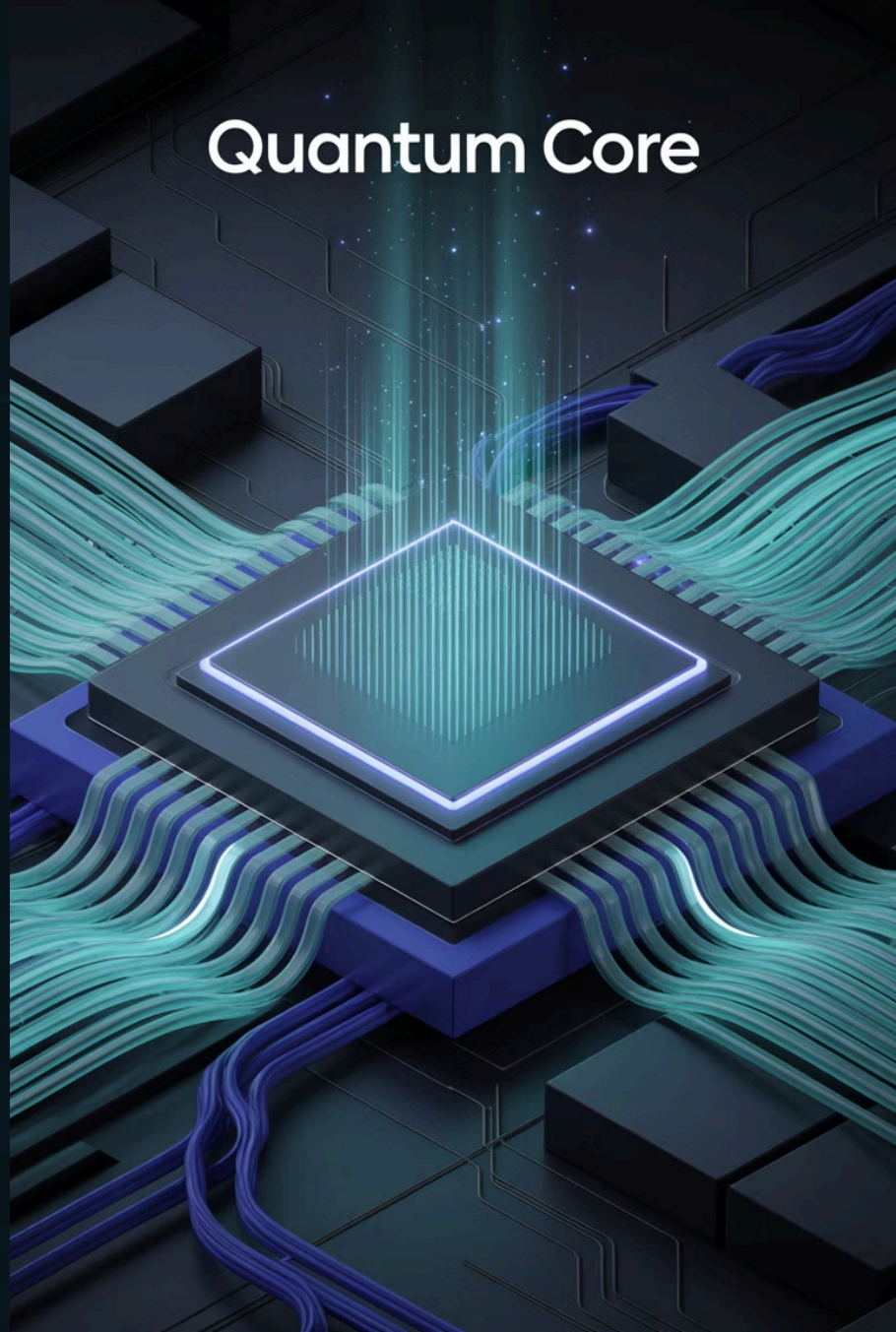
# Process Management and Scheduling in Operating Systems

An operating system's core functionality lies in its ability to manage processes efficiently. This presentation explores the fundamental concepts of process management, differentiation between processes and threads, the process lifecycle, and the various scheduling algorithms that determine how CPU time is allocated.

We'll examine both preemptive and non-preemptive process handling methods, alongside detailed explanations of scheduling algorithms including First Come First Served (FCFS), Shortest Job First (SJF), Priority Scheduling, Round Robin, and Multilevel Queues. Additionally, we'll delve into the fascinating Belady's Anomaly and its implications for memory management.



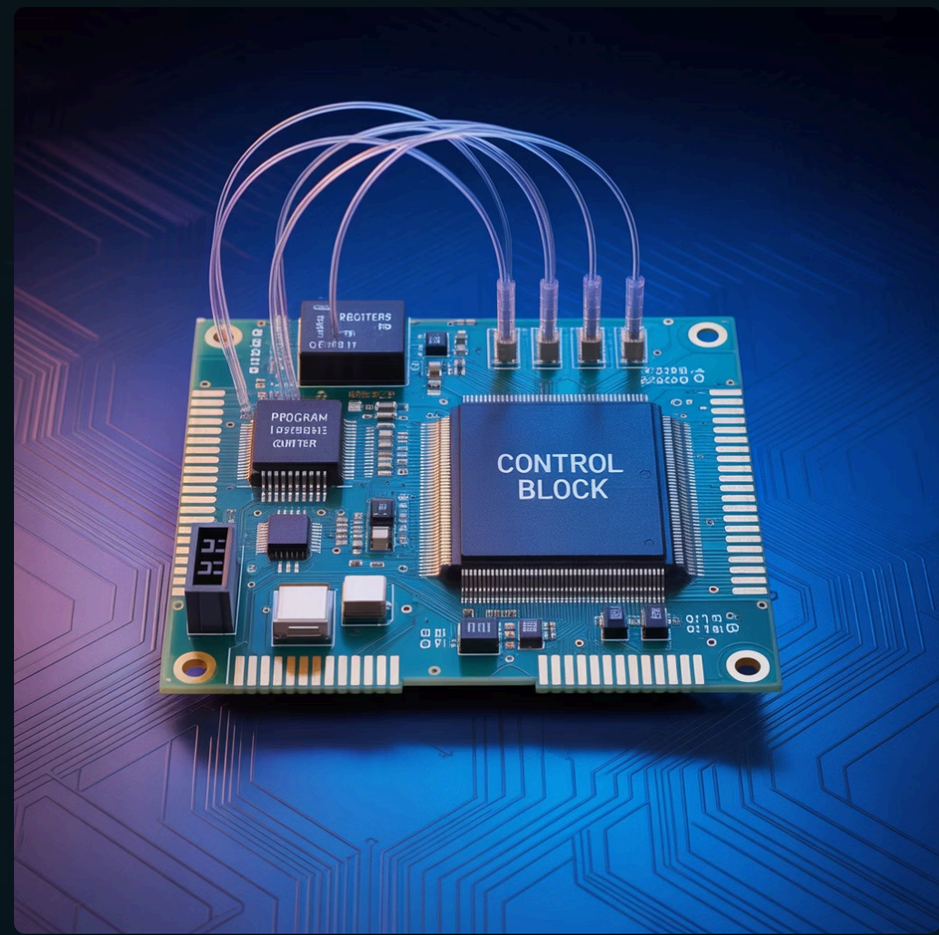
# What is a Process?

