates and deallocated memory blocks dynamically. Network fragmentation occurs when a packet of data is divided into smaller fragments for transmission over a network.

PAGING

Paging is a memory management scheme that eliminates the need for a contiguous allocation of physical memory. The process of retrieving processes in the form of pages from the secondary storage into the main memory is known as **paging**. The basic purpose of paging is to separate each procedure into pages. Additionally, frames will be used to split the main memory. This scheme permits the physical address space of a process to be non – contiguous.

In paging, the physical memory is divided into fixed-size blocks called **page frames**, which are the same size as the pages used by the process. The process's logical address space is also divided into fixed-size blocks called pages, which are the same size as the page frames. When a process requests memory, the operating system allocates one or more page frames to the process and maps the process's **logical pages** to the physical page frames.

The mapping between logical pages and physical page frames is maintained by the page table, which is used by the **memory management unit** to translate logical addresses into physical addresses. The page table maps each logical page number to a physical page frame number.

SEGMENTATION

A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the exact sizes are called segments. Segmentation gives the user's view of the process which paging does not provide. Here the user's view is mapped to physical memory.

Types of Segmentation in Operating Systems

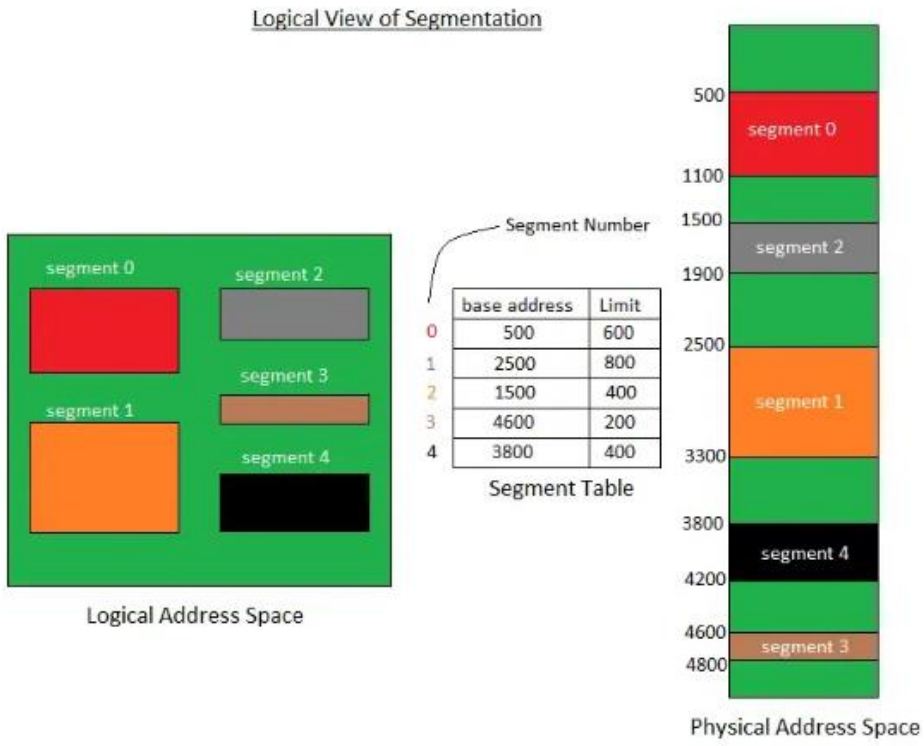
- **Virtual Memory Segmentation:** Each process is divided into a number of segments, but the segmentation is not done all at once. This segmentation may or may not take place at the run time of the program.
- **Simple Segmentation:** Each process is divided into a number of segments, all of which are loaded into memory at run time, though not necessarily contiguously.

There is no simple relationship between logical addresses and physical addresses in segmentation. A table stores the information about all such segments and is called Segment Table.

What is Segment Table?

It maps a two-dimensional Logical address into a one-dimensional Physical address. It's each table entry has:

- **Base Address:** It contains the starting physical address where the segments reside in memory.
- **Segment Limit:** Also known as segment offset. It specifies the length of the segment.



Segmentation is crucial in efficient memory management within an operating system.

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.

Parameter	LOGICAL ADDRESS	PHYSICAL ADDRESS
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.

CACHE MEMORY

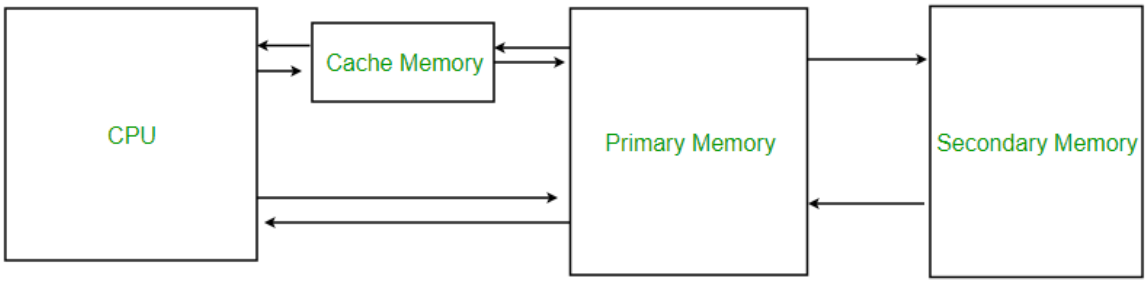
Cache memory is a small, high-speed storage area in a computer. The cache is a smaller and faster memory that stores copies of the data from frequently used main memory locations. There are various independent caches in a CPU, which store instructions and data. The most important use of cache memory is that it is used to reduce the average time to access data from the main memory.

