**Module 1: Introduction to Operating Systems - Detailed Explanation**

This module lays the groundwork for understanding what an operating system is, why it's essential, and how it has evolved.

**1. What is an OS? Role and Purpose**

An **Operating System (OS)** is system software that manages computer hardware and software resources and provides common services for computer programs. Think of it as the conductor of an orchestra, ensuring all the different instruments (hardware components) and musicians (software applications) play together harmoniously and efficiently.

**Key Roles and Purposes:**

* **Resource Management:** This is the most fundamental role. The OS manages all the computer's resources:
  + **CPU (Processor):** Decides which program gets to use the CPU and for how long (CPU scheduling).
  + **Memory (RAM):** Allocates memory to running programs and ensures they don't interfere with each other (memory management).
  + **I/O Devices:** Manages communication with peripherals like keyboards, mice, printers, hard drives, network cards, etc. (device management).
  + **Files and Storage:** Organizes, stores, retrieves, and protects data on secondary storage (file system management).
* **User Interface:** Provides a way for users to interact with the computer. This can be:
  + **Command Line Interface (CLI):** Users type commands (e.g., bash, cmd).
  + **Graphical User Interface (GUI):** Users interact with visual elements like icons, windows, and menus (e.g., Windows, macOS, modern Linux desktops).
* **Program Execution:** Loads and executes application programs, providing the necessary environment for them to run. It handles their creation, termination, and resource allocation.
* **Error Handling:** Detects and responds to errors, both hardware (e.g., memory errors, printer offline) and software (e.g., application crashes), to maintain system stability.
* **Protection and Security:** Protects system resources from unauthorized access and prevents malicious programs from harming the system or user data. This involves user authentication, access control, and enforcing permissions.
* **Networking:** Manages network connections and communication protocols, allowing computers to share resources and communicate over a network.
* **System Accounting:** Keeps track of resource usage by various users and processes, which can be used for billing or performance analysis.

**Analogy:** Imagine a busy office building.

* The **building manager (OS)** ensures that the offices (applications) have electricity (CPU time), furniture (memory), and access to shared resources like printers (I/O devices).
* They manage who gets which office (resource allocation), prevent tenants from messing with each other's spaces (protection), and provide a way for new tenants to move in (program loading).
* Without a manager, it would be chaos, with everyone fighting for resources.

**2. History and Evolution of OS**

The evolution of OS is closely tied to the evolution of computer hardware and user needs.

* **1940s-1950s: No Operating Systems (Manual Operation)**
  + **Context:** Early electronic computers were enormous, expensive, and used by a single programmer/operator at a time.
  + **Operation:** Programs were loaded manually (e.g., via punch cards, toggle switches) directly into memory. No OS existed; the programmer directly interacted with the hardware.
  + **Problem:** Incredibly inefficient and time-consuming.
* **Mid-1950s - Mid-1960s: Batch Processing Systems**
  + **Concept:** To improve efficiency, jobs with similar requirements were "batched" together. A simple monitor program (early form of OS) would automatically load and execute these jobs one after another.
  + **Features:**
    - **Automatic Job Sequencing:** The monitor would load the next job once the current one finished.
    - **Resident Monitor:** A small part of the OS permanently resided in memory.
    - **Job Control Language (JCL):** Users would specify job requirements (e.g., RUN FORTRAN PROGRAM).
  + **Examples:** General Motors Operating System (GMOS) for IBM 701.
  + **Problem:** Still no direct user interaction during execution; if a job crashed, the CPU would sit idle.
* **Late 1960s - 1970s: Multiprogramming and Time-Sharing Systems**
  + **Context:** Computers became faster and more expensive, leading to the desire for multiple users to share a single machine.
  + **Multiprogramming:**
    - **Concept:** While one program is waiting for an I/O operation to complete, the OS switches the CPU to another program that is ready to execute. This keeps the CPU busy and increases system utilization.
    - **Mechanism:** Requires memory protection (programs can't overwrite each other) and CPU scheduling.
  + **Time-Sharing (Interactive Computing):**
    - **Concept:** An extension of multiprogramming where the CPU time is divided into small slices (time slices or quanta). Each user/program gets a small slice of CPU time in a round-robin fashion, giving the illusion that each user has dedicated access to the machine.
    - **Features:** Interactive terminals, rapid context switching.
    - **Examples:** Compatible Time-Sharing System (CTSS), Multics, UNIX.
  + **Impact:** Revolutionized computing by making it interactive and accessible to many users simultaneously.
* **1980s: Personal Computer (PC) Operating Systems**
  + **Context:** Affordable microprocessors led to the widespread adoption of personal computers.
  + **Features:** Primarily single-user, single-tasking initially. Focused on ease of use for individual users.
  + **Examples:** MS-DOS (Command-line based), Apple Macintosh System Software (introduced GUI to the masses).
  + **Evolution:** Later evolved to support multitasking (e.g., early versions of Windows, OS/2).
* **1990s - Present: Distributed, Networked, Mobile, and Embedded Systems**
  + **Distributed OS:** Manages a collection of networked computers as if they were a single, coherent system. Focuses on resource sharing, fault tolerance, and transparency.
  + **Network OS:** Similar to distributed but explicitly manages resources across a network, with individual machines maintaining their autonomy. (e.g., Windows Server, Linux servers).
  + **Real-Time OS (RTOS):** Designed for applications with strict timing constraints (e.g., industrial control, medical devices). Guarantees a response within a specific deadline.
  + **Mobile OS:** Optimized for smartphones and tablets, focusing on touch interfaces, power management, connectivity, and app ecosystems (e.g., Android, iOS).
  + **Embedded OS:** Small, specialized OS designed for specific devices (e.g., smart appliances, car systems). Often resource-constrained.
  + **Cloud OS / Virtualization:** Modern OS often run as virtual machines on cloud infrastructure, enabling flexible resource allocation and scalability.
  + **Examples:** Windows NT/XP/7/10/11, various Linux distributions, macOS, Android, iOS.

**3. Types of OS (Batch, Time-sharing, Distributed, Real-time, Mobile, Embedded)**

We touched upon these in the history, but let's categorize them clearly:

* **Batch Operating Systems:**
  + **Characteristics:** Executes jobs in batches without direct user interaction. Jobs are processed sequentially.
  + **Pros:** High throughput for large, repetitive tasks. Efficient use of CPU for a single task.
  + **Cons:** No interactivity, long turnaround time if a job fails or takes long.
  + **Use Cases:** Bill processing, payroll systems, large scientific simulations (though less common now for pure batch).
* **Time-Sharing Operating Systems (Interactive OS):**
  + **Characteristics:** Allows multiple users to share a single computer simultaneously. CPU time is divided into "time slices."
  + **Pros:** Provides an interactive experience, high CPU utilization by switching between tasks, fair resource allocation.
  + **Cons:** Increased overhead due to context switching, potential for thrashing if too many processes are loaded.
  + **Use Cases:** Desktop PCs (Windows, macOS, Linux), Servers (web servers, database servers).
* **Distributed Operating Systems:**
  + **Characteristics:** Manages a collection of independent networked computers, making them appear to users as a single, coherent system. Components are distributed across multiple machines.
  + **Pros:** Resource sharing, increased reliability (if one node fails, others can take over), scalability.
  + **Cons:** Complex to design and implement, issues with network transparency, synchronization, and fault tolerance.
  + **Use Cases:** Cloud computing platforms, distributed file systems (e.g., Hadoop Distributed File System - HDFS), network rendering farms.
* **Real-Time Operating Systems (RTOS):**
  + **Characteristics:** Designed for applications where operations must be completed within strict time constraints. Focuses on predictability and deterministic behavior rather than high throughput.
  + **Types:**
    - **Hard Real-Time Systems:** Missing a deadline is a catastrophic failure (e.g., missile guidance, flight control, medical life support systems).
    - **Soft Real-Time Systems:** Missing a deadline is undesirable but not catastrophic; performance degrades (e.g., multimedia streaming, virtual reality).
  + **Pros:** Guarantees timely responses, highly reliable for critical applications.
  + **Cons:** Less flexible, often specialized hardware, difficult to develop and debug.
  + **Use Cases:** Industrial control systems, automotive systems (ABS, airbag), robotics, scientific experiments, medical imaging.
* **Mobile Operating Systems:**
  + **Characteristics:** Optimized for portable devices like smartphones and tablets. Focus on touch interfaces, power efficiency, app ecosystems, and connectivity.
  + **Pros:** Highly user-friendly, optimized for small form factors, extensive app stores.
  + **Cons:** Resource constraints (battery, CPU), security challenges due to open app ecosystems, fragmentation (especially Android).
  + **Use Cases:** Smartphones (Android, iOS), Tablets.
* **Embedded Operating Systems:**
  + **Characteristics:** Designed for specific, non-general-purpose devices. Often very small, resource-constrained, and optimized for a particular function.
  + **Pros:** Highly efficient for their specific task, low cost, small footprint.
  + **Cons:** Limited functionality, often no user interface or a very simple one, difficult to update.
  + **Use Cases:** Washing machines, smart TVs, digital cameras, GPS devices, car infotainment systems.

**4. OS Services and System Calls**

The OS provides various services to both users and programs to make the computer system convenient to use and efficient.

**OS Services:**

These are the functions that the OS performs for users and applications.

* **User Interface (UI):** As discussed, provides CLI or GUI.
* **Program Execution:** Loads a program into memory and runs it. Ends execution, either normally or abnormally (error).
* **I/O Operations:** Allows user programs to perform input/output operations. Users cannot directly access I/O hardware; they must request the OS.
* **File-System Manipulation:** Programs need to read, write, create, and delete files and directories. The OS manages these operations and permissions.
* **Communications:** Allows processes to exchange information, either on the same computer or between computers over a network. This can be through shared memory or message passing.
* **Error Detection:** Ensures correct computing by detecting errors in the CPU, memory, I/O devices, or user programs. It takes appropriate action to ensure consistency and correctness.
* **Resource Allocation:** Allocates resources (CPU cycles, memory, file storage, I/O devices) to multiple users or multiple jobs running concurrently.
* **Accounting:** Keeps track of which users use how much and what kind of computer resources. This is useful for resource quotas or billing.
* **Protection and Security:** Protects system resources (CPU, memory, files, etc.) from unauthorized access. Security involves defending the system from external and internal attacks (viruses, worms, unauthorized access).

**System Calls:**

* A system call is a request made by a user-level program to the operating system's kernel to perform a privileged operation. User programs typically cannot directly access hardware or certain critical system resources for security and stability reasons. System calls provide a controlled interface for programs to request these services from the OS.
* **Why System Calls?** User programs run in "user mode," which has limited privileges. The OS kernel runs in "kernel mode" (or supervisor/privileged mode), which has full access to hardware. System calls provide a controlled way for user programs to transition into kernel mode and request privileged operations.
* **How it works (simplified):**
  1. A user program executes an instruction that generates a **software interrupt** (trap).
  2. This trap transfers control to a specific location in the operating system's kernel.
  3. The kernel identifies the requested system call by an index number or a specific parameter called system call number.
  4. The kernel executes the requested service (e.g., reads data from disk).
  5. Upon completion, the kernel returns control to the user program, often with a return code indicating success or failure.
* **Application Programming Interface (API):** Most system calls are not directly invoked by application programmers. Instead, they use a higher-level **Application Programming Interface (API)**, such as the Win32 API for Windows or the POSIX API for UNIX-like systems (Linux). The API functions then internally make the necessary system calls. This provides portability and abstraction.
* **Categories of System Calls:**
  1. **Process Control:** create, terminate, load, execute, wait, signal, allocate/free memory.
  2. **File Management:** open, read, write, close, create, delete, seek.
  3. **Device Management:** request device, release device, read, write, reposition.
  4. **Information Maintenance:** get time, set time, get process attributes, set process attributes.
  5. **Communication:** create communication channel, send message, receive message.
  6. **Protection:** set permissions.

