Operating system:

UNIT-1

Introduction:

What is an Operating System (OS)?

An **Operating System (OS)** is system software that acts as a **bridge between the user and computer hardware**.
It manages hardware, software, memory, processes, and all other resources of the system.

✅ Why Do We Need an Operating System?

- To make the **hardware usable**.
- To **manage and allocate resources** efficiently.
- To **run applications** and enable user interaction.
- To provide **security and access control**.

✅ Basic Definition

"An Operating System is a software that manages the computer hardware and provides services for computer programs."

✅ Main Roles of an Operating System

1. **Resource Manager** – Manages CPU, memory, I/O devices, etc.
2. **Control Program** – Controls the execution of programs.
3. **Interface** – Provides user interface (CLI/GUI).
4. **Coordinator** – Coordinates between software and hardware.

✅ Examples of Operating Systems

- **Windows** (Microsoft)
- **Linux** (Open-source)

- **macOS** (Apple)
- **Android** (Mobile OS by Google)
- **iOS** (Apple mobile OS)

✅ Types of Operating Systems

Type	Description	Example
Batch OS	Executes batches of jobs with no user interaction	Early IBM systems
Time-Sharing OS	Allows multiple users at the same time	UNIX
Distributed OS	Runs on multiple computers working together	LOCUS
Real-Time OS	Responds instantly to input	RTLinux, VxWorks
Mobile OS	Designed for smartphones	Android, iOS

✅ Key Characteristics

- **Multitasking**: Run multiple tasks at once.
- **Multiprogramming**: Multiple programs reside in memory.
- **Security & Protection**: Ensures authorized access.
- **Concurrency**: Supports execution of multiple processes.

✅ Conclusion

An Operating System is essential for the functioning of a computer system. It acts as the backbone, managing hardware and software efficiently, ensuring users can run applications, and enabling smooth and secure operations.

OS FUNCTIONS:

Functions of Operating System

The **Operating System (OS)** performs several important functions to manage the hardware and software resources of a computer.

✅ 1. Process Management

- Handles the **creation, scheduling, and termination** of processes (programs in execution).
- Ensures **CPU time** is fairly shared among all processes.
- Key Concepts:
  - **Multitasking**: Run multiple processes at once.
  - **Process Scheduling**: Decide which process runs next.
  - **Context Switching**: Save and load process states.

✅ 2. Memory Management

- Manages the system's **RAM** (main memory).
- Allocates and deallocates memory to processes.
- Keeps track of **which part of memory is in use** and by whom.
- Techniques used:
  - **Paging**
  - **Segmentation**
  - **Virtual Memory**

✅ 3. File System Management

- Controls how **data is stored, accessed, and organized** on storage devices.
- Functions include:
  - Creating, reading, writing, deleting files and directories.
  - Managing **permissions** and **file naming**.
  - Ensuring **data security and access control**.

✅ 4. Device Management (I/O Management)

- Manages **input/output devices** like keyboards, printers, and hard drives.
- Uses **device drivers** to communicate with hardware.
- Ensures efficient **data transfer** between devices and system.

✅ 5. Security and Protection

- Prevents **unauthorized access** to system resources.
- Provides **user authentication** (e.g., passwords, login systems).
- Ensures that processes do not interfere with each other's memory.

✅ 6. User Interface

- Provides an interface for user interaction with the system.
- Two main types:
  - **CLI (Command Line Interface)** – e.g., Terminal
  - **GUI (Graphical User Interface)** – e.g., Windows desktop

---

✅ 7. Job Scheduling and Resource Allocation

- Manages **system resources** like CPU, memory, and I/O devices.
- Uses **schedulers** to decide the order in which tasks are executed.
- Ensures **fair and efficient use** of resources.

✅ 8. Error Detection and Handling

- Continuously monitors the system for **errors** (e.g., hardware failures, software bugs).
- Takes appropriate action to **maintain stability and security**.

✅ 9. Communication Services

- Enables **inter-process communication (IPC)**.
- Supports communication between processes on the **same or different systems** (in a distributed OS).

OS OPEARIONS:

Operating System Operations

The **Operating System (OS)** is responsible for managing and coordinating the activities of computer hardware and software. It ensures that the system operates efficiently and securely.

✅ 1. Interrupt-Driven Operation

- The OS is **event-driven** and responds to **interrupts**.
- **Interrupt**: A signal from hardware or software indicating an event that needs immediate attention.
- Example:
    - A keyboard press sends an interrupt to the CPU.
    - Hardware devices (like printers) signal when tasks are complete.

📝 **Types of Interrupts**:

- **Hardware interrupts** (e.g., mouse click, disk ready)
- **Software interrupts** (e.g., system calls)

✅ 2. Dual-Mode Operation

- To protect the system, OS works in **two modes**:
    - **User Mode**: Limited access; user programs run here.
    - **Kernel Mode**: Full access; OS runs in this mode.

🛡️ **Why Dual Mode?**
To **prevent user programs** from performing unauthorized operations that may damage the system.

✅ 3. Transition Between User and Kernel Mode

- Happens when:
    - A **system call** is made (e.g., to access a file).
    - An **interrupt** or **exception** occurs.
- The CPU switches from **user mode to kernel mode** to safely execute OS-level instructions.

✅ 4. Timer Control

- The OS uses a **timer** to control program execution.
- Prevents a process from running **too long** and blocking others.
- Used in **process scheduling** and **preemption**.

✅ 5. I/O Operations

- OS handles **input/output operations** using device drivers.
- Uses **buffers** and **queues** to manage data efficiently.
- Examples:
    - Reading data from disk
    - Displaying output to screen

✅ 6. Process and Memory Management

- OS controls the **creation, scheduling, and termination** of processes.
- Allocates memory to processes and handles **virtual memory**.

✅ 7. System Calls and Exception Handling

- System calls allow user programs to request OS services.
- OS handles **exceptions** (errors) like division by zero or invalid memory access.

✅ 8. Booting the System

- On power-up, the system runs a **bootstrap program** (from ROM) that loads the OS into memory.
- This process is known as **booting**.

📝 Summary Table

Operation	Description
Interrupt Handling	Respond to hardware/software events
Dual-Mode	User mode (limited) and Kernel mode (full access)
Timer Use	Prevent infinite loops by setting execution limits
I/O Management	Handles input/output via drivers
System Calls	Interface for programs to access OS services
Exception Handling	Handles unexpected errors during execution
Booting	Starts OS from power-on using bootstrap loader

COMPUTING ENVIRONMENT:

Computing Environments in Operating Systems

A **computing environment** refers to how computer systems are set up and used to perform tasks. The **Operating System (OS)** is at the core of managing these environments, depending on how users, applications, and hardware interact.

✅ 1. Traditional Computing

- Earlier model with **standalone systems**.
- One user per system; applications run locally.
- Examples: Personal desktops, early mainframe computers.

📝 Key Features:

- Batch processing
- Time-sharing
- Limited user access
- Minimal network dependency

✅ 2. Client-Server Computing

- **Clients** request services, and **servers** provide them.
- Clients can be PCs, tablets, or apps.
- Servers are powerful systems that handle multiple requests.

📝 Example:

- Web browser (client) requesting a webpage from a web server.

🛠️ OS Role:

- Manages communication, resource access, and security between clients and servers.

✅ 3. Peer-to-Peer (P2P) Computing

- All systems are **equal** (peers) and share resources **directly**.
- No central server required.

📝 Example:

- File sharing apps like BitTorrent.

🛠️ OS Role:

- Enables device discovery, communication, and resource sharing.

✅ 4. Distributed Computing

- A group of computers work together to complete tasks as a **single system**.
- Systems may be geographically spread out.

📝 Example:

- Google Search backend
- SETI@Home

🛠️ OS Role:

- Coordinates communication and task allocation between systems.

✅ 5. Virtualization

- Running **multiple operating systems** on a single physical machine using a **hypervisor**.
- Each OS behaves like a separate computer (called a **virtual machine**).

