1. **Memory management and Techniques(First Fit, Best Fit, Worst Fit)**

**Memory Management** involves handling and optimizing the use of the main memory. Its primary objective is to ensure that programs run efficiently by allocating memory spaces dynamically, keeping track of each byte in a computer's memory, and maximizing system performance.

Effective memory management is essential to avoid scenarios like memory leaks, which can degrade system performance over time, or memory corruption, which can lead to system crashes and unpredictable behavior.

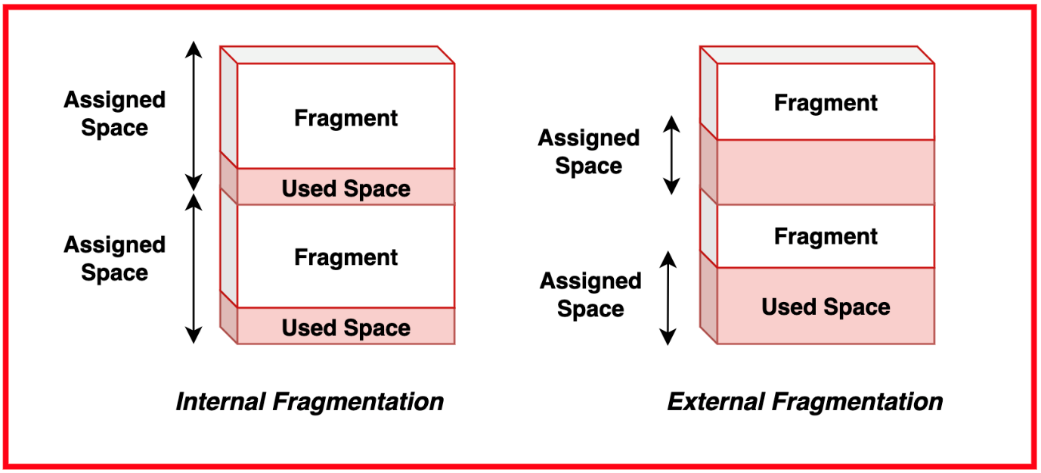
**Memory Allocation Strategies**

Memory allocation strategies are methods used to assign memory blocks to various programs and processes within a computer system. The most common strategies include static allocation, where memory is assigned at compile time, and dynamic allocation, where memory is allocated at runtime.

These strategies are crucial in managing the limited resource of memory, ensuring that all running applications receive the necessary resources while reducing wastage. Dynamic memory allocation techniques, such as heap allocation and stack allocation, allow for flexible and efficient use of memory, adapting to the changing needs of applications.

**Fragmentation: Internal and External**







Fragmentation in memory management refers to the inefficient use of memory that reduces the amount of usable memory. It occurs in two forms: internal and external fragmentation.

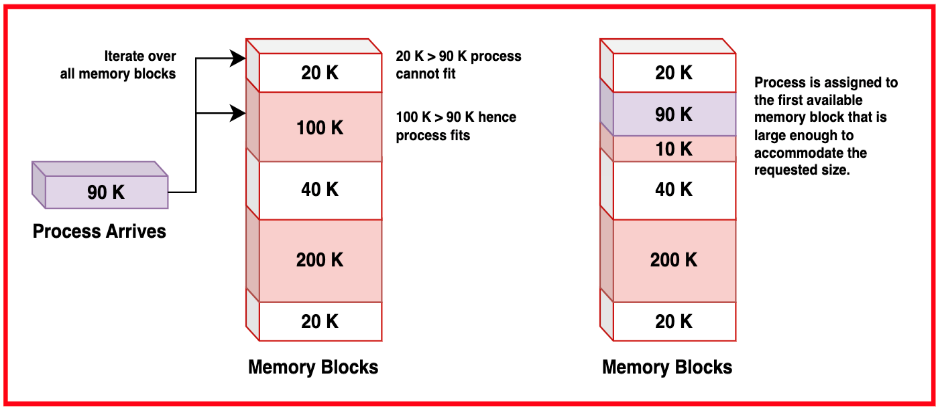
* **Internal Fragmentation**: Happens when allocated memory blocks have unused space within them, typically because the memory requested is slightly smaller than the allocated block size.
* **External Fragmentation**: Occurs when free memory is divided into small, non-contiguous blocks, making it challenging to find a block large enough to satisfy a memory request, despite having sufficient total free memory.

Both types of fragmentation can significantly impact system performance by reducing the effective memory available for applications.

**First Fit Algorithm**

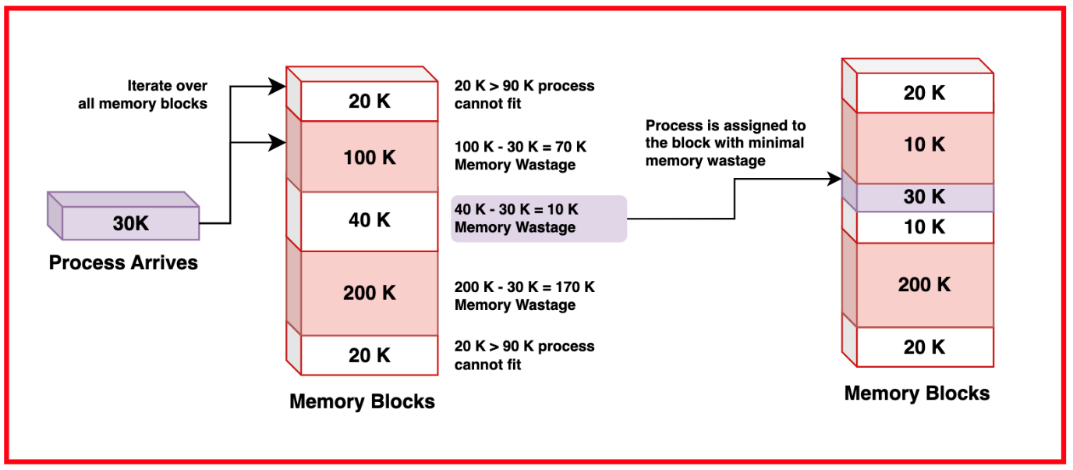
The **First Fit** algorithm is a memory allocation strategy that assigns the first available memory block that is large enough to accommodate the requested size. It scans memory from the beginning and chooses the first sufficiently large block it encounters. This approach is simple and generally fast since it minimizes the search time.

However, it can lead to fragmentation over time, as the algorithm tends to leave small unusable gaps in the memory, especially when there are numerous allocation and deallocation operations.



**Best Fit Algorithm**

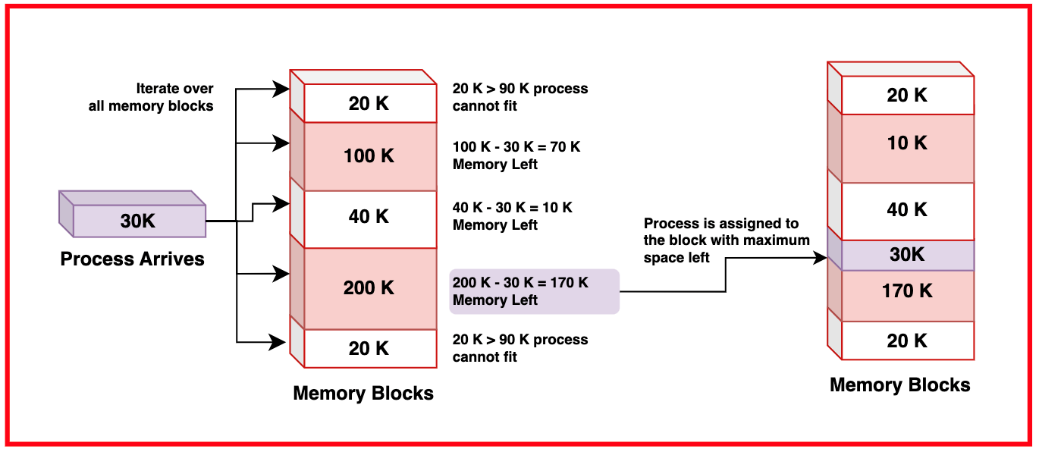
The **Best Fit** algorithm allocates memory by searching for the smallest free block that is large enough to satisfy the memory request. This strategy aims to reduce wasted space by using the most appropriately sized block available.



While this method can reduce internal fragmentation, it often increases external fragmentation because it tends to leave very small gaps that are difficult to use for future requests. Additionally, the Best Fit algorithm may involve a more extensive search, making it potentially slower than other allocation methods.

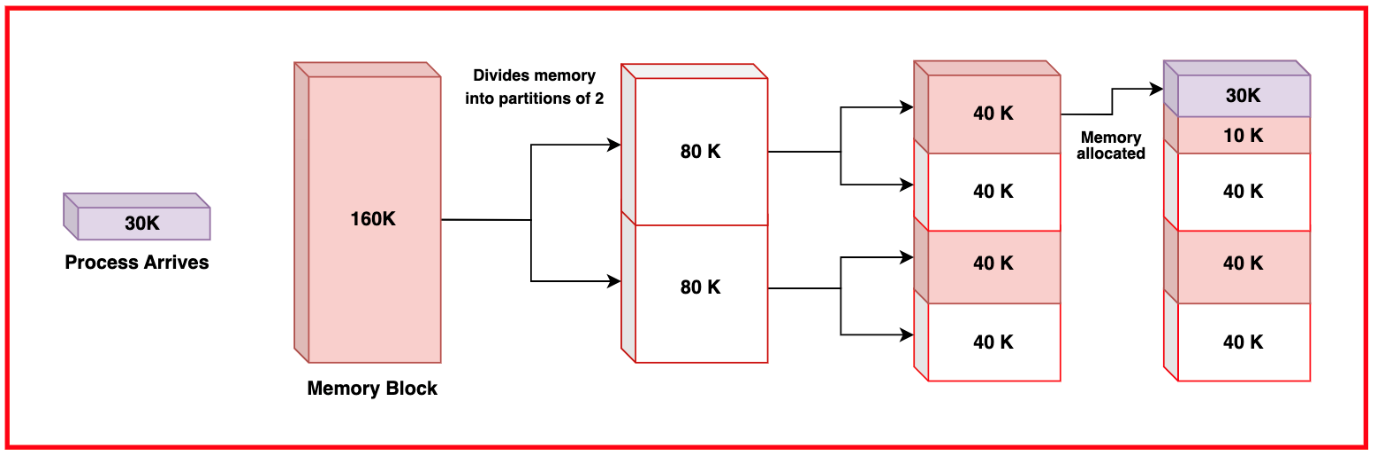
**Worst Fit Algorithm**

The **Worst Fit** algorithm allocates memory by searching for the largest available block that can satisfy the memory request. The idea is to leave the largest possible leftover block after allocation, which might be more useful for subsequent requests. This approach can help reduce external fragmentation by avoiding the creation of small, unusable gaps. However, it can lead to inefficient memory usage if the largest blocks are not split effectively, and it often requires a longer search time compared to simpler algorithms like First Fit.