By storing this information closer to the CPU, cache memory helps speed up the overall processing time. Cache memory is much faster than the main memory (RAM). When the CPU needs data, it first checks the cache. If the data is there, the CPU can access it quickly. If not, it must fetch the data from the slower main memory.

Characteristics of Cache Memory

- Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU.
- Cache Memory holds frequently requested data and instructions so that they are immediately available to the CPU when needed.
- Cache memory is costlier than main memory or disk memory but more economical than CPU registers.

- Cache Memory is used to speed up and synchronize with a high-speed CPU.



Cache Memory

Levels of Memory

- **Level 1 or Register:** It is a type of memory in which data is stored and accepted that are immediately stored in the CPU. The most commonly used register is Accumulator, Program counter , Address Register, etc.
- **Level 2 or Cache memory:** It is the fastest memory that has faster access time where data is temporarily stored for faster access.
- **Level 3 or Main Memory:** It is the memory on which the computer works currently. It is small in size and once power is off data no longer stays in this memory.
- **Level 4 or Secondary Memory:** It is external memory that is not as fast as the main memory but data stays permanently in this memory.

Cache Performance

When the processor needs to read or write a location in the main memory, it first checks for a corresponding entry in the cache.

- If the processor finds that the memory location is in the cache, a Cache Hit has occurred and data is read from the cache.
- If the processor does not find the memory location in the cache, a **cache miss** has occurred. For a cache miss, the cache allocates a new entry and copies in data from the main memory, then the request is fulfilled from the contents of the cache.

The performance of cache memory is frequently measured in terms of a quantity called **Hit ratio.**

Hit Ratio(H) = hit / (hit + miss) = no. of hits/total accesses
Miss Ratio = miss / (hit + miss) = no. of miss/total accesses = 1 - hit ratio(H)

We can improve Cache performance using higher cache block size, and higher associativity, reduce miss rate, reduce miss penalty, and reduce the time to hit in the cache.

Cache Mapping

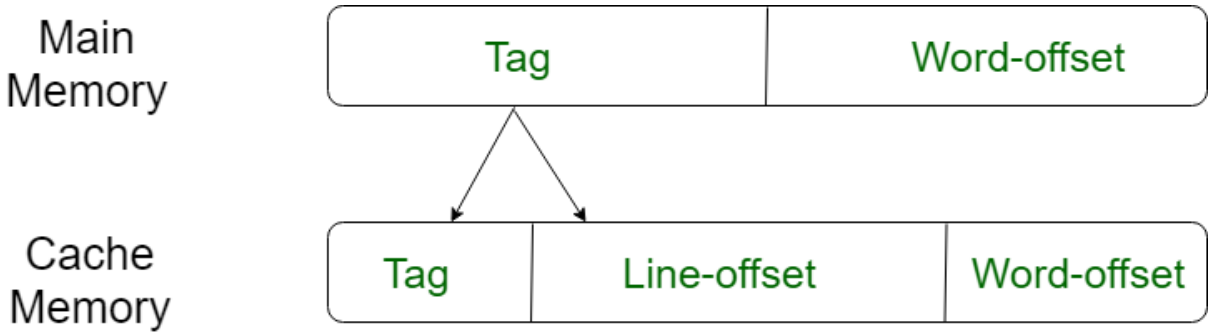
There are three different types of mapping used for the purpose of cache memory which is as follows:

- Direct Mapping
- Associative Mapping
- Set-Associative Mapping

1. Direct Mapping

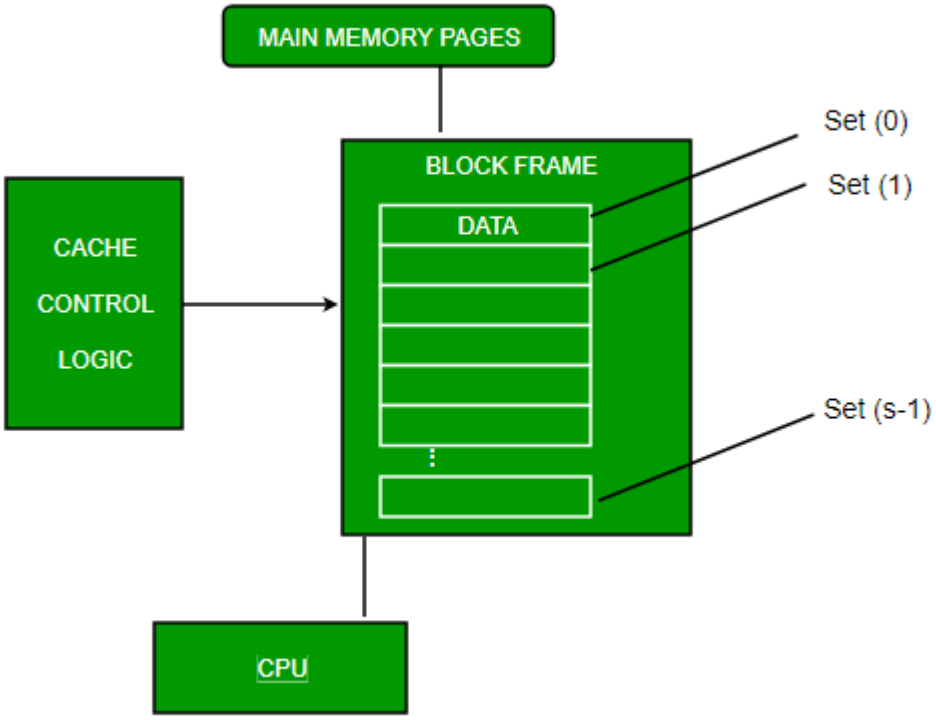
The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. or In Direct mapping, assign each memory block to a specific line in the cache. If a line is previously taken up by a memory block when a new block needs to be loaded, the old block is trashed. An address space is split into two parts index field and a tag field. The cache is used to store the tag field whereas the rest is stored in the main memory. Direct mapping`s performance is directly proportional to the Hit ratio.