📝 Example:

- Running Linux inside Windows using VirtualBox.

🛠️ OS Role:

- Host OS manages resources; guest OS runs in virtual environments.

✅ 6. Cloud Computing

- Provides **on-demand resources** (like storage, computing power) over the **Internet**.
- Users don't manage the physical infrastructure.

📝 Example:

- Google Drive, AWS, Microsoft Azure

🛠️ OS Role:

- Manages virtual machines, scaling, storage, and service delivery on cloud platforms.

✅ 7. Mobile Computing

- Computing through **portable devices** such as smartphones and tablets.
- Devices are connected via wireless networks.

📝 Example:

- Android OS on smartphones, iOS on iPhones

🛠️ OS Role:

- Manages sensors, wireless communication, battery, and apps.

📝 Summary Table

Environment	Description	Example
Traditional	Standalone systems	Desktop PCs
Client-Server	Clients request, servers respond	Web browsing
Peer-to-Peer (P2P)	Equal systems sharing resources	File sharing (Torrent)
Distributed	Multiple computers working as one	Google data centers
Virtualization	Multiple OSes on one machine	VirtualBox, VMware
Cloud Computing	Internet-based computing resources	AWS, Google Cloud

Environment	Description	Example
Mobile Computing	Portable devices with wireless access	Android, iOS smartphones

Open-Source Operating Systems & System Structures: Operating System Services

---

🟢 1. Open-Source Operating Systems

✅ What is an Open-Source OS?

An **open-source operating system** is an OS whose **source code is freely available** for anyone to view, modify, and distribute.

✅ Key Features:

- **Free to use**
- **Community-developed**
- **Highly customizable**
- Encourages **learning** and **innovation**

✅ Popular Examples:

Open-Source OS	Description
Linux	Most popular; used in servers, desktops, embedded systems
FreeBSD	UNIX-like OS known for performance and security
Ubuntu	User-friendly Linux distribution
Debian	Stable and widely used Linux-based OS
Android (partially)	Based on Linux kernel, open-source at core

✅ Benefits:

- Cost-effective
- Transparency
- No vendor lock-in
- Large support communities

🏗️ 2. System Structures: Operating System Services

Operating System Services are the **core functions** provided by the OS to help both **users and programs** perform essential tasks.

✅ Types of OS Services:

Service	Description
1. Program Execution	OS loads programs into memory and executes them
2. I/O Operations	Handles input and output through devices (keyboard, printer, etc.)
3. File-System Manipulation	Allows programs and users to read, write, create, delete files
4. Communication	Enables processes to exchange information (same or different systems)
5. Error Detection	Detects and handles errors (hardware and software)
6. Resource Allocation	Manages CPU, memory, and I/O among multiple users/programs
7. Accounting	Keeps track of system usage (who used what and for how long)
8. Protection and Security	Controls access to system resources; ensures only authorized access

User and Operating-System Interface

An **interface** is the point where interaction happens between the **user** and the **operating system (OS)**. The OS provides different types of **interfaces** to make it easier for users and applications to use system resources.

✅ Why Is an Interface Needed?

- Users and programs **cannot interact with hardware directly**.
- The OS provides a **safe, user-friendly way** to perform tasks like file access, program execution, and device control.

🔷 Types of Interfaces in Operating Systems

✅ 1. Command-Line Interface (CLI)

- User types **text commands** to perform actions.
- Requires **exact syntax** and understanding of commands.

📝 **Examples**:

- Windows Command Prompt (`cmd`)
- Linux Terminal (`bash`, `sh`)

🟢 **Advantages**:

- Powerful and flexible
- Uses less system resources

🔴 **Disadvantages**:

- Difficult for beginners
- Error-prone due to manual input

✅ 2. Graphical User Interface (GUI)

- Uses **windows, icons, menus, and pointers** (WIMP).
- Users interact using **mouse, keyboard, or touchscreen**.

📝 **Examples**:

- Windows OS
- macOS
- Ubuntu (GNOME, KDE)

🟢 **Advantages**:

- User-friendly and easy to learn
- Visual and interactive

🔴 **Disadvantages**:

- Requires more memory and processing power
- Less efficient for advanced users

✅ 3. Touchscreen Interface

- Used in **mobile devices** and tablets.
- Users interact directly by touching the screen.

📝 **Examples**:

- Android OS
- iOS

✅ 4. Voice-based and Natural Language Interfaces (NLI)

- Allow interaction using **voice commands** or **natural language**.

📝 **Examples**:

- Siri, Google Assistant, Alexa

🔷 Programmer Interface: System Calls

When a **program (application)** needs to use hardware resources, it **calls OS services** using **System Calls**.

✅ What are System Calls?

- **System Calls** are the interface between **user programs and the OS kernel**.
- They provide access to OS services like:
  - File operations
  - Process control
  - Device handling
  - Communication

📝 **Example**:

```
c
CopyEdit
int fd = open("file.txt", O_RDONLY);
```

Here, `open()` is a system call in C to open a file.

📝 Summary Table

Interface Type	Description	Example
CLI	Text-based command input	Linux Terminal, cmd.exe
GUI	Visual, icon-based interaction	Windows, macOS
Touchscreen	Direct touch interaction	Android, iOS
Voice/NLI	Voice or language-based interaction	Alexa, Siri
System Calls	Interface for programs to access OS	`read()`, `write()` in C

‘

Types of System Calls

System calls are the programming interface between the **user programs** and the **operating system**. They allow programs to request services from the OS kernel.

System calls are generally classified into the following types based on their functionality:

---

1. Process Control System Calls

- Manage processes (programs in execution).
- Examples of operations:
  - Create and terminate processes
  - Load and execute a program
  - Wait for process completion
  - Get process ID

Examples:

- `fork()` — create a new process
- `exit()` — terminate a process
- `wait()` — wait for a process to finish

---

2. File Management System Calls

- Handle file operations.
- Examples:

    - Create and delete files
    - Open and close files
    - Read from and write to files
    - Get file attributes

Examples:

- `open()` — open a file
- `read()` — read data from a file
- `write()` — write data to a file
- `close()` — close a file

---

3. Device Management System Calls

- Manage I/O devices.
- Examples:
    - Request device
    - Release device
    - Read/write from device
    - Get device attributes

Examples:

- `ioctl()` — control device parameters
- `read()` and `write()` — also used for devices

---

4. Information Maintenance System Calls

- Retrieve and set system information.
- Examples:
    - Get system time and date
    - Set system time and date
    - Get process or file attributes

Examples:

- `getpid()` — get process ID
- `alarm()` — set alarm clock

---

5. Communication System Calls

- Support interprocess communication (IPC).
- Examples:
    - Create and delete communication channels
    - Send and receive messages between processes

Examples:

- `pipe()` — create a communication channel
- `shmget()` — get shared memory segment

---

6. Security System Calls (in some OS)

- Manage security and protection.
- Examples:
    - Authenticate users
    - Access control

 Summary Table

Type of System Call	Purpose	Example System Calls
Process Control	Manage processes	`fork()`, `exit()`, `wait()`
File Management	Handle file operations	`open()`, `read()`, `write()`
Device Management	Control and access I/O devices	`ioctl()`, `read()`, `write()`
Information Maintenance	System info and attributes	`getpid()`, `alarm()`
Communication	Interprocess communication	`pipe()`, `shmget()`
Security	User authentication and access	(varies by OS)

System Programs in Operating Systems

System programs are a collection of programs provided by the operating system that help users and programmers to **perform common tasks** related to managing files, processes, and system resources.

They act as **tools** or **utilities** to simplify working with the system.

✅ What Are System Programs?

- Programs designed to **perform system-related tasks**.
- They provide an environment to **develop, test, and run application software**.
- They offer **basic functionality** such as file management, program editing, and system monitoring.

