# REPORT

# OPERATING SYSTEM PRINCIPLES AND NUMBER SYSTEMS

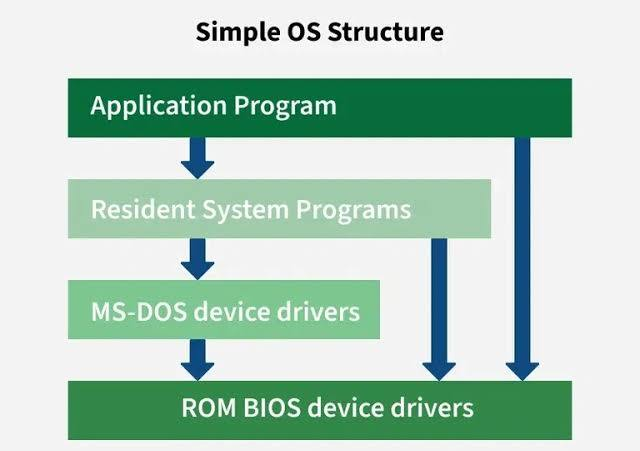
## Part I: Core Fundamentals of Operating Systems

This section introduces the fundamental concepts of operating systems, including their definition, key functions, and the distinction between a program and a process.

### 1.1. Definition and Role of the Operating System

An Operating System (OS) is system software that manages computer hardware and provides basic services for application software.Its main task is to control the hardware and organize the execution of computer programs.Essentially, the OS acts as an intermediary layer between the user and the computer's hardware.

The OS has two primary roles: as an "extended machine" that simplifies the complexity of the hardware for the user, and as a "resource manager" that efficiently allocates the CPU, memory, and I/O devices among programs. Balancing convenience and efficient management is the core challenge in operating system design.

**

### 1.2. Analysis of Key Services

An operating system provides several key services for the system to operate efficiently, including:

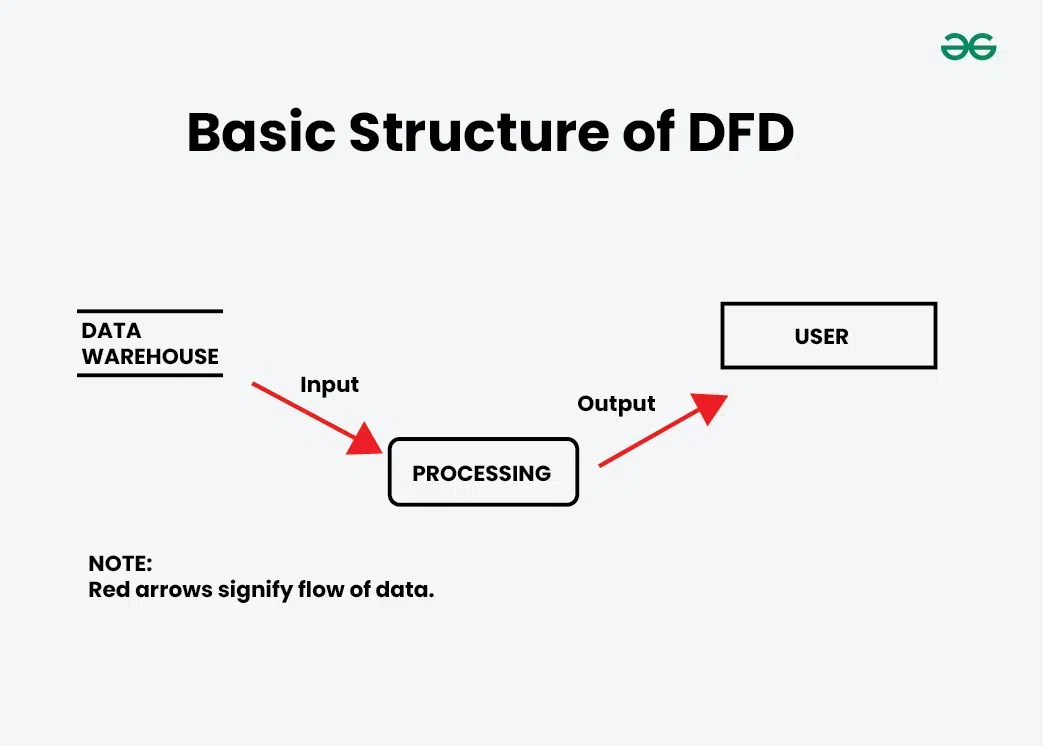
* **Process Management:** Creating, deleting, scheduling, and synchronizing processes.
* **Memory Management:** Allocating and deallocating memory space.
* **File System Management:** Organizing files and directories on secondary storage.
* **Device Management:** Interacting with hardware through drivers.
* **Security:** Protecting system resources through authentication and access control.

