Virtual Memory, Demand Paging & Page Replacement

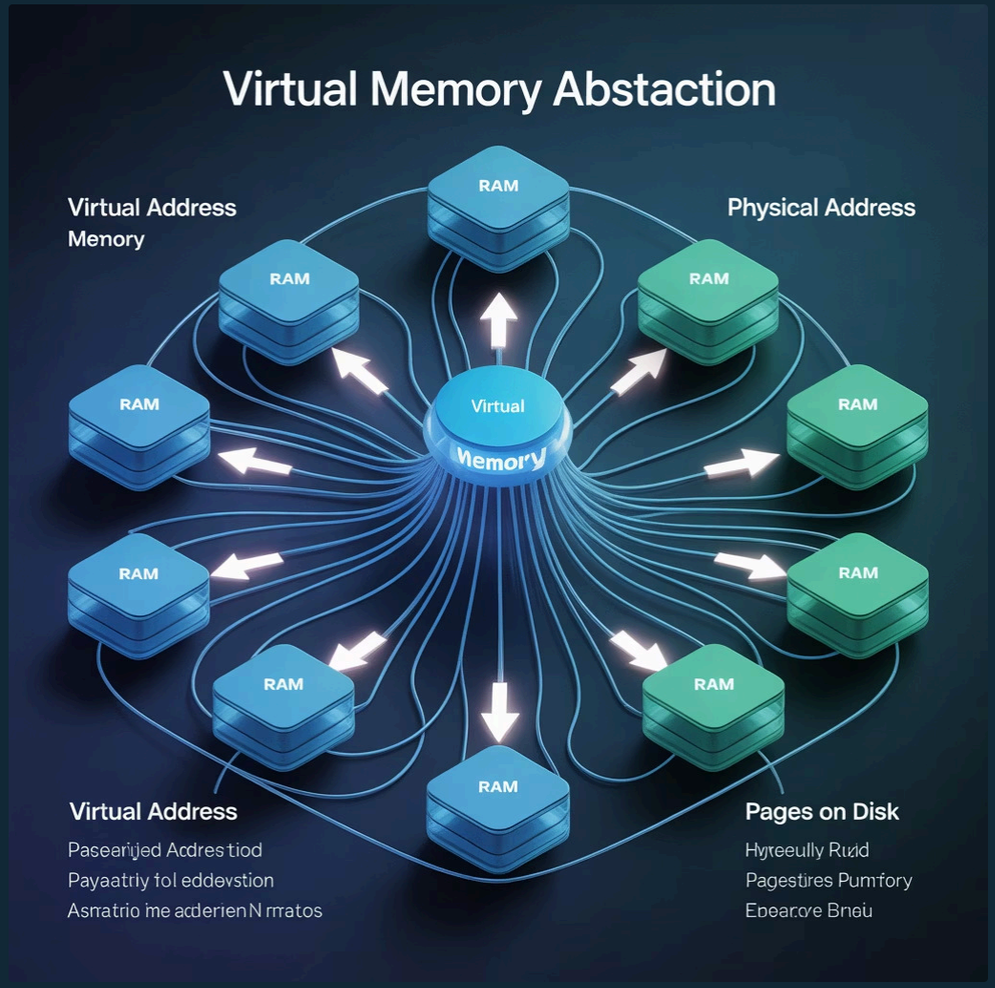
A comprehensive exploration of core memory management techniques that power modern operating systems. These fundamental concepts enable efficient resource utilisation, process isolation, and system stability across all contemporary computing platforms.

What is Virtual Memory?

Virtual memory is a sophisticated memory management technique that provides an abstraction layer over physical memory (RAM). It creates the illusion that each process has access to a vast, contiguous address space, regardless of actual physical memory constraints.

This elegant solution effectively decouples logical memory addresses (what programs see) from physical memory addresses (actual RAM locations), enabling a multitude of benefits for modern computing systems.

The virtual memory system transforms the computer's RAM into a cache for the address space stored on secondary storage, typically hard disk drives or solid-state drives. This transformation is managed by the operating system kernel with hardware support.



Each process operates within its own virtual address space, unaware of the actual physical memory organisation. The operating system dynamically maps these virtual addresses to physical memory locations as needed, creating a seamless experience for applications.