i = j modulo m
where
i = cache line number
j = main memory block number
m = number of lines in the cache



Direct Mapping

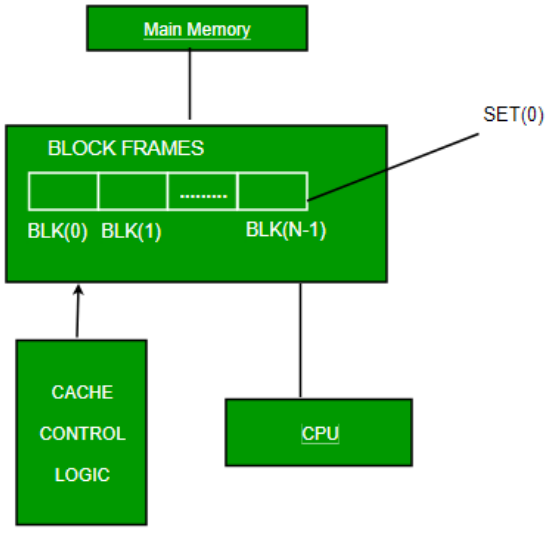
For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory. In most contemporary machines, the address is at the byte level. The remaining s bits specify one of the 2 s blocks of main memory. The cache logic interprets these s bits as a tag of s-r bits (the most significant portion) and a line field of r bits. This latter field identifies one of the m=2 r lines of the cache. Line offset is index bits in the direct mapping.



Direct Mapping – Structure

2. Associative Mapping

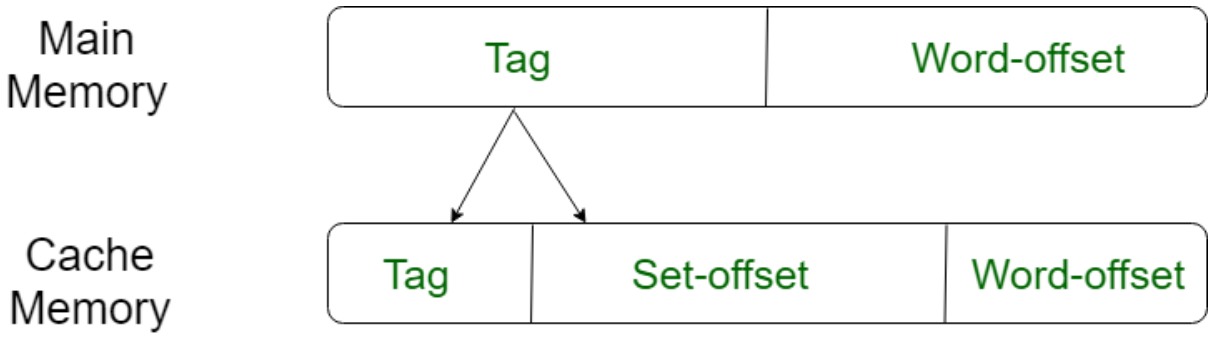
In this type of mapping, associative memory is used to store the content and addresses of the memory word. Any block can go into any line of the cache. This means that the word id bits are used to identify which word in the block is needed, but the tag becomes all of the remaining bits. This enables the placement of any word at any place in the cache memory. It is considered to be the fastest and most flexible mapping form. In associative mapping, the index bits are zero.



Associative Mapping – Structure

3. Set-Associative Mapping

This form of mapping is an enhanced form of direct mapping where the drawbacks of direct mapping are removed. Set associative addresses the problem of possible thrashing in the direct mapping method. It does this by saying that instead of having exactly one line that a block can map to in the cache, we will group a few lines together creating a ***set*** . Then a block in memory can map to any one of the lines of a specific set. Set-associative mapping allows each word that is present in the cache can have two or more words in the main memory for the same index address. Set associative cache mapping combines the best of direct and associative cache mapping techniques. In set associative mapping the index bits are given by the set offset bits. In this case, the cache consists of a number of sets, each of which consists of a number of lines.