These services work closely together. For example, launching an application triggers a chain of activities from the User Interface, File System Management, Process Management, and Memory Management to locate the file, create a process, and allocate necessary resources.

### 1.3. Program vs. Process: A Comparative Analysis

A "program" is a static entity, a file of instructions stored on a disk. In contrast, a "process" is a dynamic entity, a program in execution. A process contains not only the program code but also its entire execution state, such as the program counter, registers, and variables.

This distinction is fundamental to multitasking systems. The OS manages each process through a data structure called the **Process Control Block (PCB)**. The PCB stores the entire context of a process, allowing the OS to pause one process and switch to executing another, creating the illusion of running multiple programs simultaneously.

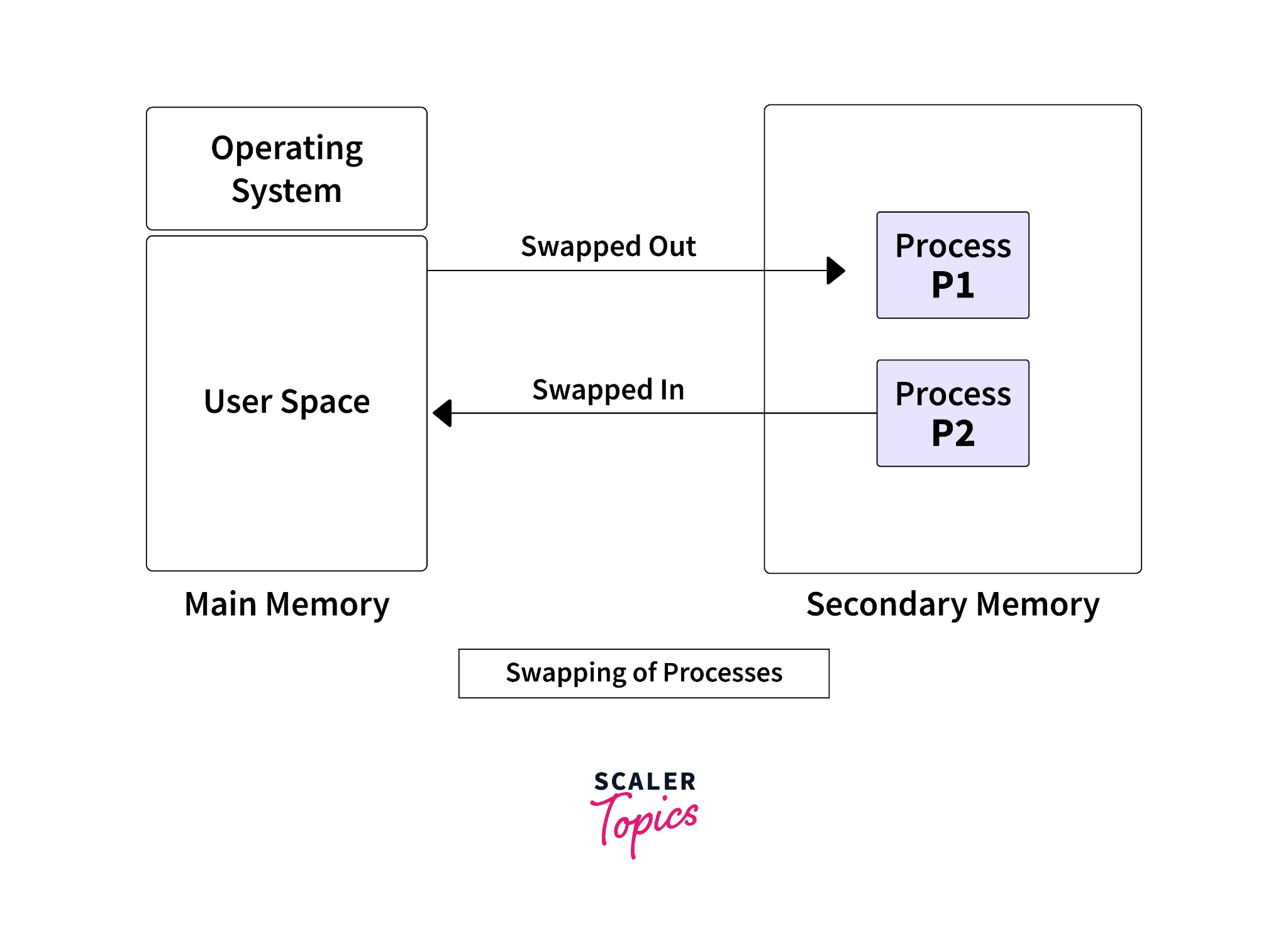
**

## Part II: Modern Memory Management Techniques

This section explores the mechanisms for managing main memory, allowing the system to run more processes than the physical RAM capacity.