**5. Operating System Structure (Monolithic, Layered, Microkernel, Modules, Hybrid)**

The internal organization of an OS significantly impacts its design, performance, and maintainability.

* **a. Monolithic Structure:**
  + **Concept:** The entire operating system kernel is implemented as a single, large, executable program. All services (process management, memory management, file system, device drivers) run in a single address space (kernel space) with full privileges.
  + **Characteristics:**
    - **Tight Coupling:** Components directly call functions within other components.
    - **High Performance:** Low overhead for inter-component communication as it's just function calls.
    - **Difficult to Debug/Maintain:** A bug in one part can crash the entire system. Hard to add new features or remove old ones without recompiling the whole kernel.
    - **Lack of Modularity:** Difficult to understand and modify.
  + **Examples:** Early UNIX, MS-DOS, Linux (though modern Linux has modularity features).
* **b. Layered Structure:**
  + **Concept:** The OS is divided into a series of layers, each built on top of the lower layers. Each layer only interacts with the layers immediately below and above it. Layer 0 is the hardware, and the highest layer is the user interface.
  + **Characteristics:**
    - **Modularity:** Clear separation of concerns, easier to debug (if a bug occurs, it's likely in the current or lower layer), and easier to maintain.
    - **Abstraction:** Each layer provides services to the layer above it, hiding the complexities of the lower layers.
    - **Reduced Performance:** Each request might have to pass through multiple layers, leading to increased overhead.
  + **Examples:** THE (Technische Hogeschool Eindhoven) OS, early Multics. (Pure layered systems are rare now due to performance overhead).
* **c. Microkernel Structure:**
  + **Concept:** Removes all non-essential components from the kernel and implements them as user-level services (servers). The microkernel itself only provides minimal essential services: inter-process communication (IPC), basic memory management, and low-level process scheduling.
  + **Characteristics:**
    - **High Reliability/Robustness:** If a user-level service (e.g., file system, device driver) crashes, it doesn't bring down the entire kernel.
    - **Extensibility:** Easy to add new services or remove old ones without modifying the microkernel.
    - **Security:** Services run in user mode with limited privileges.
    - **Lower Performance:** Increased overhead due to message passing (IPC) between the microkernel and user-level servers.
  + **Examples:** Mach (used as the basis for macOS's Darwin kernel), QNX, Minix.
* **d. Modules (Loadable Kernel Modules - LKM):**
  + **Concept:** A more practical approach that combines the benefits of monolithic kernels (performance) with the modularity of layered/microkernel designs. The core kernel is still monolithic, but many services (especially device drivers, file systems) can be loaded and unloaded dynamically at runtime as modules.
  + **Characteristics:**
    - **Flexibility:** Allows extending kernel functionality without recompiling the entire kernel.
    - **Performance:** Modules run in kernel space, providing direct access and good performance.
    - **Ease of Development:** Easier to develop and test individual kernel components.
  + **Examples:** Linux is a prime example of a modular monolithic kernel. Most modern OS use this approach.
* **e. Hybrid Structure:**
  + **Concept:** Combines elements from both monolithic and microkernel architectures to achieve a balance of performance and modularity/stability.
  + **Characteristics:** Often, a smaller, performance-critical part resides in the kernel space (like a monolithic kernel), while less critical services are run as user-mode processes (like a microkernel).
  + **Examples:** Windows NT family (NT, 2000, XP, Vista, 7, 8, 10, 11) is often cited as a hybrid kernel. macOS (Darwin) also uses a hybrid approach, with a Mach microkernel augmented by a BSD layer in kernel space.

**6. Virtual Machines**

A virtual machine (VM) is a software implementation of a computer system that executes programs like a physical machine. It provides an abstract layer between the hardware and the software, allowing multiple independent operating systems to run concurrently on a single physical machine.

* **Concept:** A VM essentially "tricks" an OS into thinking it's running on dedicated hardware, even though it's sharing resources with other VMs on the same physical machine.
* **Hypervisor (Virtual Machine Monitor - VMM):** This is the crucial software layer that creates and manages the VMs. It intercepts VM instructions and translates them to actual hardware operations, allocating resources and isolating VMs from each other.
  + **Type 1 Hypervisor (Bare-Metal):** Runs directly on the physical hardware, without a host OS. It's the OS itself in many ways, providing virtualization services (e.g., VMware ESXi, Microsoft Hyper-V, Xen). Offers better performance and security.
  + **Type 2 Hypervisor (Hosted):** Runs as an application on top of an existing host operating system (e.g., Oracle VirtualBox, VMware Workstation, Parallels Desktop). Easier to set up for personal use, but incurs overhead from the host OS.
* **Guest OS:** The operating system running inside the virtual machine. It could be Windows, Linux, macOS, etc., regardless of the host OS.
* **Benefits of Virtual Machines:**
  + **Resource Utilization:** Efficiently utilizes hardware by consolidating multiple workloads onto fewer physical machines.
  + **Isolation:** Each VM is isolated from others. A crash in one VM does not affect others.
  + **Portability:** VMs can be easily moved or copied between physical machines.
  + **Security:** Provides a sandboxed environment for running suspicious software or for development/testing.
  + **Development/Testing:** Allows developers to test software on various OS and configurations without needing multiple physical machines.
  + **Disaster Recovery:** Easier to back up and restore entire systems.
  + **Legacy Applications:** Can run old applications that require specific OS versions.

**7. System Boot Process**

The boot process is the sequence of operations that a computer performs when it is powered on until the operating system is fully loaded and ready for use. It's a critical, low-level process.

**General Steps (may vary slightly between architectures and OS):**

1. **Power On and Power-On Self-Test (POST):**
   * When you press the power button, power flows to the components.
   * The CPU starts executing firmware instructions from a special ROM chip (often called **BIOS - Basic Input/Output System** on older PCs, or **UEFI - Unified Extensible Firmware Interface** on modern PCs).
   * POST runs diagnostics to ensure essential hardware components (CPU, RAM, keyboard, video card, etc.) are working correctly. It might emit beeps if a critical component fails.
2. **BIOS/UEFI Initialization and Boot Device Selection:**
   * If POST is successful, the BIOS/UEFI initializes other hardware components and loads basic drivers.
   * It then consults the **boot order** settings (configured by the user in the BIOS/UEFI setup) to determine which storage device (hard drive, SSD, USB, network) to try booting from first.
3. **Loading the Bootloader (Stage 1):**
   * The BIOS/UEFI reads the first sector (typically 512 bytes) of the designated boot device. This sector usually contains the **Master Boot Record (MBR)** on older systems, or the **GPT (GUID Partition Table)** and **EFI System Partition (ESP)** on UEFI systems.
   * The MBR/GPT contains a small program called the **bootloader (Stage 1)**. This tiny program's primary job is to find and load the next stage of the bootloader.
4. **Loading the Operating System Kernel (Stage 2/Kernel Load):**
   * The Stage 1 bootloader (e.g., GRUB for Linux, Windows Boot Manager for Windows) is loaded into memory and executed.
   * This Stage 2 bootloader is more sophisticated. It understands file systems and knows where to find the **operating system kernel** on the hard drive.
   * It loads the compressed kernel image into RAM and then uncompresses it.
   * Before the kernel starts, it might also load an **initial RAM disk (initrd/initramfs)**, which is a temporary root file system used to mount the real root file system.
5. **Kernel Initialization:**
   * Once the kernel is in memory, it takes control.
   * The kernel initializes its core data structures, sets up memory management, loads essential device drivers, and starts internal processes.
   * It detects and configures hardware devices.
6. **Starting Init/Systemd Process:**
   * The kernel's final step in the boot process is to start the very first user-space process, traditionally called init (on older UNIX/Linux systems) or systemd (on modern Linux systems).
   * This init/systemd process is the "parent" of all other user-space processes. It's responsible for bringing up the rest of the system.
7. **User-Space Services and Login Prompt:**
   * init/systemd reads configuration files to determine which services to start (e.g., networking, logging, graphical display manager, shell).
   * Once all necessary services are running, the system presents a login prompt (either a text-based console or a graphical login screen).
   * At this point, the operating system is fully booted and ready for user interaction.

**Module 2: Process Management**!

**module 2: Process Management - Detailed Explanation**

This module delves into the concept of a "process," how the operating system manages it throughout its lifecycle, and the critical issues involved in scheduling and coordination.

**1. Process Concept: Definition, Process vs. Program**

At the heart of an operating system's function is the management of processes.

* **Program:** A program is a passive entity, residing on disk, containing a set of instructions that can be executed by a computer. It's essentially a file that is "code." Think of it as a blueprint or a recipe.
  + *Example:* The .exe file for Microsoft Word, a Python script, or a Java .jar file.
* **Process:** A process is an active entity. It is a program in execution. When you double-click an icon, type a command, or launch an application, you are creating a process. It's the "doing" of the program. Think of it as the actual building constructed from the blueprint, or the meal being cooked from the recipe.

A process is more than just the program code; it includes:

* + **Program Counter (PC):** Indicates the next instruction to be executed.
  + **Stack:** Contains temporary data (function parameters, return addresses, local variables).
  + **Data Section:** Contains global variables.
  + **Heap:** Dynamically allocated memory during runtime.
  + **Process State:** (Discussed next)
  + **CPU Registers:** Values of CPU registers when the process was last running.
  + **Open Files:** List of files the process currently has open.
  + **I/O Devices:** List of I/O devices currently allocated to the process.

**Why is this distinction important?**

* Multiple processes can be associated with the same program. For example, you can open multiple instances of a web browser; each instance is a separate process.
* The OS manages processes, not programs. A program needs to be loaded into memory and have resources allocated to become a process.

**2. Process State Model (New, Ready, Running, Waiting, Terminated)**

As a process executes, it changes state. These states represent the current activity of a process.

* **New:** The process is being created. It's in the initial stage, but not yet ready to be executed by the CPU. The OS is allocating resources (memory, PCB).
* **Ready:** The process has all the necessary resources and is waiting for the CPU to become available. It is ready to run as soon as the CPU scheduler selects it.
* **Running:** The process is currently executing instructions on the CPU. At any given instant, only one process per CPU core can be in the running state.
* **Waiting (or Blocked):** The process is waiting for some event to occur (e.g., completion of an I/O operation, receiving a signal, waiting for a resource to become available). While waiting, it cannot execute, even if the CPU is free.
* **Terminated:** The process has finished execution, either normally (completed its task) or abnormally (due to an error or explicit termination by the OS/user). The OS is in the process of deallocating its resources.

These first five states are the most commonly depicted in simpler 5-state models. The "7 states" model adds two more to handle situations where processes might be swapped out of main memory.