A process is a program in execution that is actively managed by the operating system. Unlike a static program file stored on disk, a process is a dynamic entity that requires system resources to perform its tasks.

## Key Characteristics:

- Each process has a unique identifier (PID) assigned by the OS
- Maintains independent memory space and resources
- Contains program code, data, heap, stack, and PCB (Process Control Block)
- Requires CPU time, memory, files, and I/O devices to execute

The operating system maintains a Process Control Block (PCB) for each process, which contains all the information needed to track the process state, including registers, scheduling information, memory management information, accounting information, and I/O status information.




The PCB serves as the repository for any information that may vary from process to process, such as the current state of the process, its priority in the scheduling queue, and its memory allocation details.

# Process States

### New

Process is being created. The operating system is performing the creation of PCB and allocating necessary resources.

### Terminated

Process has finished execution or has been terminated by the operating system. Resources are deallocated.

### Ready

Process is waiting to be assigned to a processor. Ready processes are waiting in the ready queue for execution.

### Running

Process is executing instructions on the CPU. Only one process can be in the running state per CPU core.

### Waiting

Process is waiting for some event to occur (such as an I/O completion or reception of a signal).

The operating system manages these state transitions through system calls, interrupts, and scheduler decisions. The process moves between these states throughout its lifetime based on CPU availability, I/O requests, and program behaviour.

State transitions are critical for multi-tasking operating systems to efficiently manage multiple processes simultaneously, creating the illusion of parallel execution even on single-core processors.

# Preemptive vs Non-Preemptive Processes

## Preemptive Scheduling

In preemptive scheduling, the operating system can forcibly remove a running process from the CPU based on a scheduling decision, without the process voluntarily yielding control.

- OS can interrupt currently running process
- Higher priority tasks can preempt lower ones
- Typically uses time slicing to share CPU time
- More complex to implement but more flexible

## Non-Preemptive Scheduling

In non-preemptive scheduling, once a process begins execution, it continues until it either completes or voluntarily yields the CPU (e.g., by making an I/O request).

- Process runs until completion or voluntary blocking
- No external interruption by scheduler
- Simpler to implement but less responsive
- Can lead to long waiting times for high-priority tasks

The choice between preemptive and non-preemptive scheduling significantly impacts system responsiveness, throughput, and fairness in resource allocation. Modern general-purpose operating systems predominantly use preemptive scheduling to ensure timely execution of critical tasks and prevent any single process from monopolising system resources.

# Characteristics of Preemptive Scheduling

Preemptive scheduling is characterised by the ability of the operating system to reclaim control of the CPU from a running process. This approach offers several distinct advantages that make it the preferred choice for modern interactive operating systems.

### Improved Resource Utilisation

CPU time is more evenly distributed among processes, preventing any single process from monopolising system resources.

### Enhanced System Responsiveness

High-priority tasks can be executed promptly by preempting lower-priority ones, ensuring critical operations receive immediate attention.

### Better Support for Real-Time Systems

Time-sensitive operations can be guaranteed specific CPU time allocations, making preemptive scheduling essential for real-time applications.

### Reduced Risk of Process Monopolisation

Prevents poorly designed or malicious processes from indefinitely controlling system resources through forced time-sharing.

### Examples of Preemptive Operating Systems

- Linux with the Completely Fair Scheduler (CFS)
- Windows with its priority-based preemptive scheduler
- macOS with Grand Central Dispatch
- Android with the Linux-based scheduler
- Real-time operating systems like QNX and VxWorks

While preemptive scheduling offers significant advantages, it also introduces additional complexity in implementation. The system must carefully save and restore process states during context switches, manage synchronisation issues, and handle potential race conditions that can arise from arbitrary interruption.

Additionally, frequent context switching can lead to increased overhead, potentially reducing overall system throughput in certain workloads where predictability is more important than responsiveness.

# Characteristics of Non-Preemptive Scheduling

## Implementation Simplicity

Non-preemptive scheduling is considerably simpler to implement because the operating system doesn't need complex mechanisms to forcibly reclaim CPU control. This simplicity translates to reduced kernel code complexity and fewer potential bugs.

## Minimal Context Switching Overhead

Since processes run to completion or until they voluntarily yield, there are fewer context switches. Each context switch requires saving and restoring registers, cache invalidation, and other overhead that can impact system performance.

## Predictable Execution

Process execution times are more predictable without external interruptions, which can be beneficial for certain types of batch processing workloads where throughput is prioritised over response time.

## Reduced Synchronisation Complexity

With non-preemptive scheduling, critical sections of code are naturally protected from concurrent access without additional synchronisation primitives, reducing the potential for race conditions.