✅ Categories of System Programs

Category	Description	Examples
File Management	Programs to create, delete, copy, rename files	`cp`, `mv`, `rm`, `ls`
Status Information	Display system info like CPU usage, memory, processes	`ps`, `top`, `df`, `free`
File Modification	Text editors and tools to modify files	`vi`, `nano`, `sed`, `awk`
Programming Language Support	Compilers, assemblers, debuggers for software development	`gcc`, `gdb`, `make`
Program Loading and Execution	Loaders and linkers that prepare programs for execution	`ld` (linker), `loader`
Communication	Programs to communicate between systems or processes	`ssh`, `ftp`, `telnet`

✅ Examples of Common System Programs

- **File Commands:**
  - `ls` — list files
  - `cp` — copy files
  - `rm` — remove files
- **Text Editors:**
  - `vim`, `nano` — edit files
- **System Monitoring:**
  - `top` — shows running processes
  - `df` — shows disk space usage
- **Compilers & Debuggers:**
  - `gcc` — compile C programs
  - `gdb` — debug programs
- **Networking Tools:**
  - `ssh` — secure remote login
  - `ping` — test network connection

✅ Why Are System Programs Important?

- They provide a **user-friendly interface** to perform system tasks.
- Help users and developers **interact with the OS without needing to know hardware details**.
- Facilitate **software development** and **system maintenance**.

📝 Summary Table

System Program Type	Function	Example Commands
File Management	Manage files and directories	`ls`, `cp`, `rm`
Status Information	Show system and process status	`ps`, `top`, `df`
File Modification	Edit and modify files	`vim`, `nano`
Programming Support	Compile and debug programs	`gcc`, `gdb`
Program Loading/Execution	Load and execute programs	`ld`, loader
Communication	Enable system communication	`ssh`, `ftp`

Operating System Design and Implementation

Designing and implementing an operating system involves creating a system that efficiently manages hardware resources while providing a reliable and user-friendly interface.

✅ 1. Design Goals of an Operating System

Goal	Description
Efficiency	Make best use of hardware and resources
Fairness	Allocate resources fairly among users/programs
Reliability	Ensure the system is stable and recovers from errors
Security	Protect data and resources from unauthorized access
Portability	Make OS easy to adapt to different hardware
Usability	Easy to use interface for users and programmers

✅ 2. Operating System Design Approaches

Approach	Description
Monolithic Design	Entire OS runs in a single large block of code (kernel). Fast but complex and less modular. Example: Early UNIX.
Layered Approach	OS is divided into layers; each layer only interacts with the one below it. Easier to manage and debug.
Microkernel Design	Minimal kernel handling core functions (memory, IPC); other services run in user space. Improves reliability and portability. Example: MINIX.
Modular Design	OS is built from separate modules that can be loaded/unloaded dynamically. Flexible and easy to extend.
Hybrid Design	Combines monolithic and microkernel features. Example: Windows NT, macOS.

✅ 3. Implementation of Operating System

- **Programming Languages:** OS is usually implemented in **low-level languages** like **C** (for performance and hardware control) and some parts in **Assembly** (for direct hardware interaction).
- **Kernel:** The core part of the OS that manages resources, memory, processes, devices, and system calls.
- **System Libraries:** Provide common functions for system calls.
- **Utilities:** Additional programs that assist users and system administrators.

✅ 4. Steps in OS Implementation

1. **Requirement Analysis:** Understand hardware features and user needs.
2. **System Specification:** Define OS functions and services.
3. **Design:** Choose architecture and modular organization.
4. **Coding:** Write kernel, device drivers, system programs.
5. **Testing:** Verify functionality, fix bugs.
6. **Deployment:** Install and maintain the OS on target machines.

✅ 5. Challenges in OS Design

- Managing **hardware complexity**.
- Supporting **multiprogramming and multiprocessing**.
- Ensuring **security** and **fault tolerance**.
- Balancing **performance** with ease of use.

- Maintaining **portability** across different devices.

📝 Summary Table

Aspect	Description
Design Goals	Efficiency, fairness, reliability, security, portability, usability
Design Approaches	Monolithic, Layered, Microkernel, Modular, Hybrid
Implementation Details	Kernel programming (C, Assembly), system libraries, utilities
Steps	Analysis → Specification → Design → Coding → Testing → Deployment
Challenges	Hardware management, security, pe

Operating System Structure

The **structure of an operating system (OS)** defines how the OS is organized internally to manage hardware resources and provide services efficiently. Different OS designs use different structures.

✅ 1. Simple Structure (Monolithic)

- The OS is a **single large program**.
- All components like process management, file management, device drivers, etc., run in **kernel mode**.
- Easy to design but difficult to maintain and debug.
- Example: Early UNIX systems.

Advantages:

- Fast and efficient due to no modular separation.

Disadvantages:

- Complex, error in one module can crash the entire OS.
- Difficult to extend or modify.

✅ 2. Layered Structure

- OS is divided into **layers**, each built on top of the lower layers.
- The **bottom layer** interacts directly with hardware.
- Each layer communicates only with the layer below or above it.
- Simplifies debugging and system design.

Example: THE operating system (by Dijkstra).

Advantages:

- Clear structure and easy to debug.
- Layers can be replaced independently.

Disadvantages:

- Performance overhead due to layered communication.
- Difficult to define appropriate layers.

✅ 3. Microkernel Structure

- Only essential services (like communication, basic memory management) run in **kernel mode**.
- Other services (file system, device drivers) run in **user mode** as separate processes.
- More secure and reliable because faults in user mode do not crash the kernel.

Examples: MINIX, Mach.

Advantages:

- Better modularity and security.
- Easier to extend and port.

Disadvantages:

- Performance overhead due to context switching between user and kernel mode.

✅ 4. Modular Structure

- OS is built with a **core kernel** and separate **loadable modules**.
- Modules can be dynamically loaded/unloaded (e.g., device drivers).
- Combines flexibility and efficiency.

Example: Modern Linux kernels.

Advantages:

- Easy to extend.
- Efficient because only needed modules run.

✅ 5. Client-Server Structure

- OS services are provided by **servers** (processes) that communicate via message passing.
- The client requests services from the server.
- Often used in distributed systems.

📝 Summary Table

OS Structure	Description	Advantages	Disadvantages	Example
Simple (Monolithic)	Single large kernel program	Efficient, simple	Hard to maintain, less secure	Early UNIX
Layered	OS divided into layers	Easy debugging, modular	Performance overhead	THE OS
Microkernel	Minimal kernel, most services in user mode	Secure, modular	Context switching overhead	MINIX, Mach
Modular	Core kernel + loadable modules	Flexible, easy to extend	Slight complexity	Linux kernel
Client-Server	OS services as separate processes	Good for distributed systems	Communication overhead	Distributed OS

Operating System Debugging

Debugging is the process of **detecting, locating, and fixing errors or bugs** in an operating system during its development or maintenance.

Since an OS interacts directly with hardware and manages critical resources, debugging it is **complex and crucial**.

✅ Why is OS Debugging Important?

- Operating systems are **complex software** with many components running concurrently.
- Bugs can cause system crashes, data loss, or security vulnerabilities.
- Ensures system **stability**, **reliability**, and **security**.

✅ Challenges in Debugging OS

- OS runs in **kernel mode** with full hardware access — errors can crash the entire system.
- Hard to isolate and reproduce bugs because of concurrency and hardware interaction.
- Debugging tools must be able to **work at a low level** (e.g., inspecting memory, registers).

✅ Common Debugging Techniques in OS

Technique	Description
Print Debugging	Insert print statements to output variable values or states (e.g., `printf`)
Kernel Debugger	Special debugger tools that allow step-by-step execution and inspection of kernel code (e.g., GDB, KGDB)
Log Files	OS writes events/errors to logs for later analysis
Emulators/Simulators	Run OS on virtual machines or simulators to safely test and debug without affecting real hardware
Memory Dump Analysis	Analyze contents of memory after a crash (core dump) to find cause
Hardware Debugging Tools	Use hardware-level debuggers and logic analyzers to inspect processor and bus activity

---

✅ Kernel Debuggers

- Allow developers to **pause execution**, **examine memory/registers**, and **step through kernel code**.