* **Suspended Ready (Ready/Suspended):**
  + **Description:** A process in the "Suspended Ready" state is essentially a "Ready" process that has been swapped out of main memory to secondary storage (disk). It is ready to execute but requires being loaded back into RAM before it can be put in the "Ready" queue.
  + **Activities:** The OS might suspend processes to free up physical memory for other processes, especially in systems with limited RAM or when a higher-priority process needs to run.
  + **Transition:**
    - From "Ready" to "Suspended Ready": When the OS decides to swap a ready process out to disk.
    - From "Suspended Blocked" to "Suspended Ready": When the event a suspended blocked process was waiting for occurs, but it's still on disk.
    - From "Suspended Ready" to "Ready": When the OS decides to load it back into main memory.
* **Suspended Blocked (Blocked/Suspended):**
  + **Description:** A process in the "Suspended Blocked" state is a "Blocked/Waiting" process that has been swapped out of main memory to secondary storage (disk). It is waiting for an event *and* is not in RAM.
  + **Activities:** This state is used to free up memory when a blocked process is consuming significant RAM and isn't immediately needed. Even if its awaited event occurs, it won't be able to run until it's swapped back in.
* **Transition:**
  + From "Blocked/Waiting" to "Suspended Blocked": When the OS decides to swap a blocked process out to disk.
  + From "Suspended Blocked" to "Suspended Ready": When the event it was waiting for occurs, but it remains swapped out.
  + From "Suspended Blocked" to "Blocked/Waiting": (Less common, usually goes to Suspended Ready first) If it's loaded back into memory *before* its event occurs.
* **Why the "Suspended" states?**
  + The "Suspended" states (also known as "Swapped Out" states) are crucial for memory management in operating systems, particularly in older systems or environments with tight memory constraints. They allow the OS to manage the degree of multiprogramming by temporarily moving processes from RAM to disk, making room for other processes that need to run immediately.

**Transitions Between States:**

* **New -> Ready:** The OS admits the process into the set of ready processes.
* **Ready -> Running:** The CPU scheduler selects the process for execution (dispatch).
* **Running -> Ready:**
  + **Time Slice Expired (Timer Interrupt):** In time-sharing systems, the process's allocated CPU time slice runs out.
  + **Preemption:** A higher-priority process becomes ready.
* **Running -> Waiting:** The process requests an I/O operation or waits for some event.
* **Waiting -> Ready:** The event for which the process was waiting (e.g., I/O completion) occurs.
* **Running -> Terminated:** The process finishes its execution.
* **Ready/Waiting -> Terminated:** The process is explicitly killed by the OS or another process (e.g., kill command).
* **New (Created):** **In Secondary storage, but sometimes at RAM** (at least partially loaded, PCB allocated).
* **Ready (Waiting for CPU):** **In RAM**.
* **Running:** **In RAM**.
* **Blocked / Waiting (Waiting for I/O or Event):** **In RAM**.
* **Terminated (Exit / Done):** **Partially in RAM** (PCB might remain for a short time) or **Not in RAM** (resources deallocated).
* **Suspended Ready (Ready/Suspended):** **Not in RAM** (swapped out to disk).
* **Suspended Blocked (Blocked/Suspended):** **Not in RAM** (swapped out to disk).

Sources

**3. Process Control Block (PCB)**

The Process Control Block (PCB), also known as a Task Control Block (TCB), is a data structure maintained by the operating system for each process. It contains all the information needed by the OS to manage and control that specific process. It's essentially the process's "identity card" or "passport."

* **Why is it needed?** When the OS switches from one process to another (context switch), it needs to save the state of the current process and load the state of the new process. The PCB is where this state information is stored.
* **Information Stored in a PCB:**
  + **Process State:** Current state of the process (New, Ready, Running, Waiting, Terminated).
  + **Program Counter (PC):** The address of the next instruction to be executed for this process.
  + **CPU Registers:** All general-purpose registers, stack pointers, index registers, etc., whose contents must be saved when the process is swapped out of the CPU.
  + **CPU Scheduling Information:** Process priority, pointers to scheduling queues (e.g., ready queue, waiting queues).
  + **Memory-Management Information:** Base and limit registers, page tables, segment tables (depending on the memory management scheme). This tells the OS where the process's code and data are located in memory.
  + **Accounting Information:** CPU usage, real time used, time limits, account numbers, job or process numbers.
  + **I/O Status Information:** List of I/O devices allocated to the process, list of open files.
  + **Process ID (PID):** A unique identifier for the process.
  + **Parent Process ID (PPID):** The ID of the process that created this process.
  + **Child Process IDs:** IDs of processes created by this process.
  + **Pointers:** Pointers to other PCBs, used for linking processes in various queues (e.g., ready queue).
* **Location:** PCBs are typically stored in a protected area of memory (kernel space) so that user-level processes cannot tamper with them.

**4. Process Scheduling**

Process scheduling is the activity of the operating system that selects a process from the ready queue to be executed by the CPU. Its goal is to maximize CPU utilization and provide an acceptable response time for users.

* **a. Scheduling Queues (Job, Ready, Device)**

Processes migrate between various queues as they change state.

* + **Job Queue (or Hold Queue):**
    - **Purpose:** Contains all processes in the system. When processes are first submitted, they are placed here.
    - **State:** Mostly in the 'New' state, waiting for admission into main memory.
    - **Managed by:** Long-term scheduler.
    - **Characteristics:** Typically a batch processing concept, where jobs wait for system resources to become available.
  + **Ready Queue:**
    - **Purpose:** Contains processes that are in main memory, are ready to execute, and are waiting for CPU allocation.
    - **State:** 'Ready' state.
    - **Managed by:** Short-term scheduler. Typically implemented as a linked list of PCBs.
    - **Characteristics:** This is the most crucial queue for CPU scheduling. Processes are added here when they are newly created, preempted from the CPU, or complete an I/O operation. Different scheduling algorithms (e.g., First-Come, First-Served, Round Robin, Priority, Shortest Job First) determine the order in which processes are selected from this queue.
  + **Device Queues (or I/O Queues):**
    - **Purpose:** Contains processes waiting for a particular I/O device to become available or for an I/O operation to complete. Each device typically has its own queue.
    - **State:** 'Waiting' or 'Blocked' state.
    - **Managed by:** Long-term or medium-term schedulers, and device-specific handlers.
    - **Characteristics:** Processes in these queues are not consuming CPU time. When the I/O operation completes or the event occurs, the process is moved from the Device Queue back to the Ready Queue.
  + **Suspended Queues (or Swapped Queues):**
* **Purpose:** These queues hold processes that have been swapped out of main memory to secondary storage (disk) to free up RAM. They correspond to the "Suspended Ready" and "Suspended Blocked" states.
* **Contents:** Processes in the "Suspended Ready" or "Suspended Blocked" states.
* **Managed by:** The **Medium-Term Scheduler (or Swapper)**. This scheduler's role is to manage the swapping of processes between main memory and secondary storage.
* **Characteristics:** Processes in these queues are temporarily out of contention for the CPU until they are swapped back into main memory.
* **b. Schedulers (Long-term, Short-term, Medium-term)**

Schedulers are OS components responsible for selecting processes for execution or for moving them between different states.

* + **Long-Term Scheduler (Job Scheduler):**
    - **Role:** Selects processes from the job queue and loads them into the ready queue (main memory) for execution. It controls the **degree of multiprogramming** (the number of processes simultaneously in memory). Maintains a balanced mix of CPU-bound and I/O-bound processes to optimize resource usage.
    - **Frequency:** Infrequent (minutes or hours).
    - **Impact:** If too many processes are admitted, memory becomes saturated, leading to thrashing. If too few, CPU may be underutilized.
    - **Goal:** To provide a balanced mix of CPU-bound and I/O-bound processes to maximize efficiency.
  + **Short-Term Scheduler (CPU Scheduler):**
    - **Role:** Selects one of the processes from the ready queue and allocates the CPU to it. This is the most frequent scheduler.
    - **Frequency:** Very frequent (milliseconds). Every time a process completes, blocks, or its time slice expires. Directly impacts system responsiveness and process turnaround times.
    - **Impact:** Determines which process gets the CPU next, directly impacting system performance (response time, throughput).
    - **Goal:** To maximize CPU utilization and minimize response time.
  + **Medium-Term Scheduler (Swapper):**
    - **Role:** Swaps processes out of memory (to disk) and later swaps them back into memory. This is part of a swapping scheme. Handles the swapping of processes in and out of main memory (suspending and resuming).
    - Responsible for moving processes from the suspended state back to the ready state as resources become available.
    - **Frequency:** Moderate.
    - **Purpose:** Reduces the degree of multiprogramming by removing a process from memory, typically to free up memory for other processes or to manage thrashing. When memory becomes available, it can swap the process back in.
    - **Impact:** Crucial for managing memory and preventing thrashing in systems with virtual memory.
    - **State Change:** Ready/Waiting (in memory) <-> Suspended-Ready/Suspended-Waiting (on disk)

**Scheduling Queues**

Scheduling queues are data structures within the OS that track the status and location of every process as it moves through its lifecycle. They enable the efficient handling of large numbers of processes by grouping similar states together.

**Main Types of Queues**

* **Job Queue:**  
  Contains all processes in the system, including those waiting to be admitted to main memory.
* **Ready Queue:**  
  Holds processes currently in main memory, ready and waiting to be assigned to the CPU. This queue plays a central role in short-term scheduling.
* **Device/I/O Queues:**  
  Each type of I/O device has its own queue where processes wait for access to that device (e.g., printer queue, disk queue).

**Process Flow Between Queues**

* When a new process is created, it enters the **job queue**.
* If admitted to memory by the long-term scheduler, it moves to the **ready queue**.
* When the short-term scheduler selects it, the process moves to the **running** state.
* If the process requests I/O, it is moved to the corresponding **device queue**.
* Upon I/O completion, it returns to the **ready queue**.
* This cycle continues until the process finishes execution and exits the system.

**Visual Summary**

| **Scheduler** | **Primary Role** | **Frequency** | **Associated Queue** |
| --- | --- | --- | --- |
| Long-Term | Admit jobs from disk to memory | Infrequent | Job Queue |
| Medium-Term | Swap processes in/out of memory | Moderate | Suspended/Ready |
| Short-Term | Dispatch ready process to CPU | Very Frequent | Ready Queue |

**Key Points**

* Schedulers coordinate system multitasking and efficient resource usage.
* Scheduling queues organize processes by state, aiding management and prioritization.
* The interplay between schedulers and queues ensures processes move smoothly through creation, execution, waiting, and completion.

This structured approach is what enables modern systems to seamlessly handle many processes and users at once.

* **c. Context Switching**

Context switching is the mechanism by which the CPU switches from executing one process to executing another. It's a fundamental operation in multiprogramming and time-sharing systems.

* + **What happens:**
    1. **Save State:** The OS saves the current state of the currently running process (its CPU registers, program counter, and other relevant information) into its Process Control Block (PCB).
    2. **Load State:** The OS loads the saved state (from its PCB) of the new process (the one selected by the short-term scheduler) into the CPU registers.
    3. **Transfer Control:** The CPU then begins executing instructions from the new process at the point where it was last interrupted.
  + **When does it occur?**
    - **Time Slice Expiration:** The timer interrupt signals that the current process's allocated time is up.
    - **System Call:** A process makes a system call (e.g., to perform I/O), causing it to enter the waiting state.
    - **Interrupts:** Hardware interrupts (e.g., I/O completion).
    - **Higher Priority Process Ready:** In preemptive scheduling, if a higher-priority process becomes ready.
  + **Overhead:** Context switching is pure overhead. The system performs no useful work while performing a context switch. Its duration depends on hardware support (e.g., number of registers, special instructions for saving/restoring context) and memory speed. Minimizing context switch time is a design goal for OS.