Despite these advantages, non-preemptive scheduling has significant limitations. Most notably, a long-running or compute-intensive process can monopolise the CPU, causing system responsiveness to suffer dramatically. Additionally, priority inversions can occur where high-priority tasks must wait for lower-priority ones to complete.

For these reasons, non-preemptive scheduling is primarily used in specialised systems like certain embedded devices, batch processing systems, or specific subsystems where predictability and simplicity outweigh the need for responsiveness.

# Examples of Scheduling Algorithms

Operating systems employ various scheduling algorithms that can be classified as either preemptive or non-preemptive based on how they handle process execution and CPU allocation.

## Preemptive Algorithms

- **Round Robin (RR)**: Allocates a fixed time quantum to each process in a circular queue. When the time quantum expires, the process is preempted and moved to the back of the queue.
- **Preemptive Priority Scheduling**: Assigns priorities to processes and preempts the running process if a higher priority process becomes ready.
- **Preemptive Shortest Job First (PSJF)**: Preempts the current process if a newly arrived process has a shorter expected completion time.
- **Shortest Remaining Time First (SRTF)**: Preempts the current process if a newly arrived process has a shorter remaining execution time.
- **Multilevel Feedback Queue**: Uses multiple queues with different priorities and time quanta, dynamically adjusting process priorities based on behaviour.

## Non-Preemptive Algorithms

- **First Come First Served (FCFS)**: Executes processes in the order they arrive, with each process running to completion.
- **Shortest Job First (SJF)**: Selects the process with the shortest expected execution time, allowing it to run to completion.
- **Non-preemptive Priority Scheduling**: Assigns priorities to processes but only considers priority at process selection time, not during execution.
- **Highest Response Ratio Next (HRRN)**: Selects processes based on a ratio of waiting time to expected execution time.
- **Deadline Scheduling**: Executes processes based on approaching deadlines in certain real-time systems.

Most modern operating systems implement hybrid approaches, combining multiple scheduling strategies to balance responsiveness, fairness, and throughput for different types of workloads. For example, Linux's Completely Fair Scheduler (CFS) uses a sophisticated approach based on fair queuing, while Windows employs a multi-level feedback queue with priority boosting.

# Process vs Thread: Key Differences


Processes and threads are both units of execution managed by the operating system, but they differ significantly in their properties and resource requirements.

## Resource Ownership

A process has its own memory space, file descriptors, and system resources. Threads within a process share these resources, including code, data, and open files.

## Communication Overhead

Inter-process communication (IPC) requires special mechanisms and is relatively expensive. Threads can communicate directly through shared memory with minimal overhead.

## Creation and Context Switching

Creating a new process is resource-intensive as the OS must duplicate the parent's address space. Thread creation is lighter as it only requires a new stack and thread control block.

## Fault Isolation

A fault in one process doesn't affect others. However, a fault in one thread can compromise the entire process and all its threads.

Threads are often called "lightweight processes" because they provide a more efficient way to achieve concurrent execution within an application. By sharing the same address space, threads enable parallel processing without the overhead of separate memory allocation, making them ideal for applications that benefit from concurrent execution on multicore processors.

# Comparative Table: Process vs Thread

Understanding the specific differences between processes and threads is crucial for effective system design and resource management. The following table provides a detailed comparison across various dimensions:

| Characteristic | Process | Thread |
|---|---|---|
| Definition | An independent program in execution with its own memory space | A lightweight execution unit within a process, sharing its resources |
| Memory Space | Has its own independent memory space (code, data, heap, stack) | Shares code, data, and heap with other threads; has its own stack |
| Resource Consumption | Heavyweight, requires more system resources | Lightweight, requires fewer system resources |
| Creation Time | Slower to create due to resource allocation | Faster to create as it shares resources with the parent process |
| Context Switch Time | More expensive, involves MMU, TLB flush, cache invalidation | Less expensive, primarily involves register state changes |
| Communication | Requires IPC mechanisms (pipes, sockets, shared memory) | Direct access to shared memory, simpler synchronisation |
| Isolation | Strong isolation; one process crash doesn't affect others | Weak isolation; thread crash affects entire process |
| Control Block | Process Control Block (PCB) | Thread Control Block (TCB) |
| System Calls | fork(), exec(), exit() | pthread_create(), pthread_exit(), pthread_join() |

This comparison highlights why modern software often employs a hybrid approach—using multiple processes for isolation and security, while leveraging multiple threads within each process for efficient concurrent execution. Web browsers, for instance, typically isolate different websites in separate processes for security while using multiple threads within each process for responsive UI and parallel processing.

# Process Management Overview

Process management is a fundamental responsibility of the operating system, encompassing all activities related to creating, scheduling, and terminating processes. Effective process management ensures optimal utilisation of system resources while maintaining system stability and responsiveness.

## Key Components of Process Management:

- **Process Creation**: Allocating memory, initialising PCB, and loading program code
- **Process Scheduling**: Determining which process executes and for how long
- **Process Synchronisation**: Managing access to shared resources to prevent conflicts
- **Process Communication**: Enabling information exchange between processes
- **Process Termination**: Releasing resources and handling exit status


## Process Management Mechanisms:

- **Dispatcher**: Responsible for context switching, transferring control to the selected process
- **Scheduler**: Implements scheduling algorithms to select the next process to run
- **PCB Manager**: Maintains and updates the Process Control Blocks
- **Memory Manager**: Allocates and deallocates memory for processes
- **I/O Manager**: Handles process requests for input/output operations

The efficiency of process management directly impacts system performance, user experience, and resource utilisation. Modern operating systems employ sophisticated techniques to balance competing goals such as maximising throughput, minimising response time, ensuring fairness, and maintaining predictability in process execution.

Process management strategies often vary based on the system's purpose—desktop operating systems prioritise interactive responsiveness, server systems focus on throughput and fairness, while real-time systems emphasise predictability and meeting deadlines.