Benefits of Virtual Memory

Program Size Flexibility

Applications can utilise more memory than physically installed in the system. This enables the execution of large, complex programs on systems with limited RAM by keeping only actively used portions in physical memory.

Process Isolation

Each process operates in its own protected address space, preventing processes from interfering with each other's memory. This isolation is fundamental to system stability and security, as it prevents faulty applications from corrupting other programs or the operating system itself.

Efficient Multitasking

The system can run more concurrent programs than would fit simultaneously in physical memory. This allows for rapid context switching between applications without completely unloading and reloading memory contents.

Memory Sharing

Common libraries and resources can be shared among multiple processes, reducing redundancy. For instance, system DLLs or shared libraries can be loaded once in physical memory but mapped into multiple virtual address spaces.

These benefits combine to create computing environments that are more efficient, secure, and capable of running complex software ecosystems that would otherwise be impossible with direct physical memory management approaches.

Logical vs Physical Address Spaces

The distinction between logical and physical address spaces lies at the heart of virtual memory systems:

Logical (Virtual) Address Space

The continuous range of memory addresses that a process can reference. Each process believes it has exclusive access to the entire memory space, typically spanning from address 0 to a maximum value (e.g., 2^64-1 on modern 64-bit systems).

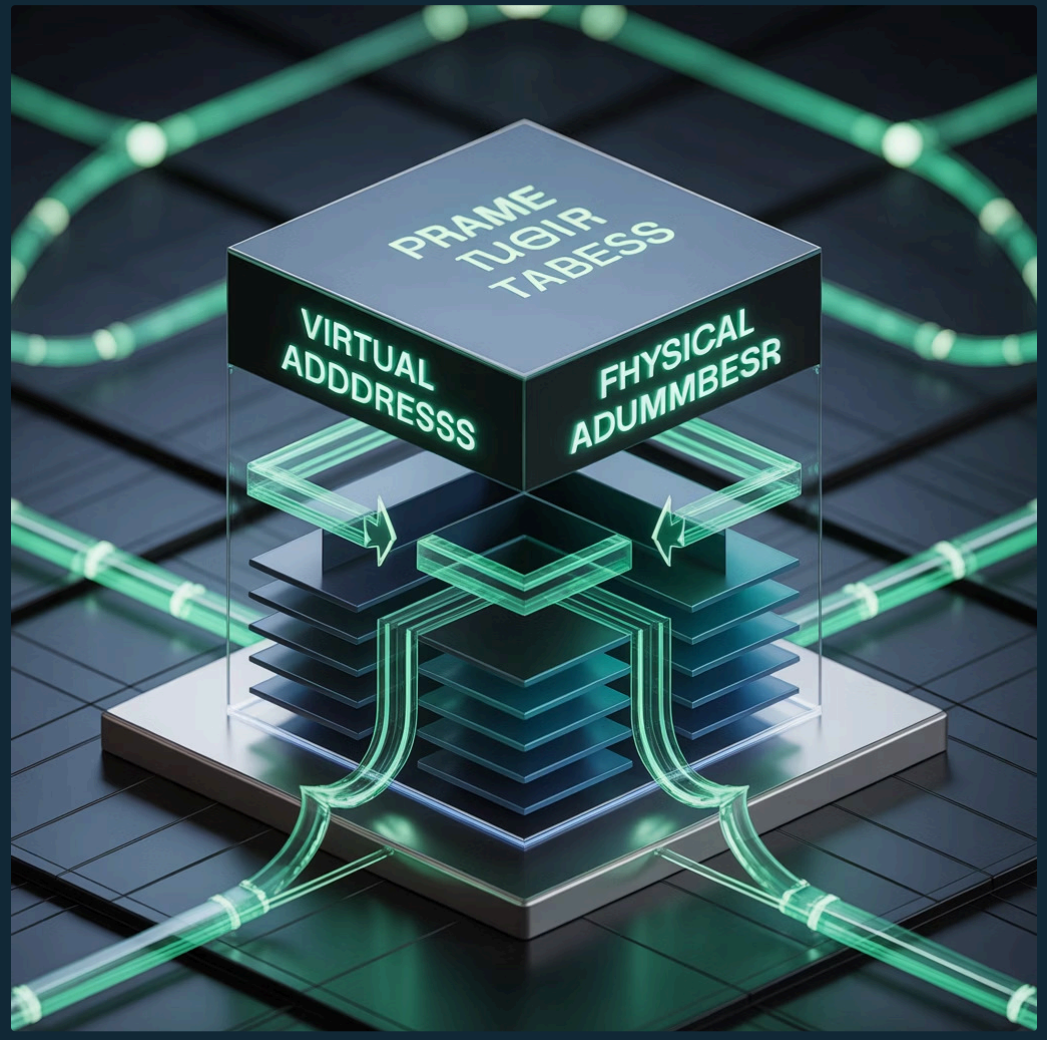
Physical Address Space