**5. Operations on Processes: Creation, Termination, Suspension, Resumption**

The OS provides mechanisms for manipulating processes.

* **a. Process Creation (fork() and exec()):**
  + **Parent-Child Relationship:** A process (the parent) can create new processes (children). The creating process is the parent, and the new process is the child. This forms a process tree.
  + **Resource Sharing:** Children may share resources (memory, files) with their parent, or they may receive a subset, or none at all.
  + **Execution Options:**
    - **Parent Continues, Child Continues:** Parent and child execute concurrently.
    - **Parent Waits, Child Continues:** Parent waits for the child to terminate before resuming.
  + **Address Space Options:**
    - **Duplicate:** Child process is a duplicate of the parent (e.g., fork() in Unix-like systems).
    - **New Program:** Child process loads a new program into its address space (e.g., exec() in Unix-like systems).
  + **Unix/Linux Example (fork() and exec()):**
    - fork(): Creates a new process that is an *exact copy* of the calling process (the parent). Both processes continue execution from the point after fork(). The fork() call returns 0 to the child process and the child's PID to the parent process.
    - exec(): Replaces the current process's address space with a new program. The process ID (PID) remains the same, but the code, data, and stack are entirely replaced. exec() is often called immediately after fork() by the child to run a different program.
* **b. Process Termination:**
  + **Normal Termination (Exit):** A process executes its last statement and asks the operating system to delete it (e.g., exit() system call). Resources are deallocated by the OS.
  + **Abnormal Termination (Abort):**
    - **Parent Termination:** A parent process can terminate its children (e.g., kill command or abort() system call). Reasons might include:
      * Child exceeding allocated resources.
      * Task assigned to child is no longer needed.
      * Parent exiting, and the OS doesn't allow child processes to continue without a parent (cascading termination).
    - **Error/Fault:** A process attempts to perform an illegal operation (e.g., division by zero, invalid memory access), leading to termination by the OS.
    - **External Signal:** Termination due to an external signal (e.g., SIGKILL on Linux).
* **c. Process Suspension and Resumption:**
  + **Suspension:** A process is temporarily removed from active contention for the CPU and main memory, typically moved to secondary storage (swapped out). It's distinct from being in the "waiting" state, as a waiting process is still in main memory.
    - **Reason:** Usually for memory management (medium-term scheduler), debugging, or a user explicitly suspending a process.
    - **State:** Ready -> Suspended-Ready, Waiting -> Suspended-Waiting.
  + **Resumption:** A suspended process is brought back into main memory and made eligible for execution again.
    - **Reason:** Memory becomes available, or the user/OS explicitly resumes it.
    - **State:** Suspended-Ready -> Ready, Suspended-Waiting -> Waiting.

**6. Inter-Process Communication (IPC)**

Processes often need to communicate and synchronize with each other to share data, coordinate activities, or exchange information. IPC mechanisms allow this.

* **a. Shared Memory:**
  + **Concept:** A region of memory is created that is shared by multiple processes. Each process can then directly read from and write to this shared region.
  + **Mechanism:** The OS establishes the shared memory region, but once established, processes can access it directly without further OS intervention (after the initial setup).
  + **Pros:**
    - **Fast:** Once set up, data transfer is at memory speeds, very efficient as no kernel intervention is required for each read/write.
    - **Simple for Data Exchange:** Direct access to data.
  + **Cons:**
    - **Synchronization Issues:** Requires explicit synchronization mechanisms (e.g., semaphores, mutexes) to prevent race conditions and ensure data consistency. This is the responsibility of the programmers.
    - **Not for Distributed Systems:** Limited to processes on the same machine.
  + **Use Cases:** High-performance data sharing, database systems, large data transfers.
* **b. Message Passing (Pipes, Message Queues)**
  + **Concept:** Processes communicate by exchanging messages. The OS provides send() and receive() primitives.
  + **Mechanism:** The OS manages a communication channel (e.g., a pipe or queue) and copies messages between the address spaces of the communicating processes.
  + **Pros:**
    - **Simpler for Synchronization:** Message passing inherently provides some synchronization (e.g., receive might block until a message arrives).
    - **Easier for Distributed Systems:** Can be extended to network communication (e.g., sockets).
    - **Less Error-Prone:** OS handles the copying, reducing direct memory manipulation errors.
  + **Cons:**
    - **Slower:** Involves more overhead (context switches, data copying) compared to shared memory, as the kernel is involved in every message transfer.
    - **Message Size Limits:** Messages often have size limits.
  + **Specific Examples:**
    - **Pipes:**
      * **Concept:** A unidirectional (or sometimes bidirectional) communication channel that allows one process to write to one end and another process to read from the other.
      * **Types:**
        + **Unnamed Pipes (Anonymous Pipes):** Used for communication between related processes (e.g., parent-child) on the same machine. They are created with pipe() system call and exist only as long as the processes using them are alive. (ls | grep .txt)
        + **Named Pipes (FIFOs):** Have a name in the filesystem and can be used for communication between unrelated processes on the same machine. They persist as long as the system is running or until explicitly deleted.
    - **Message Queues:**
      * **Concept:** A linked list of messages stored within the kernel. Processes can add messages to a queue (producer) or retrieve messages from it (consumer). Messages can be prioritized or selected by type.
      * **Pros:** More flexible than pipes (can send different message types, non-FIFO ordering possible, multiple readers/writers).
      * **Cons:** Still involves kernel overhead.
* **c. Synchronization Mechanisms (Semaphores, Mutexes, Monitors) - *Introduction***

While IPC allows processes to exchange data, **synchronization** is about coordinating their execution to ensure data consistency and avoid race conditions when shared resources are accessed. This is a critical concept, and we'll dive much deeper into it in Module 4. Here's a brief introduction:

* + **Race Condition:** Occurs when multiple processes or threads access and manipulate the same shared data concurrently, and the outcome of the execution depends on the particular order in which the access takes place. This leads to unpredictable and incorrect results.
  + **Critical Section:** A segment of code where shared resources (data structures, files, I/O devices) are accessed. Only one process/thread should be allowed to execute its critical section at a time.
  + **Synchronization Tools:** Mechanisms to ensure that critical sections are mutually exclusive, preventing race conditions.
    - **Mutex (Mutual Exclusion Lock):** A simple binary lock. Only one process can acquire the mutex at a time. A process must acquire the mutex before entering a critical section and release it upon exiting. If a process tries to acquire a mutex that is already held, it blocks until the mutex is released.
    - **Semaphore:** A more generalized synchronization tool. It's an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() (or P()) and signal() (or V()).
      * **Binary Semaphore (similar to Mutex):** Value is 0 or 1.
      * **Counting Semaphore:** Value can be any non-negative integer. Used to control access to a resource that has multiple instances.
    - **Monitor:** A higher-level language construct that provides mutual exclusion and synchronization. It encapsulates shared data and the procedures that operate on that data. Only one process can be active within a monitor at any given time. Monitors often use condition variables for more complex waiting/signaling.

**CPU Scheduling**

**1. Basic Concepts: CPU-I/O Burst Cycle, CPU Scheduler, Preemptive vs. Non-preemptive**

To understand CPU scheduling, we need to grasp a few foundational concepts:

* **CPU-I/O Burst Cycle:**
  + Processes typically alternate between two phases:
    - **CPU Burst:** A period of time when the process is executing instructions on the CPU (performing computations).
    - **I/O Burst:** A period of time when the process is waiting for an I/O operation to complete (e.g., reading from disk, waiting for user input, sending data over a network).
  + Most processes exhibit this cycle. A process begins with a CPU burst, followed by an I/O burst, then another CPU burst, and so on. The last CPU burst ends with a system call to terminate execution.
  + **CPU-bound processes:** Have very long CPU bursts and infrequent I/O bursts (e.g., scientific computations, heavy data processing).
  + **I/O-bound processes:** Have many short CPU bursts and frequent I/O bursts (e.g., interactive applications, web servers).
  + **Significance:** The nature of these bursts heavily influences which scheduling algorithm performs best. An algorithm designed for CPU-bound tasks might perform poorly for I/O-bound tasks and vice-versa.
* **CPU Scheduler (Short-Term Scheduler):**
  + As discussed in Module 2, this is the component of the OS that selects a process from the Ready queue and allocates the CPU to it.
  + It's invoked frequently (on the order of milliseconds), so its decision-making process must be very fast.
* **Preemptive vs. Non-preemptive Scheduling:** This defines when a currently running process can lose control of the CPU.
  + **Non-preemptive (or Cooperative) Scheduling:**
    - **Concept:** Once a process is given the CPU, it keeps the CPU until it either:
      1. Terminates.
      2. Switches to the Waiting state (e.g., for an I/O request).
      3. **Characteristics:** Simple to implement, as no complex interruption logic is needed. No need for special kernel data structures to handle preemption.
      4. **Drawbacks:** A long-running CPU-bound process can monopolize the CPU, leading to poor response times for other processes (especially interactive ones). Not suitable for time-sharing systems where responsiveness is key.
      5. **Example:** Early versions of Windows (Windows 3.x), MS-DOS.
  + **Preemptive Scheduling:**
    - **Concept:** The CPU can be taken away from a running process even if it hasn't completed its CPU burst or hasn't voluntarily given up the CPU. This happens due to:
      1. **Time Slice Expiration:** A timer interrupt occurs, indicating the process's allocated time is up.
      2. **Higher Priority Process Ready:** A new process enters the Ready state and has a higher priority than the currently running one.
      3. **I/O Completion:** An I/O operation completes, unblocking a process that now has higher priority.
    - **Characteristics:** More complex to implement, as it requires careful handling of shared data (critical sections) within the kernel to avoid race conditions during preemption. Leads to frequent context switches.
    - **Benefits:** Essential for time-sharing and real-time systems. Provides good responsiveness, ensures fairness, and prevents a single process from hogging the CPU.
    - **Example:** All modern operating systems (Linux, Windows, macOS).

**2. Scheduling Criteria (CPU Utilization, Throughput, Turnaround Time, Waiting Time, Response Time)**

When evaluating the effectiveness of a CPU scheduling algorithm, several criteria are considered. Often, these criteria are conflicting, meaning an algorithm optimizing for one might perform poorly on another.

* **CPU Utilization:**
  + **Definition:** The percentage of time the CPU is busy executing processes.
  + **Goal:** Maximize CPU utilization (keep the CPU as busy as possible). In a real system, CPU utilization typically ranges from 40% (lightly loaded) to 90% (heavily loaded).
* **Throughput:**
  + **Definition:** The number of processes completed per unit of time.
  + **Goal:** Maximize throughput (finish as many jobs as possible in a given period). For long-running processes, this might be measured in jobs per hour; for short transactions, transactions per second.
* **Turnaround Time:**
  + **Definition:** The total time from the submission of a process until its completion. This includes the time spent waiting in the ready queue, executing on the CPU, and performing I/O.
  + **Goal:** Minimize turnaround time (get jobs done quickly).
  + *Calculation:* Completion Time - Arrival Time
* **Waiting Time:**
  + **Definition:** The total amount of time a process spends waiting in the ready queue. It does *not* include time spent doing I/O or executing.
  + **Goal:** Minimize waiting time (processes should spend as little time as possible idle in the queue).
  + *Calculation:* Turnaround Time - Burst Time (assuming single CPU burst)
* **Response Time:**
  + **Definition:** The time from when a request is submitted until the *first response is produced*. This is particularly relevant for interactive systems. It's the time until a process *starts* responding, not when it finishes.
  + **Goal:** Minimize response time (provide quick initial feedback to the user). For interactive systems, this is often perceived as the most important metric.
  + *Calculation:* Time of first response - Arrival Time

**Optimization Goals:**

* **Maximize:** CPU utilization, Throughput.
* **Minimize:** Turnaround time, Waiting time, Response time.