**Buddy System Allocation**

The **Buddy System** is a memory allocation technique that divides memory into partitions of size that are powers of two. When a request is made, the system splits the available block into smaller "buddies" until it finds a block size that best fits the request.



If the allocated block is later freed, and its buddy is also free, the system merges them back into a larger block. This approach helps manage fragmentation by simplifying the coalescing of free memory blocks and maintaining a structured method for splitting and merging memory. The Buddy System balances the need for efficiency and simplicity in managing memory allocation.

**Slab Allocation**

**Slab Allocation** is a memory management technique used primarily in kernel memory allocation, where efficiency and speed are crucial. It involves pre-allocating chunks of memory, called slabs, for objects of the same size. When an object is needed, it is taken from a pre-allocated slab, and when it is no longer needed, it is returned to the slab.

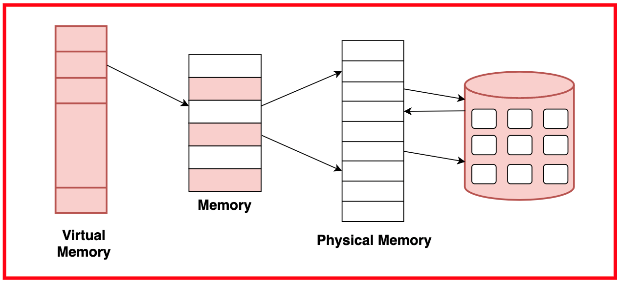
This method reduces fragmentation and overhead by minimizing frequent allocations and deallocations, providing a fast and predictable allocation pattern. Slab allocation is particularly effective in environments where objects of fixed sizes are frequently created and destroyed, such as in operating systems.

1. **Virtual Memory**

**Virtual memory** is a memory management technique that provides an "idealized abstraction of the storage resources" that are actually available on a given machine, creating the illusion to users of a very large (main) memory.

This is achieved by using both the computer's physical memory (RAM) and a portion of the storage device (hard disk or SSD). When a program requires more memory than what is physically available, the operating system temporarily transfers inactive parts of the program's data to a pre-configured space on the disk, known as the swap space or page file.

This allows for more efficient and flexible use of memory, enabling systems to run larger applications and multitask more effectively.

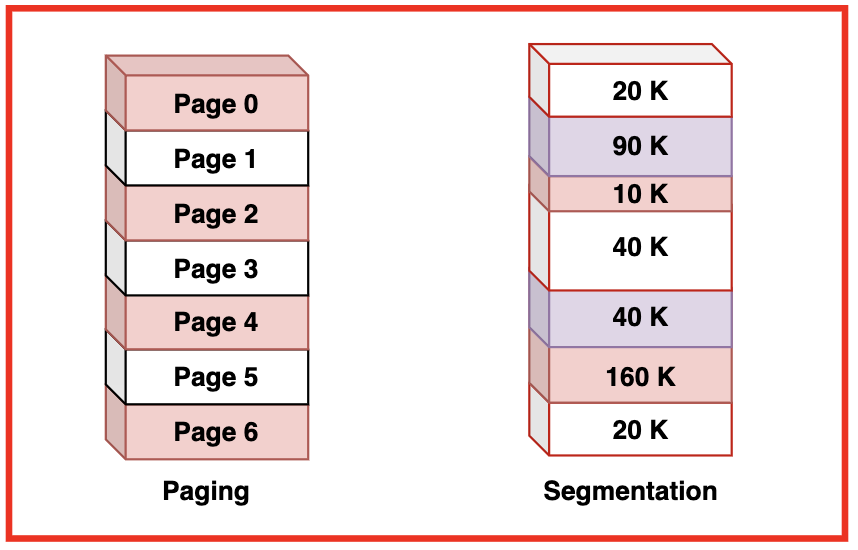


**Advantages of Virtual Memory**

* Increases the effective memory capacity.
* Allows larger applications to run on systems with limited physical RAM.
* Enables more programs to run simultaneously without exhausting physical memory.
* Creates a layer of abstraction between physical hardware and applications.
* Improves system stability by preventing applications from interfering with each other’s memory.
* Enhances security and reliability, ensuring one process’s crash or misbehavior doesn’t affect others.

**Paging vs Segmentation**

Paging and segmentation are two approaches to memory management in operating systems.



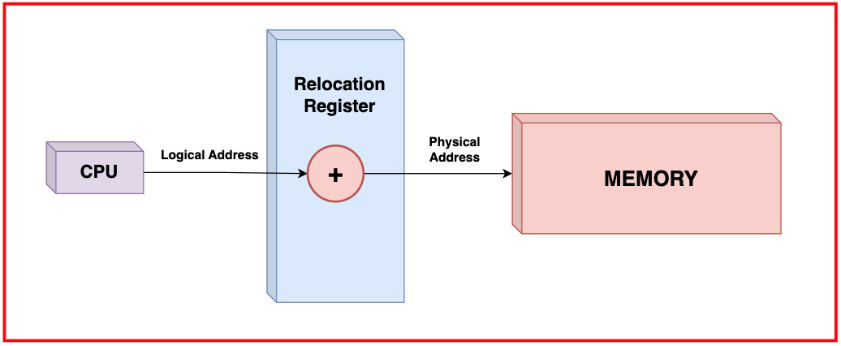
**Paging** divides the virtual memory into fixed-size blocks called pages, while the physical memory is divided into blocks of the same size called frames. This simplifies memory allocation and management, as any page can be loaded into any frame, eliminating fragmentation issues.

**Segmentation**, on the other hand, divides the memory into variable-sized segments based on logical divisions, such as functions, arrays, or modules. Each segment can grow or shrink independently, providing a more logical way of memory allocation that matches the program structure but may suffer from fragmentation and complex memory management.

**Comparison: Paging vs Segmentation**

| **Feature** | **Paging** | **Segmentation** |
| --- | --- | --- |
| **Memory Division** | Fixed-sized blocks called pages | Variable-size blocks called segments |
| **Physical Memory** | Divided into frames of the same size as pages | Divided into segments of varying sizes |
| **Address Structure** | Single-level address with page number and offset | Two-level address with segment number and offset |
| **Fragmentation** | Eliminates external fragmentation, may cause internal fragmentation within pages | Can lead to external fragmentation, but no internal fragmentation |
| **Logical Division** | Divides memory without considering logical structure | Divides memory according to logical structure (e.g., functions, arrays) |
| **Ease of Management** | Simpler due to fixed-size pages | More complex due to variable-size segments |
| **Protection and Sharing** | Easier to manage protection and sharing at page level | More intuitive for logical groupings but complex |
| **Memory Access** | Uniform size makes access time consistent | Variable sizes can lead to varying access times |
| **Use Case** | Commonly used in modern operating systems | Less common, used for specific applications requiring logical division |

**Logical and Physical Address Space**





**Logical Address Space** is also known as the virtual address space, is a set of addresses that a program can use to access memory. These addresses are generated by the CPU during program execution and are independent of the actual physical memory addresses. The logical address space provides an abstraction layer, allowing programs to use a continuous and consistent range of addresses, which simplifies programming and enhances portability.

**Physical Address Space** is the actual location in the computer’s physical memory (RAM). These addresses are used by the memory management unit (MMU) to access data stored in the system's hardware memory. The physical address space is finite and directly corresponds to the physical hardware installed in the machine.

Logical Address space is a virtualized abstraction used by the software, while the physical address space is the actual memory hardware. These combined allow the operating system to manage memory more efficiently, providing each process with its own logical address space, mapped onto the physical memory by the Memory Management Unit.

**Address Translation**

**Address Translation** is the process of converting logical addresses into physical addresses. This translation is typically handled by the memory management unit (MMU) and involves the following steps:

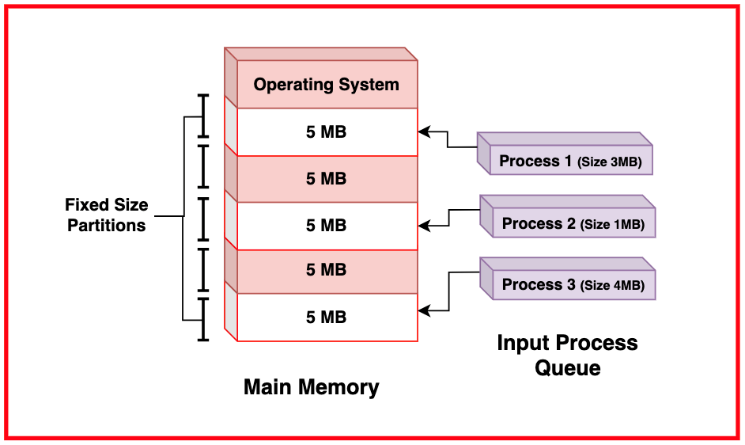
* The CPU generates a logical address during program execution. The address is divided into two parts: the page number and offset.
* The MMU uses the page number to index into the page table, a data structure that maintains the mapping between logical pages and physical frames. Each entry in the page table corresponds to a page frame number in physical memory.
* The frame number obtained from the page table is combined with the offset to form the complete physical address. This address points to the exact location in physical memory where the data is stored.
* The physical address is used to access the data in the physical memory, completing the translation process.