# Process Life Cycle: Overview

The process life cycle represents the complete journey of a process from its creation to termination. Understanding this cycle is crucial for effective process management and system design. Each phase in the cycle corresponds to different resource requirements and system interactions.

The operating system maintains data structures to track processes in each state, typically using various queues such as the ready queue, I/O wait queues, and job queues. Transitions between states are triggered by scheduler decisions, system calls, interrupts, and I/O completion events.

Efficient management of these transitions is critical for system performance. For example, minimising the time processes spend in the waiting state can improve overall throughput, while ensuring fair allocation of CPU time to ready processes prevents starvation and ensures responsive system behaviour.

> Note that this process life cycle focuses on operating system processes, which differs from the business process life cycle that would include planning, design, implementation, and monitoring phases in a business context.

# Process Life Cycle: Phases Explained

## Creation

The OS creates a new process either in response to a system call (e.g., fork() in Unix-like systems) or as part of starting a new application. Resources are allocated, and a PCB is created to track the process state.

## Ready

The process is loaded into memory and is waiting for CPU time. It has all the resources it needs except for the processor. Ready processes are placed in the ready queue, waiting for the scheduler to select them.

## Running

The process is actively executing on the CPU. In a single-core system, only one process can be in the running state at any given time. The process remains in this state until it completes, makes an I/O request, or is preempted.

## Waiting/Blocked

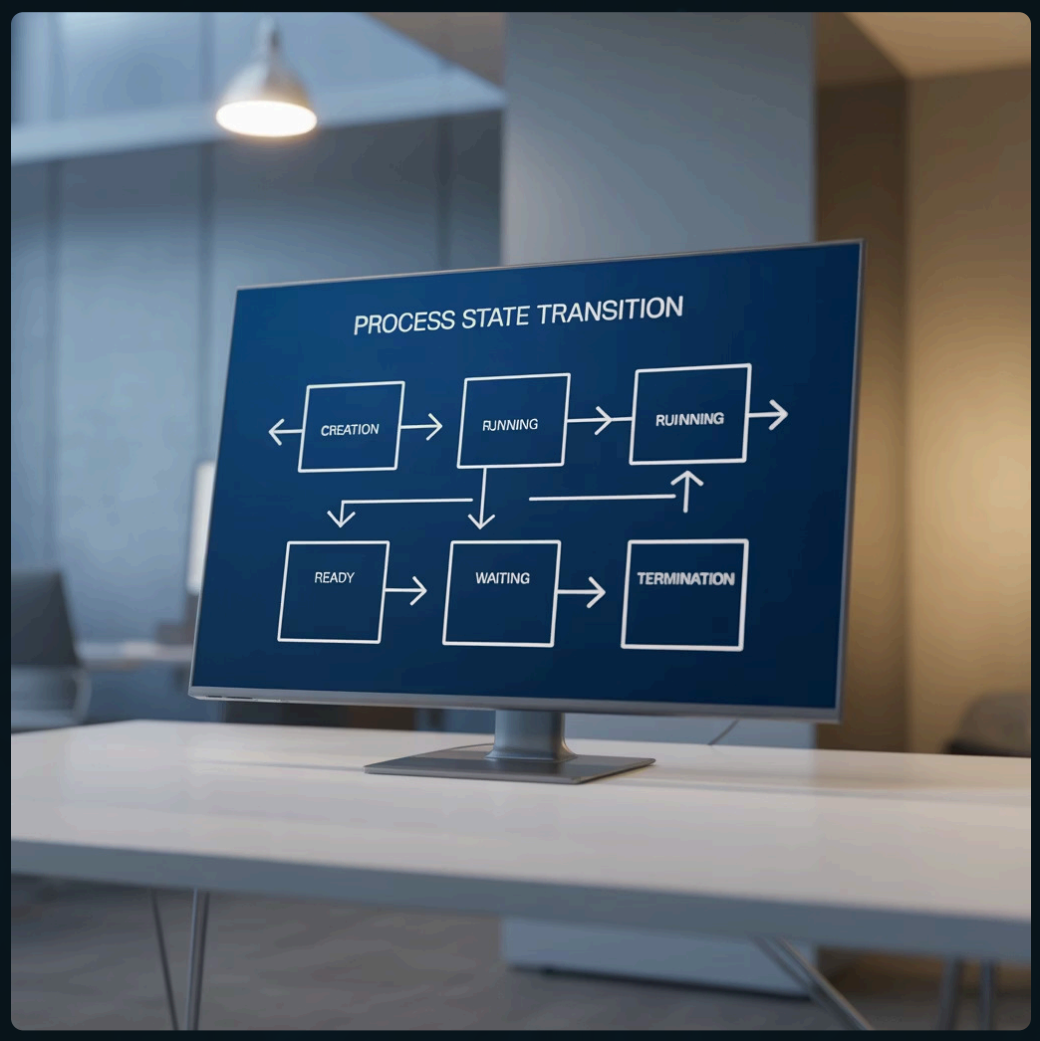
The process is waiting for some event to occur (such as I/O completion, resource availability, or a signal from another process). The process cannot execute until this event occurs, freeing the CPU for other processes.

## Termination

The process has finished execution or has been terminated by the system. The OS reclaims all resources allocated to the process, updates accounting information, and removes the PCB from the system tables.

### Additional States in Modern Systems:

- **Suspended Ready**: Process is temporarily removed from memory to secondary storage but is otherwise ready to execute when restored
- **Suspended Blocked**: Process is both blocked (waiting for an event) and swapped out of memory
- **Zombie**: Process execution has completed, but its entry remains in the process table until the parent collects its exit status




In Unix/Linux systems, the `ps` command reveals these states with single-letter codes: R (running), S (sleeping/waiting), Z (zombie), T (stopped), and D (uninterruptible sleep).

# What are Schedulers?

Schedulers are essential components of an operating system that determine which processes receive CPU time and in what order. They implement scheduling algorithms to optimise system performance based on different criteria such as throughput, response time, resource utilisation, and fairness.