**3. Scheduling Algorithms**

These are the core strategies the CPU scheduler uses to decide which process to run next. We'll illustrate with examples.

* **Assumptions for examples:**
  + All processes arrive at time 0 unless specified.
  + CPU burst times are given in milliseconds.
  + Lower numbers indicate higher priority where applicable.
* **a. First-Come, First-Served (FCFS)**
  + **Concept:** The process that requests the CPU first gets the CPU first. It's a non-preemptive algorithm.
  + **Mechanism:** Processes are executed in the order they arrive in the ready queue, similar to a FIFO queue.
  + **Pros:** Simple to understand and implement.
  + **Cons:**
    - **Convoy Effect:** A long CPU-bound process at the front of the queue can make all subsequent processes (especially I/O-bound ones) wait, leading to low CPU utilization and poor response times.
    - Average waiting time can be high.
* **b. Shortest-Job-First (SJF)**
  + **Concept:** The process with the smallest *next CPU burst* is executed next. This can be either preemptive or non-preemptive.
  + **Mechanism:** The scheduler needs to know or estimate the length of the next CPU burst. This is the main challenge with SJF in practice.
  + **Optimality:** SJF gives the minimum average waiting time for a given set of processes.
  + **Types:**
    - **Non-preemptive SJF:** Once a process starts, it runs to completion, even if a shorter job arrives later.
    - **Preemptive SJF (also called Shortest-Remaining-Time-First - SRTF):** If a new process arrives with a shorter remaining CPU burst than the currently executing process, the current process is preempted.
  + **Pros:** Optimal in terms of average waiting time and average turnaround time.
  + **Cons:**
    - **Starvation:** Long processes may never get to run if there's a continuous stream of short processes.
    - **Predicting Next Burst:** The main practical difficulty is accurately predicting the length of the next CPU burst. This is usually done by exponential averaging of past burst times.
* **c. Priority Scheduling**
  + **Concept:** Each process is assigned a priority, and the CPU is allocated to the process with the highest priority. Equal priority processes are typically handled by FCFS or Round Robin.
  + **Mechanism:** Can be preemptive (if a higher priority process arrives, the current one is preempted) or non-preemptive.
  + **Pros:** Can be very useful for real-time systems or to give preference to important tasks.
  + **Cons:**
    - **Starvation (Indefinite Blocking):** Low-priority processes may never execute if there's a continuous stream of higher-priority processes.
    - **Solution to Starvation: Aging:** Gradually increase the priority of processes that have been waiting for a long time.
  + **Priority Assignment:** Can be internal (e.g., based on time limits, memory requirements, I/O-to-CPU burst ratio) or external (e.g., based on importance, funding).
* **d. Round Robin (RR)**
  + **Concept:** Designed for time-sharing systems. Each process gets a small unit of CPU time, called a **time quantum** (or time slice), typically 10 to 100 milliseconds. When the quantum expires, the process is preempted and added to the end of the ready queue.
  + **Mechanism:** The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time quantum.
  + **Pros:**
    - **Fairness:** Provides fair allocation of CPU time among processes.
    - **Good Response Time:** Excellent for interactive systems as all processes get a chance to run relatively quickly.
    - No starvation.
  + **Cons:**
    - **High Context Switching Overhead:** If the time quantum is too small, excessive context switching reduces efficiency.
    - **Poor Throughput:** If the time quantum is too large, it degenerates to FCFS, and response time suffers.
    - Performance heavily depends on the choice of time quantum.
  + **Rule of Thumb for Quantum:** 80% of CPU bursts should be shorter than the quantum.
* **e. Multilevel Queue Scheduling**
  + **Concept:** Divides the ready queue into multiple separate queues, often based on process characteristics (e.g., foreground/interactive processes, background/batch processes). Each queue has its own scheduling algorithm.
  + **Mechanism:**
    - **Process Assignment:** Processes are permanently assigned to a queue based on their type.
    - **Inter-queue Scheduling:** There must be scheduling *between* the queues. This is typically done using fixed-priority preemptive scheduling (e.g., foreground queue always has priority over background queue).
    - **Example:**
      * **System Processes:** Highest priority (e.g., FCFS)
      * **Interactive Processes:** Next priority (e.g., Round Robin)
      * **Batch Processes:** Lowest priority (e.g., FCFS or SJF)
  + **Pros:** Good for systems with diverse process types, allows for specialized scheduling for different needs.
  + **Cons:**
    - **Starvation:** Lower-priority queues can suffer starvation if higher-priority queues are always busy.
    - Fixed assignment of processes can be rigid.
* **f. Multilevel Feedback Queue Scheduling**
  + **Concept:** A more flexible version of multilevel queue scheduling. Processes can move between queues based on their behavior (e.g., if a process uses too much CPU, it moves to a lower-priority queue; if it waits for I/O, it might move to a higher-priority queue).
  + **Mechanism:**
    - **Multiple Queues:** Typically, a set of queues with different priorities and/or time quantum sizes.
    - **Dynamic Priority:** Processes are assigned different priorities (and moved between queues) based on their CPU burst characteristics.
    - **Aging:** Used to prevent starvation by gradually moving processes that have been waiting too long to higher-priority queues.
  + **Pros:**
    - **Adaptive:** Adapts to process behavior, favoring I/O-bound and interactive processes while still allowing CPU-bound processes to run.
    - **Prevents Starvation:** Aging helps.
    - Most general purpose scheduling algorithm.
  + **Cons:**
    - **Complex to Implement:** Requires careful tuning of parameters (number of queues, scheduling algorithm for each queue, criteria for moving processes between queues, aging mechanism).
  + **Example:**
    - Queue 0 (Highest Priority): RR with quantum 8 ms
    - Queue 1: RR with quantum 16 ms
    - Queue 2 (Lowest Priority): FCFS
    - A new process enters Queue 0. If it doesn't complete within 8 ms, it's moved to Queue 1. If it doesn't complete within 16 ms in Queue 1, it's moved to Queue 2.

**4. Algorithm Evaluation (Gantt Charts, Average Times)**

After defining various scheduling algorithms, it's essential to evaluate and compare their performance.

* **Gantt Charts:**
  + **Purpose:** A visual representation of the schedule. It shows which process is running on the CPU at which time intervals.
  + **How to use:** Helps in tracing the execution flow and calculating completion times, start times, and waiting times for each process.
  + *Example:* All the Gantt charts provided for the algorithms above.
* **Average Times (Waiting Time, Turnaround Time, Response Time):**
  + **Purpose:** Quantifiable metrics for comparing scheduling algorithms.
  + **Calculation:** For each process, calculate its individual waiting time, turnaround time, and response time. Then sum these values and divide by the total number of processes to get the average.
  + **Steps to calculate for a process P\_i:**
    1. **Completion Time (C\_i):** The time at which P\_i finishes execution.
    2. **Turnaround Time (T\_i):** C\_i - Arrival Time (A\_i)
    3. **Burst Time (B\_i):** The total CPU time required for P\_i.
    4. **Waiting Time (W\_i):** T\_i - B\_i OR (Time P\_i started - Arrival Time) + sum of times P\_i was preempted and waited again.
    5. **Response Time (R\_i):** Time P\_i first started executing - Arrival Time (A\_i)
  + **Significance:** While average values are useful, it's also important to consider the *variance* of these times. An algorithm with a low average waiting time but high variance might lead to some processes waiting for an extremely long time, which could be unacceptable.
* **Other Evaluation Methods:**
  + **Deterministic Modeling:** Analyze the performance for a given fixed set of processes and their burst times. (This is what we did with the Gantt charts). It's simple but only applies to specific scenarios.
  + **Queuing Models:** Use mathematical formulas (from queuing theory) to model process arrival and CPU burst distributions. This allows for analytical calculation of performance metrics without running simulations. Useful for general analysis, but requires assumptions about distributions.
  + **Simulations:** Run a computer program that simulates the behavior of the scheduling algorithm given a stream of processes (generated randomly or from actual system traces). Provides more realistic results than deterministic modeling and can handle complex algorithms, but can be time-consuming.
  + **Implementation and Measurement:** Implement the algorithm in a real OS and measure its performance. The most accurate but most expensive and time-consuming method.

**Process Synchronization**

**1. Background: Race Condition, Critical Section Problem**

Before diving into solutions, it's essential to understand the problems that synchronization aims to solve.

* **Race Condition:**
  + **Definition:** A situation where multiple processes or threads access and manipulate the same shared data concurrently, and the outcome of the execution depends on the particular order in which the access takes place. The final value of the shared data depends on which process finishes last.
  + **Example:** Consider a shared counter variable, initially 0.
    - Process A wants to increment it (counter++).
    - Process B wants to increment it (counter++).
    - counter++ typically involves three machine instructions:
      1. register = counter (Load counter's value into a CPU register)
      2. register = register + 1 (Increment the value in the register)
      3. counter = register (Store the new value back to counter)

If the execution interleaves as follows:

* + 1. Process A: registerA = counter (counter = 0, registerA = 0)
    2. Process B: registerB = counter (counter = 0, registerB = 0)
    3. Process A: registerA = registerA + 1 (registerA = 1)
    4. Process B: registerB = registerB + 1 (registerB = 1)
    5. Process A: counter = registerA (counter = 1)
    6. Process B: counter = registerB (counter = 1)

Expected result: counter = 2. Actual result: counter = 1. This is a race condition.

* **Critical Section Problem:**
  + **Definition:** A segment of code in a process where shared resources (data structures, files, I/O devices) are accessed. The critical section problem is to design a protocol that ensures that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
  + **Goal:** Ensure **mutual exclusion** for critical sections.
  + **Requirements for a solution to the Critical Section Problem:**
    1. **Mutual Exclusion:** If process P1 is executing in its critical section, then no other process P2 can be executing in its critical section. i.e At most one process should be inside the critical section at any time.
    2. **Progress:** If no process is executing in its critical section and some processes want to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and thi1s selection cannot be postponed indefinitely. (i.e., if no one is in the critical section, and someone wants to enter, they shouldn't be blocked forever). In simple terms If no process is in the critical section, then **only those processes that want to enter** should decide **who gets in next**—and **the decision shouldn’t be postponed forever.**
    3. **Bounded Waiting:** There exists a limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. (i.e., a process shouldn't have to wait indefinitely). In simpler terms, There must be a **limit** on the number of times **other processes** are allowed to enter the critical section **before a waiting process is allowed**.

**2. Software Solutions (e.g., Peterson's Solution)**

These are algorithmic solutions implemented purely in software, without special hardware instructions.