Address translation allows the operating system to provide each process with its own logical address space, ensuring that processes do not interfere with each other’s memory. It also enables features like paging and segmentation, which help in efficient memory management and protection

1. **Contiguous allocation , Paging and Segmentation**

**Contiguous Memory Allocation**

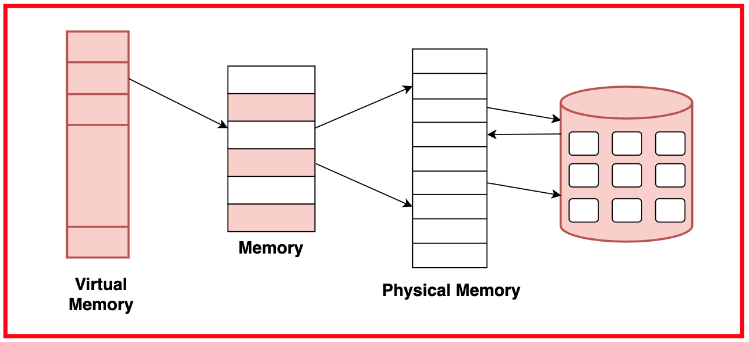
Contiguous Memory Allocation is a memory management technique where each process is allocated a single continuous block of memory. This approach is simple and efficient in terms of access speed because the CPU can easily calculate the address of any location within the block.



However, contiguous memory allocation suffers from fragmentation.Additionally, the process of fitting processes into memory can be complex, requiring strategies like first-fit, best-fit, or worst-fit to manage available memory effectively.

**Paging**

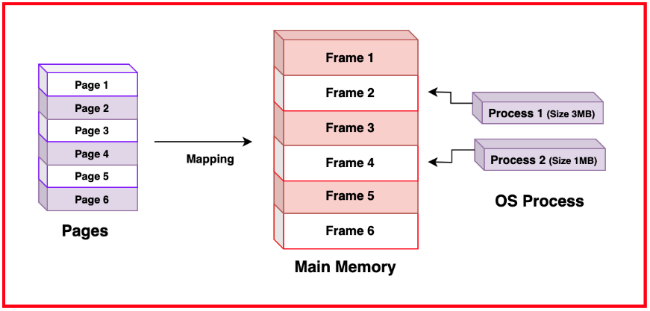
Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. In paging, the physical memory is divided into fixed-size blocks called frames, and the logical memory (used by processes) is divided into blocks of the same size called pages. When a process is loaded into memory, its pages can be loaded into any available memory frames, and a page table keeps track of the mapping between the process’s pages and the physical frames.



This allows the physical address space to be non-contiguous, which solves the problem of external fragmentation and makes it easier to allocate memory efficiently. Paging also simplifies memory management by eliminating the need for complex allocation algorithms.

**Page Table Structure**

The page table is a data structure used in paging to store the mapping between virtual addresses and physical addresses. Each entry in a page table corresponds to a page of the process's logical address space and contains the address of the frame in physical memory where the page is stored.

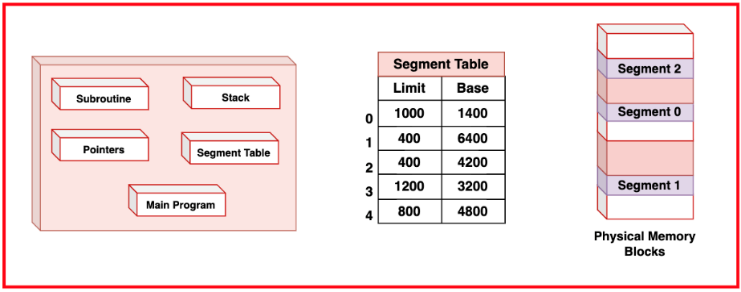


Page tables can be implemented in several ways, including single-level, multi-level, and inverted page tables. A single-level page table is a simple array, but it can become very large if the address space is large.

* **Advantages:**
  + Avoids gaps between allocated memory blocks, ensuring efficient use of available memory.
  + Pages can be placed anywhere in physical memory, simplifying memory allocation.
  + Fixed-size pages mean that memory allocation does not require complex algorithms.
  + Allows larger programs to run even with limited physical memory.
  + Virtual memory allows parts of a program to be stored on disk and brought into memory as needed.
* **Disadvantages:**
  + Each process requires a page table, which can consume a lot of memory for large address spaces.
  + Each memory access requires a lookup in the page table, which can add delay.
  + Unused space in the last page of a process can lead to internal fragmentation.

**Segmentation: Concept and Organisation**

Segmentation is a memory management technique that divides the process's memory into variable-sized segments, each of which can be a logical unit such as a function, array, or data structure. Unlike paging, segments vary in length, reflecting the logical structure of the process.



Each segment has a segment number and a length, and memory addresses within a segment are specified by an offset from the segment's base address.

A segment table keeps track of each segment's base address and length. Segmentation facilitates sharing and protection, as segments can be independently protected and shared between processes, enhancing modularity and security.

* **Advantages:**
  + Improves program readability and manageability.
  + Each segment can have its own access rights, enhancing security.
  + Reduces wasted space within segments.
* **Disadvantages:**
  + Can suffer from external fragmentation.
  + Requires more complex algorithms for allocation and deallocation.

**Paging vs Segmentation**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Paging** | **Segmentation** |
| Memory Division | Fixed-sized blocks called pages | Variable-size blocks called segments |
| Physical Memory | Divided into frames of the same size as pages | Divided into segments of varying sizes |
| Address Structure | Single-level address with page number and offset | Two-level address with segment number and offset |
| Fragmentation | Eliminates external fragmentation, may cause internal fragmentation within pages | Can lead to external fragmentation, but no internal fragmentation |
| Logical Division | Divides memory without considering logical structure | Divides memory according to logical structure (e.g., functions, arrays) |
| Ease of Management | Simpler due to fixed-size pages | More complex due to variable-size segments |
| Protection and Sharing | Easier to manage protection and sharing at page level | More intuitive for logical groupings but complex |
| Memory Access | Uniform size makes access time consistent | Variable sizes can lead to varying access times |
| Use Case | Commonly used in modern operating systems | Less common, used for specific applications requiring logical division |

1. **Dynamic binding**

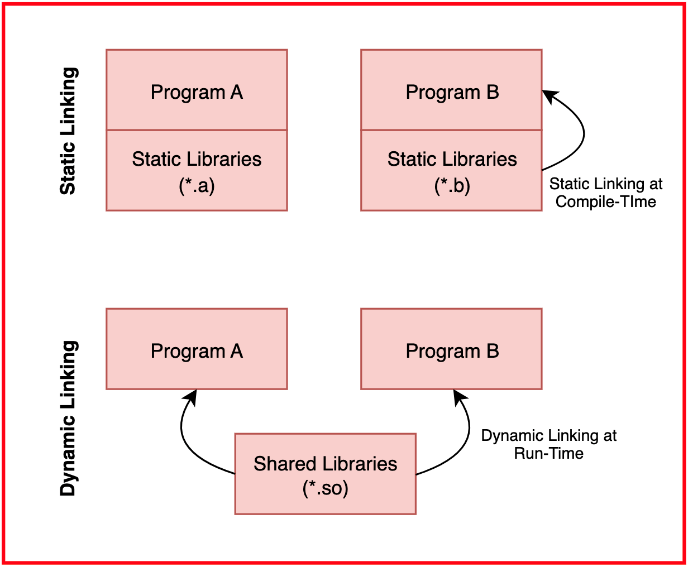
Dynamic Binding, also known as **late binding**, is a programming mechanism where the method to be called is determined at runtime, not during compile-time. It is a key part of **polymorphism** in object-oriented programming, allowing objects to be handled more generically.

For example, a base class pointer can refer to objects of derived classes, and the correct method is chosen based on the object's type at runtime.

**Benefits of Dynamic Binding:**

* Enhances flexibility and extensibility, as new classes and methods can be added with minimal changes.
* Supports modular and maintainable code, which is crucial for modern software development.



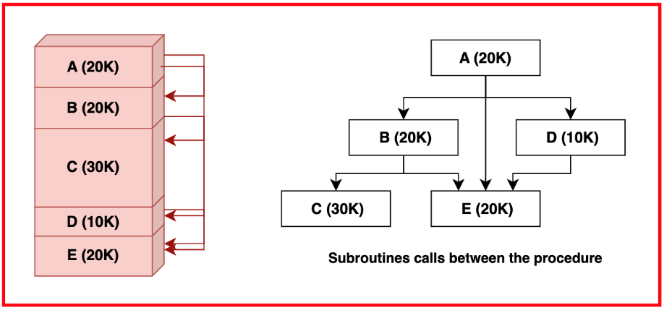




**Dynamic Linking vs Dynamic Loading**

**Dynamic Linking** links required libraries or modules to a program at runtime, instead of during compile-time. The compiler includes references to shared libraries, and the operating system loads these libraries when the program starts.

**Dynamic Loading** is when an application explicitly loads or unloads libraries at runtime. This is often used for loading plugins or optional modules.



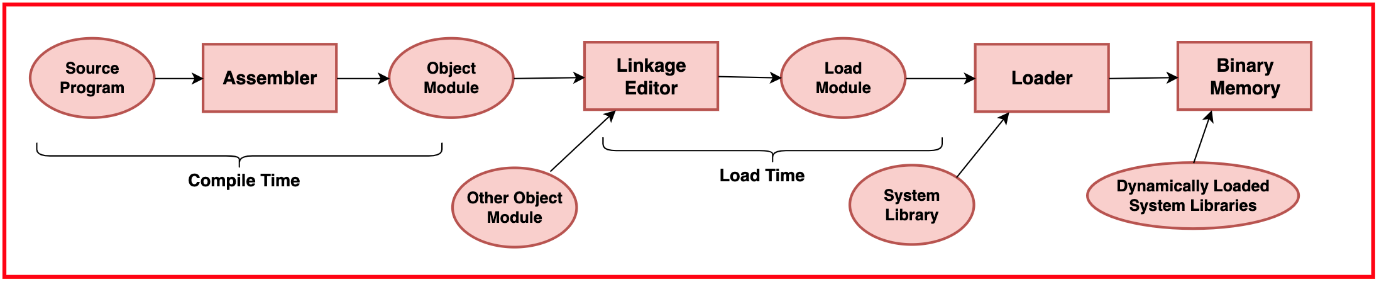
**Differences between Dynamic Linking and Dynamic Loading**

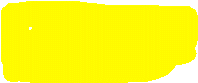
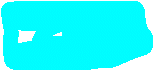
* **Dynamic Linking:** Automatically managed by the OS at program start, shares library code across multiple programs.
* **Dynamic Loading:** Controlled by the application, ideal for plugins and modules that load based on user needs.
* **Dynamic Linking** saves disk space by not duplicating library code.
* **Dynamic Loading** is highly modular, allowing for precise control over loaded modules.
* **Dynamic Linking** may have version issues ("DLL Hell"), while Dynamic Loading reduces these issues.