### 2.1. The Swapping Mechanism

Swapping is a technique of moving an entire process from main memory (RAM) to secondary storage (hard disk) to free up RAM. The purpose is to run more processes than the physical RAM capacity allows. Although effective, swapping can reduce performance due to the latency of disk I/O operations. This is a classic trade-off between space and time.

**

### 2.2. Dynamic Memory Management and Addressing

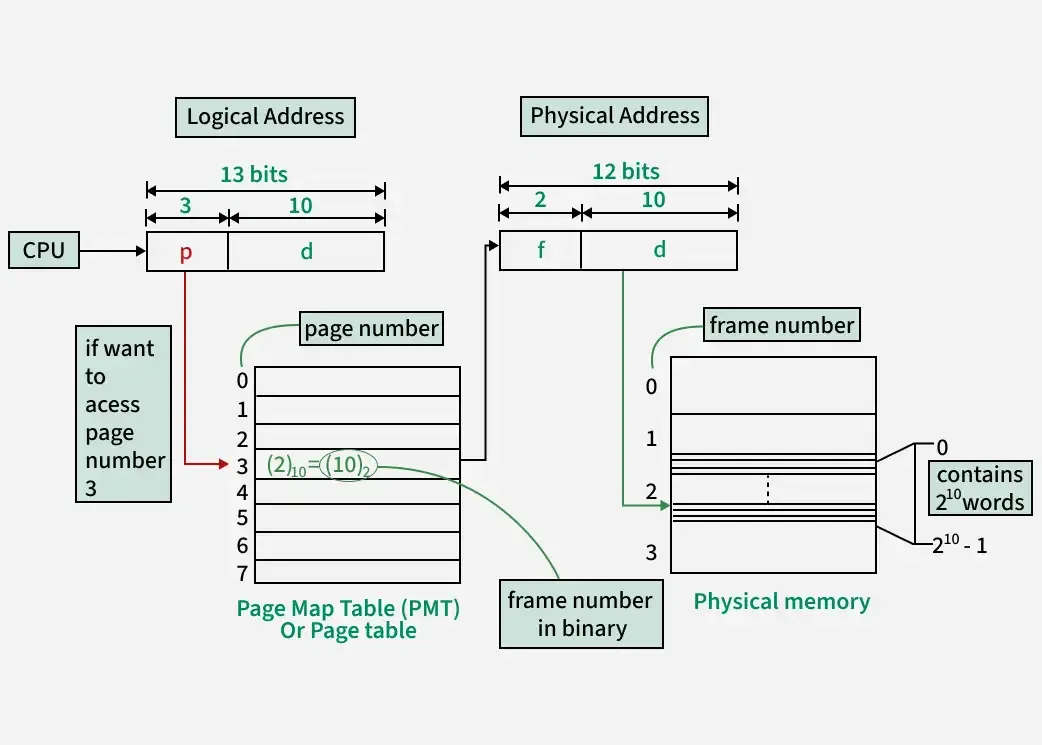
When a process can be loaded into different locations in memory, the address generated by the CPU (logical address) needs to be translated into a physical address in RAM. The **Memory Management Unit (MMU)** is the hardware that performs this translation. This mechanism allows for:

1. **Relocation:** Loading a program anywhere in physical memory.
2. **Protection:** Preventing processes from accessing each other's memory.
3. **Efficiency:** Allowing non-contiguous use of physical memory.

### 2.3. The Paging Technique

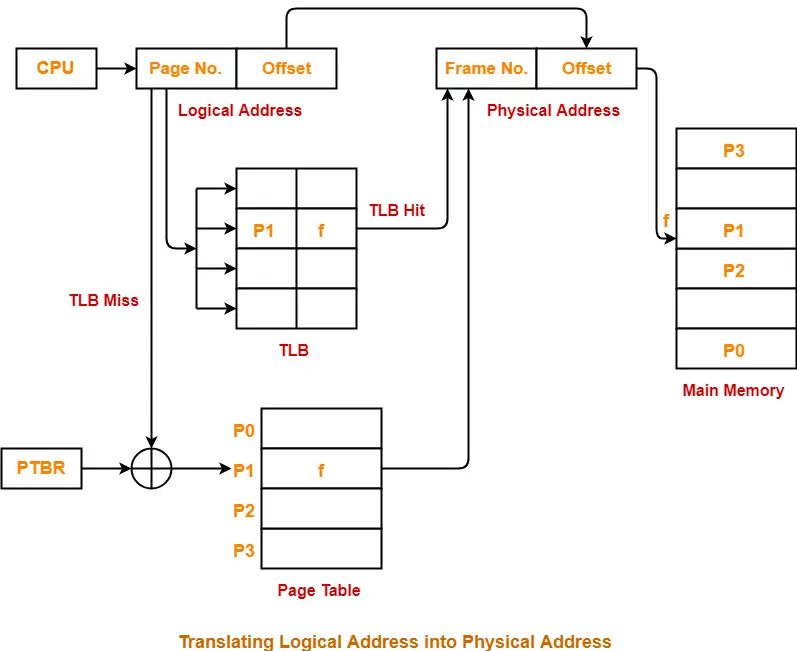
Paging divides a process's virtual address space into fixed-size "pages" and physical memory into corresponding "frames". The OS uses a "page table" to map each virtual page to a physical frame. This allows a process's pages to be stored non-contiguously in RAM.