- Can debug live systems or crash dumps.
- Examples: **KGDB** (Linux kernel debugger), **WinDbg** (Windows debugger).

✅ Debugging Process

1. **Reproduce the bug** — Try to recreate the issue reliably.
2. **Isolate the cause** — Narrow down which part of the OS causes the problem.
3. **Use debugging tools** — Print debugging, breakpoints, or kernel debuggers.
4. **Fix the bug** — Modify the code to correct the issue.
5. **Test thoroughly** — Ensure the fix works and does not create new bugs.

✅ Best Practices

- Use **modular design** to isolate and debug parts independently.
- Maintain **good logging** to trace issues.
- Test on **virtual environments** before deploying on real hardware.
- Use **version control** to track changes and fixes.

📝 Summary Table

Debugging Technique	Purpose/Use	Example Tools
Print Debugging	Simple output of variable values	`printf`
Kernel Debugger	Step-through kernel debugging	KGDB, WinDbg
Log Files	Record system events/errors	System logs
Emulators/Simulators	Safe testing environment	QEMU, VMware
Memory Dump Analysis	Post-crash error analysis	Core dump analyzers
Hardware Debugging Tools	Low-level hardware inspection	JTAG, logic analyzers

System Boot in Operating Systems

System Booting is the process of starting a computer and loading the operating system into the main memory so that the system can begin operation.

✅ What is Booting?

- Booting is the process that **powers up the computer, initializes hardware**, and **loads the OS**.
- It prepares the computer to run user programs and system processes.
- The program that initiates this process is called the **bootstrap program** or **bootloader**.

✅ Types of Booting

Type	Description
Cold Boot	Starting the computer from a completely powered-off state (turning it on).
Warm Boot	Restarting the computer without turning off the power (reboot or reset).

✅ Steps in the Booting Process

1. **Power-On Self Test (POST)**
   - When the computer is powered on, the BIOS (Basic Input Output System) or UEFI firmware performs POST to check hardware like RAM, disk drives, keyboard, etc.
   - If POST fails, the system usually halts with error messages or beep codes.
2. **Bootstrap Loader Execution**
   - After POST, BIOS/UEFI looks for the bootloader program in the bootable device (e.g., hard drive, SSD, USB).
   - The bootloader is a small program that loads the OS kernel into memory.
3. **Loading the Operating System**
   - The bootloader loads the OS kernel into RAM.
   - It sets up the initial environment and passes control to the OS.
4. **Kernel Initialization**
   - The OS kernel initializes hardware and system processes.
   - It sets up system services, device drivers, and user interfaces.
5. **User Login / Ready to Use**
   - Once initialization is complete, the system displays the login prompt or desktop, ready for user interaction.

✅ Common Bootloaders

- **GRUB (GRand Unified Bootloader)** — used in Linux systems.

- **NT Loader (NTLDR)** — used in older Windows systems.
- **Windows Boot Manager (BOOTMGR)** — used in modern Windows.

📝 Summary Table

Step	Description
Power-On Self Test (POST)	Hardware checks and diagnostics
Bootstrap Loader	Load bootloader from storage device
OS Loading	Bootloader loads OS kernel into memory
Kernel Initialization	OS sets up system services and drivers
User Interface	System ready for user login and operations

UNIT – II

PROCESS CONCEPT:

Process Scheduling in Operating Systems

Process Scheduling is the method by which an operating system decides the order in which processes run on the CPU.

Since many processes compete for the CPU, scheduling helps **manage multitasking efficiently**.

✅ Why is Scheduling Needed?

- CPU is a **shared resource**.
- Multiple processes may be **ready to run**.
- Scheduling ensures **fairness**, **efficient CPU use**, and **responsiveness**.

✅ Process States

A process moves through various states:

```
sql
CopyEdit
New → Ready → Running → Waiting → Terminated
```

- **Ready:** Process is ready to run, waiting for CPU.
- **Running:** Process is currently executing.
- **Waiting:** Process is waiting for some event (e.g., I/O).
- **Terminated:** Process has finished execution.

The **scheduler** selects a process from the ready queue to run.

✅ Types of Scheduling

Type	Description
Preemptive	OS can interrupt a running process to schedule another (e.g., Round Robin).
Non-preemptive	Process runs till completion or waits (e.g., FCFS).

✅ Common Scheduling Algorithms

Algorithm	How it Works	Advantages	Disadvantages
First-Come, First-Served (FCFS)	Runs processes in order of arrival	Simple, easy to implement	Poor average wait time, Convoy effect
Shortest Job Next (SJN)	Runs process with shortest CPU burst next	Minimizes average waiting time	Difficult to predict burst time, possible starvation
Round Robin (RR)	Each process gets a fixed time slice (quantum)	Fair, good for time-sharing systems	Context switching overhead
Priority Scheduling	Processes run based on priority	Good for prioritizing important tasks	Lower priority can starve

---

✅ Terminology

- **Context Switch:** Saving the state of a running process and loading the state of another.
- **Time Quantum (Time Slice):** Fixed amount of CPU time given to each process in Round Robin.
- **Throughput:** Number of processes completed per unit time.
- **Turnaround Time:** Total time taken from process submission to completion.
- **Waiting Time:** Time a process spends waiting in the ready queue.

✅ Scheduler Types

- **Long-term Scheduler:** Decides which jobs enter the system.
- **Short-term Scheduler (CPU Scheduler):** Selects processes for CPU execution.
- **Medium-term Scheduler:** Temporarily removes processes from memory (swapping).

📝 Summary Table

Aspect	Description
Scheduling Purpose	Efficient CPU utilization and fairness
Scheduling Types	Preemptive and Non-preemptive
Algorithms	FCFS, SJN, Round Robin, Priority
Key Terms	Context switch, time quantum, waiting time

Operations on Processes

A **process** is a program in execution, and the operating system performs several key operations on processes to manage them effectively.

✅ Main Operations on Processes

1. **Process Creation**
    - A new process is created by an existing process (called the **parent**).
    - The newly created process is called a **child** process.
    - The OS allocates resources and initializes process control block (PCB).
    - Examples:
        - `fork()` system call in Unix creates a new process.
2. **Process Termination**
    - A process finishes execution and is removed from the system.
    - It can terminate **normally** (successful completion) or **abnormally** (due to error or kill).
    - Resources allocated to the process are released.
3. **Process Suspension and Resumption**

    - A process can be **suspended** (temporarily stopped) to free CPU or resources.
    - Suspended processes are stored in secondary memory.
    - Later, the process can be **resumed** and moved back to ready state.
4. **Process State Transitions**
    - Processes move through different states:
        - **New → Ready → Running → Waiting → Terminated**
    - Operations cause transitions, for example:
        - Creation → New to Ready
        - Scheduler picks process → Ready to Running
        - Process waits for I/O → Running to Waiting
        - I/O completes → Waiting to Ready
        - Process ends → Running to Terminated

✅ Process Control Block (PCB)

- The OS maintains information about each process in a PCB.
- PCB stores:
    - Process ID
    - Process state
    - CPU registers and program counter
    - Memory limits
    - Scheduling information
    - I/O status

📝 Summary Table

Operation	Description
Process Creation	Create a new process (parent-child relationship)
Process Termination	End process and release resources
Process Suspension	Temporarily stop process and swap out
Process Resumption	Restart suspended process
State Transitions	Change process state based on events

Inter-Process Communication (IPC)

Inter-Process Communication (IPC) is the mechanism that allows processes to **exchange data and coordinate** their actions.

Since processes run independently and may be in different states or even on different machines, IPC is essential for cooperation.

✅ Why IPC is Needed?

- Processes may need to **share data**.
- Synchronize actions (e.g., producer-consumer problem).
- Coordinate resource sharing.
- Communicate in distributed systems.