**Late Binding vs Early Binding**

**Late binding**, or dynamic binding, happens when the method or function to be called is determined at runtime. This is common in object-oriented programming using virtual functions or interfaces.

**Early binding**, or static binding, happens when the method or function is determined during compilation. The method to be called is known at compile-time, which results in faster execution.





**Comparison between Late Binding and Early Binding**

* **Late Binding:** Determined at runtime, supports polymorphism and flexibility, but may have runtime errors.
* **Early Binding:** Determined at compile-time, faster due to no method lookup overhead, but less flexible.
* **Late binding** is used in languages like Python and JavaScript.
* **Early binding** is common in languages like Java and C# for performance-critical applications.

**Benefits of Dynamic Binding**

* **Greater Flexibility:** It allows methods to be chosen based on the actual runtime context, adapting to different scenarios.
* **Supports Polymorphism:** Enables different classes to be treated the same way, which allows code reuse and easier extensibility.
* **Simplifies Code Maintenance:** New methods can be added without changing existing code.
* **Useful in Dynamic Languages:** Languages that determine types at runtime use dynamic binding for flexibility.

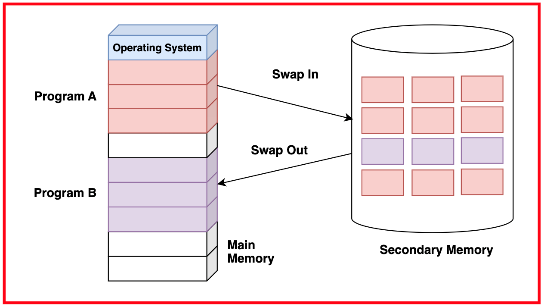
**Drawbacks of Dynamic Binding**

* **Runtime Overhead:** Method lookup can slow down execution, which may be a concern in performance-critical applications.
* **Errors Surface During Runtime:** Debugging can be harder since errors are found when the program runs.
* **Complex Code Management:** Managing dynamic method resolution adds complexity, making code harder to understand in large projects.
* **Risk of Type Errors:** In dynamically typed languages, it’s easier to call methods incorrectly, leading to runtime errors

1. **Page Fault**

A page fault is an exception that occurs when a program accesses a memory page that is not currently present in physical memory (RAM). In modern operating systems that use virtual memory management, memory is divided into fixed-size pages, and only the pages that are actively being used by a process are kept in physical memory. When a program attempts to access a page that is not in RAM, a page fault occurs, triggering the operating system to load the required page from secondary storage (such as a hard disk or SSD) into physical memory.

Page faults are a fundamental part of virtual memory systems, allowing programs to address more memory than is physically available. They enable efficient use of memory resources by swapping pages in and out of physical memory as needed, based on the demands of running processes. While page faults incur some overhead due to the need to access slower secondary storage, they are essential for providing the illusion of a larger memory space to processes than actually exists in physical memory.

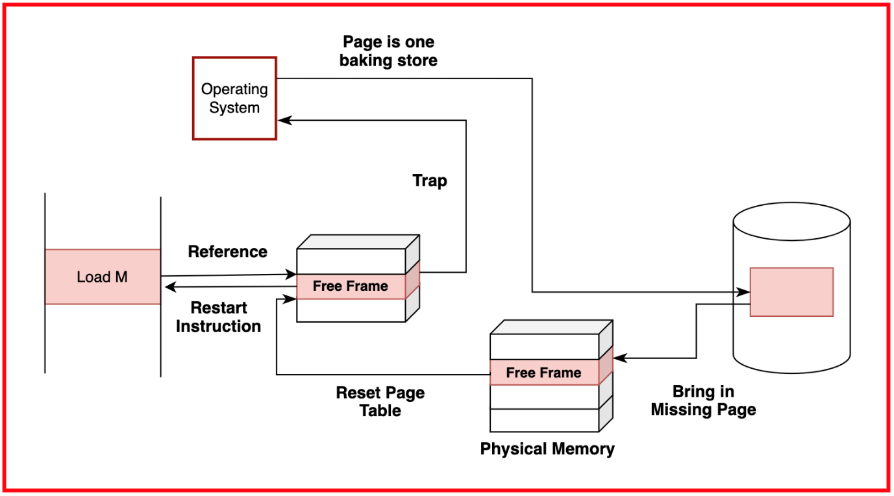




**Causes of Page Faults**

* **Demand Paging:** Most modern operating systems use demand paging, where only the pages of memory that are actively being used by a process are loaded into physical memory. As a result, when a process accesses a page that is not currently in RAM, a page fault occurs, and the required page is fetched from disk into memory.
* **Page Replacement:** If physical memory is full and a process needs to bring in a new page, the operating system must choose a page to evict from memory to make space. This involves selecting a victim page for replacement, often based on algorithms like Least Recently Used (LRU) or First-In, First-Out (FIFO).
* **Copy-on-Write:** Some systems use copy-on-write mechanisms, where multiple processes can share the same memory page until one of them attempts to modify it. When a modification occurs, the page is copied, leading to a page fault for the modifying process.

**Handling Page Faults**





When a page fault occurs, the operating system intervenes to handle the exception and ensure that the required page is brought into physical memory. The steps involved in handling a page fault are:

* **Page Table Lookup:** The operating system consults the page table of the process to determine whether the accessed memory address corresponds to a valid page in physical memory or if it has been swapped out to disk.
* **Disk Access:** If the required page is not in physical memory, the operating system initiates a disk access to read the required page from secondary storage into a free page frame in physical memory.
* **Updating Page Table:** Once the required page is loaded into physical memory, the operating system updates the page table entry for the accessed memory address to indicate that the page is now resident in RAM.
* **Resuming Execution:** Finally, the operating system restarts the instruction that caused the page fault, allowing the process to continue its execution with the required page now available in physical memory.

**Impact of Page Faults on System Performance**

Page faults can have a significant impact on system performance, primarily due to the latency introduced by disk accesses compared to accessing data from physical memory. When a page fault occurs, the CPU must wait for the required page to be fetched from secondary storage, leading to a temporary pause in program execution known as a "page fault stall."

Frequent page faults can result in decreased overall system performance, as the CPU spends more time waiting for data to be fetched from disk rather than executing program instructions. To mitigate the impact of page faults on performance, operating systems employ various techniques, such as:

* **Optimising page replacement algorithms**
* **Utilising disk caching**
* **Employing prefetching strategies** to anticipate and load pages into memory before they are needed

Additionally, system administrators may tune system parameters, such as the size of the page file or swap space, to balance memory usage and disk access latency. Overall, managing page faults effectively is crucial for maintaining system performance and ensuring efficient utilisation of memory resources

1. **Page replacement algorithms( LRU, Optimal, FIFO)**

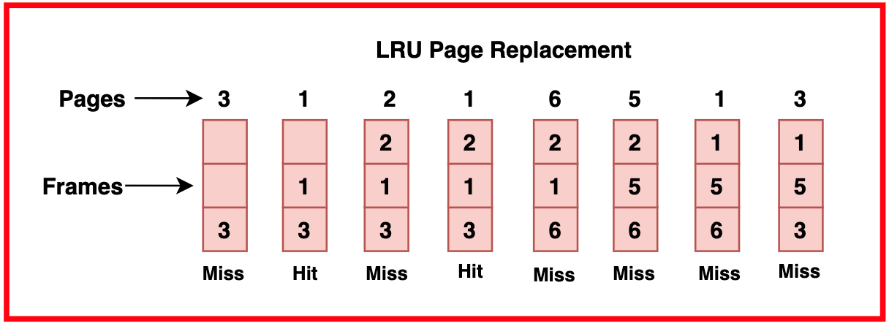
It involves the process of selecting a page in physical memory to be evicted or swapped out to secondary storage (such as a hard disk or SSD) when a new page needs to be brought into memory.

Page replacement directly impacts system performance, as it involves accessing slower secondary storage to swap pages in and out of physical memory. Frequent page replacement can lead to increased disk I/O and latency, which can degrade overall system performance. Therefore, selecting an efficient page replacement algorithm is crucial for optimising system performance and minimising the impact of page faults on system responsiveness. Additionally, system designers may employ techniques such as prefetching, caching, or optimising disk access to mitigate the performance impact of page replacement and improve overall system efficiency.

Least Recently Used (LRU) Algorithm

The Least Recently Used (LRU) algorithm is a page replacement policy used in virtual memory management. It operates on the principle that pages that have been least recently used are the ones least likely to be used in the near future. When a page needs to be replaced due to a page fault, the LRU algorithm selects the page in memory that has not been accessed for the longest period and replaces it with the new page.

LRU can be implemented using various data structures, such as linked lists or hash maps, to track the order of page accesses. Each time a page is accessed, it is moved to the front of the list or marked as most recently used. When a page fault occurs, the algorithm selects the page at the end of the list or the least recently used page for replacement.

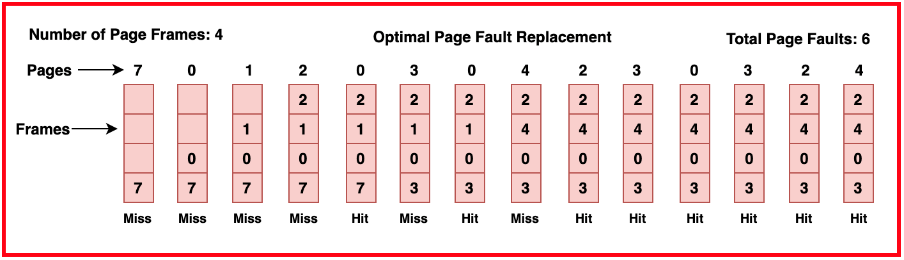


LRU is relatively simple to implement and often performs well in practice, especially when there is good temporal locality in memory access patterns.