## Key Functions of Schedulers

- **Process Selection**: Choosing which process from the ready queue should be allocated CPU time next
- **Resource Allocation**: Managing the assignment of CPU and other resources to processes
- **Load Balancing**: Distributing computational load evenly across multiple processors in multicore systems
- **Priority Management**: Handling process priorities and ensuring high-priority tasks receive preferential treatment
- **Deadlock Prevention**: Implementing strategies to avoid or resolve deadlock situations

## Scheduling Criteria

- **CPU Utilisation**: Keeping the CPU as busy as possible, ideally approaching 100%
- **Throughput**: Maximising the number of processes completed per unit time
- **Turnaround Time**: Minimising the time from process submission to completion
- **Waiting Time**: Reducing the time processes spend waiting in the ready queue
- **Response Time**: Decreasing the time from submission to first response (critical for interactive systems)
- **Fairness**: Ensuring all processes receive a fair share of CPU time

Different scheduling algorithms prioritise different criteria, creating trade-offs. For example, algorithms that optimise for throughput might sacrifice response time, while those focused on fairness might not maximise overall system utilisation. The choice of scheduler depends on the system's purpose and workload characteristics.

# Types of Schedulers

## Long-Term Scheduler (Job Scheduler)

Controls the degree of multiprogramming by selecting which processes from the submission queue should be admitted to the ready queue and loaded into memory.

- Invoked infrequently (seconds to minutes)
- Determines overall system load
- Balances mix of CPU-bound and I/O-bound processes
- Makes admission decisions based on resource availability

## Medium-Term Scheduler (Swapper)

Temporarily removes processes from memory to reduce system load or make room for other processes, implementing a technique called swapping.

- Manages memory allocation across processes
- Swaps out inactive processes to secondary storage
- Restores swapped processes when resources are available
- Helps maintain optimal memory utilisation

## Short-Term Scheduler (CPU Scheduler)

Selects which process from the ready queue should be executed next and allocates the CPU accordingly.

- Invoked very frequently (milliseconds)
- Must make decisions quickly to minimise overhead
- Implements specific scheduling algorithms (FCFS, SJF, RR, etc.)
- Directly impacts system responsiveness and fairness

These three levels of scheduling work together to manage system resources effectively. The long-term scheduler controls how many processes are in the system, the medium-term scheduler manages memory allocation, and the short-term scheduler determines which ready process receives CPU time.

In many modern desktop operating systems, the distinction between long-term and medium-term schedulers has blurred, with their functions often combined into a more general memory and process manager. However, the conceptual separation remains useful for understanding resource allocation strategies.

# First Come First Served (FCFS) Scheduling

First Come First Served (FCFS) is the simplest scheduling algorithm, executing processes in the exact order they arrive in the ready queue. It is a non-preemptive algorithm, meaning once a process begins execution, it continues until completion or until it voluntarily releases the CPU.

## Key Characteristics:

- **Implementation**: Typically uses a FIFO (First-In-First-Out) queue data structure
- **Non-preemptive**: Processes run to completion once started
- **Simple**: Easy to understand and implement
- **Fair**: No process is prioritised over others; all are treated equally

> ⚠ **Convoy Effect**
>
> FCFS is susceptible to the convoy effect, where a single CPU-bound process can cause numerous I/O-bound processes to wait, leading to inefficient CPU and I/O device utilisation. This results in poor system performance despite potentially high CPU usage.


## Advantages:

- Simple to implement and understand
- No starvation as every process eventually gets CPU time
- Minimal scheduler overhead

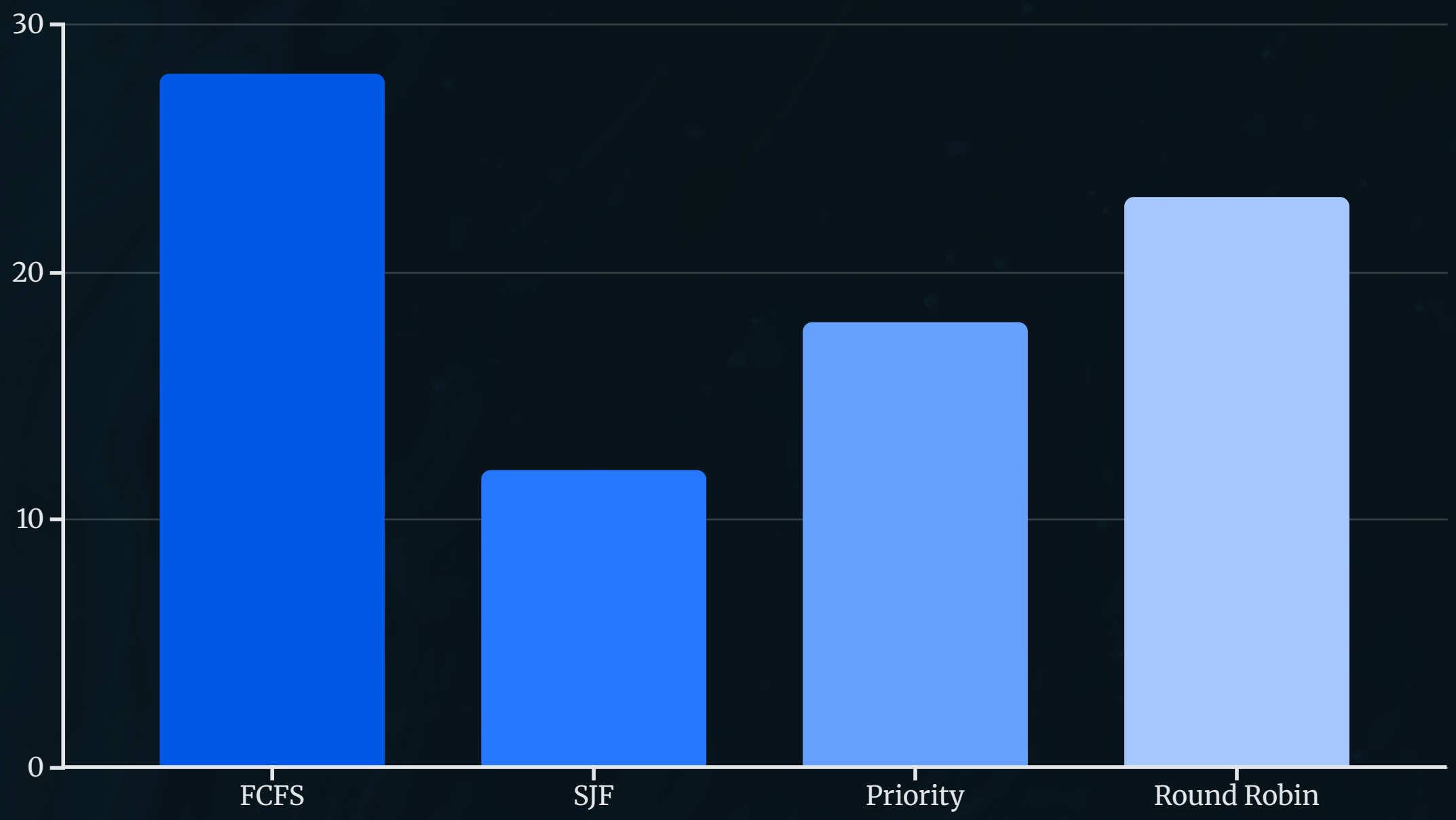
## Disadvantages:

- Long average waiting time if short processes are behind long ones
- Poor responsiveness for interactive systems
- Not suitable for time-sharing systems
- Performance heavily dependent on arrival order

FCFS is rarely used as the primary scheduling algorithm in modern general-purpose operating systems due to its limitations. However, it often serves as a fallback or secondary algorithm within more complex scheduling schemes, particularly for processes of equal priority.

# Shortest Job First (SJF) Scheduling

Shortest Job First (SJF) is a scheduling algorithm that selects the process with the smallest execution time first. In its non-preemptive form, once a process begins execution, it runs to completion regardless of new arrivals.




## Key Characteristics:

- **Optimal Average Waiting Time**: Theoretically provides the minimum average waiting time for a given set of processes
- **Non-preemptive by Default**: Once selected, a process runs to completion (though a preemptive variant exists called SRTF—Shortest Remaining Time First)
- **Requires Job Length Estimation**: Implementation requires knowledge or prediction of process execution times
- **Potential for Starvation**: Long processes may be indefinitely delayed if shorter processes continuously arrive

## Implementation Challenges:

- **Execution Time Prediction**: Accurate prediction of process execution time is difficult in practice
- **Estimation Techniques**: Often uses historical data and exponential averaging to estimate future execution times
- **Overhead Consideration**: The algorithm itself adds computational overhead to determine the shortest job
- **Aging Mechanism**: Typically requires an aging mechanism to prevent starvation of longer processes

SJF is particularly effective in batch processing environments where job execution times are known in advance. It's less suitable for interactive systems where user experience depends on responsive handling of all processes, regardless of their length. Modern implementations often use SJF principles within more complex scheduling frameworks, combined with mechanisms to prevent starvation of longer processes.

# Priority Scheduling

Priority scheduling assigns a priority value to each process, and the CPU is allocated to the process with the highest priority. Priority can be determined by various factors such as memory requirements, time constraints, or importance of the task.

## Types of Priority Scheduling

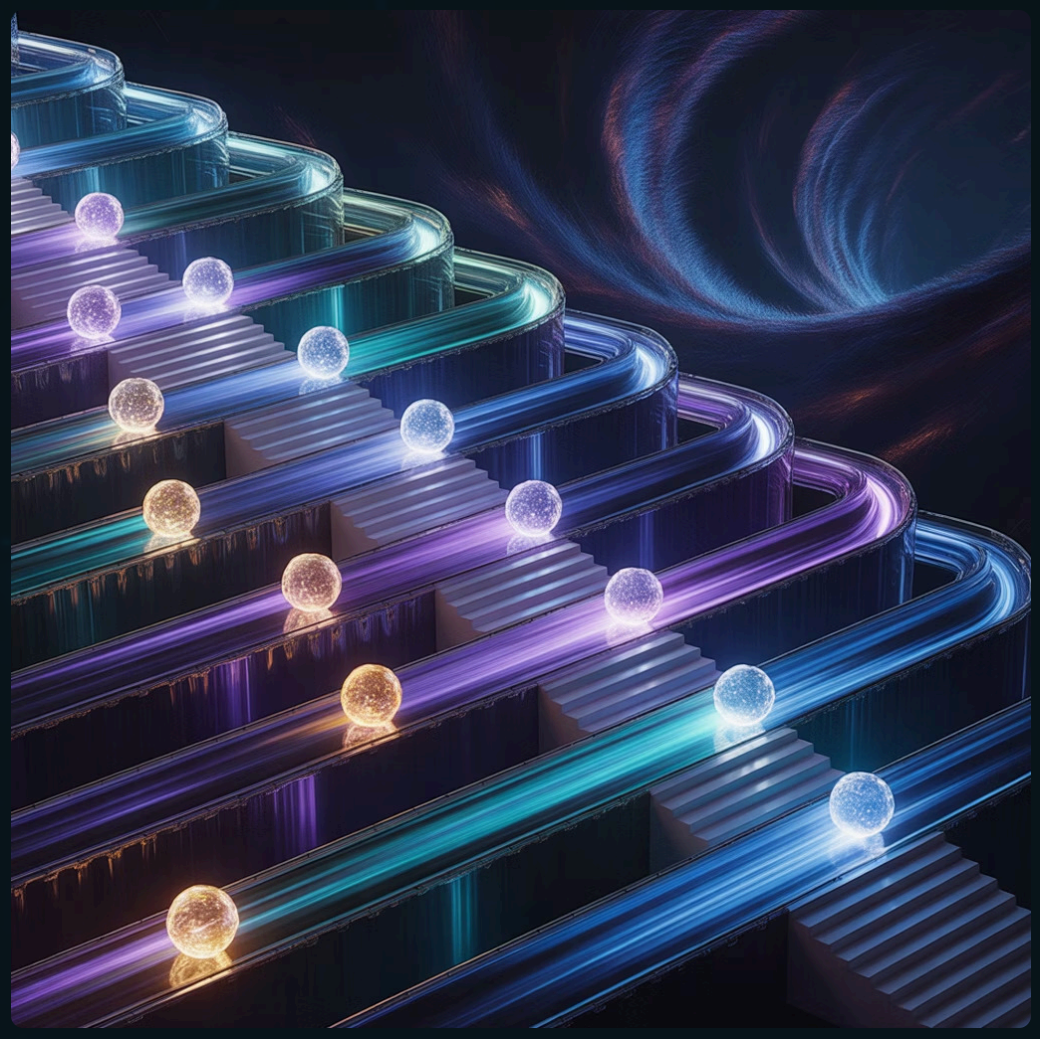
- **Non-preemptive Priority**: Once a process starts executing, it continues until completion or voluntary release, regardless of the arrival of higher-priority processes
- **Preemptive Priority**: A running process is preempted if a higher-priority process arrives, immediately giving CPU control to the higher-priority process