* **Peterson's Solution:**
  + **Purpose:** A classic software-based solution to the critical section problem for *two processes* (P0 and P1). It satisfies all three requirements: mutual exclusion, progress, and bounded waiting.
  + **Variables:**
    - boolean flag[2]; // flag[i] is true if Pi wants to enter its critical section.
    - int turn; // Indicates whose turn it is to enter the critical section.
  + **Explanation:**
    - When Pi wants to enter, it sets its flag[i] to TRUE and then sets turn to j (giving Pj a chance).
    - The while loop is the waiting condition: Pi waits if Pj also wants to enter (flag[j] == TRUE) AND it's Pj's turn (turn == j).
    - If Pj is not interested (flag[j] == FALSE), Pi enters.
    - If Pj is interested but it's Pi's turn (turn == i), Pi enters.
    - If both are interested, turn will be set by one of them last. The one whose turn it is *not* will wait.
  + **Pros:** Elegant, demonstrates the logic of synchronization.
  + **Cons:** Limited to two processes, relies on busy waiting (spinning in a loop, wasting CPU cycles), and not practical for modern multi-core systems due to memory caching issues (compiler optimizations or CPU reordering can break it).

**3. Hardware-assisted Solutions: TestAndSet(), CompareAndSwap()**

Modern systems provide special atomic (indivisible) hardware instructions that simplify synchronization and avoid issues like those in Peterson's solution.

* **Atomic Operations:** An operation that is executed as a single, uninterruptible unit. Either it completes entirely, or it doesn't happen at all.
* **TestAndSet() Instruction:**
  + **Concept:** An atomic instruction that reads the value of a memory word, returns the old value, and simultaneously sets the memory word to TRUE.
  + **Signature:** boolean TestAndSet (boolean \*target)
  + **How it works:**
    - When a process calls TestAndSet(&lock), if lock was FALSE, it becomes TRUE atomically, and the function returns FALSE. The process then enters the critical section.
    - If lock was already TRUE, TestAndSet returns TRUE, and the process keeps looping (busy waiting).
  + **Pros:** Simple to implement, guarantees mutual exclusion.
  + **Cons:** Still uses busy waiting, which wastes CPU cycles. Does not guarantee bounded waiting (a process could get unlucky and keep losing the race to acquire the lock).
* **CompareAndSwap() (CAS) Instruction:**
  + **Concept:** An atomic instruction that compares the value of a memory word with an expected value. If they match, it replaces the memory word with a new value. It returns the original value of the memory word.
  + **Signature:** int CompareAndSwap (int \*value, int expected, int new\_value)
  + **Implementation of Mutual Exclusion:**
  + **How it works:**
    - A process tries to CompareAndSwap(&lock, 0, 1). If lock is 0 (unlocked), it atomically sets lock to 1 and returns 0. The process enters the critical section.
    - If lock is 1 (locked), it returns 1, and the process keeps looping (busy waiting).
  + **Pros:** More general than TestAndSet, can be used for more complex atomic operations (e.g., atomic updates to linked lists). Guarantees mutual exclusion.
  + **Cons:** Still uses busy waiting. Does not guarantee bounded waiting.

**4. Synchronization Tools: Mutex Locks, Semaphores, Monitors**

These are higher-level synchronization constructs built upon hardware primitives or OS support, providing more robust and easier-to-use mechanisms than raw hardware instructions.

* **a. Mutex Locks (Binary Semaphores):**
  + **Concept:** A synchronization primitive that provides mutual exclusion. It's a binary lock, meaning it can be in one of two states: locked or unlocked.
  + **Operations:**
    - acquire(): A process calls acquire() before entering its critical section. If the lock is unlocked, it acquires it and enters. If the lock is already locked, the process blocks (goes into a waiting state, not busy waiting) until the lock is released.
    - release(): A process calls release() after exiting its critical section, making the lock available for other processes.
  + **Implementation:** Typically implemented using TestAndSet or CompareAndSwap at a low level, but the blocking mechanism (putting a process to sleep and waking it up) is handled by the OS scheduler, avoiding busy waiting.
  + **Pros:** Simple to use for mutual exclusion, avoids busy waiting (most of the time).
  + **Cons:** Can lead to deadlocks if not used carefully.
* **b. Semaphores:**
  + **Concept:** A more generalized synchronization tool than a mutex. It's an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() (or P() for *proberen* - to test) and signal() (or V() for *verhogen* - to increment).
  + **Types:**
    - **Counting Semaphore:** Can take on any non-negative integer value. Used to control access to a resource that has multiple instances.
      * wait(S): Decrements S. If S becomes negative, the process blocks.
      * signal(S): Increments S. If S was negative (meaning processes were blocked), one of the blocked processes is unblocked.
    - **Binary Semaphore:** Can only be 0 or 1. Similar to a mutex lock.
      * wait(S): Decrements S. If S is 0, the process blocks.
      * signal(S): Increments S. If S was 0 and processes were blocked, one is unblocked.
  + **Implementation:** Like mutexes, wait() and signal() operations are atomic and involve OS-level blocking/unblocking mechanisms to avoid busy waiting.
  + **Pros:** Highly versatile, can solve various synchronization problems (mutual exclusion, producer-consumer, reader-writer).
  + **Cons:**
    - **Error Prone:** Incorrect use (e.g., forgetting a signal(), calling wait() twice) can lead to deadlocks or violations of mutual exclusion.
    - **Difficult to Debug:** Errors are hard to trace.
* **c. Monitors:**
  + **Concept:** A high-level language construct (not a low-level primitive) that provides mutual exclusion and a mechanism for processes to wait for certain conditions to become true. It encapsulates shared data and the procedures that operate on that data.
  + **Key Features:**
    - **Mutual Exclusion:** Only one process can be active within a monitor at any given time. The compiler/runtime system automatically handles locking/unlocking.
    - **Condition Variables:** Used for more complex synchronization scenarios where processes need to wait for a specific condition.
      * wait(condition): A process calling wait() on a condition variable releases the monitor lock and blocks until another process signals that condition.
      * signal(condition): A process calling signal() on a condition variable wakes up one (or all) processes waiting on that condition.
  + **Pros:**
    - **Easier to Use:** Simplifies synchronization logic by handling mutual exclusion automatically.
    - **Less Error Prone:** Reduces the chance of programming errors like forgotten acquire()/release() calls.
    - **Stronger Guarantees:** Compiler/runtime can enforce correct usage.
  + **Cons:** Not available in all programming languages (e.g., Java has synchronized blocks and wait()/notify() which are similar to monitors; C/C++ require external libraries like Pthreads).

**5. Classic Synchronization Problems**

These are well-known problems used to test the effectiveness and correctness of synchronization mechanisms.

* **a. Bounded-Buffer Problem (Producer-Consumer Problem):**
  + **Scenario:** A producer process produces items and puts them into a shared fixed-size buffer. A consumer process consumes items from the buffer.
  + **Challenges:**
    - **Mutual Exclusion:** Only one process (producer or consumer) can access the buffer at a time.
    - **Buffer Full:** Producer must wait if the buffer is full.
    - **Buffer Empty:** Consumer must wait if the buffer is empty.
  + **Solution using Semaphores:**
    - mutex: Binary semaphore, initialized to 1 (for mutual exclusion to the buffer).
    - empty: Counting semaphore, initialized to N (size of buffer, represents empty slots).
    - full: Counting semaphore, initialized to 0 (represents full slots).
* **b. Readers-Writers Problem:**
  + **Scenario:** Multiple processes want to read from or write to a shared data resource.
  + **Constraints:**
    - Multiple readers can read concurrently.
    - Only one writer can write at a time.
    - If a writer is writing, no reader can read.
    - If a reader is reading, no writer can write.
  + **Variations:**
    - **First Readers-Writers Problem:** Prioritizes readers. If a reader is present, new readers are admitted, potentially starving writers.
    - **Second Readers-Writers Problem:** Prioritizes writers. If a writer is waiting, new readers are blocked until the writer finishes, potentially starving readers.
  + **Solution using Semaphores (First Readers-Writers - reader-preferred):**
    - mutex: Binary semaphore, initialized to 1 (for mutual exclusion on read\_count).
    - rw\_mutex: Binary semaphore, initialized to 1 (for mutual exclusion for writers, and for first/last reader).
    - read\_count: Integer, initialized to 0 (number of readers currently reading).
* **c. Dining-Philosophers Problem:**
  + **Scenario:** Five philosophers are seated around a circular table. In the center of the table is a bowl of rice, and between each pair of philosophers is a single chopstick. To eat, a philosopher needs two chopsticks (one from their left and one from their right).
  + **Challenge:** Design a protocol that allows philosophers to eat without deadlocking or starving.
  + **Problem:** If all philosophers simultaneously pick up their left chopstick, they will all be waiting for their right chopstick (which is held by their neighbor), leading to a **deadlock**.
  + **Solutions (to avoid deadlock):**
    1. Allow at most N-1 philosophers to be seated at the table.
    2. Allow a philosopher to pick up chopsticks only if both are available (atomic operation).
    3. Asymmetric solution: Odd-numbered philosophers pick up their left chopstick first, then their right. Even-numbered philosophers pick up their right chopstick first, then their left.
  + **Significance:** A classic example to illustrate the challenges of resource allocation and deadlock prevention in concurrent systems.

**6. Deadlock: Conditions, Prevention, Avoidance, Detection & Recovery**

Deadlock is a critical issue in concurrent systems where a set of processes are blocked indefinitely, waiting for resources held by other processes in the same set.2

* a. Deadlock Conditions (Necessary Conditions):3

For a deadlock to occur, all four of these conditions must hold simultaneously:4

* 1. **Mutual Exclusion:** At lea5st one resource must be held in a non-sharable mode (only one process can use it at a time).
  2. **Hold and Wait:** A process holding at least one resource is waiting to acquire additional resources currently held by other processes.
  3. **No Preemption:** Resources cannot be forcibly taken from a process; they must be released voluntarily by the process that is holding them.
  4. **Circular Wait:** A set of processes {P0, P1, ..., Pn} exists such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.
* **b. Deadlock Prevention:**
  1. **Concept:** Ensure that at least one of the four necessary conditions for deadlock cannot hold.
  2. **Strategies:**
     1. **Deny Mutual Exclusion:** Not always possible (e.g., printers are inherently non-sharable). For sharable resources (like read-only files), this is fine.
     2. **Deny Hold and Wait:**
        + **Option 1:** A process must request and be allocated all its resources *before* it begins execution. (Inefficient, resource waste, potential starvation).
        + **Option 2:** A process must release all its currently held resources before requesting new ones. (May require frequent saving/restoring state).
     3. **Deny No Preemption:**
        + If a process holding resources requests another resource that cannot be immediately allocated, it must release all its currently held resources. These released resources are added to the list of resources for which the process is waiting. The process will restart only when it can reacquire its old resources plus the new ones.
        + Or, if a resource is requested by a higher-priority process, the lower-priority process holding it can be preempted.
     4. **Deny Circular Wait:**
        + Impose a total ordering of all resource types. Processes must request resources in increasing order of enumeration.
        + Example: If resources are R1, R2, R3... and R1 < R2 < R3. A process requests R1, then R2, then R3. It cannot request R3 then R1. This breaks the cycle.
* **c. Deadlock Avoidance:**
  1. **Concept:** Requires the OS to have some prior information about the resources a process will request and release. The OS then dynamically checks if granting a resource request will lead to a safe state.
  2. **Safe State:** A state is safe if there exists a sequence of processes
  3. **Algorithm:**
     1. **Banker's Algorithm:** A well-known deadlock avoidance algorithm.
        + Each process must declare its maximum number of resources of each type it may need.
        + When a process requests a resource, the system checks if granting the request would leave the system in a safe state. If it does, the request is granted; otherwise, it's denied (or the process waits).
  4. **Pros:** Less restrictive than prevention.
  5. **Cons:** Requires prior knowledge of maximum resource needs, which is often difficult to obtain in practice. Overhead of running the algorithm.
* **d. Deadlock Detection and Recovery:**
  1. **Concept:** Allow the system to enter a deadlocked state, then detect it, and finally recover from it.
  2. **Detection:**
     1. **Resource-Allocation Graph:** A directed graph that shows processes, resources, and their assignments/requests. Cycles in the graph indicate a deadlock (if resources have single instances) or a potential deadlock (if resources have multiple instances).
     2. **Algorithm:** Periodically run an algorithm (similar to Banker's) to check for cycles in the resource-allocation graph or to check if all processes can eventually complete.
  3. **Recovery:**
     1. **Process Termination:**
        + **Terminate all deadlocked processes:** Simple but drastic.
        + **Terminate one process at a time:** Terminate processes until the deadlock cycle is broken. Choose the process to terminate based on factors like priority, time spent, resources used, etc.
     2. **Resource Preemption:**
        + Successively preempt resources from processes and give them to other deadlocked processes until the deadlock is broken.
        + **Rollback:** If a resource is preempted, the process that lost it must be rolled back to some safe state and restarted from that state. This requires checkpoints.
  4. **Pros:** More flexible than prevention or avoidance.
  5. **Cons:** Can be costly (detection overhead, recovery cost).