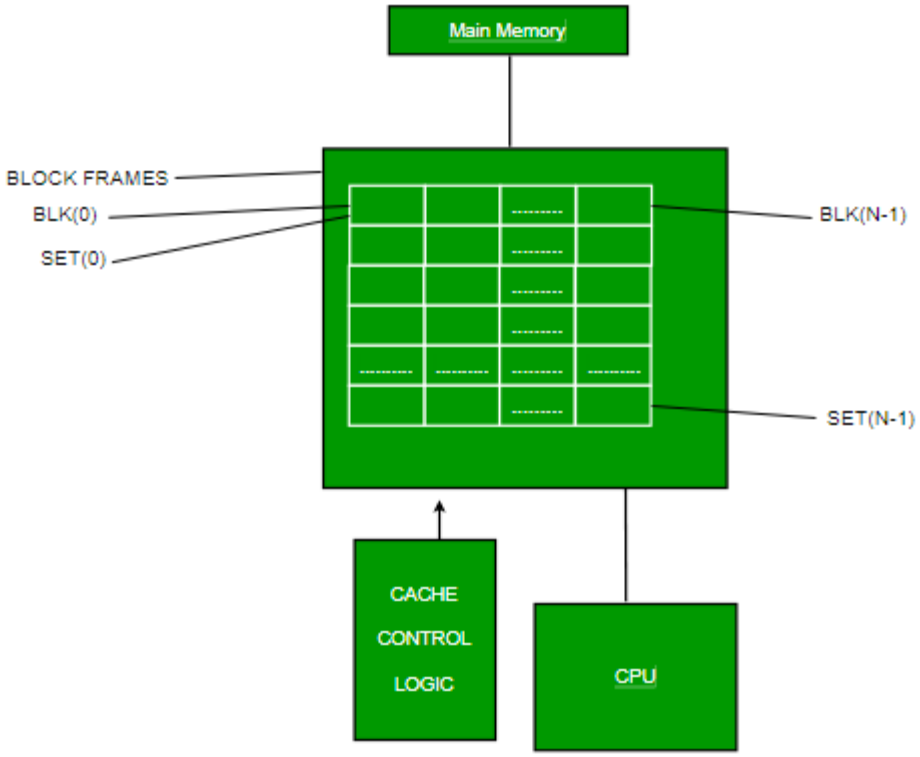


Set-Associative Mapping

Relationships in the Set-Associative Mapping can be defined as:

m = v * k
i= j mod v

where
i = cache set number
j = main memory block number
v = number of sets
m = number of lines in the cache number of sets
k = number of lines in each set



Set-Associative Mapping – Structure

How Virtual Memory Works?

Virtual Memory is a technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

- All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of the main memory such that it occupies different places in the main memory at different times during the course of execution.
- A process may be broken into a number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and the use of a page or segment table permits this.

If these characteristics are present then, it is not necessary that all the pages or segments are present in the main memory during execution. This means that the required pages need to be loaded into memory whenever required. Virtual memory is implemented using Demand Paging or Demand Segmentation.

Types of Virtual Memory

In a computer, virtual memory is managed by the Memory Management Unit (MMU), which is often built into the CPU. The CPU generates virtual addresses that the MMU translates into physical addresses.

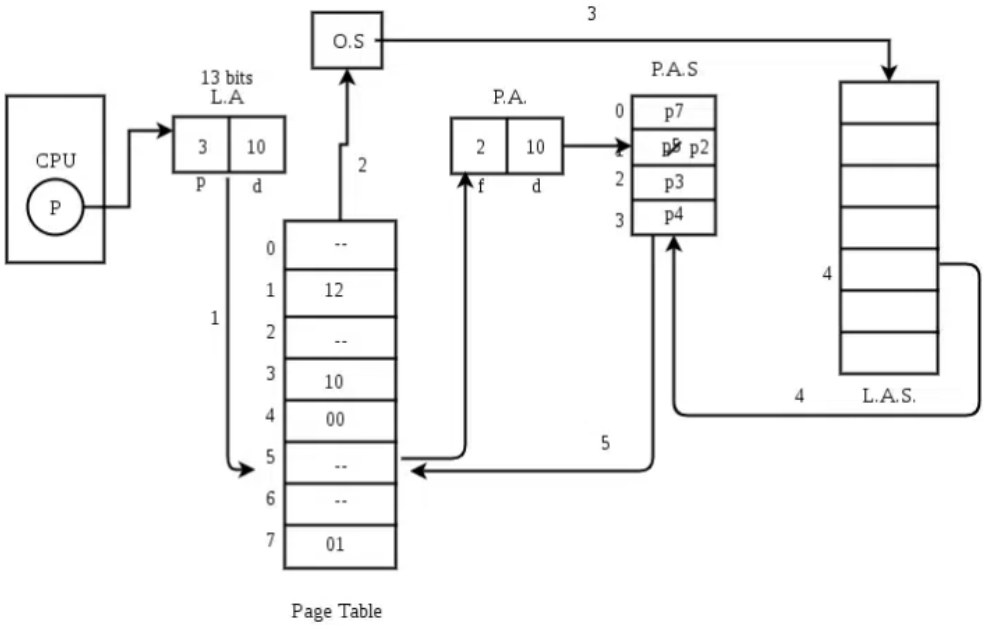
There are two main types of virtual memory:

- Paging
- Segmentation

Paging

Paging divides memory into small fixed-size blocks called pages. When the computer runs out of RAM, pages that aren't currently in use are moved to the hard drive, into an area called a swap file. The swap file acts as an extension of RAM. When a page is needed again, it is swapped back into RAM, a process known as page swapping. This ensures that the operating system (OS) and applications have enough memory to run.

Demand Paging: The process of loading the page into memory on demand (whenever a page fault occurs) is known as demand paging. The process includes the following steps are as follows:



Demand Paging

- If the CPU tries to refer to a page that is currently not available in the main memory, it generates an interrupt indicating a memory access fault.
- The OS puts the interrupted process in a blocking state. For the execution to proceed the OS must bring the required page into the memory.
- The OS will search for the required page in the logical address space.

- The required page will be brought from logical address space to physical address space. The page replacement algorithms are used for the decision-making of replacing the page in physical address space.
- The page table will be updated accordingly.
- The signal will be sent to the CPU to continue the program execution and it will place the process back into the ready state.

Hence whenever a page fault occurs these steps are followed by the operating system and the required page is brought into memory.

What is Page Fault Service Time?

The time taken to service the page fault is called page fault service time. The page fault service time includes the time taken to perform all the above six steps.

Let Main memory access time is: m

Page fault service time is: s

Page fault rate is : p

Then, Effective memory access time = (p*s) + (1-p)*m

Segmentation

Segmentation divides virtual memory into segments of different sizes. Segments that aren't currently needed can be moved to the hard drive. The system uses a segment table to keep track of each segment's status, including whether it's in memory, if it's been modified, and its physical address. Segments are mapped into a process's address space only when needed.

Combining Paging and Segmentation

Sometimes, both paging and segmentation are used together. In this case, memory is divided into pages, and segments are made up of multiple pages. The virtual address includes both a segment number and a page number.