✅ Types of IPC

Type	Description	Example
Shared Memory	Processes share a common memory area for communication. Faster but requires synchronization.	POSIX shared memory
Message Passing	Processes communicate by sending and receiving messages through the OS. Easier but slower due to kernel involvement.	Pipes, message queues

---

✅ Methods of IPC

1. **Shared Memory**
   - A memory segment is shared between processes.
   - Processes read/write data to this shared space.
   - Requires **synchronization** mechanisms (like semaphores) to avoid conflicts.
2. **Message Passing**
   - Processes send discrete messages to each other.
   - Messages can be **synchronous** (blocking) or **asynchronous** (non-blocking).
   - Useful for communication between processes on different machines.

✅ Common IPC Mechanisms

Mechanism	Description
Pipes	One-way communication channel between related processes (parent-child).
Named Pipes (FIFOs)	Like pipes but can be used between unrelated processes.

Mechanism	Description
Message Queues	Store messages in queues; processes read/write messages asynchronously.
Semaphores	Used to synchronize access to shared resources.
Sockets	Communication over a network, used in client-server systems.

✅ Example Scenario

- In a **producer-consumer** problem, producer and consumer processes use IPC to share data buffer and synchronize actions to avoid conflicts.

📝 Summary Table

IPC Type	Advantages	Disadvantages
Shared Memory	Fast data exchange	Complex synchronization needed
Message Passing	Simpler synchronization, works across machines	Slower due to kernel involvement

Communication in Client-Server Systems

A **client-server system** is a distributed system architecture where:

- The **client** requests services or resources.
- The **server** provides these services or resources.

Communication between client and server is key to making the system work.

✅ How Communication Happens

Clients and servers communicate over a network using standard protocols and techniques.

✅ Common Communication Methods

Method	Description
Remote Procedure Calls (RPC)	Client calls a procedure on a remote server as if it were local. Hides network communication complexity.
Sockets	Low-level network communication endpoints for sending/receiving data (TCP/UDP). Flexible and widely used.
Message Queues	Asynchronous communication where messages are sent to a queue and processed later by the receiver.
HTTP/REST APIs	Clients use HTTP requests to interact with web servers (common in web services).

✅ Key Concepts

- **Synchronous Communication:** Client waits for the server to respond (e.g., RPC).
- **Asynchronous Communication:** Client sends request and continues without waiting; server responds later (e.g., message queues).
- **Connection-Oriented:** Communication requires establishing a connection (e.g., TCP sockets).
- **Connectionless:** No connection setup, just sending messages (e.g., UDP sockets).

✅ Typical Communication Flow

1. **Client sends request** to server.
2. **Server receives request**, processes it.
3. **Server sends response** back to client.
4. **Client receives response** and continues.

✅ Example: Using Sockets

- The client creates a socket and connects to the server's socket.
- They exchange data through the connection.
- After communication, sockets are closed.

📝 Summary Table

Communication Type	Description	Examples
RPC	Remote function calls	ONC RPC, XML-RPC
Sockets	Network communication	TCP, UDP sockets
Message Queues	Asynchronous message passing	RabbitMQ, MSMQ
HTTP/REST APIs	Web-based client-server	RESTful web services

Multithreaded Programming

A **thread** is a lightweight process — a unit of execution within a process. A process can have multiple threads running concurrently.

---

1 Multithreading Models

Model	Description
Many-to-One	Many user-level threads mapped to one kernel thread. Easy but no true parallelism.
One-to-One	Each user thread maps to a kernel thread. Enables true parallelism but overhead is higher.
Many-to-Many	Many user threads mapped to many kernel threads. Combines benefits of both.

---

2 Thread Libraries

- Libraries provide APIs to create and manage threads.

Library	Description
POSIX Threads (Pthreads)	Standard threading library for Unix/Linux.
Windows Threads	Thread APIs in Windows OS.
Java Threads	Built-in thread support in Java language.

---

3 Threading Issues

Issue	Description
Race Conditions	When threads access shared data concurrently without synchronization, leading to inconsistent results.
Deadlock	Two or more threads waiting indefinitely for each other to release resources.
Starvation	Some threads never get CPU time because others monopolize resources.
Context Switching Overhead	Frequent switching between threads can degrade performance.

---

4 Example (Pthreads in C)

```
c
CopyEdit
#include <pthread.h>
#include <stdio.h>

void* printHello(void* arg) {
    printf("Hello from thread!\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, printHello, NULL);
    pthread_join(thread, NULL);
    return 0;
}
```

⏳ Process Scheduling

Process scheduling decides **which process runs on the CPU** and when.

---

1 Basic Concepts

- **CPU Scheduler**: Selects a process from the ready queue.
- **Context Switch**: Saving and loading process states when switching processes.
- **Preemptive vs Non-preemptive Scheduling**

---

2 Scheduling Criteria

Criterion	Description
CPU Utilization	Keep CPU as busy as possible (ideally 100%).
Throughput	Number of processes completed per time unit.
Turnaround Time	Time from submission to completion.
Waiting Time	Time a process waits in the ready queue.
Response Time	Time from submission until first response.

---

3 Common Scheduling Algorithms

Algorithm	Description	Preemptive or Non-preemptive
First-Come, First-Served (FCFS)	Run in order of arrival.	Non-preemptive
Shortest Job First (SJF)	Run shortest burst next.	Can be preemptive or not
Round Robin (RR)	Each process gets a fixed time quantum.	Preemptive
Priority Scheduling	Based on process priority.	Can be preemptive or not

---

Summary Table

Topic	Key Points
Multithreading Models	Many-to-One, One-to-One, Many-to-Many
Thread Libraries	Pthreads, Windows Threads, Java Threads
Threading Issues	Race conditions, deadlock, starvation
Scheduling Criteria	CPU utilization, throughput, turnaround time
Scheduling Algorithms	FCFS, SJF, Round Robin, Priority

Topic	Key Points

Multiple Processor Scheduling

- **Multiple Processor Scheduling** deals with scheduling processes or threads on systems with **more than one CPU/core**.
- Goals: Maximize CPU utilization, load balancing, and minimize response time.

Types of Multiprocessor Scheduling:

Type	Description
Asymmetric Scheduling	One processor schedules all processes; others execute only tasks assigned to them.
Symmetric Scheduling	Each processor runs its own scheduler and schedules processes independently.

 Thread Scheduling

- Threads are scheduled by the OS **within processes**.
- Scheduling can be:
  - **User-level threads:** Managed by thread libraries, invisible to OS scheduler.
  - **Kernel-level threads:** Scheduled by the OS directly.

Thread Scheduling Examples:

- **Round Robin:** Threads get CPU for a fixed time quantum in a cyclic order.
- **Priority Scheduling:** Threads with higher priority run first.

 Inter-Process Communication (IPC) Concepts

Topic	Key Points

1 Race Conditions

- Occur when two or more processes or threads access shared data **concurrently**, and the outcome depends on the order of execution.
- Example: Two threads incrementing the same counter simultaneously, leading to incorrect results.

---

2 Critical Regions

- A **critical region** is a code section where shared resources are accessed.
- Only one process/thread should execute in this region at a time to prevent race conditions.

---

3 Mutual Exclusion Mechanisms

a) Busy Waiting

- Process continuously checks a condition to enter the critical region.
- Inefficient as CPU cycles are wasted spinning.

b) Sleep and Wakeup

- Processes go to sleep (block) if they cannot enter critical region and get woken up when it's available.
- More efficient than busy waiting.

---

4 Semaphores

- A **semaphore** is a synchronization variable that can be used to solve mutual exclusion and synchronization problems.
- Types:
  - **Binary semaphore (mutex):** Takes only values 0 or 1.
  - **Counting semaphore:** Can have any non-negative integer value.
- Operations:

Topic	Key Points

- `wait()` or `P()`: Decrement semaphore, block if value is 0.
- `signal()` or `V()`: Increment semaphore, possibly waking a blocked process.

---

5 Mutexes

- A **mutex** is a locking mechanism to ensure mutual exclusion.
- Only one thread can own the mutex at a time.
- Mutex must be **locked before** entering critical section and **unlocked after** leaving.