The main drawback of the LRU algorithm is its high implementation complexity and the overhead of maintaining the access history for every page in memory. Additionally, LRU may suffer from "thrashing" in scenarios where the working set of a process exceeds the available physical memory, leading to frequent page faults and poor performance.

Optimal Algorithm

The Optimal algorithm, also known as the MIN or MINOPT algorithm, is a theoretical page replacement policy that selects the page for replacement that will not be accessed for the longest time in the future. In other words, it replaces the page whose next access is farthest in the future.



The Optimal algorithm requires knowledge of future page access patterns, which is usually not feasible in practice. Therefore, it is often used as a benchmark for comparing the performance of other page replacement algorithms rather than as a practical solution.

Optimal algorithm provides the lowest possible page fault rate among all page replacement algorithms, as it always selects the optimal page to replace. The main drawback of the Optimal algorithm is its impracticality for real-world systems, as it requires knowledge of future page accesses, which is generally not available. Therefore, it is primarily used as a reference point for evaluating the performance of other algorithms.

First-In-First-Out (FIFO) Algorithm

The First-In-First-Out (FIFO) algorithm is a simple page replacement policy that replaces the oldest page in memory when a page fault occurs. It operates on the principle of a queue, where the page that was brought into memory earliest is the first one to be replaced.

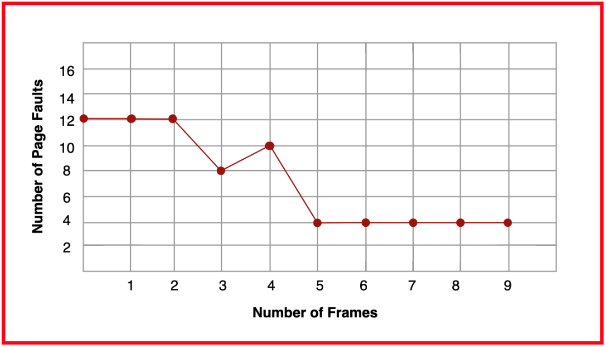
FIFO is typically implemented using a queue data structure, where new pages are added to the back of the queue, and the page at the front of the queue (the oldest page) is replaced when a page fault occurs.

FIFO is easy to implement and requires minimal bookkeeping, making it suitable for systems with limited resources or where simplicity is prioritised over performance.

FIFO suffers from the "Belady's Anomaly," where increasing the number of page frames may lead to an increase in page faults rather than a decrease. This anomaly occurs because the FIFO algorithm does not consider the frequency of page accesses or the temporal locality of memory references

1. **Belady's Anomaly**

Belady's Anomaly is a phenomenon observed in page replacement algorithms, where increasing the number of page frames allocated to a process can unexpectedly lead to an increase in the number of page faults, rather than a decrease as expected. It challenges the intuitive notion that providing more memory resources should result in better performance by reducing the frequency of page faults.



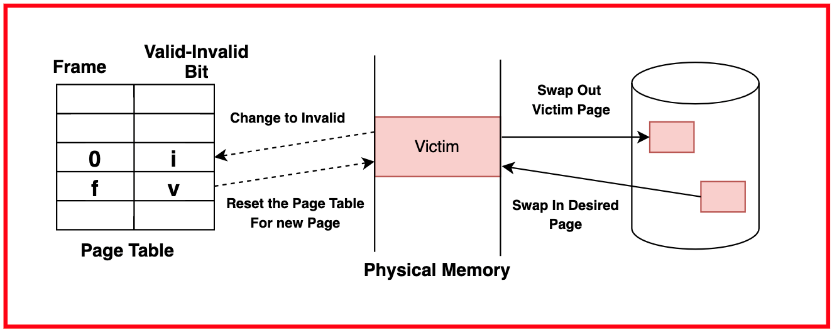
Belady's Anomaly occurs when a page replacement algorithm, such as First-In-First-Out (FIFO), replaces a page that will be needed soon, despite having additional free page frames available. This anomaly highlights the limitations of certain page replacement policies that do not consider the future access patterns of pages. In other words, increasing the number of page frames can result in pages being evicted prematurely, leading to more frequent page faults.

Example

Consider a scenario where a process accesses a sequence of memory pages. With a smaller number of page frames available, the page replacement algorithm may evict pages that are not immediately needed to make space for incoming pages, resulting in a certain number of page faults. However, increasing the number of page frames may unexpectedly lead to more page faults, as the algorithm now has more options to choose from and may evict pages that would have been needed in the near future.

Causes

* Page Replacement Policies: The choice of page replacement algorithm significantly influences the occurrence of Belady's Anomaly. Algorithms like FIFO (First-In-First-Out) do not consider the future access patterns of pages and may evict pages that will be needed soon, leading to an increase in page faults when the number of page frames is increased.
* Lack of Future Access Prediction: Many page replacement algorithms, including FIFO, are based solely on past page access patterns. They do not take into account future page accesses or the temporal locality of memory references, making it difficult to predict which pages will be needed in the near future.
* Allocation of Additional Memory: Increasing the number of page frames allocated to a process can exacerbate Belady's Anomaly if the page replacement algorithm is not designed to handle the additional memory effectively. The algorithm may evict pages that would have been needed soon, resulting in more frequent page faults despite the availability of additional memory resources.

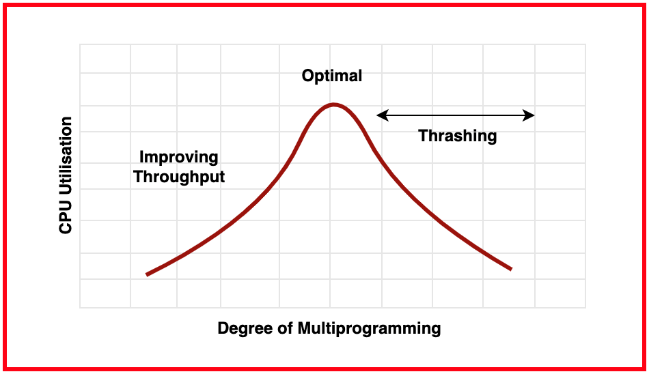


Implications of Belady’s Anomaly

* Unexpected Performance Degradation: Belady's Anomaly can lead to unexpected performance degradation in virtual memory systems, where increasing the amount of allocated memory does not result in the expected reduction in page faults. This can be particularly problematic in scenarios where system administrators allocate additional memory to improve performance, only to find that it has the opposite effect.
* Challenges in System Optimization: Belady's Anomaly presents challenges in optimising virtual memory systems for performance. It underscores the importance of selecting page replacement algorithms carefully and designing them to minimise future page faults rather than simply replacing the oldest page in memory. Mitigating Belady's Anomaly often requires using more sophisticated page replacement policies, such as Least Recently Used (LRU) or Optimal algorithms, which consider the temporal locality of memory references and aim to minimise future page faults.
* Importance of System Tuning: System administrators must be aware of Belady's Anomaly when tuning virtual memory settings and allocating memory resources. They should carefully monitor system performance and adjust memory allocation and page replacement policies to mitigate the impact of Belady's Anomaly and improve overall system efficiency. Additionally, system designers may explore adaptive replacement policies that dynamically adjust the page replacement strategy based on observed access patterns to address the challenges posed by Belady's Anomaly and optimise system performance.

1. **Thrashing**

Thrashing refers to a state in a computer system where excessive paging activity occurs, leading to a significant decrease in overall system performance. In a thrashing scenario, the system spends more time swapping pages between physical memory and secondary storage than executing actual program instructions.



As a result, the CPU is heavily burdened with managing the paging activities, leading to a severe degradation in system responsiveness and throughput.

**Causes of Thrashing**

* **Insufficient Memory:** Thrashing often occurs when the system does not have enough physical memory to accommodate the working sets of active processes. When the demand for memory exceeds available physical memory, the operating system begins swapping pages in and out of memory frequently to make space for active pages, leading to thrashing.
* **Overallocation of Memory:** Allocating more memory to processes than is physically available can also trigger thrashing. In this scenario, the operating system may spend excessive time swapping pages between physical memory and secondary storage, trying to accommodate the memory demands of all active processes.
* **High Degree of Multiprogramming:** Thrashing can occur in systems with a high degree of multiprogramming, where multiple processes are simultaneously active and competing for limited physical memory resources. If the combined memory demands of all active processes exceed the available physical memory, thrashing may ensue.

**Effects of Thrashing on System Performance**

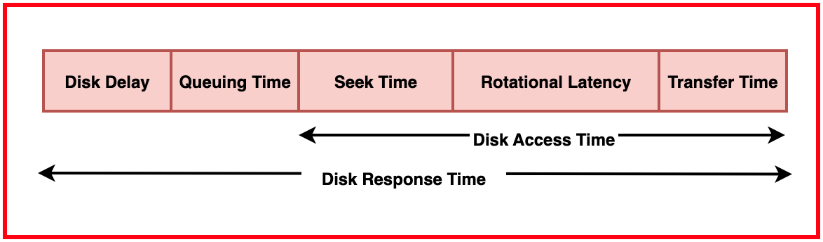
* **Severe Degradation in Performance:** Thrashing results in a significant degradation in overall system performance, as the CPU spends more time managing paging activities than executing useful program instructions. This leads to sluggish responsiveness and increased latency for user interactions and system operations.
* **Increased Disk I/O:** Thrashing increases the frequency of disk I/O operations, as the operating system continuously swaps pages between physical memory and secondary storage. This excessive disk activity can lead to increased wear and tear on storage devices and can degrade system performance further due to the inherent latency associated with disk accesses.
* **Poor Resource Utilisation:** Thrashing leads to poor utilisation of system resources, as a significant portion of CPU and disk bandwidth is consumed by paging activities rather than executing user processes. This can result in underutilization of CPU resources and decreased throughput for compute-intensive tasks.