Virtual Memory vs Physical Memory

Feature	**Virtual Memory**	**Physical Memory (RAM)**
Definition	An abstraction that extends the available memory by using disk storage	The actual hardware (RAM) that stores data and instructions currently being used by the CPU
Location	On the hard drive or SSD	On the computer's motherboard

Feature	Virtual Memory	Physical Memory (RAM)
Speed	Slower (due to disk I/O operations)	Faster (accessed directly by the CPU)
Capacity	Larger, limited by disk space	Smaller, limited by the amount of RAM installed
Cost	Lower (cost of additional disk storage)	Higher (cost of RAM modules)
Data Access	Indirect (via paging and swapping)	Direct (CPU can access data directly)
Volatility	Non-volatile (data persists on disk)	Volatile (data is lost when power is off)

What is Swapping?

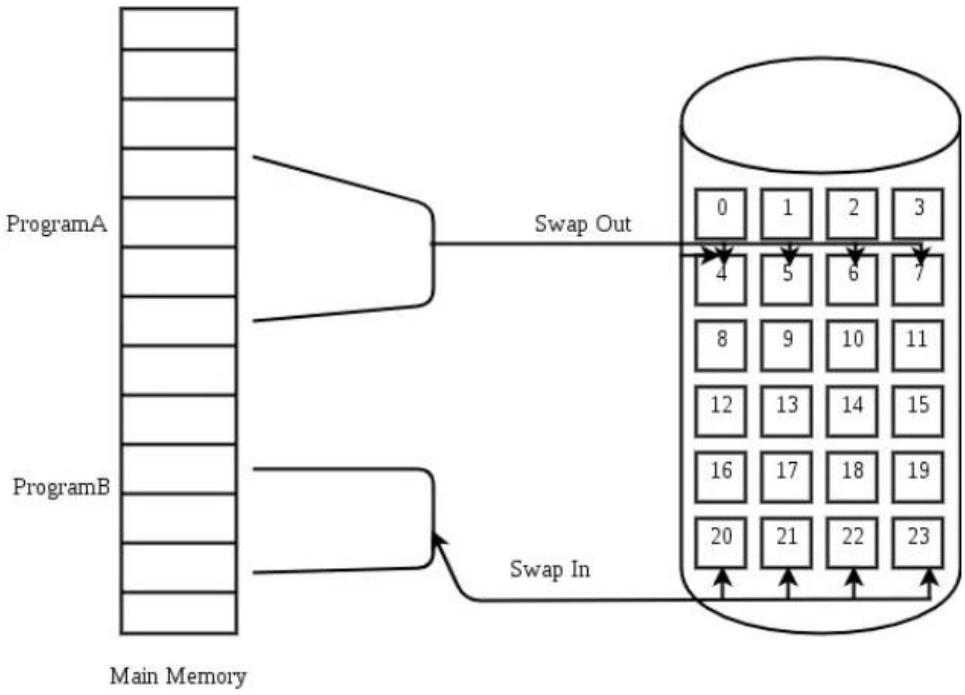
Swapping is a process out means removing all of its pages from memory, or marking them so that they will be removed by the normal page replacement process. Suspending a process ensures that it is not runnable while it is swapped out. At some later time, the system swaps back the process from the secondary storage to the main memory. When a process is busy swapping pages in and out then this situation is called thrashing.

What is Thrashing?

At any given time, only a few pages of any process are in the main memory, and therefore more processes can be maintained in memory. Furthermore, time is saved because unused pages are not swapped in and out of memory. However, the OS must be clever about how it manages this scheme. In the steady state practically, all of the main memory will be occupied with process pages, so that the processor and OS have direct access to as many processes as possible. Thus when the OS brings one page in, it must throw another out. If it throws out a page just before it is used, then it will just have to get that page again almost immediately. Too much of this leads to a condition called Thrashing. The system spends most of its time swapping pages rather than executing instructions. So a good page replacement algorithm is required.

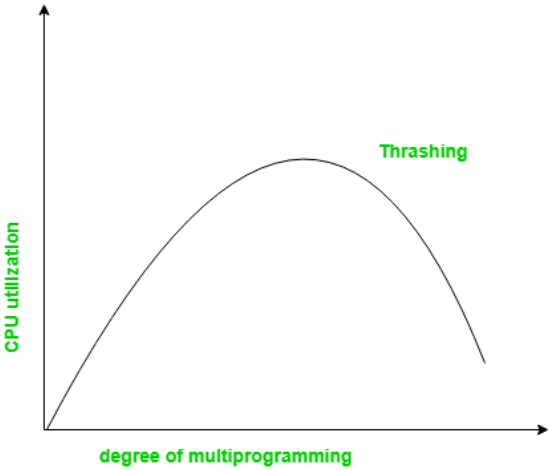
In the given diagram, the initial degree of multiprogramming up to some extent of point(lambda), the CPU utilization is very high and the system resources are utilized 100%. But if we further increase the degree of multiprogramming the CPU utilization will drastically

fall down and the system will spend more time only on the page replacement and the time taken to complete the execution of the process will increase. This situation in the system is called thrashing.



Causes of Thrashing

Thrashing occurs in a computer system when the CPU spends more time swapping pages in and out of memory than executing actual processes. This happens when there is insufficient physical memory, causing frequent page faults and excessive paging activity. Thrashing reduces system performance and makes processes run very slowly. There are many cause of thrashing as discussed below.



1. High Degree of Multiprogramming: If the number of processes keeps on increasing in the memory then the number of frames allocated to each process will be decreased. So, fewer frames will be available for each process. Due to this, a page fault will occur more frequently

and more CPU time will be wasted in just swapping in and out of pages and the utilization will keep on decreasing.