---

6 Monitors

- A high-level synchronization construct that **combines mutual exclusion and condition variables**.
- Only one process can be active inside the monitor.
- Monitors manage access automatically, simplifying synchronization.

---

7 Message Passing

- Processes communicate by sending and receiving messages.
- Can be **synchronous** (blocking) or **asynchronous** (non-blocking).
- Useful for distributed systems.

---

8 Barriers

- Synchronization point where multiple processes or threads must wait until all have reached it.
- Useful for parallel computing to ensure all tasks complete a phase before moving on.

Topic	Key Points

📝 Summary Table

Concept	Description	Example/Use Case
Race Conditions	Concurrent access causing errors	Shared counter incrementing
Critical Region	Code section needing mutual exclusion	Updating shared variables
Busy Waiting	Continuous checking (inefficient)	Spinlock
Sleep & Wakeup	Block and wake processes	Blocking on resource availability
Semaphores	Synchronization variable	Producer-consumer problem
Mutexes	Mutual exclusion lock	Thread-safe data access
Monitors	Combined mutual exclusion + condition variables	High-level synchronization
Message Passing	Exchange of messages between processes	Client-server communication
Barriers	Synchronization point for threads	Parallel loop synchronization

Classical IPC Problems

These problems illustrate challenges in **process synchronization and inter-process communication (IPC)**, mainly focusing on **deadlock, starvation, and mutual exclusion**.

1️⃣ Dining Philosophers Problem

Scenario:

- Five philosophers sit around a circular table.
- Between each pair is one fork (total 5 forks).
- To **eat**, a philosopher needs **both forks** on their left and right.
- Philosophers alternate between **thinking** and **eating**.

Topic	Key Points

Problem:

- If every philosopher picks up the fork on their left simultaneously, no one can pick up the right fork → **deadlock**.
- Also, some philosophers might starve if others keep eating.

Solution Approaches:

- **Resource hierarchy:** Pick up forks in a certain order to avoid circular wait.
- **At most four philosophers can try to eat simultaneously** to prevent deadlock.
- Use **semaphores or mutexes** to control access to forks.

---

2 Readers-Writers Problem

Scenario:

- Multiple processes can be **readers** or **writers** accessing a shared data.
- **Readers** can read simultaneously without issues.
- **Writers** need exclusive access (no other readers or writers).

Problems to solve:

- Prevent **writers** from writing while others read.
- Prevent **readers** from reading while a writer writes.
- Avoid **writer starvation** (writers waiting forever if readers keep coming).
- Avoid **reader starvation** (if writers keep priority).

Variants:

- **First Readers-Writers Problem:** No reader kept waiting unless a writer is already writing.
- **Second Readers-Writers Problem:** Once a writer is ready, no new readers allowed (writer priority).

Synchronization:

Topic	Key Points

- Use **semaphores, mutexes, and counters** to manage reader/writer access.

Summary Table

Problem	Key Issue	Goal	Common Solution
Dining Philosophers	Deadlock & starvation of philosophers	Ensure no deadlock; fair access to forks	Semaphores, resource ordering
Readers-Writers	Synchronization between readers and writers	Allow concurrent reading, exclusive writing	Semaphores, mutexes, counters

---

Example Pseudo-code snippet for Dining Philosophers (simplified):

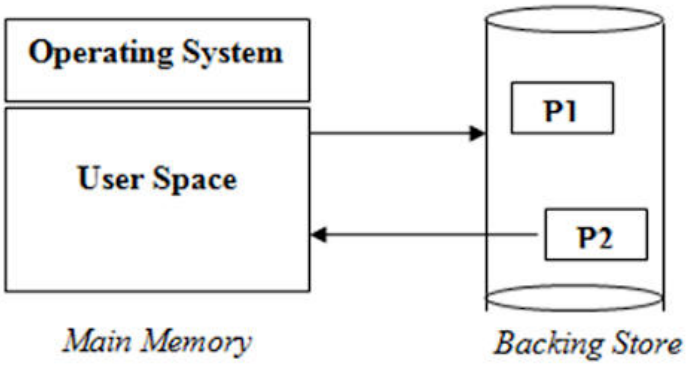
```
wait(fork[i]);     // Pick left fork
wait(fork[(i+1)%5]); // Pick right fork

// Eat

signal(fork[i]);     // Put down left fork
signal(fork[(i+1)%5]); // Put down right fork
```

UNIT-III

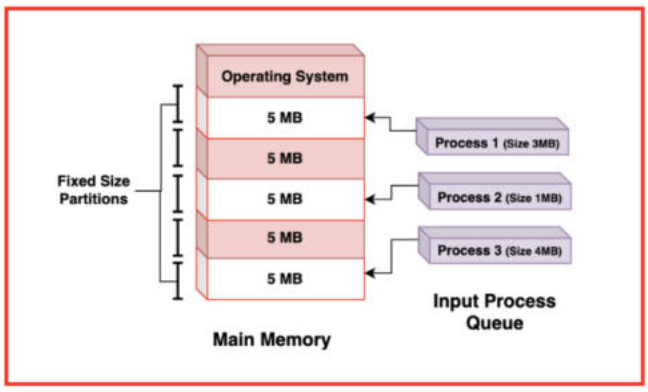
Memory-Management Strategies

Memory management is crucial for efficient system operation. Think of it like a librarian organizing books on a shelf to make sure they're easy to find and that there's enough space for new books. The strategies below are the different ways the librarian might organize their shelves.

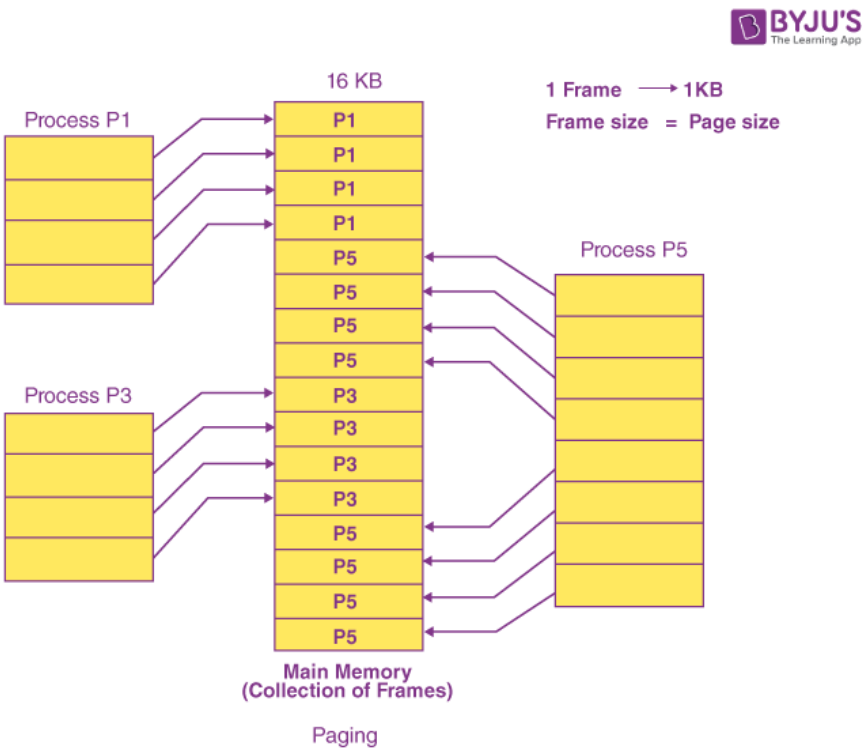
- **Swapping** is a core concept, but it's important to understand the **swapping process** itself. A process is completely moved from RAM to disk and then brought back. This is different from paging, where only parts of a process are moved. Swapping can be slow because it involves a lot of I/O, but it's essential for handling processes that are too large to fit in memory at the same time.

- 

- **Contiguous Memory Allocation** is simple but has a major drawback: **external fragmentation**. Imagine you have a parking lot with two large, empty spots. A small car comes and takes one. Another small car takes the other. Now, a bus arrives. There's plenty of empty space, but it's in two separate blocks, so the bus can't park. This is external fragmentation.