**Techniques to Prevent Thrashing**

* **Optimising Memory Allocation:** Properly sizing memory allocations for processes can help prevent thrashing. System administrators should monitor memory usage and adjust memory allocations to ensure that the working sets of active processes fit within the available physical memory without excessive paging.
* **Using Effective Page Replacement Policies:** Employing efficient page replacement policies, such as Least Recently Used (LRU) or Optimal algorithms, can help minimise thrashing by ensuring that the most frequently accessed pages remain in physical memory. These algorithms aim to maximise memory utilisation and minimise page faults, reducing the likelihood of thrashing.
* **Limiting Degree of Multiprogramming:** Limiting the number of active processes or the degree of multiprogramming can help prevent thrashing by reducing the overall memory demands on the system. System administrators can adjust system parameters or use workload management techniques to control the number of concurrent processes and prioritise memory allocation for critical tasks.
* **Using Memory Management Techniques:** Techniques such as memory paging or segmentation can help optimise memory usage and prevent thrashing by efficiently managing memory allocation and access patterns. These techniques partition memory into smaller units and allocate memory dynamically based on the needs of active processes, reducing the likelihood of memory contention and thrashing

1. **Disk Scheduling**

Disk scheduling is a crucial aspect of operating system design that deals with efficiently managing the order in which disk I/O requests are serviced by the disk subsystem. In computer systems, data is stored on disk drives, which consist of spinning platters with magnetic surfaces. Disk scheduling algorithms determine the optimal sequence in which these data requests are processed to minimise access latency and maximise throughput.



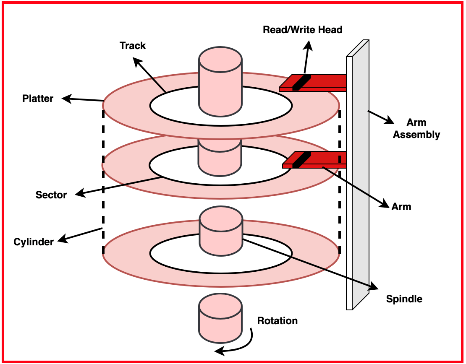


**Role of Disk Scheduling in Operating Systems**

The role of disk scheduling in operating systems is to optimise the utilisation of disk resources and minimise the time required to access data stored on disk drives. Disk I/O operations are typically one of the slowest operations in computer systems due to the mechanical nature of disk drives.

Disk scheduling algorithms play a critical role in coordinating the access of multiple processes or threads to shared disk resources. By prioritising and organising disk I/O requests, disk scheduling algorithms ensure fair access to disk resources while maximising throughput and minimising access latency. This is particularly important in multi-user or multi-tasking environments where multiple processes may concurrently access disk resources.

**Factors Influencing Disk Scheduling**



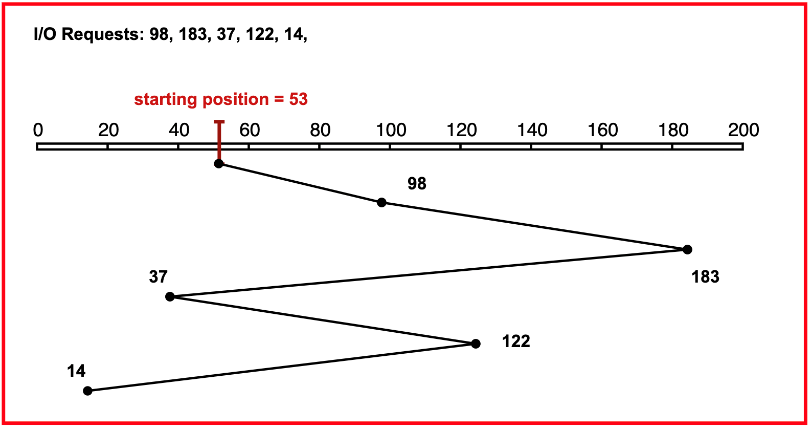
Several factors influence the design and selection of disk scheduling algorithms:

* **Seek Time:** Seek time is the time required for the disk arm to move to the desired track on the disk platter. Disk scheduling algorithms aim to minimise seek time by ordering disk I/O requests to reduce the distance the disk arm needs to travel.
* **Rotational Latency:** Rotational latency refers to the time it takes for the desired disk sector to rotate under the disk head once the disk arm is positioned over the correct track. Disk scheduling algorithms may attempt to reduce rotational latency by optimising the order of disk I/O requests to minimise waiting time for the desired sector to rotate into position.
* **Transfer Time:** Transfer time is the time required to read or write data once the disk head is positioned over the desired track and sector. While transfer time is primarily influenced by the disk's rotational speed and data transfer rate, disk scheduling algorithms may optimise the order of disk I/O requests to maximise data transfer rates and minimise idle time.
* **Concurrency and Fairness:** Disk scheduling algorithms must consider the concurrency and fairness requirements of multiple processes or threads accessing disk resources concurrently. Fairness ensures that all processes have equitable access to disk resources, while concurrency management aims to maximise disk throughput without sacrificing fairness

1. **Disk Scheduling Algorithms**

**First-Come-First-Served (FCFS) Algorithm**

In the FCFS algorithm, disk I/O requests are serviced in the order they arrive. The disk arm moves to the requested track and services the request without considering the distance to the next request. FCFS is simple to implement but can lead to high average seek times, especially if there are large variations in seek times between requests.



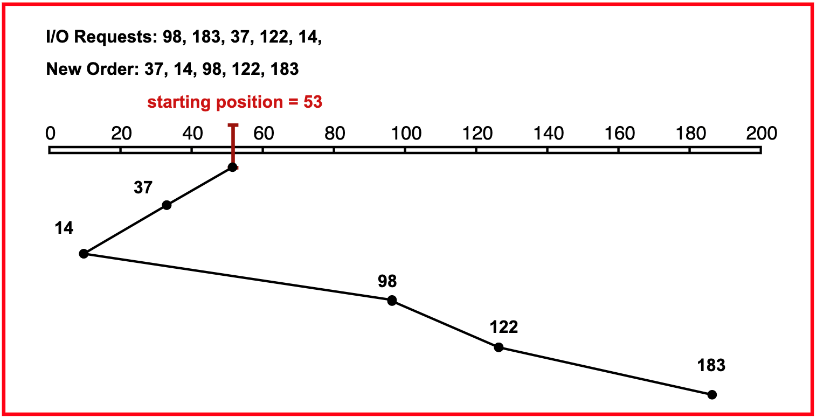
**Example:** Consider a disk with requests to access tracks 98, 183, 37, 122, 14, and the current position of the disk arm is at track 53.

**Advantages:** FCFS is easy to implement and ensures fairness, as requests are serviced in the order they are received.

**Disadvantages:** FCFS can lead to poor performance due to high seek times, especially if there are large variations in seek times between requests.

**Shortest Seek Time First (SSTF) Algorithm**

SSTF selects the request with the shortest seek time from the current position of the disk arm. This algorithm aims to minimise the total seek time by always servicing the nearest request next. SSTF can result in a significant reduction in average seek time compared to FCFS but may suffer from starvation, where requests at the outer tracks are frequently skipped.



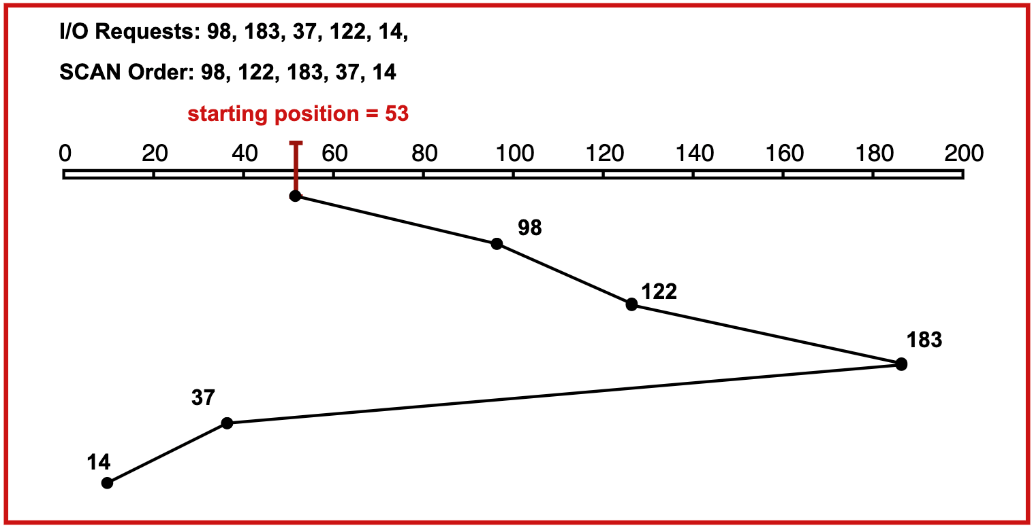
**Example:** Using the same example as before, if the disk arm is at track 53 and the requests are to access tracks 98, 183, 37, 122, and 14, SSTF would service the request to track 37 first, as it has the shortest seek time from track 53.

**Advantages:** SSTF can significantly reduce average seek time compared to FCFS, leading to improved disk performance.

**Disadvantages:** SSTF may suffer from starvation, where requests at the outer tracks are frequently skipped in favour of servicing requests closer to the current position of the disk arm.

**SCAN Algorithm**

The SCAN algorithm (also known as the elevator algorithm) moves the disk arm in one direction across the disk, servicing requests along the way. When the arm reaches the end of the disk, it reverses direction and scans back to the other end. SCAN aims to minimise the average seek time by servicing requests in a back-and-forth manner.