The actual physical RAM addresses in the hardware. This space is shared among all processes and the operating system. Its size is limited by the installed physical memory (e.g., 16GB).

Address Translation

The Memory Management Unit (MMU) is a critical hardware component that performs real-time translation between virtual and physical addresses:

- When a process attempts to access memory, it generates a virtual address
- The MMU intercepts this request and consults page tables maintained by the OS
- If the address is valid and mapped, the MMU converts it to the corresponding physical address
- If unmapped, it triggers a page fault exception for the OS to handle



The Need for Efficient Memory Use

Principle of Locality

Programs exhibit locality of reference—they tend to access a relatively small portion of their address space at any given time. For example, a video editing application might only need to access the frames currently being processed, not the entire video file.

Resource Optimisation

By keeping only actively needed memory regions in physical RAM, the system can support many more concurrent processes than would otherwise be possible. This dramatically improves system utilisation and throughput in multi-user or multi-tasking environments.

Performance Enhancements

Efficient memory management reduces unnecessary I/O operations. Rather than loading entire programs at launch, the OS can load components on demand, reducing startup times and overall system overhead.

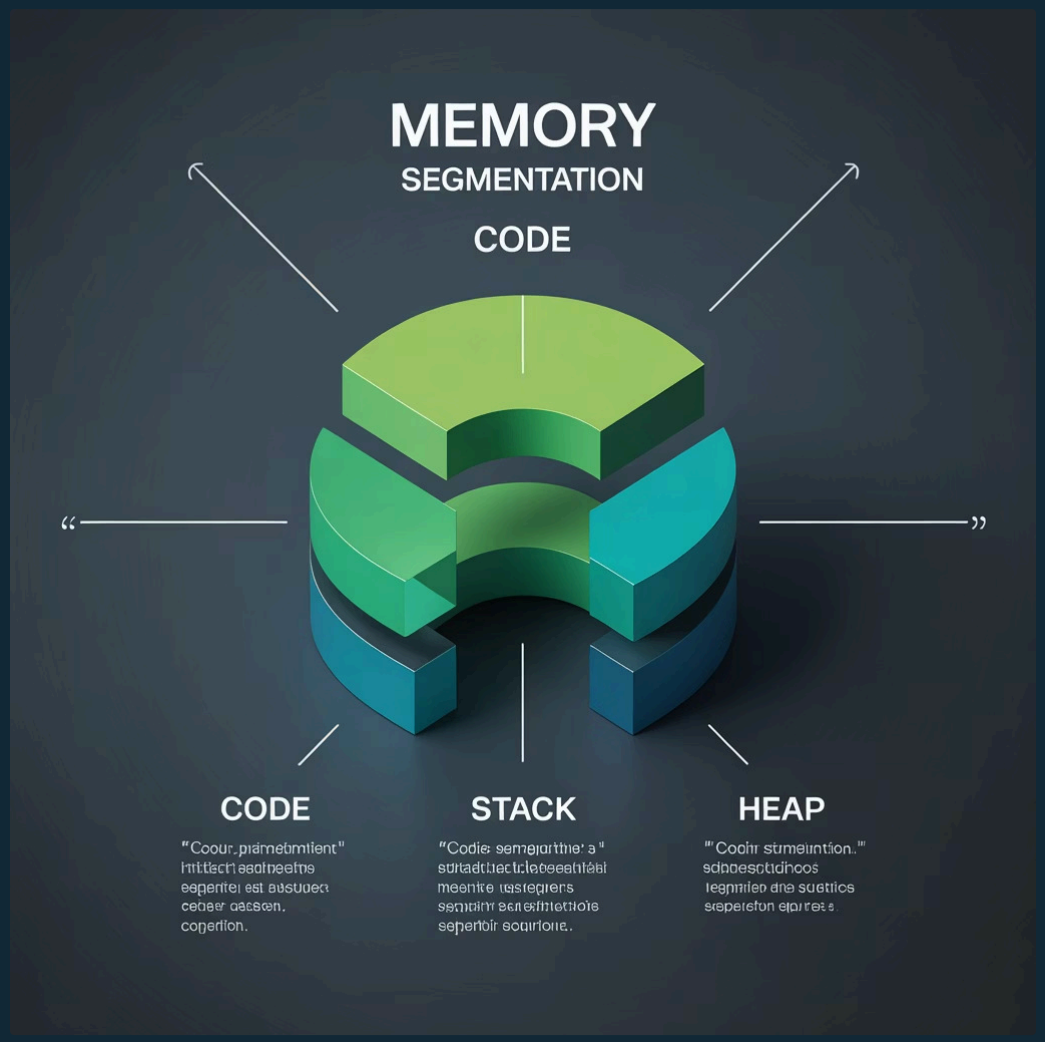
Modern computing environments run dozens or even hundreds of processes simultaneously. Without efficient memory management techniques, even systems with substantial RAM would quickly exhaust available resources.