## Priority Assignment Approaches

- **Static Priorities**: Fixed at process creation time and remain unchanged throughout execution
- **Dynamic Priorities**: Adjusted during execution based on aging, resource usage, or other factors

## Priority Inversion Problem

A situation where a high-priority process is indirectly preempted by a lower-priority process. This occurs when a high-priority process needs a resource that is currently held by a low-priority process, which itself is preempted by a medium-priority process. Solutions include priority inheritance or priority ceiling protocols.



### Advantages:

- Allows important processes to receive preferential treatment
- Suitable for real-time systems where meeting deadlines is critical
- Flexible prioritisation based on system or business needs
- Can be combined with other scheduling techniques for hybrid approaches

### Disadvantages:

- Potential starvation of low-priority processes
- Complexity in determining appropriate priority values
- Priority inversion problems can lead to unexpected behaviour
- Overhead in maintaining and updating priority information

To address the starvation problem, most priority scheduling implementations include an aging mechanism that gradually increases the priority of processes that have been waiting for a long time. This ensures that even low-priority processes eventually get CPU time, maintaining system fairness while still respecting the general priority ordering.

# Round Robin (RR) Scheduling

Round Robin is a preemptive scheduling algorithm specifically designed for time-sharing systems. It allocates a small unit of time, called a time quantum or time slice, to each process in a circular queue fashion.

### Key Characteristics:

- **Time Quantum**: Each process is allocated a fixed time slice, typically 10-100 milliseconds
- **Circular Queue**: Processes are arranged in a circular ready queue
- **Preemptive**: When the time quantum expires, the process is preempted and moved to the back of the queue
- **Fair Allocation**: All processes receive equal CPU time in a cyclic manner

### Time Quantum Considerations:

- **Too Small**: Excessive context switching overhead, reducing effective CPU utilisation
- **Too Large**: Degenerates to FCFS behaviour, reducing responsiveness
- **Optimal Range**: Typically set so that 80% of CPU bursts are shorter than the quantum
- **Adaptive Approaches**: Some systems dynamically adjust the quantum based on system load

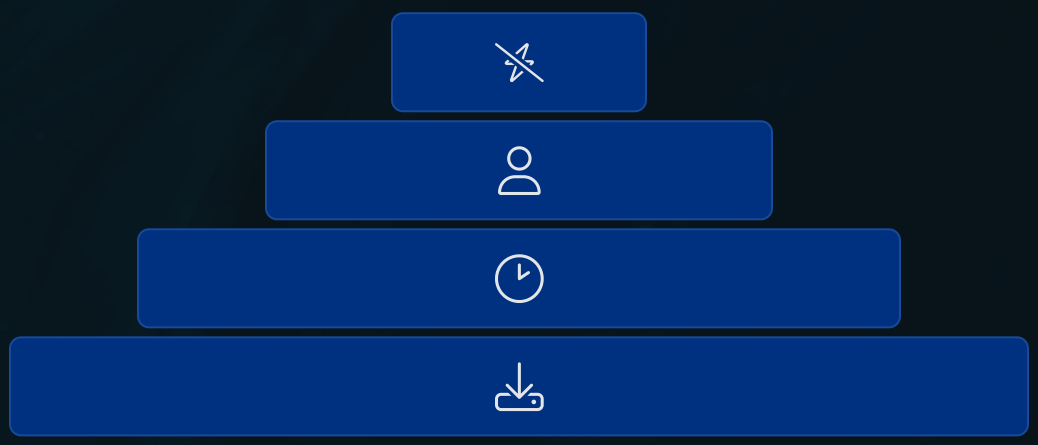
Round Robin is particularly effective in interactive environments where quick response time is critical. It ensures that no process monopolises the CPU for an extended period, providing good responsiveness for short interactive commands while still making progress on longer computational tasks.

Performance metrics for Round Robin are heavily influenced by the time quantum setting. With a properly tuned quantum, RR provides a good balance of fairness, responsiveness, and throughput. Most modern operating systems use variants of Round Robin combined with priority mechanisms to create multilevel feedback queue schedulers that adapt to different workload characteristics.

# Multilevel Queue Scheduling

Multilevel Queue scheduling partitions the ready queue into several separate queues, each with its own scheduling algorithm. Processes are permanently assigned to one queue based on certain properties like process type, memory size, or priority.