**Memory Management**

**1. Background: Basic Hardware, Address Binding, Logical vs. Physical Address Space, MMU**

To understand memory management, we first need to establish some foundational concepts.

* **Basic Hardware:**
  + **RAM (Random Access Memory):** The main memory where programs and data are stored for quick access by the CPU. It's volatile, meaning its contents are lost when power is off.
  + **CPU (Central Processing Unit):** Fetches instructions and data from RAM.
  + **Memory Controller:** A hardware component that manages the flow of data between the CPU and RAM.
* **Address Binding:**
  + **Definition:** The process of mapping program instructions and data to physical memory addresses.
  + **Stages of Binding:**
    1. **Compile Time:** If the memory location of a process is known beforehand (e.g., in a simple, single-program system), absolute code can be generated. The program must be recompiled if the starting address changes.
    2. **Load Time:** If the memory location is not known at compile time (e.g., the program might be loaded anywhere in memory), then relocatable code is generated. Binding occurs when the program is loaded into memory.
    3. **Execution Time (Run Time):** Most common in modern operating systems. Binding is delayed until run time. This allows a process to be moved during its execution (e.g., swapping), which is essential for virtual memory. Requires hardware support (MMU).

**🔵 1. Compile-Time Binding (Fixed Before Running)**

**🔧 When?**

* Binding happens during **compilation** (before the program even runs)

**💬 How?**

* The compiler **hardcodes absolute addresses** into your program.
* You must know **exactly where** in memory the program will be loaded.

**🎯 Analogy:**

You are building a house with **fixed room numbers** written on a blueprint.  
You say: Kitchen is at 1050. You can never move it.

**📦 Example:**

* You compile the program to start at physical address 1000.
* The compiler sets variable x to address 1050.

🛑 So if you try to run the program at address 2000, it **won’t work** — the addresses are all wrong.

**✅ Advantages:**

* Simple and fast (no address translation at runtime)

**❌ Disadvantages:**

* Inflexible — program **must be loaded in the same place** every time
* Can’t move in memory, can’t run multiple copies

**🟡 2. Load-Time Binding (Resolved When Loaded into RAM)**

**🔧 When?**

* Binding happens **when the program is loaded into memory**

**💬 How?**

* The compiler generates **relocatable code** (uses offsets like "x is 50 bytes from the start")
* At **load time**, the **loader** adds the actual starting address.

**🎯 Analogy:**

You’re moving into a new flat. You say: "The kitchen is the **second room from the entrance**."  
Wherever you enter, kitchen is 2 rooms ahead — **location is adjusted when you move in**.

**📦 Example:**

* Variable x is at offset 50
* Program is loaded at physical address 1000
* Final physical address of x = 1000 + 50 = 1050

**✅ Advantages:**

* More flexible — program can load at **different addresses**
* Still no special hardware needed

**❌ Disadvantages:**

* Once the program is loaded, **it can’t move** anymore
* Less dynamic than modern systems

**🔴 3. Execution-Time Binding (Dynamic During Execution)**

**🔧 When?**

* Binding happens **while the program is running**
* Requires **hardware support** like MMU (Memory Management Unit)

**💬 How?**

* The program uses **logical/virtual addresses**
* Every time the CPU accesses memory, the **MMU translates** the logical address to a physical one using **page tables** or **offsets**

**🎯 Analogy:**

You walk into a hotel and ask for "Room 101".  
The **receptionist (MMU)** checks a mapping and says: "Room 101 is currently Room 301 today."  
This can **change anytime**, and **you don’t know where it's physically located**.

**📦 Example:**

* Logical address = 500
* MMU adds base = 4000 → Physical address = 4500
* This can change on the fly, or even be paged to disk!

**✅ Advantages:**

* Very flexible — process can **move during execution**
* Enables **multitasking, paging, virtual memory**
* Multiple processes can share memory safely

**❌ Disadvantages:**

* Requires hardware (MMU)
* Slight runtime overhead

**🧠 Visualization Table**

| **Feature** | **Compile-Time Binding** | **Load-Time Binding** | **Execution-Time Binding** |
| --- | --- | --- | --- |
| When it happens | During compilation | When program is loaded | During execution (runtime) |
| Uses relocatable code? | ❌ No | ✅ Yes | ✅ Yes |
| Can program move? | ❌ No | ❌ No (after loading) | ✅ Yes (anytime) |
| Hardware needed? | ❌ No | ❌ No | ✅ Yes (MMU) |
| Used in... | Early/embedded systems | Older OS like DOS | Modern OS (Linux, Windows) |
| Flexibility | ❌ Low | ⚠️ Medium | ✅ High |

**🔚 Summary**

| **Binding Type** | **Flexibility** | **Needs MMU?** | **Real Use Case** |
| --- | --- | --- | --- |
| Compile-Time | ❌ No | ❌ No | Fixed embedded systems |
| Load-Time | ⚠️ Some | ❌ No | Early DOS-like systems |
| Execution-Time | ✅ Yes | ✅ Yes | Modern OS with virtual memory (Linux, Windows, Mac) |

* **Logical vs. Physical Address Space:**
  + **Logical Address (Virtual Address):** An address generated by the CPU. This is the address that programs "see" and use. It's a conceptual address within the process's own address space.
  + **Physical Address:** An address seen by the memory unit (RAM). This is the actual address in the main memory.
  + **Relationship:** In modern systems with virtual memory, the logical address space can be much larger than the physical address space. The MMU translates logical addresses to physical addresses.
* **Memory-Management Unit (MMU):**
  + **Definition:** A hardware device that maps logical addresses to physical addresses at run time. (part of the CPU or chipset)
  + **Mechanism:** A simple MMU scheme involves a **relocation register** and a **limit register**.
    1. **Relocation Register:** Contains the value of the smallest base physical address.
    2. **Limit Register:** Contains the range of logical addresses (size of the program/data).
    3. Every logical address generated by the CPU is added to the value in the relocation register to get the physical address. Before addition, the logical address is checked against the limit register to ensure it's within the valid range for the process.
  + **Significance:** Enables dynamic relocation, protection (preventing processes from accessing memory outside their allocated space), and the implementation of virtual memory.

**Example:**

* **Base Register = 4000**
* **Limit Register = 1000 (program size)**

**If CPU tries to access address 1500, MMU checks:** **1500 < 1000?**

**❌ → Trap (Memory violation)**

**If valid, physical address = Base + Logical = 4000 + 500 = 4500**

**2. Swapping**

* **Concept:** A memory management technique where a process (or parts of it) can be temporarily swapped out of main memory to a backing store (e.g., a fast disk) and then swapped back in later to continue execution.
* **Purpose:** Allows the total physical address space of all processes to exceed the actual physical memory available. It's a form of virtual memory.
* **Backing Store:** A fast disk that is large enough to accommodate copies of all memory images for all users and provides direct access to these memory images.
* **Process:**
  1. A process is currently in memory.
  2. If another process needs to run and there isn't enough free memory, the current process is "swapped out" to the backing store.
  3. When the swapped-out process is needed again, it's "swapped in" from the backing store to main memory.
* **Challenges:**
  1. **Context Switch Time:** Swapping adds significant overhead to context switch time because it involves disk I/O.
  2. **Roll Out, Roll In:** Swapping a process out and then swapping it back in.
  3. **Memory Allocation:** Where in memory should the swapped-in process go? This leads to fragmentation issues.

**3. Contiguous Memory Allocation**

* **Concept:** Each process is allocated a single, contiguous block of memory.
* **Types:**
  + **Fixed-Partition Allocation:** Memory is divided into a fixed number of partitions, each of a fixed size. A process is loaded into a partition that is large enough.
    - **Pros:** Simple to implement.
    - **Cons:**
      * **Internal Fragmentation:** If a process is smaller than the partition, the unused space within the partition is wasted. **Fragmentation** happens when memory is **wasted** because it's not used efficiently.
      * Limited number of processes.
  + **Variable-Partition Allocation:** Memory is initially one large block. As processes arrive, they are allocated a contiguous block of exactly the size they need. When a process terminates, its memory is freed, creating holes.
    - **Pros:** No internal fragmentation.
    - **Cons:**
      * **External Fragmentation:** As processes are loaded and unloaded, memory becomes fragmented into many small, non-contiguous holes. Even if the total free space is sufficient, it might not be contiguous enough for a new, larger process.
      * **Compaction:** A technique to solve external fragmentation by moving all allocated memory blocks together to form one large free block. Very expensive (high CPU overhead, I/O for moving data).
* **Dynamic Storage-Allocation Problem:** How to satisfy a request of size n from a list of free holes.
  + **First Fit:** Allocate the first hole that is big enough.
  + **Best Fit:** Allocate the smallest hole that is big enough. (Leaves the smallest leftover hole, but requires searching the entire list or sorting it).
  + **Worst Fit:** Allocate the largest hole. (Leaves the largest leftover hole, which might be useful for future large requests, but also requires searching).
  + **Performance:** First fit and best fit are generally better than worst fit in terms of speed and storage utilization.