- **Paging** solves external fragmentation. The process and memory are broken into fixed-size chunks (pages and frames). The key is the **page table**, which acts as a lookup dictionary. When the CPU needs to access a memory address, it looks up the page number in the page table to find the corresponding frame number in physical memory. This process is very fast, as the page table is often stored in a special hardware cache called the **Translation Lookaside Buffer (TLB)**.



- **Segmentation** is more aligned with how a programmer thinks. Instead of just numbers, you have logical blocks like "code," "data," or "stack." This makes memory management more intuitive. For example, a shared library could be its own segment, easily shared among multiple processes without needing to be duplicated.

Virtual Memory Management

Virtual memory creates the illusion of a much larger memory space, decoupling logical memory from physical memory. It's like having a backpack that feels bottomless because items are only brought out when you need them.

- **Demand Paging** is the core of virtual memory. When a **page fault** occurs, it's not a program error, but an intentional interrupt that tells the OS to fetch the needed page from the disk. This is a very common and efficient way to handle memory. A key metric is the **page-fault rate**; a high rate indicates that the system is spending too much time swapping pages, which can lead to thrashing.

- **Copy-on-Write (COW)** is a powerful optimization. It's most commonly seen when a new process is created using the `fork()` system call in Unix-like systems. The parent and child processes initially share the same pages in memory. Only when one of them tries to write to a page is a copy made. This saves memory and speeds up process creation.

- **Page Replacement Algorithms** are a critical component. The goal is to minimize page faults. The **Least Recently Used (LRU)** algorithm is a good balance between performance and complexity. It works by keeping track of the last time each page was accessed. Another notable algorithm is **Second-Chance**, which is a more efficient approximation of LRU.

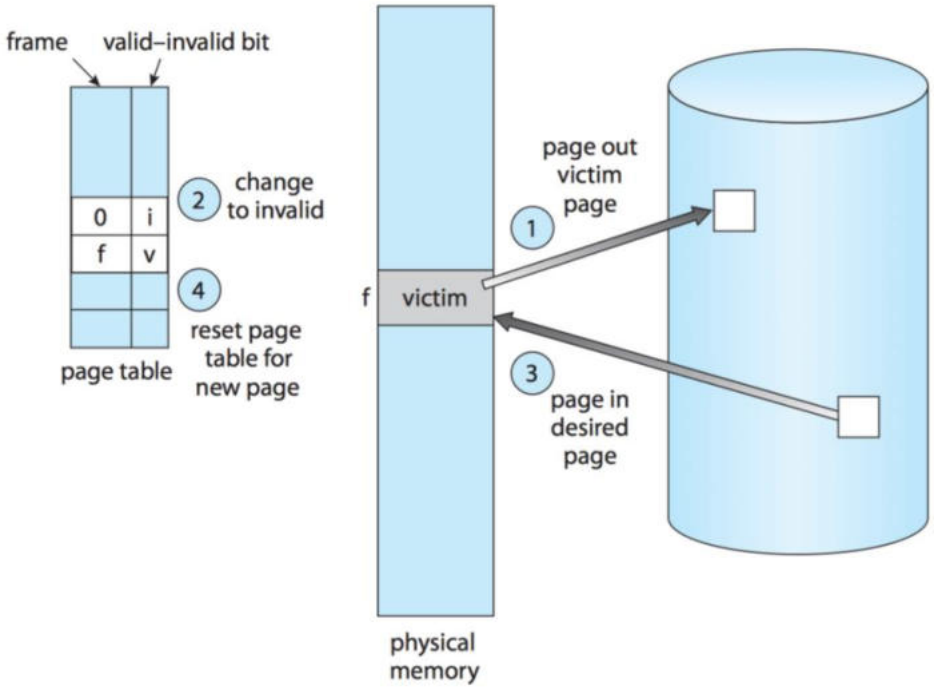
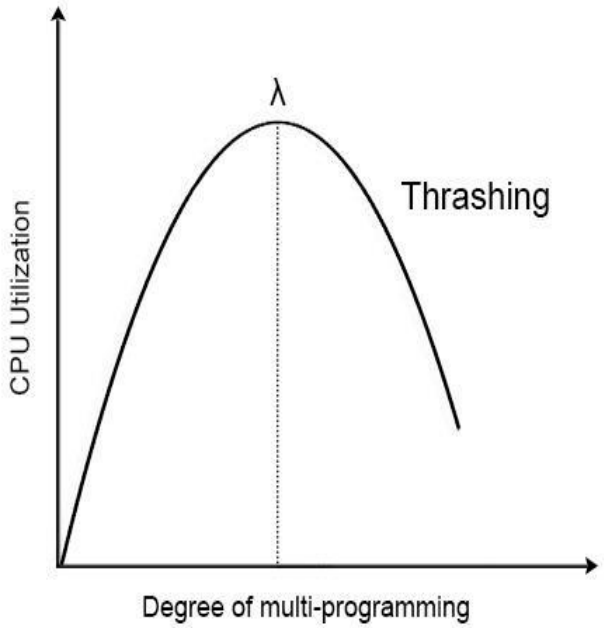


Figure 9.10 Page replacement.

S

- **Thrashing** is a severe performance issue. It's the point of diminishing returns in a multitasking system. The solution is often to reduce the number of active processes or to increase the amount of available physical memory. **The working-set model** is a way to prevent thrashing by ensuring a process has all the pages it needs (its "working set") in memory before it is allowed to run.



- **Memory-Mapped Files** provide a simple and elegant way to handle large files. Instead of using traditional `read()` and `write()` system calls, you can simply access the file data as if it were an array in memory. This is highly efficient for random access within a large file.

- **Kernel Memory Allocation** is a specialized area. The kernel needs its own fast and reliable memory allocator. For example, the **buddy system** allocator splits memory into power-of-two sized blocks. When a request for memory comes in, it finds the smallest block that fits and, if necessary, splits a larger block into two "buddies." This is efficient at preventing fragmentation within the kernel's memory space.

UNIT-4

Deadlocks

A **deadlock** occurs when a set of processes are blocked because each process is holding a resource and waiting to acquire a resource held by another process in the set.

Resources

A **resource** is anything a process needs to complete its task. Resources are categorized into two types:

- **Reusable resources:** Can be used by one process at a time and are not depleted (e.g., CPU, memory, files, devices).
- **Consumable resources:** Can be created and destroyed (e.g., signals, messages, interrupts).

Conditions for Resource Deadlocks

For a deadlock to occur, four conditions must be met simultaneously:

1. **Mutual Exclusion:** A resource can only be used by one process at a time.
2. **Hold and Wait:** A process is holding at least one resource and is waiting to acquire additional resources held by other processes.
3. **No Preemption:** A resource cannot be forcibly taken from a process holding it. It must be released voluntarily.
4. **Circular Wait:** A set of processes are in a circular chain where each process is waiting for a resource held by the next process in the chain.

Ostrich Algorithm

The **Ostrich algorithm** is a method for dealing with deadlocks by ignoring them. The name comes from the saying "burying one's head in the sand." This is a practical approach for systems where deadlocks are rare, as it is cheaper to simply reboot the system when one occurs than to implement complex prevention or avoidance strategies.

Deadlock Detection and Recovery

Deadlock detection involves periodically checking the system's state to see if a deadlock has occurred. This is often done using a **resource-allocation graph**. If a deadlock is detected, the system must recover by:

- **Process Termination:** Aborting one or more processes to break the circular wait. This can be done by terminating all deadlocked processes or one by one.
- **Resource Preemption:** Forcibly taking a resource from a process and giving it to another.

Deadlock Avoidance

Deadlock avoidance requires the operating system to have prior information about the resources a process will request. The system dynamically checks if a resource allocation will lead to an unsafe state. A **safe state** is one where the system can allocate resources to each process in some order without causing a deadlock. The **Banker's algorithm** is a well-known example of a deadlock avoidance algorithm.

Deadlock Prevention

Deadlock prevention ensures that at least one of the four necessary conditions for a deadlock can never occur.