For example:
Let free frames = 400
Case 1: Number of processes = 100
Then, each process will get 4 frames.

Case 2: Number of processes = 400
Each process will get 1 frame.
Case 2 is a condition of thrashing, as the number of processes is increased, frames per process are decreased. Hence CPU time will be consumed just by swapping pages.

2. Lacks of Frames: If a process has fewer frames then fewer pages of that process will be able to reside in memory and hence more frequent swapping in and out will be required. This may lead to thrashing. Hence a sufficient amount of frames must be allocated to each process in order to prevent thrashing.

Recovery of Thrashing

- Do not allow the system to go into thrashing by instructing the long-term scheduler not to bring the processes into memory after the threshold.
- If the system is already thrashing then instruct the mid-term scheduler to suspend some of the processes so that we can recover the system from thrashing.

Performance in Virtual Memory

- Let p be the page fault rate(0 <= p <= 1).
- if p = 0 no page faults
- if p =1, every reference is a fault.

Effective access time (EAT) = (1-p)* Memory Access Time + p * Page fault time.
Page fault time = page fault overhead + swap out + swap in +restart overhead

The performance of a virtual memory management system depends on the total number of page faults, which depend on “paging policies” and “frame allocation“

Frame Allocation

A number of frames allocated to each process in either static or dynamic.

- **Static Allocation:** The number of frame allocations to a process is fixed.
- **Dynamic Allocation:** The number of frames allocated to a process changes.

Paging Policies

- **Fetch Policy:** It decides when a page should be loaded into memory.
- **Replacement Policy:** It decides which page in memory should be replaced.

- **Placement Policy:** It decides where in memory should a page be loaded.

What are the Applications of Virtual memory?

Virtual memory has the following important characteristics that increase the capabilities of the computer system. The following are five significant characteristics of Lean.

- **Increased Effective Memory:** One major practical application of virtual memory is, virtual memory enables a computer to have more memory than the physical memory using the disk space. This allows for the running of larger applications and numerous programs at one time while not necessarily needing an equivalent amount of DRAM.
- **Memory Isolation:** Virtual memory allocates a unique address space to each process and that also plays a role in process segmentation. Such separation increases safety and reliability based on the fact that one process cannot interact with and or modify another's memory space through a mistake, or even a deliberate act of vandalism.
- **Efficient Memory Management:** Virtual memory also helps in better utilization of the physical memories through methods that include paging and segmentation. It can transfer some of the memory pages that are not frequently used to disk allowing RAM to be used by active processes when required in a way that assists in efficient use of memory as well as system performance.
- **Simplified Program Development:** For case of programmers, they don't have to consider physical memory available in a system in case of having virtual memory. They can program 'as if' there is one big block of memory and this makes the programming easier and more efficient in delivering more complex applications.

How to Manage Virtual Memory?

Here are 5 key points on how to manage virtual memory:

1. Adjust the Page File Size

- **Automatic Management**: All contemporary operating systems including Windows contain the auto-configuration option for the size of the empirical page file. But depending on the size of the RAM, they are set automatically, although the user can manually adjust the page file size if required.
- **Manual Configuration:** For tuned up users, the setting of the custom size can sometimes boost up the performance of the system. The initial size is usually advised to be set to the minimum value of 1. To set the size of the swap space equal to 5 times the amount of physical RAM and the maximum size 3 times the physical RAM.

2. Place the Page File on a Fast Drive

- **SSD Placement:** If this is feasible, the page file should be stored in the SSD instead of the HDD as a storage device. It has better read and write times, and the virtual memory may prove benefecial in an SSD.

- **Separate Drive:** Regarding systems having multiple drives involved, the page file needs to be placed on a different drive than the os and that shall in turn improve its performance.

3. Monitor and Optimize Usage

- **Performance Monitoring:** Employ the software tools used in monitoring the performance of the system in tracking the amounts of virtual memory. High page file usage may signify that there is a lack of physical RAM or that virtual memory needs a change of settings or addition in physical RAM.
- **Regular Maintenance:** Make sure there is no toolbar or other application running in the background, take time and uninstall all the tool bars to free virtual memory.

4. Disable Virtual Memory for SSD

- **Sufficient RAM:** If for instance your system has a big physical memory, for example 16GB and above then it would be advised to freeze the page file in order to minimize SSD usage. But it should be done, in my opinion, carefully and only if the additional signals that one decides to feed into his applications should not likely use all the available RAM.

5. Optimize System Settings

- **System Configuration:** Change some general properties of the system concerning virtual memory efficiency. This also involves enabling additional control options in Windows such as adjusting additional system setting option on the operating system, or using other options in different operating systems such as Linux that provides different tools and commands to help in adjusting how virtual memory is utilized.
- **Regular Updates:** Ensure that your drivers are run in their newest version because new releases contain some enhancements and issues regarding memory management.

What are the Benifits of Using Virtual Memory?

- Many processes maintained in the main memory.
- A process larger than the main memory can be executed because of demand paging. The OS itself loads pages of a process in the main memory as required.
- It allows greater multiprogramming levels by using less of the available (primary) memory for each process.
- It has twice the capacity for addresses as main memory.
- It makes it possible to run more applications at once.
- Users are spared from having to add memory modules when RAM space runs out, and applications are liberated from shared memory management.
- When only a portion of a program is required for execution, speed has increased.

- Memory isolation has increased security.
- It makes it possible for several larger applications to run at once.
- Memory allocation is comparatively cheap.
- It doesn’t require outside fragmentation.
- It is efficient to manage logical partition workloads using the CPU.
- Automatic data movement is possible.