## Queue Hierarchies:



### Foreground (Interactive)

Highest priority, typically uses Round Robin scheduling

### Interactive Editing

Medium priority, often uses priority scheduling with Round Robin

### Batch Processes

Lower priority, may use FCFS scheduling

### Student Processes

Lowest priority, typically FCFS or SJF


## Inter-Queue Scheduling Approaches:

- **Fixed Priority**: Higher queues always served first (may lead to starvation)
- **Time Slice**: Each queue gets a certain percentage of CPU time
- **Hybrid**: Combination of priority and time sharing between queues

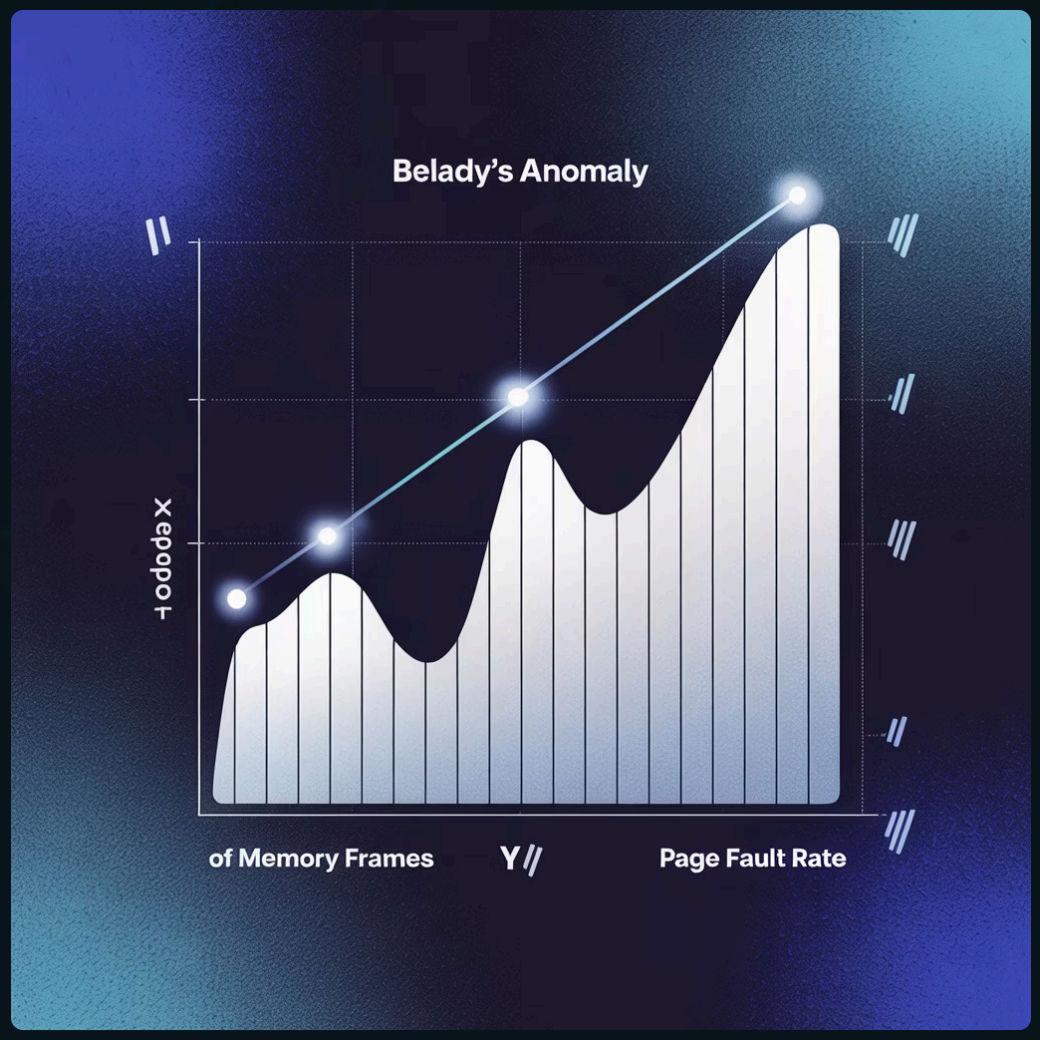
A related but more flexible approach is the Multilevel Feedback Queue, which allows processes to move between queues based on their behaviour. Processes that use their entire time quantum repeatedly are moved to lower-priority queues, while I/O-bound processes that release the CPU voluntarily move to higher-priority queues.

Multilevel Queue scheduling is particularly effective in systems with diverse workloads, such as a combination of real-time, interactive, and batch processes. By segregating different types of processes and applying the most appropriate scheduling algorithm to each, the system can optimise multiple performance metrics simultaneously.

Most modern operating systems, including Linux, Windows, and macOS, use variants of multilevel queue scheduling to balance the competing demands of different types of processes while maintaining system responsiveness and throughput.

# Belady's Anomaly

Belady's Anomaly is a counterintuitive phenomenon in page replacement algorithms where increasing the number of page frames (physical memory allocated to a process) can sometimes lead to an increase in page faults rather than a decrease.




## Key Aspects:

- **Discovery**: First observed by László Belády in 1969 while studying the FIFO (First-In-First-Out) page replacement algorithm
- **Affected Algorithms**: Primarily occurs in FIFO page replacement; does not occur in LRU (Least Recently Used) or optimal algorithms
- **Root Cause**: Lack of stack property in certain page replacement algorithms
- **Stack Property**: An algorithm has the stack property if the set of pages in memory for n frames is always a subset of the pages that would be in memory for n+1 frames

> While Belady's Anomaly is primarily a concern in memory management rather than process scheduling, it illustrates an important principle: seemingly intuitive improvements to resource allocation can sometimes lead to unexpected performance degradation due to complex system interactions.

## Example Scenario:

Consider a reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

With 3 frames using FIFO:

- 9 page faults occur

With 4 frames using FIFO:

- 10 page faults occur

This counterintuitive result occurs because the additional frame changes the timing of replacements, leading to less optimal page residence patterns despite having more physical memory available.

Belady's Anomaly has significant implications for memory management system design. It demonstrates that simply adding more resources doesn't guarantee better performance and highlights the importance of using page replacement algorithms with the stack property (like LRU) in virtual memory systems.

Modern operating systems typically avoid algorithms susceptible to Belady's Anomaly, instead opting for approximations of LRU or more sophisticated approaches like the Clock algorithm that provide more predictable performance as memory resources scale.