Segmentation and Paging

Segmentation

Segmentation divides memory into logical segments of varying sizes based on the program's structure:

- Each segment represents a logical unit (code, stack, heap, etc.)
- Segments can grow or shrink independently
- Addressing uses segment:offset format
- Suffers from external fragmentation issues



Paging

Paging divides memory into fixed-size blocks called pages (typically 4KB):

- Virtual address space divided into pages
- Physical memory divided into frames of identical size
- Pages map to frames via page tables
- Eliminates external fragmentation
- Enables fine-grained memory protection

Most modern systems use paging rather than pure segmentation, though some architectures implement a hybrid approach. The fixed-size nature of pages simplifies memory allocation and reclamation operations significantly.

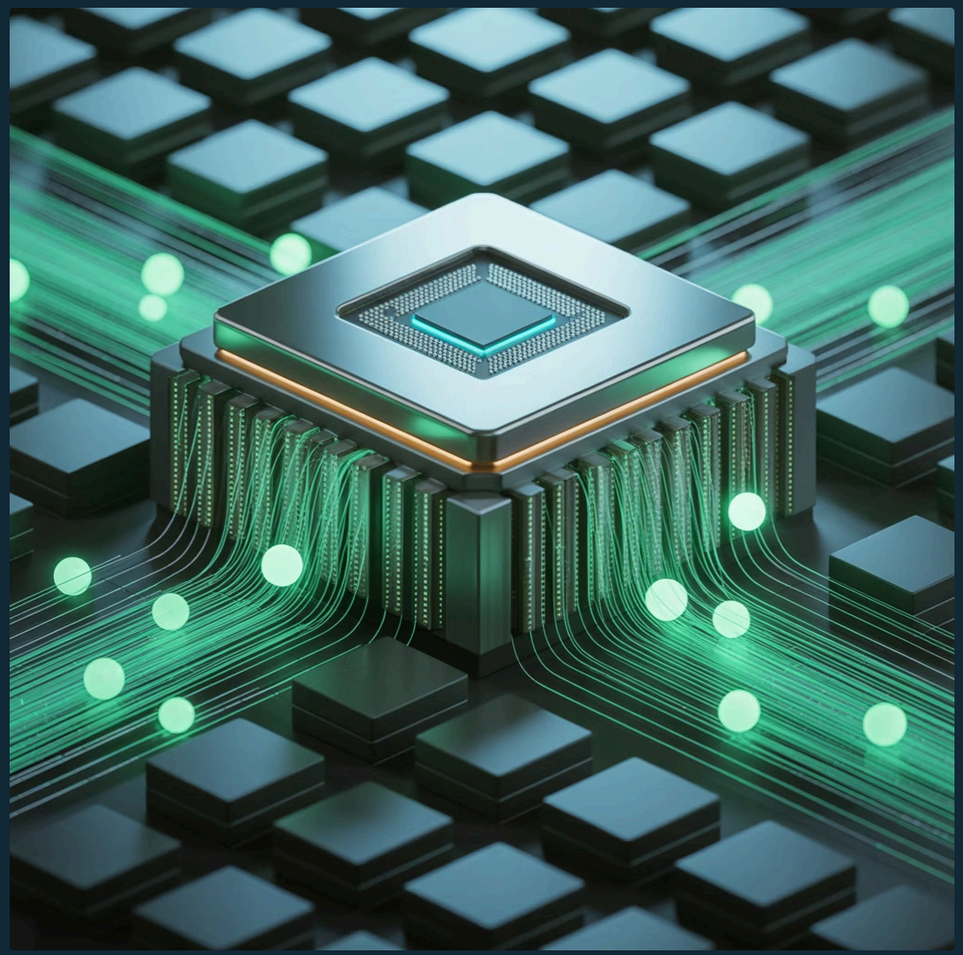
What is Demand Paging?