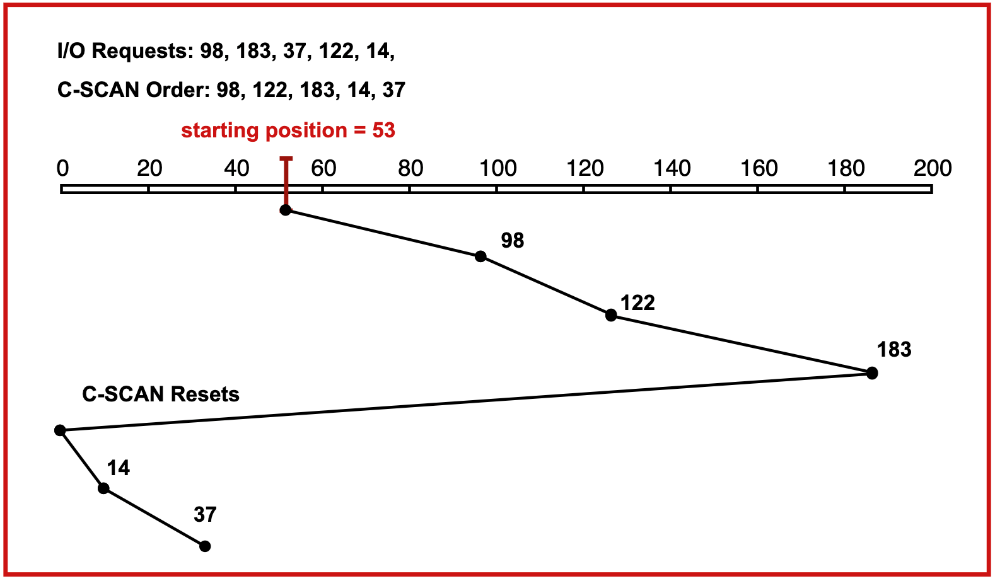
**Example:** Using the same example, if the disk arm is at track 53 and the requests are to access tracks 98, 183, 37, 122, and 14, SCAN would first service requests in the direction of movement (e.g., tracks 98, 122), then reverse direction and service requests in the other direction (e.g., tracks 37, 14).

**Advantages:** SCAN can provide a good balance between fairness and performance by servicing requests in a back-and-forth manner.

**Disadvantages:** SCAN may not be optimal for systems with high variance in request distribution, as requests at the ends of the disk may experience longer wait times.

**C-SCAN Algorithm**

The C-SCAN algorithm is a variant of SCAN that only scans in one direction, servicing requests until it reaches the end of the disk, at which point it jumps back to the beginning of the disk and continues scanning. C-SCAN aims to reduce the variance in service times compared to SCAN by ensuring that all requests are serviced in the same direction.



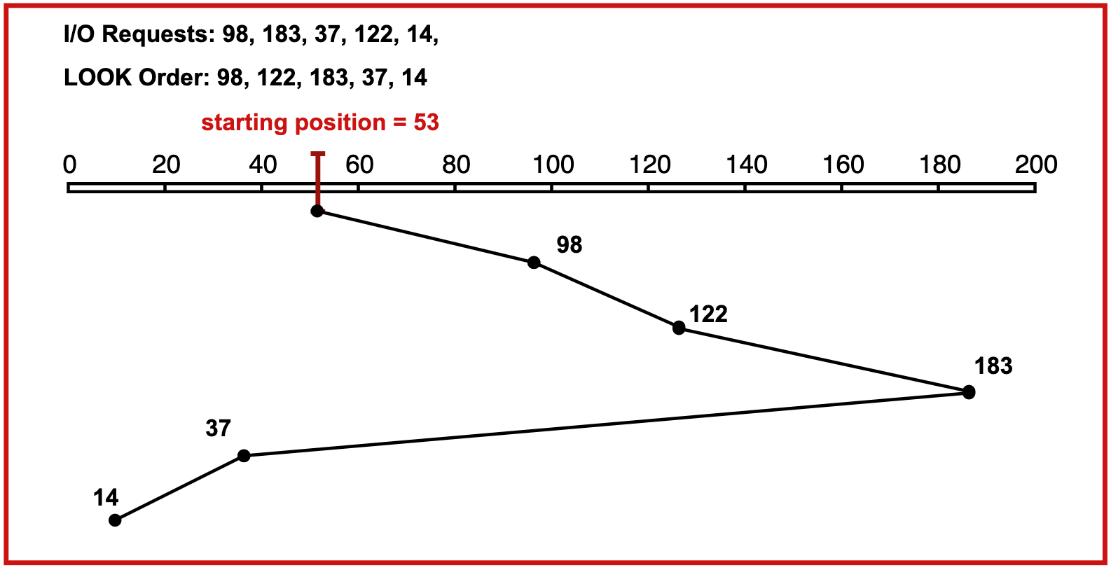
**Example:** Using the same example, if the disk arm is at track 53 and the requests are to access tracks 98, 183, 37, 122, and 14, C-SCAN would first service requests in the direction of movement (e.g., tracks 98, 122), then jump back to the beginning of the disk and continue servicing requests in the same direction (e.g., tracks 14, 37).

**Advantages:** C-SCAN can reduce the variance in service times compared to SCAN by ensuring that all requests are serviced in the same direction.

**Disadvantages:** C-SCAN may lead to increased average seek times compared to SCAN, especially if there are requests at both ends of the disk.

**LOOK Algorithm**

The LOOK algorithm is a variant of SCAN that services requests only until it reaches the last request in the current direction, at which point it reverses direction without reaching the end of the disk. LOOK aims to reduce the average seek time compared to SCAN by avoiding unnecessary scanning to the ends of the disk.



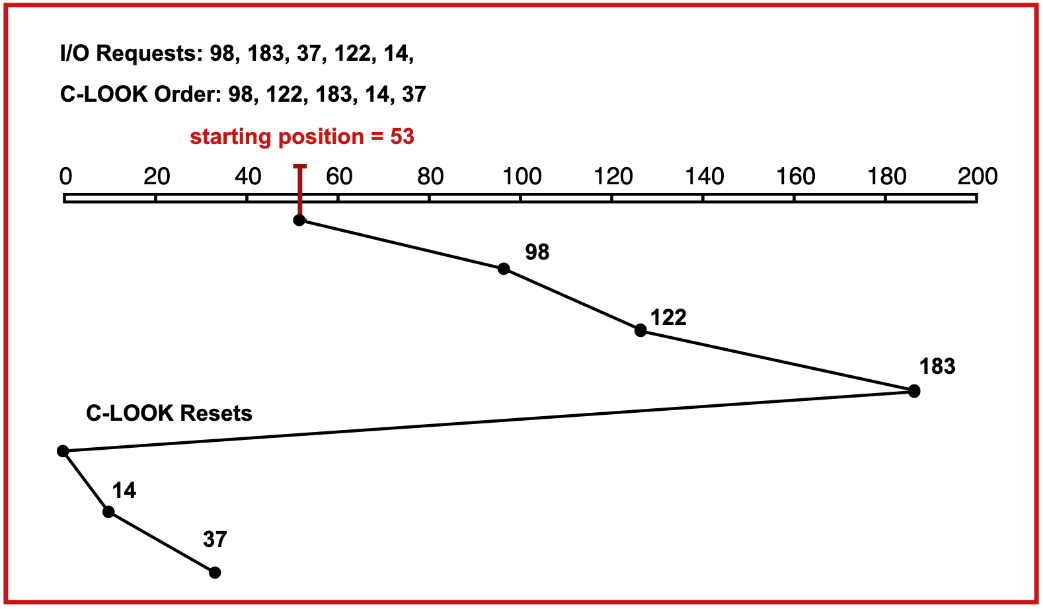
**Example:** Using the same example, if the disk arm is at track 53 and the requests are to access tracks 98, 183, 37, 122, and 14, LOOK would first service requests in the direction of movement (e.g., tracks 98, 122), then reverse direction and service requests in the other direction (e.g., tracks 37, 14), stopping before reaching the ends of the disk.

**Advantages:** LOOK can reduce the average seek time compared to SCAN by avoiding unnecessary scanning to the ends of the disk.

**Disadvantages:** LOOK may not be optimal for systems with high variance in request distribution, as requests at the ends of the disk may experience longer wait times.

**C-LOOK Algorithm**

The C-LOOK algorithm is a variant of LOOK that only scans in one direction, servicing requests until it reaches the last request in that direction before jumping back to the beginning of the disk. C-LOOK aims to reduce the variance in service times compared to LOOK by ensuring that all requests are serviced in the same direction.



**Example:** Using the same example, if the disk arm is at track 53 and the requests are to access tracks 98, 183, 37, 122, and 14, C-LOOK would first service requests in the direction of movement (e.g., tracks 98, 122), then jump back to the beginning of the disk and continue servicing requests in the same direction (e.g., tracks 14, 37).

**Advantages:** C-LOOK can reduce the variance in service times compared to LOOK by ensuring that all requests are serviced in the same direction.

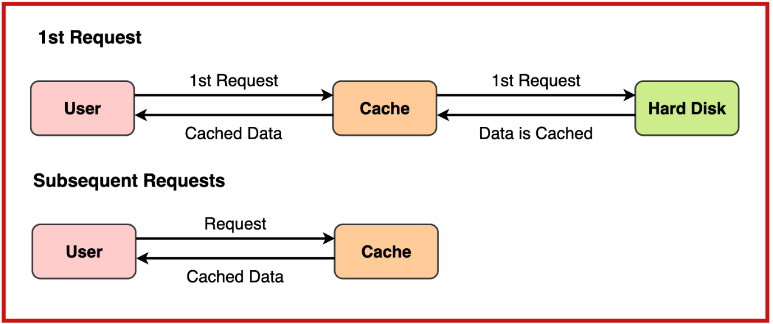
**Disadvantages:** C-LOOK may lead to increased average seek times compared to LOOK, especially if there are requests at both ends of the disk

**Various Disk Scheduling Policies**

| **Algorithm** | **Description** | **Advantages** | **Disadvantages** |
| --- | --- | --- | --- |
| First-Come-First-Served | Services requests in the order they arrive. | Simple implementation ensures fairness. | May result in high average seek times. |
| Shortest Seek Time First | Selects the request with the shortest seek time from the current position of the disk arm. | Minimises average seek time. | May suffer from starvation. |
| SCAN | Moves the disk arm in one direction across the disk, servicing requests along the way, then reverses direction and scans back. | Provides a good balance between fairness and performance. | May not be optimal for systems with high variance in request distribution. |
| C-SCAN | Scans in one direction until it reaches the end of the disk, then jumps back to the beginning. | Reduces variance in service times. | May lead to increased average seek times. |
| LOOK | Services requests only until it reaches the last request in the current direction, then reverses direction. | Reduces average seek time compared to SCAN. | Requests at the ends of the disk may experience longer wait times. |
| C-LOOK | Scans in one direction until it reaches the last request, then jumps back to the beginning. | Reduces variance in service times. | May lead to increased average seek times. |

1. **Cache**

A cache is a high-speed memory component used to store frequently accessed data and instructions to improve system performance by reducing the latency of memory accesses. Caches are placed between the CPU and main memory (RAM) and exploit the principle of locality, which states that programs tend to access a relatively small portion of memory frequently or exhibit spatial and temporal locality.