**4. Paging**

* **Concept:** A non-contiguous memory allocation scheme that solves the external fragmentation problem. It allows a process's physical address space to be non-contiguous.
* **Mechanism:**
  + **Physical Memory (RAM):** Divided into fixed-size blocks called **frames**.
  + **Logical Memory (Process Address Space):** Divided into fixed-size blocks called **pages**.
  + **Page Size:** Typically a power of 2 (e.g., 4 KB, 8 KB). Page size and frame size are the same.
  + **Page Table:** A data structure (per process) that maps logical pages to physical frames.
    - Each entry in the page table contains the base address of the physical frame where the corresponding page is loaded.
    - When the CPU generates a logical address, the MMU uses the page table to translate it into a physical address.
      * Logical Address = (Page Number, Offset)
      * Page Number (p): Used as an index into the page table.
      * Offset (d): Combined with the frame address from the page table entry to get the physical address.
      * Physical Address = (Frame Number, Offset)
* **Pros:**
  + **Eliminates External Fragmentation:** Any free frame can be used for any page.
  + **Simpler Allocation:** Just find a free frame.
  + **Allows Virtual Memory:** Processes can use a logical address space larger than physical memory.
* **Cons:**
  + **Internal Fragmentation:** A process's last page might not be fully used, leading to wasted space within that frame.
  + **Page Table Overhead:** Page tables can be very large, especially for large logical address spaces. This can consume significant memory.
  + **Translation Lookaside Buffer (TLB):** A small, fast hardware cache that stores recent page-to-frame translations. Speeds up address translation by avoiding frequent page table lookups in main memory. If a translation is found in the TLB (a TLB hit), the physical address is immediately available. If not (a TLB miss), the MMU must access the page table in main memory.

**5. Structure of the Page Table**

To manage the potentially large size of page tables, various techniques are used:

* **a. Hierarchical Paging (Multilevel Page Tables):**
  + **Concept:** Break the logical address space into multiple page tables. A common approach is a **two-level page table**.
  + **Mechanism (Two-level):** The page number is divided into an outer page number and an inner page number (or page offset).
    - The outer page number indexes into an **outer page table** (or page directory).
    - An entry in the outer page table points to an **inner page table**.
    - The inner page number indexes into the inner page table to find the frame number.
  + **Pros:** Only the necessary parts of the page table need to be in memory. Reduces the memory required for the page table itself.
  + **Cons:** Increased number of memory accesses for address translation (e.g., two memory accesses for a two-level page table without TLB).
* **b. Hashed Page Tables:**
  + **Concept:** Used for address spaces larger than 32 bits (e.g., 64-bit systems). A hash function maps the page number to an entry in a hash table.
  + **Mechanism:** Each entry in the hash table contains a linked list of elements that hash to the same location. Each element in the list contains the virtual page number, the physical frame number, and a pointer to the next element.
  + **Pros:** Efficient for sparse address spaces.
  + **Cons:** Hash collisions can lead to longer search times.
* **c. Inverted Page Tables:**
  + **Concept:** Instead of having a page table for each process, there is only *one* page table for the entire system.
  + **Mechanism:** Each entry in the inverted page table stores the virtual address (page number) of the page that is currently stored in a *specific physical frame*, along with the process ID that owns that page.
  + **Pros:** Reduces the memory required for page tables (proportional to physical memory, not virtual address space).
  + **Cons:**
    - **Increased Search Time:** When a process generates a logical address, the entire inverted page table must be searched to find the corresponding physical frame.
    - Requires a TLB for efficient operation.

**6. Segmentation**

* **Concept:** A memory management scheme that views memory as a collection of variable-sized logical units called **segments**. Each segment corresponds to a logical unit of a program (e.g., code, data, stack, heap, subroutines).
* **Mechanism:**
  + Logical addresses consist of a **segment number** and an **offset within the segment**.
  + **Segment Table:** Each entry in the segment table contains the **base address** (physical starting address) and **limit** (length) of the segment.
  + When the CPU generates a logical address, the MMU uses the segment number to find the base and limit. It checks if the offset is within the limit. If valid, it adds the offset to the base to get the physical address.
* **Pros:**
  + **User View of Memory:** Maps directly to the programmer's view of a program (code, data, etc.).
  + **Protection:** Easier to apply protection (read-only, execute-only) at the segment level.
  + **Sharing:** Easier to share segments between processes (e.g., shared code libraries).
* **Cons:**
  + **External Fragmentation:** Since segments are variable-sized, segmentation can suffer from external fragmentation, similar to variable-partition allocation.
  + **Complex Allocation:** Finding a sufficiently large contiguous block for a segment can be challenging.

**7. Virtual Memory: Demand Paging, Copy-on-Write**

* **Concept:** A technique that allows the execution of processes that are not entirely in memory. It separates the user's logical memory from physical memory.
* **Benefits:**
  + **Larger Programs:** Programs can be larger than physical memory.
  + **More Programs:** More programs can run concurrently (increased multiprogramming).
  + **Less I/O:** Only load parts of the program that are actually needed.
* **Implementation:** Most commonly implemented using **demand paging**.
* **a. Demand Paging:**
  + **Concept:** Pages are loaded into memory only when they are needed (demanded) during program execution.
  + **Mechanism:**
    - When a process starts, only a few pages (or none) are loaded initially.
    - When the CPU tries to access a page that is not in memory, a **page fault** occurs.
    - **Page Fault Handling:**
      1. The MMU detects that the page is not in memory (invalid bit in page table entry).
      2. A trap (interrupt) is generated to the OS.
      3. The OS determines if the reference was valid but the page is simply not in memory.
      4. If valid, the OS finds a free frame.
      5. If no free frame, a **page replacement algorithm** is used to select a victim page to be swapped out.
      6. The desired page is read from disk into the free frame.
      7. The page table is updated.
      8. The instruction that caused the page fault is restarted.
  + **Pure Demand Paging:** Start a process with no pages in memory.
  + **Locality of Reference:** Programs tend to access a relatively small portion of their address space at any given time. This principle makes demand paging effective.
  + **Performance:** Measured by **Effective Access Time (EAT)**, which considers the page fault rate.
    - EAT = (1 - p) \* ma + p \* page\_fault\_time
      1. p: page fault rate (0 <= p <= 1)
      2. ma: memory access time
      3. page\_fault\_time: time to handle a page fault (includes swap out, swap in, restart instruction).
* **b. Copy-on-Write (COW):**
  + **Concept:** A technique used in virtual memory management to allow multiple processes to share the same pages in memory. When a process forks (creates a child process), the parent and child initially share all the same pages.
  + **Mechanism:**
    - Pages are marked as "copy-on-write" (read-only) in the page tables of both processes.
    - If either process tries to *write* to a shared page, a page fault occurs.
    - The OS intercepts the write, makes a *private copy* of that specific page for the writing process, and updates its page table to point to the new copy.
    - The original page remains shared (or becomes private to the other process).
  + **Pros:**
    - **Efficiency:** Reduces memory usage and the overhead of copying entire address spaces during fork().
    - **Faster Process Creation:** Child processes are created much faster.
  + **Cons:** Can incur a slight overhead on the first write to a shared page.

**8. Page Replacement Algorithms**

When a page fault occurs and there are no free frames, the OS must choose a "victim" page to swap out to make room for the incoming page. The goal is to minimize the page fault rate.

* **a. FIFO (First-In, First-Out):**
  + **Concept:** The page that has been in memory the longest is replaced.
  + **Mechanism:** Maintain a queue of pages in memory. The page at the front of the queue is replaced. When a new page is brought in, it's added to the rear.
  + **Pros:** Simple to implement.
  + **Cons:** Can suffer from **Belady's Anomaly** (increasing the number of frames can sometimes *increase* the page fault rate). It doesn't consider how frequently or recently a page is used.
* **b. Optimal Page Replacement (OPT or MIN):**
  + **Concept:** Replace the page that will *not be used for the longest period of time* in the future.
  + **Mechanism:** Requires future knowledge of the page reference string.
  + **Pros:** Lowest possible page fault rate (theoretically optimal).
  + **Cons:** Impossible to implement in practice because it requires knowing the future. Used as a benchmark for other algorithms.
* **c. LRU (Least-Recently Used):**
  + **Concept:** Replace the page that has not been used for the longest period of time in the past.
  + **Mechanism:** Requires keeping track of when each page was last used (e.g., using timestamps or a stack).
  + **Pros:** A good approximation of OPT. Does not suffer from Belady's Anomaly.
  + **Cons:**
    - **Expensive to Implement:**
      * **Counters:** Requires a time-stamp counter for each page, updated on every memory access.
      * **Stack:** Maintaining a stack of page numbers, moving the referenced page to the top. Both involve significant overhead.
    - Often approximated in practice.
* **d. LFU (Least-Frequently Used):**
  + **Concept:** Replace the page with the smallest count of references.
  + **Mechanism:** Maintain a counter for each page. Increment the counter on each reference. Replace the page with the lowest count.
  + **Pros:** Pages that are heavily used remain in memory.
  + **Cons:**
    - **Expensive to Implement:** Counters need to be updated.
    - Doesn't consider recency: A page that was heavily used in the past but is no longer needed might stay in memory, while a recently popular page is swapped out.
* **e. Second-Chance (Clock) Algorithm:**
  + **Concept:** A practical approximation of LRU that is much cheaper to implement.
  + **Mechanism:** Uses a circular queue and a "reference bit" for each page (set to 1 when the page is referenced, 0 otherwise).
    - When a page needs to be replaced, the algorithm inspects pages in a circular fashion.
    - If a page's reference bit is 1, it gives it a "second chance" (sets the bit to 0 and moves to the next page).
    - If a page's reference bit is 0, it's the victim.
  + **Pros:** Simple, relatively efficient, avoids Belady's Anomaly.
  + **Cons:** Still an approximation, not as good as true LRU.
* **f. Counting-Based Algorithms (e.g., MFU - Most Frequently Used):**
  + **MFU:** Replaces the page that has been used most frequently. The rationale is that the page with the smallest count has probably just been brought in and is about to be used. (Generally does not perform well).

**9. Allocation of Frames**

* **Concept:** How to distribute the available free memory frames among the competing processes.
* **Types:**
  + **Fixed Allocation:**
    - **Equal Allocation:** If there are m frames and n processes, each process gets m/n frames.
    - **Proportional Allocation:** Each process gets a number of frames proportional to its size (or priority).
  + **Dynamic Allocation (Priority Allocation):**
    - A higher-priority process can take frames from a lower-priority process.
    - When a page fault occurs, a victim frame is chosen from either the faulting process's own frames or from other processes' frames (if priority allows).
* **Global vs. Local Replacement:**
  + **Global Replacement:** A process can select a replacement frame from *all* frames in the system (including those allocated to other processes).
    - **Pros:** Higher CPU utilization, more flexible.
    - **Cons:** A process's page fault rate can be affected by the behavior of other processes.
  + **Local Replacement:** A process can only select a replacement frame from *its own set* of allocated frames.
    - **Pros:** Consistent performance for each process, easier to manage.
    - **Cons:** Can lead to lower overall CPU utilization as a process might have many free frames while another suffers many page faults.

**10. Thrashing**

* **Concept:** A situation in virtual memory systems where a process spends more time paging (swapping pages in and out of memory) than executing instructions.
* **Cause:** Occurs when a process does not have enough frames allocated to it to hold its entire working set (the set of pages actively being used). The page fault rate becomes very high.
* **Symptoms:**
  + Low CPU utilization (CPU is idle, waiting for I/O from paging).
  + High disk activity (constant swapping).
  + Increased page fault rate.
* **Solutions:**
  + **Provide more frames:** Increase the amount of physical memory or reduce the number of active processes.
  + **Working Set Model:** An algorithm that tries to keep the "working set" of a process (the set of pages recently used) in memory. If a process's working set cannot fit, it's suspended.
  + **Page-Fault Frequency:** Control the page fault rate. If it's too high, allocate more frames. If it's too low, deallocate frames.
  + **Admission Control:** Limit the degree of multiprogramming (number of processes running concurrently) to prevent overallocation of memory.