Demand paging is a sophisticated memory management technique that forms the cornerstone of virtual memory systems in modern operating systems. Rather than loading an entire program into memory before execution, demand paging loads memory pages only when they are accessed or "demanded" by the executing process.

This lazy loading approach offers several significant advantages:

- Faster program startup times as initial loading is minimal
- Lower memory consumption for inactive program regions
- More efficient use of available physical memory
- Support for larger applications than physical memory could contain

All contemporary operating systems—Windows, macOS, Linux, iOS, and Android—implement demand paging as their primary memory management strategy.



> "Demand paging is the most sophisticated and widespread form of virtual memory implementation, allowing systems to function as if they had more memory than physically installed."

The effectiveness of demand paging relies on the principle of locality—the observation that programs typically access a small subset of their total memory space during any given time period.

How Demand Paging Works

Each memory page in a virtual memory system has associated metadata maintained by the operating system:

Page Table Entry Bits

- **Present/Absent bit:** Indicates if the page is currently in physical memory
- **Modified bit (dirty bit):** Indicates if the page has been modified since loading
- **Referenced bit:** Indicates if the page has been accessed recently
- **Protection bits:** Control read/write/execute permissions

Performance Considerations

The system maintains a careful balance between keeping frequently accessed pages in memory whilst relegating rarely used pages to secondary storage. This optimisation is critical for maintaining system responsiveness.

Page size selection is also crucial—larger pages reduce table overhead but may waste memory with unused portions, while smaller pages offer finer granularity but increase table size and translation overhead.

Page Faults Explained

A page fault occurs when a program attempts to access a memory page that is mapped in its virtual address space but not currently present in physical memory (RAM). This critical event triggers a sequence of operating system operations to rectify the situation.

Types of Page Faults:

- **Minor page fault:** The page exists in memory but isn't mapped in the process table or requires permission changes
- **Major page fault:** The page must be loaded from disk or other secondary storage
- **Invalid page fault:** The program attempts to access an unmapped or protected address (causes segmentation faults or access violations)

Performance Impact

Page faults introduce significant performance penalties because accessing disk storage is orders of magnitude slower than accessing RAM:

- RAM access: Nanoseconds (10-9 seconds)
- SSD access: Microseconds (10-6 seconds)
- HDD access: Milliseconds (10-3 seconds)

When a system experiences an excessive rate of page faults, causing continuous paging activity, it enters a condition known as "thrashing"—where the system spends more time managing page faults than executing actual program instructions.

Handling a Page Fault: The Steps

Fault Detection

When the CPU attempts to access a page marked as not present in memory, the MMU generates a trap (interrupt) to the operating system. The current instruction is halted mid-execution.

Process Suspension

The OS saves the current state of the process, including registers and program counter. The process is moved from the "running" state to a "blocked" or "waiting" state until the page fault is resolved.

Page Location

The OS examines the virtual address that caused the fault and locates the corresponding page on disk, typically in the swap file/partition or in the original executable file for code pages.

Frame Allocation

The OS allocates a physical memory frame to hold the page. If memory is full, a page replacement algorithm (discussed later) is used to select a victim page to evict from memory.

Page Loading

The required page is read from disk into the allocated frame. If the victim page was modified (dirty), it must first be written back to disk to preserve its contents.

Table Update & Restart

The page table is updated to reflect the new page location, marking it as present. The process is then restarted from the instruction that caused the fault, which now completes successfully.

This entire sequence happens transparently to the running application, which remains unaware that its memory access triggered such a complex series of operations.