What are the Limitation of Virtual Memory?

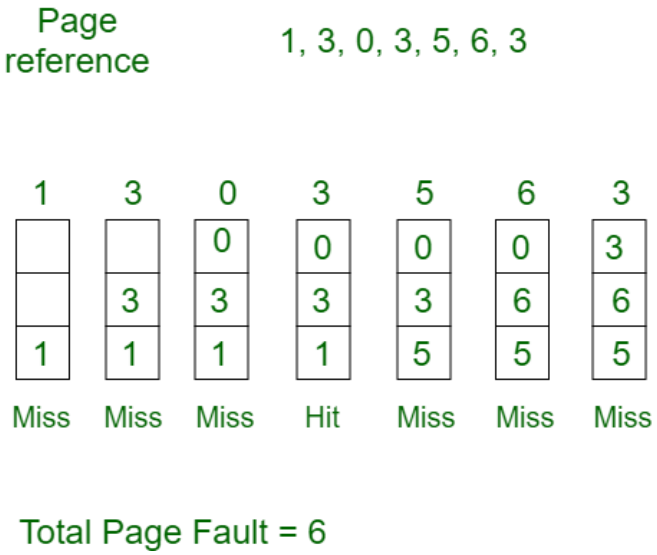
- It can slow down the system performance, as data needs to be constantly transferred between the physical memory and the hard disk.
- It can increase the risk of data loss or corruption, as data can be lost if the hard disk fails or if there is a power outage while data is being transferred to or from the hard disk.
- It can increase the complexity of the memory management system, as the operating system needs to manage both physical and virtual memory.

PAGE REPLACEMENT POLICIES

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.



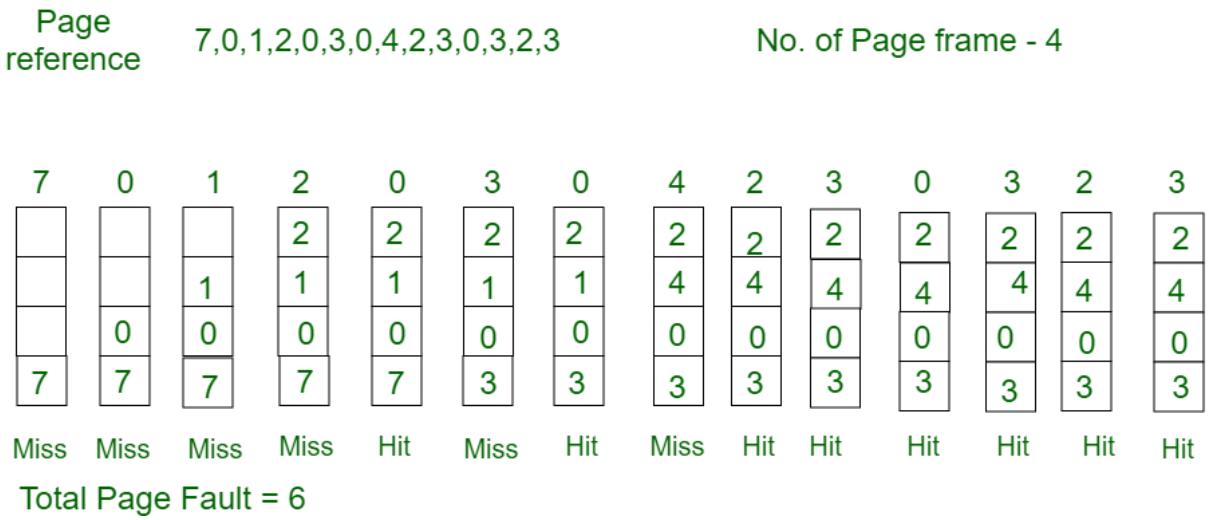
*Initially, all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> **3 Page Faults.***
*when 3 comes, it is already in memory so —> **0 Page Faults.** Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —> **1 Page Fault.** 6 comes, it is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

Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference strings 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4, and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10-page faults.

Optimal Page Replacement

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

Example-2: 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.



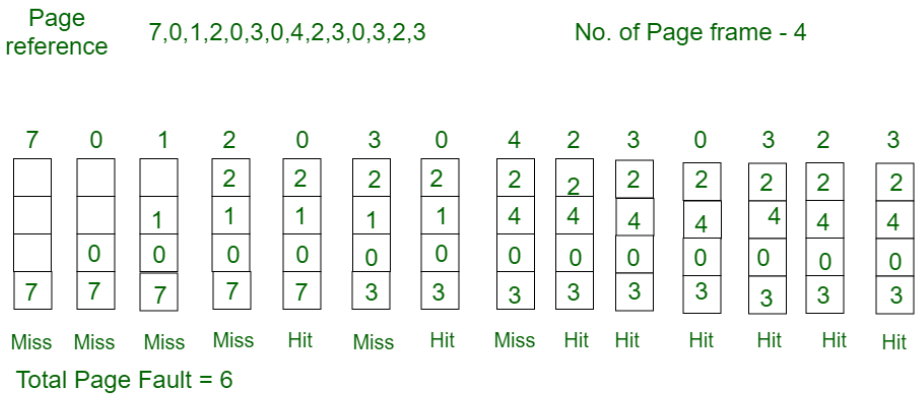
*Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults***
*0 is already there so —> **0 Page fault.** when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—> **1 Page fault.** 0 is already there so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



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

*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 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.*

WINDOWS MEMORY MANAGEMENT

Windows memory management is a vital component of the operating system that orchestrates the allocation, tracking, and retrieval of memory resources. It ensures that applications receive adequate memory for smooth operation, while optimizing resource use to handle multiple tasks concurrently.