- **Mutual Exclusion:** This is not always possible to prevent, as some resources are inherently non-shareable.
- **Hold and Wait:** A process must request all its resources at once before it begins execution, or it must release all its held resources before requesting new ones.
- **No Preemption:** The system can forcibly preempt a resource from a process.
- **Circular Wait:** Impose a total ordering of all resource types and require each process to request resources in increasing order.

File Systems

A **file system** is a method and data structure used by an operating system to control how data is stored and retrieved.

Files

A **file** is a named collection of related information stored on a secondary storage device. Files are abstract data types that can be organized, accessed, and managed by the file system.

Directories

A **directory** is a specialized file that contains a list of other files and directories. It provides a structured hierarchy for organizing and navigating the file system.

File System Implementation

File systems can be implemented in various ways. Key components include:

- **Layers:** A file system is often layered, with the file system interface providing an API for user programs, and the storage layer managing physical disk blocks.
- **Allocation Methods:** How disk space is allocated to files.
    - **Contiguous Allocation:** Each file occupies a set of contiguous blocks on the disk. Simple but suffers from external fragmentation.
    - **Linked Allocation:** Each file is a linked list of disk blocks. No fragmentation, but slow random access.
    - **Indexed Allocation:** A separate block (the **index block**) contains pointers to all the blocks of a file. Provides fast random access but can have wasted space.

Management and Optimization

File system management and optimization techniques include:

- **Free-Space Management:** Keeping track of available disk blocks using a **bit vector** or a **linked list**.
- **Efficiency and Performance:** Caching frequently accessed blocks in memory, using **i-nodes** (a data structure in Unix-like systems that stores information about a file, such as its location on disk) for quick access, and using different block sizes to balance performance and waste.
- **Journaling:** A technique used to ensure file system consistency after a crash by logging changes before they are made.

Secondary-Storage Structure

Secondary storage is non-volatile memory that retains data when the power is off. Hard disks are a common type.

Overview of Disk Structure and Attachment

A magnetic disk consists of platters, each with two surfaces. Each surface is divided into concentric circles called **tracks**, which are further divided into **sectors**. The set of tracks at the same distance from the center on all surfaces is a **cylinder**. Disks are attached to a computer via an I/O bus, such as SATA or SCSI.

Disk Scheduling

Disk scheduling algorithms determine the order in which disk I/O requests are serviced to minimize seek time (the time it takes for the disk head to move to the correct track).

- **FCFS (First-Come, First-Served):** Simplest, but not very efficient.
- **SSTF (Shortest Seek Time First):** Services the request closest to the current head position. Can lead to starvation for requests at the edges.
- **SCAN:** The disk arm moves from one end of the disk to the other, servicing requests along the way.
- **C-SCAN (Circular SCAN):** Similar to SCAN, but the head sweeps in only one direction, returning to the beginning to start the next sweep.

RAID Structure

RAID (**R**edundant **A**rray of **I**ndependent **D**isks) is a technology that combines multiple physical disks into a single logical unit to improve performance and/or provide redundancy for data

protection. Different RAID levels (e.g., RAID 0, RAID 1, RAID 5) offer various trade-offs between speed, data integrity, and cost.

Stable Storage Implementation

Stable storage is a concept of storage that can withstand a single failure, such as a disk crash. It is often implemented using a RAID level that provides redundancy (e.g., RAID 1 or RAID 5) or by maintaining two physical copies of the data and carefully managing writes to both copies to ensure data consistency.

UNIT-5

System Protection

System protection refers to mechanisms that control access to resources. Its primary goal is to prevent processes from harming each other or the system itself.

Goals of Protection

The main goals are:

- **Preventing misuse** of resources.
- **Ensuring data integrity** and consistency.
- **Preventing unauthorized access** to data.
- **Maintaining confidentiality** and privacy.

Principles and Domain of Protection

The **principle of least privilege** is a fundamental concept: a process should only be given the minimum rights it needs to perform its task. A **domain of protection** specifies the set of resources that a process can access. Domains can be implemented as user IDs, process IDs, or a combination of both.

Access Matrix

An **access matrix** is a model used to formalize protection. It's a two-dimensional table where rows represent **domains** (who can access) and columns represent **objects** (what can be accessed). Each cell in the matrix contains the **access rights** (e.g., read, write, execute) that the domain has over the object.

Access Control

Access control is the practical implementation of the access matrix. It can be implemented in two main ways:

- **Access Control List (ACL):** Each object has a list of which domains can access it and what rights they have.
- **Capability List:** Each domain has a list of the objects it can access and what rights it has.

Revocation of Access Rights

The ability to revoke access is crucial. **Revocation** can be immediate or delayed. Common methods include removing a capability from a list or deleting an entry from an access control list.

System Security

System security focuses on protecting a system from external and internal threats.

Introduction

The goal of security is to maintain **confidentiality** (preventing unauthorized data disclosure), **integrity** (preventing unauthorized data modification), and **availability** (ensuring resources are accessible to authorized users).

Program Threats

These are malicious programs designed to harm a system. Examples include:

- **Viruses:** Attach to legitimate programs and spread.
- **Worms:** Self-replicating programs that spread across networks.
- **Trojan Horses:** Disguised as legitimate software to trick users.
- **Logic Bombs:** Code that activates when a specific condition is met.

System and Network Threats

These target the operating system or network infrastructure. Examples include:

- **Denial-of-Service (DoS) Attacks:** Overloading a system to make it unavailable.
- **Man-in-the-Middle Attacks:** An attacker secretly relays and possibly alters communication between two parties.
- **Port Scanning:** Searching for open network ports to find vulnerabilities.

Cryptography as a Security Tool

Cryptography is the science of secure communication. It's used to:

- **Encrypt** data to ensure confidentiality.
- **Digitally sign** data to ensure integrity and authenticity.
- **Hash** data for secure storage and comparison.

User Authentication

Authentication verifies a user's identity. Common methods include:

- **Passwords:** The most common method.
- **Biometrics:** Using unique physical characteristics (e.g., fingerprints, facial recognition).
- **Tokens:** Using a physical device or a one-time password generator.

Implementing Security Defenses

Effective security requires a layered approach:

- **Firewalling:** Controlling network traffic.
- **Intrusion Detection Systems (IDS):** Monitoring for malicious activity.
- **Anti-virus Software:** Detecting and removing malware.
- **Regular Patching:** Keeping software up to date to fix vulnerabilities.

Firewalling to Protect Systems and Networks

A **firewall** is a network security system that monitors and controls incoming and outgoing network traffic based on predefined security rules. Firewalls can be **packet-filtering** (based on IP addresses and ports), **stateful** (aware of the context of a connection), or **application-level** (inspecting traffic at the application layer).

Computer Security Classification

Governments and organizations use classification levels (e.g., Confidential, Secret) to manage access to sensitive information. The **Orange Book** (formally the **Trusted Computer System Evaluation Criteria** or TCSEC) is a historical standard for evaluating computer systems' security features.

Case Studies: Linux and Microsoft Windows

Both Linux and Windows have robust protection and security features, but they approach them differently.

Linux

- **Protection:** Linux uses a **Discretionary Access Control (DAC)** model, where the owner of a file can set permissions for other users (read, write, execute). It also supports **Mandatory Access Control (MAC)** through modules like SELinux and AppArmor, which enforce a more rigid set of rules regardless of the file owner.
- **Security:** The open-source nature of Linux means many eyes can find and fix vulnerabilities quickly. It is often seen as more secure by default due to its permissions model and fewer users running with root (administrator) privileges.

Microsoft Windows

- **Protection:** Windows uses an **ACL-based** protection model. Every file and object has an Access Control List that defines permissions for different users and groups.
- **Security:** Windows has invested heavily in security features, including built-in firewalls, encryption tools (e.g., BitLocker), and robust user authentication (e.g., Active Directory). Its widespread use makes it a more frequent target for attackers, but also means there is a large industry dedicated to securing it.