Introduction to Page Replacement Algorithms

When physical memory is full and a new page needs to be loaded, the operating system must decide which existing page to evict. This critical decision is made by page replacement algorithms, which significantly impact system performance.

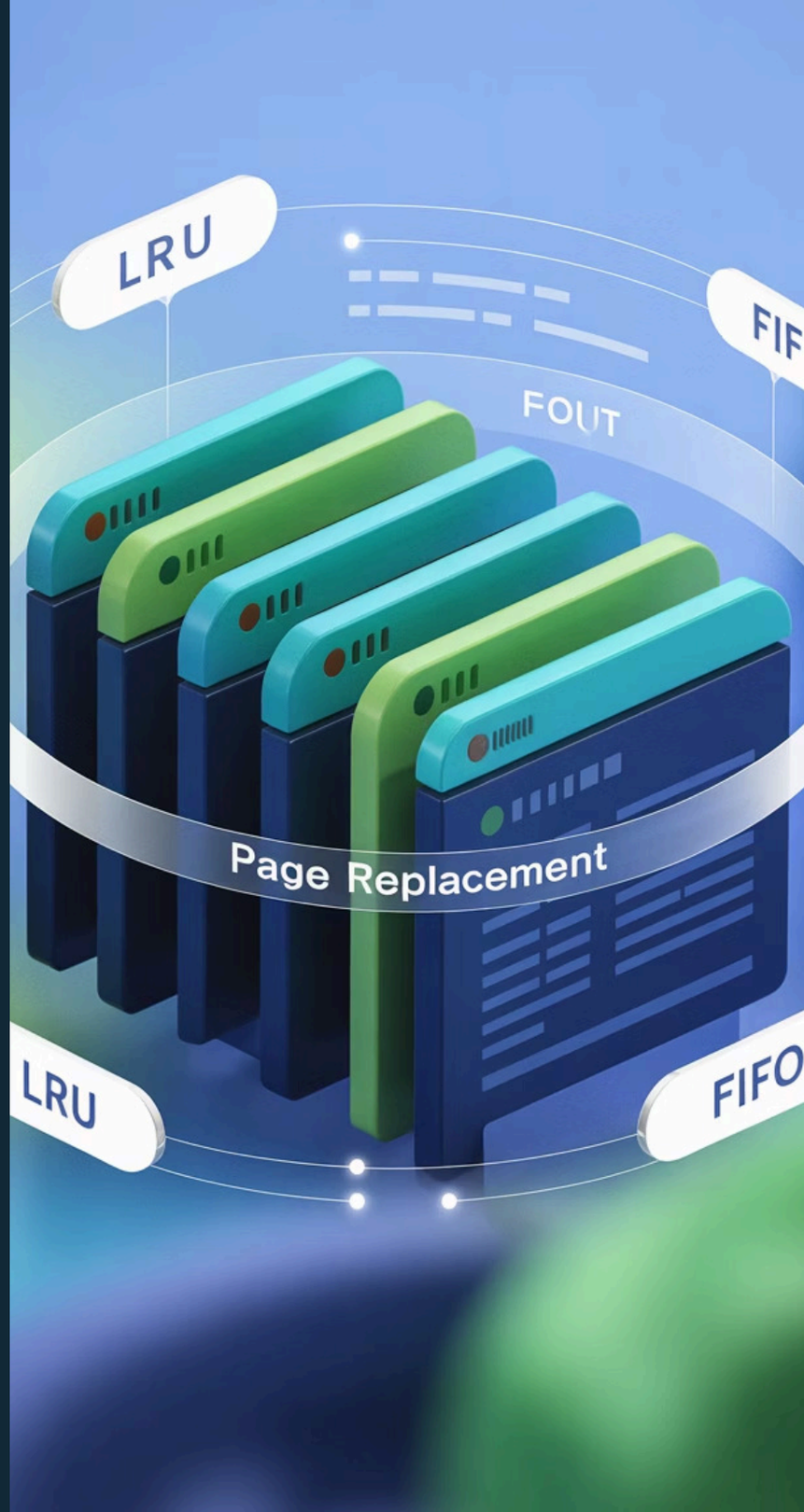
Key Objectives

- Minimise the number of page faults
- Reduce disk I/O operations
- Maintain acceptable application performance
- Balance memory utilisation across processes

Evaluation Metrics

- Page fault rate (lower is better)
- Memory throughput
- System responsiveness
- Algorithm implementation complexity
- Overhead of tracking page usage

The ideal replacement algorithm would predict future memory access patterns perfectly, but since this is impossible in practice, various heuristic approaches have been developed based on observed program behaviour patterns.