A central concept in Windows memory management is virtual memory, which provides each application with its own virtual address space. This means that applications do not directly interact with physical memory but rather operate within a separate memory environment, such as a 4GB space in 32-bit systems or up to 128TB in 64-bit systems. When physical memory runs low, Windows employs a technique called paging. In paging, parts of the virtual memory are temporarily moved to a disk-based paging file, allowing the system to swap data between physical RAM and disk to create the effect of expanded memory. When data needed by a program is located in the paging file rather than RAM, a page fault occurs, and the system retrieves the data from disk.

Physical memory management is facilitated through a structure called the Page Frame Number (PFN) database, which tracks each physical memory page in RAM. Using this database, Windows can monitor which pages are actively in use. The system allocates memory both statically and dynamically, depending on process requirements, allowing for

efficient use of available physical RAM. To further manage memory efficiently, Windows uses memory pools, which are dedicated regions of memory allocated for specific tasks. The paged pool can be moved to disk when not in use, while the non-paged pool remains in physical RAM at all times, reserved for crucial kernel operations requiring immediate access to data.

Each process running in Windows has a "working set," which is the subset of its virtual memory currently held in physical memory. Windows dynamically adjusts this working set based on the system's memory availability and the process's activity. In low-memory situations, Windows may trim the working sets of less active processes, freeing up memory for those with higher demands. For applications that require access to more memory than the virtual address space allows, Address Windowing Extensions (AWE) enable programs to directly access large amounts of physical memory on 32-bit systems, a feature beneficial for high-performance applications.

Memory-mapped files are another important part of Windows memory management, allowing files to be mapped into a process’s address space. This approach enables fast file access by treating file data as part of memory, a technique especially useful for large files. For applications built on managed languages, such as those running in the .NET environment, Windows employs garbage collection to handle memory allocation dynamically, ensuring that unused memory is released. Additionally, memory in Windows is organized using heaps, where blocks are allocated and freed as needed.

Windows manages memory separately for user-mode and kernel-mode processes, with user-mode processes isolated from each other to prevent one application from corrupting another's memory. Kernel-mode components, however, share memory space, allowing them quick access to system resources. Memory protection features, such as Data Execution Prevention (DEP), prevent unauthorized access and execution of malicious code in memory areas intended for data only.

Finally, the Windows Memory Manager dynamically allocates memory based on system demand, balancing resources between kernel and user-mode applications. It employs demand paging and least-recently-used (LRU) algorithms to ensure that memory is used efficiently across all running processes. By combining these various strategies, Windows memory management achieves a balance between performance, security, and reliability, supporting stable system operation even under heavy workloads.

LINUX MEMORY MANAGEMENT

Linux memory management is a comprehensive system designed to efficiently handle memory allocation, protection, and optimization for running applications. It uses several advanced techniques to manage resources, balance system performance, and ensure stability. The foundation of Linux memory management is virtual memory, which provides each process with its own address space. This address space isolates applications from each other, enhancing security and stability by preventing processes from inadvertently accessing or modifying each other’s memory. Linux virtual memory is typically larger than physical

memory, allowing the system to extend available memory through a process known as swapping.

When physical memory (RAM) is fully used, Linux offloads less frequently accessed memory pages to a disk-based swap space. This frees up RAM for active processes while keeping swapped pages accessible when needed. If a process needs data that has been swapped to disk, a page fault occurs, prompting Linux to retrieve the data and load it back into RAM. This efficient use of active and swapped pages enables Linux to optimize memory resources.

Physical memory in Linux is managed through paging, with memory divided into small blocks, or pages, typically 4KB in size. These pages are tracked in a data structure called the page table, which maps virtual addresses to physical ones, allowing Linux to allocate and reclaim memory with minimal fragmentation. For kernel memory, Linux uses fixed-size allocation pools to provide quick and predictable access for essential processes. In user mode, Linux employs dynamic memory allocation strategies, such as the Buddy System and Slab Allocator. The Buddy System divides memory into blocks of sizes that are powers of two, which can be split or merged as needed. The Slab Allocator is efficient for managing frequently allocated and freed small memory objects, making it ideal for caching.

Linux also optimizes performance through caching and buffering. Data read from disk is stored in a page cache, reducing future disk access times and making frequently used files more accessible from memory. The Virtual Memory Manager (VMM) is responsible for allocating and deallocating virtual memory pages, determining which pages to keep in physical memory and which to swap to disk. By tracking page access patterns, the VMM prioritizes pages that are accessed frequently and swaps out pages that are rarely used.

Memory protection is another crucial aspect of Linux memory management. Each process has its own protected memory space, preventing unauthorized access and modification. Linux implements security features like Kernel Address Space Layout Randomization (KASLR) and Data Execution Prevention (DEP) to prevent code execution in non-executable memory areas, adding a layer of protection against memory-related vulnerabilities.

Additionally, Linux supports Transparent Huge Pages (THP), which combines smaller pages into larger "huge" pages for applications that require large contiguous memory. This approach improves performance by reducing Translation Lookaside Buffer (TLB) misses. When Linux runs low on memory, the Out-of-Memory (OOM) killer may be invoked to terminate processes with high memory usage, ensuring system stability by freeing up memory for essential tasks. Lastly, Linux facilitates efficient communication between processes through shared memory, a feature of Inter-Process Communication (IPC) that enables applications to quickly pass large amounts of data without duplicating it in memory.

In summary, Linux memory management is a sophisticated system that integrates virtual memory, dynamic allocation, paging, and caching to provide a robust and secure memory environment. These mechanisms allow Linux to efficiently utilize resources, support multitasking, and adapt to various workloads, from desktop environments to large-scale servers, ensuring a stable and responsive operating system.