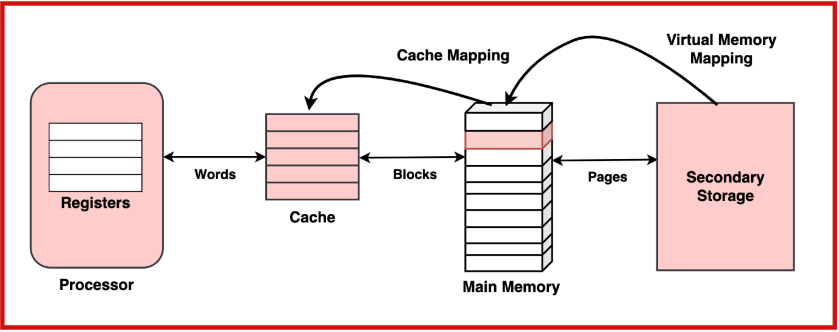


By storing copies of frequently accessed data in a smaller, faster cache memory, the CPU can retrieve data more quickly, thereby reducing the average memory access time.

**Cache Organisation: Direct-Mapped, Set-Associative, Fully-Associative**

* 1. **Direct-Mapped Cache**

In a direct-mapped cache, each main memory address is mapped to a specific location in the cache using a hash function or modulo arithmetic. This mapping ensures that each block of main memory can only reside in one specific cache location. Direct-mapped caches are simple and require minimal hardware, but they may suffer from conflicts where multiple main memory blocks map to the same cache location, leading to cache thrashing.



* 1. **Set-Associative Cache**

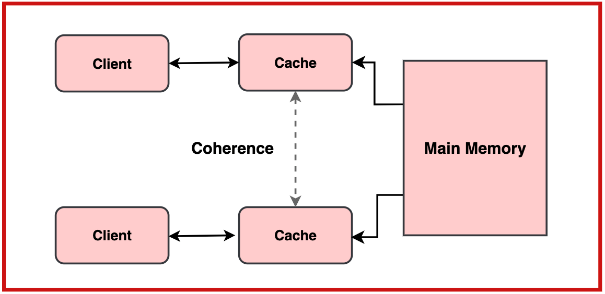
In a set-associative cache, each main memory address is mapped to a set of cache locations, and the data can be stored in any location within the set. This organisation reduces the likelihood of conflicts compared to direct-mapped caches while maintaining simplicity and low hardware overhead. Set-associative caches strike a balance between simplicity and flexibility, making them widely used in modern CPU designs.

* 1. **Fully-Associative Cache**

In a fully-associative cache, each main memory address can be stored in any cache location, without any restrictions on mapping. This organisation offers the highest degree of flexibility and minimises the potential for conflicts, but it requires complex hardware for tag comparison and cache management, leading to higher power consumption and cost.

**Cache Coherence**

Cache coherence refers to the consistency of data stored in multiple caches that reference the same memory location. In multiprocessor systems, each processor typically has its own cache memory to improve performance. However, when multiple processors access and modify the same memory location concurrently, ensuring cache coherence becomes crucial to prevent data inconsistencies.



Cache coherence protocols, such as MESI (Modified, Exclusive, Shared, Invalid) or MOESI (Modified, Owned, Exclusive, Shared, Invalid), are used to maintain coherence by coordinating cache operations, invalidating or updating cache lines as needed, and ensuring that all processors observe a consistent view of memory.

**Cache Replacement Policies**

Cache replacement policies determine which cache line to evict when the cache is full and a new line needs to be loaded. Common cache replacement policies include:

* 1. **Least Recently Used (LRU)**

LRU replaces the cache line that has not been accessed for the longest time. This policy aims to minimise the likelihood of evicting recently accessed data, assuming that recently accessed data is more likely to be accessed again soon.

* 1. **First-In, First-Out (FIFO)**

FIFO replaces the cache line that was loaded into the cache earliest. This policy is simple to implement but may not always reflect access patterns accurately, especially in scenarios with non-uniform memory access patterns.

* 1. **Random Replacement**

Random replacement selects a cache line to evict randomly from the set of available cache lines. While simple to implement and not prone to algorithmic complexity, random replacement may result in suboptimal performance compared to more sophisticated policies like LRU.

* 1. **Least Frequently Used (LFU)**

LFU replaces the cache line that has been accessed the fewest times. This policy aims to evict cache lines that are least frequently accessed, assuming that they are less likely to be accessed in the future. However, accurately tracking access frequency can be challenging and may require additional hardware support

1. **Direct and Associative mapping**

**Concept of Direct Mapping**

Direct mapping is a cache mapping technique where each block of main memory is mapped to exactly one location in the cache using a simple modulo operation. The address of a memory block is divided into three fields: the tag, index, and block offset. The index field identifies the specific cache line, while the tag field helps to verify if the data in the indexed cache line corresponds to the requested memory block.

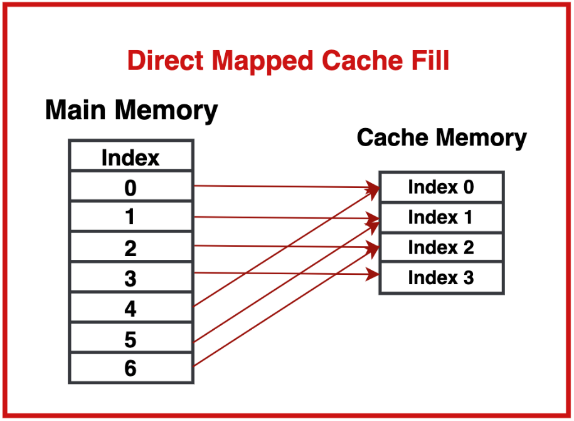
This mapping method is straightforward to implement due to its simplicity, making it suitable for hardware implementations that prioritize speed. However, direct mapping may lead to frequent cache misses in cases where multiple memory blocks map to the same cache line, a situation known as cache thrashing.

**Implementation of Direct Mapping**

In direct mapping, each memory block is placed in a cache line determined by the formula:

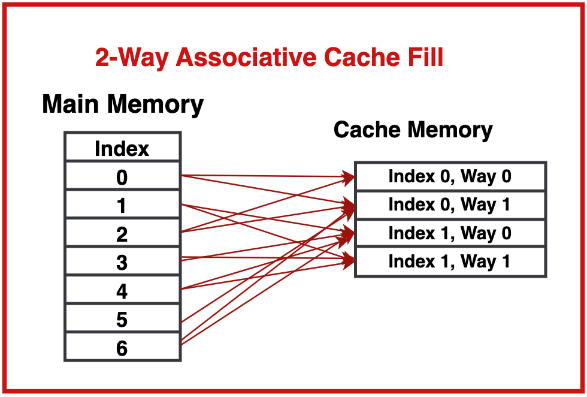
Cache Line Number = (Memory Block Address) % (Number of Cache Lines)

For example, if there are 16 cache lines, and the memory block address is 18, the block will be stored in the cache line number 18 % 16 = 2. The tag field is then used to ensure that the data in cache line 2 corresponds to the requested memory block during access.



**Concept of Associative Mapping**

Associative mapping is a more flexible cache mapping technique where a memory block can be stored in any cache line rather than being restricted to a specific one. This is achieved by using a tag field that identifies which memory block is currently stored in each cache line. During access, the cache searches all lines in parallel for the tag that matches the requested memory address.While this technique offers more flexibility and reduces the risk of cache thrashing, it requires more complex hardware to compare tags across all cache lines simultaneously. The increased hardware complexity can lead to higher costs and power consumption, making associative mapping more suitable for smaller caches.



**Implementation of Associative Mapping**

In fully associative mapping, any memory block can be placed in any cache line. During data retrieval, the cache checks each line for a matching tag, which indicates that the desired data is present. The implementation relies on a process called tag comparison, where all cache lines are searched in parallel for a match.

For example, if a memory block with address 25 is requested, the cache will search through all lines to see if any contain a tag that matches the address. If a match is found, the corresponding data is retrieved; otherwise, the memory block is loaded into an available cache line, possibly replacing an existing one based on the cache replacement policy.

**Comparison Between Direct and Associative Mapping**

| **Aspect** | **Direct Mapping** | **Associative Mapping** |
| --- | --- | --- |
| **Simplicity** | Simpler to implement with a straightforward mapping function. Requires less hardware. | More complex due to the need for tag comparison across all cache lines. |
| **Flexibility** | Each memory block is mapped to a specific cache line, increasing chances of conflicts. | Offers greater flexibility since any memory block can be stored in any cache line. |
| **Performance** | Prone to cache thrashing, which can lead to lower hit rates in some scenarios. | Typically higher hit rates due to reduced conflict misses, leading to better performance. |
| **Cost and Complexity** | Lower cost and power consumption due to simpler hardware requirements. | Higher cost and power consumption due to complex comparison logic. |

**Hybrid Mapping Techniques**

Hybrid mapping techniques combine elements of both direct and associative mapping to balance performance and hardware complexity. The most common hybrid technique is the \*set-associative cache\*, where the cache is divided into multiple sets, and each memory block maps to a specific set, but can be placed in any line within that set.

Set-associative mapping is typically implemented as *n-way set-associative*, where *n* refers to the number of lines in each set. For example, in a 4-way set-associative cache, each memory block maps to a specific set and can occupy any of the 4 cache lines within that set.

This approach offers a middle ground between direct and fully associative caches, reducing the likelihood of conflicts compared to direct mapping while avoiding the high hardware complexity of fully associative caches. Set-associative caches are widely used in modern CPU designs due to their balanced approach.