FIFO (First-In First-Out) Algorithm

The FIFO (First-In First-Out) page replacement algorithm is one of the simplest approaches, maintaining a queue of pages in memory based solely on their arrival time. When a replacement is needed, the oldest page (first in the queue) is selected for eviction.

Implementation:

- Maintain a queue of page references
- New pages are added to the rear of the queue
- Page replacement removes from the front of the queue
- No consideration of page usage frequency or recency

Advantages:

- Simple implementation with minimal overhead
- No need for usage tracking hardware
- Predictable behaviour

Limitations:

- Does not consider page usage patterns
- Frequently used pages may be replaced
- Susceptible to Bélády's anomaly (increasing frames can increase faults)

Example Scenario:

With reference string: 1, 3, 0, 3, 5, 6, 3 and 3 frames:

Page faults occur for pages 1, 3, 0, 5, 6 (total: 5 faults)

When page 3 is referenced again, it's already in memory for the second reference but causes a fault when referenced the third time (after being evicted as the oldest page).

LRU (Least Recently Used) Algorithm

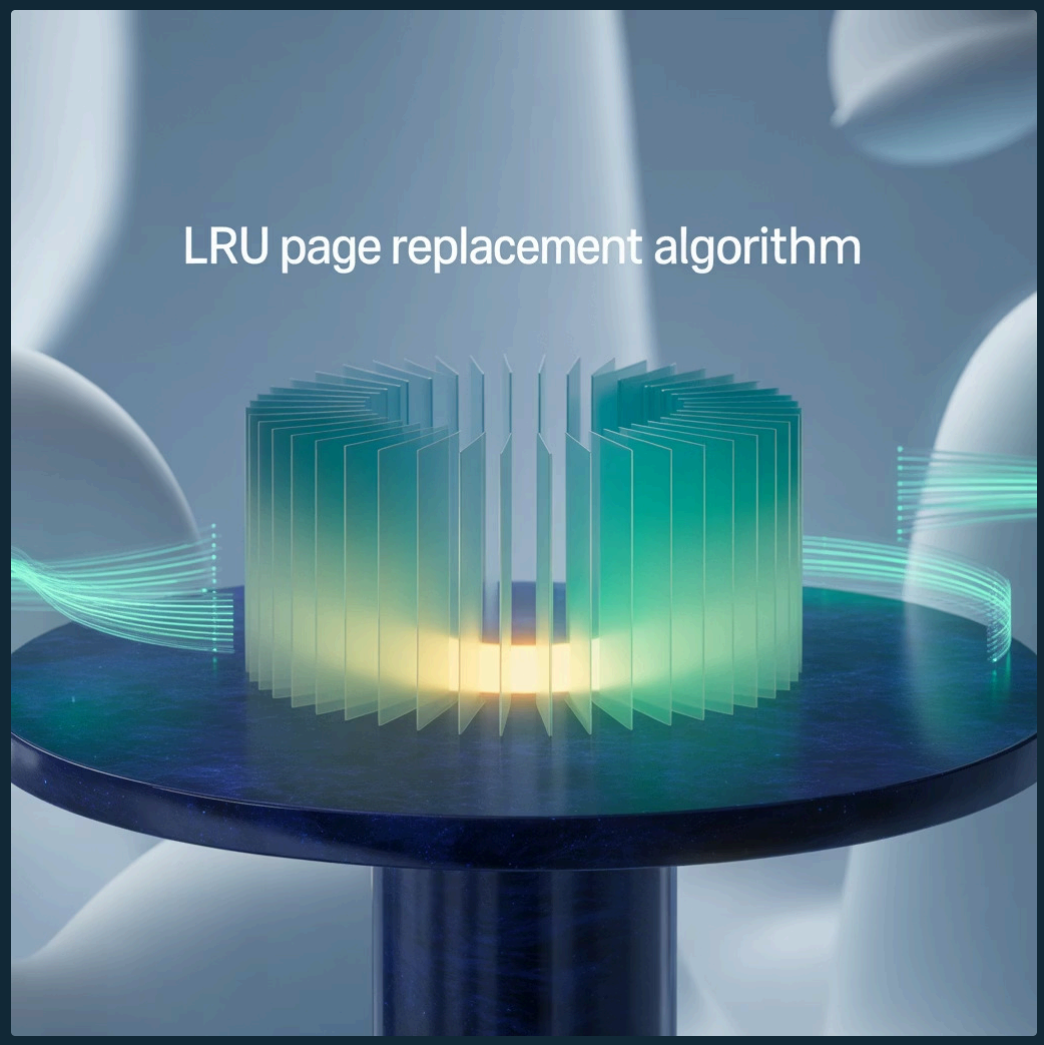
The LRU algorithm is based on the principle of temporal locality—the observation that recently accessed pages are likely to be accessed again in the near future. It replaces the page that has not been used for the longest period of time.