"Demand paging" is a technique where a page is loaded into RAM only when it is needed. When a process accesses a page not in RAM, a "page fault" occurs, and the OS loads that page from the disk.

**

### 2.4. Optimization with the Translation Lookaside Buffer (TLB)

The **Translation Lookaside Buffer (TLB)** is a small, fast hardware cache that stores recent address translation results. When an address needs to be translated, the MMU first checks the TLB. If found (a TLB hit), the physical address is retrieved immediately.If not (a TLB miss), the MMU must look up the page table in main memory, a much slower process.The TLB significantly reduces the performance overhead of paging by exploiting the principle of locality.

**

## Part III: CPU Scheduling Strategies

This section compares CPU scheduling algorithms, which decide which process gets the CPU next.

### 3.1. Introduction to Types of Scheduling

There are three main types of scheduling:

* **Long-term scheduling:** Decides which processes are admitted to the system.
* **Short-term scheduling (CPU scheduler):** Selects the next process to execute.
* **Medium-term scheduling:** Involved in swapping processes.

This analysis focuses on short-term scheduling, which directly impacts system performance.

### 3.2. Analysis of Common Scheduling Algorithms

* **First-Come, First-Served (FCFS):** The process that arrives first is served first. It is a non-preemptive algorithm, simple but can cause the "convoy effect," where short processes wait for a long one.
* **Shortest-Job-First (SJF):** Prioritizes the process with the shortest next CPU burst. This algorithm is optimal for average waiting time but it's difficult to predict the burst time and can lead to "starvation" for long processes.
* **Round Robin (RR):** Designed for time-sharing systems, RR allocates a "time quantum" to each process. If not finished, the process is preempted and moved to the back of the queue. This algorithm provides good response time, but its performance depends on the quantum size.
* **Priority Scheduling:** Assigns a priority to each process, and the CPU is allocated to the process with the highest priority. The main issue is the risk of starvation for low-priority processes, which can be addressed with "aging".

### 3.3. Comparative Summary of Scheduling Algorithms

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criteria | FCFS (First-Come, First-Served) | SJF (Shortest-Job-First) | Round Robin (RR) | Priority Scheduling |
| **Principle** | First come, first served | Prioritizes shortest burst time | Time-sharing per quantum | Prioritizes highest priority process |
| **Type** | Non-preemptive | Preemptive or Non-preemptive | Preemptive | Preemptive or Non-preemptive |
| **Avg. Waiting Time** | Poor, affected by convoy effect | Optimal (lowest) | Average, depends on quantum | Low for high-priority processes |
| **Starvation Risk** | No | Yes (for long processes) | No | Yes (for low-priority processes) |
| **Main Advantage** | Simple, fair | Maximum throughput | Good for time-sharing systems | Flexible, prioritizes important tasks |
| **Main Disadvantage** | Convoy effect | Hard to predict burst time, causes starvation | Context switch overhead | Causes starvation, needs aging solution |

## Part IV: Guide to Number System Conversion

This section provides methods and examples for converting between common number systems. Using binary as an intermediate for converting between octal and hexadecimal is an effective method.

*Image note: Quick lookup table for conversions between Hexadecimal, Decimal, and Binary.*

### 4.1. Conversion from Binary

* **To Decimal:** Multiply each binary digit by to the power of its position and sum the results.
  + **Example:**



* + **Example:**



* **To Hexadecimal:** Group bits into sets of 4 from right to left, then convert each group to its corresponding hexadecimal digit.
  + **Example:**



### 4.2. Conversion from Decimal

* **To Binary:**
  + **Integer part:** Repeatedly divide by 2, take the remainders, and read them in reverse.
  + **Fractional part:** Repeatedly multiply by 2, take the integer parts, and read them in order.
  + **Example:**



* **To Hexadecimal:** Repeatedly divide the integer part by 16 and take the remainders.
  + **Example:**



* + **Example:**



### 4.3. Conversions Between Other Number Systems

* **Octal to Hexadecimal:** Convert each octal digit to a 3-bit binary group, then regroup into 4-bit sets to convert to hexadecimal.
  + **Example:**



* **Hexadecimal to Decimal:** Multiply each digit by to the power of its position and sum the results.
  + **Example:**



* **Hexadecimal to Binary:** Convert each hexadecimal digit to its corresponding 4-bit binary group.
  + **Example:**