Implementation Approaches:

- **Counter-based:** Each page entry is tagged with the time of last access
- **Stack-based:** Pages are maintained in a stack, moving to the top when accessed
- **Reference bits:** Hardware sets bits when pages are accessed, software periodically examines patterns

Advantages:

- Exploits temporal locality for better performance
- Adapts to changing program behaviour
- Not susceptible to Bélády's anomaly
- Generally outperforms FIFO in real-world scenarios



Limitations:

- Requires hardware support for efficient implementation
- Higher overhead than FIFO
- Pure LRU can be expensive to implement

Approximation Methods:

Due to implementation challenges, most systems use approximations:

- **Clock algorithm:** Circular buffer with reference bits
- **NFU (Not Frequently Used):** Counters track access frequency
- **Aging:** Bit shifting to track recent usage patterns

Modern operating systems often implement sophisticated variants of LRU that consider both recency and frequency of access to achieve optimal performance under various workloads.

Optimal and Other Algorithms

1

Optimal (OPT/MIN) Algorithm

The theoretical best algorithm that replaces the page that will not be used for the longest time in the future. Impossible to implement in practice as it requires knowledge of future memory references, but serves as a benchmark for evaluating other algorithms.

- Used primarily for theoretical comparison
- Provides the lowest possible page fault rate
- Requires perfect knowledge of future references

2

MRU (Most Recently Used)

The opposite of LRU, replacing the most recently used page. While counterintuitive, this can be effective for certain access patterns like sequential scans where the most recently used page may not be needed again soon.

- Useful for specific workloads (e.g., database operations)
- Performs well with cyclic/looping access patterns

3

Second Chance (Clock)

A compromise between FIFO simplicity and LRU effectiveness. Uses a circular queue like FIFO but gives pages a "second chance" if they've been referenced recently.

- Uses a reference bit for each page
- Lower overhead than pure LRU
- Widely implemented in real systems

4

Working Set Model

Based on the concept that a process has a "working set" of pages it actively uses. The algorithm attempts to keep the entire working set in memory to minimise page faults.

- Adaptively adjusts memory allocation
- Effective at preventing thrashing
- Requires tracking page references over time windows

Conclusion: Why These Concepts Matter

System Performance

Efficient virtual memory management directly impacts overall system performance. Well-implemented page replacement algorithms can significantly reduce latency and improve throughput in memory-constrained environments.

Understanding these concepts enables developers and system administrators to optimise applications and system configurations for maximum performance.

Modern Computing

These concepts underpin all contemporary computing systems—from mobile phones to supercomputers. As applications grow more complex and memory-intensive, effective virtual memory management becomes increasingly critical.

Cloud computing environments rely heavily on these techniques to maximise resource utilisation across virtualized infrastructure.

Troubleshooting

Knowledge of virtual memory operations is essential for diagnosing performance issues. When systems exhibit slowdowns, understanding page fault patterns and memory pressure can quickly identify root causes.

Tools like Windows Performance Monitor, Linux's vmstat, and macOS's Activity Monitor reveal these underlying mechanisms in action.

Virtual memory, demand paging, and page replacement algorithms represent the elegant solutions computer scientists have developed to balance the competing demands of memory capacity, performance, and cost. These fundamental concepts continue to evolve as computing hardware and software requirements advance.