Unit 1: Overview

1.1 Introduction

An Operating System (OS) is a system software that manages computer hardware and software resources. It provides essential services for applications and users. Examples include Windows, Linux, and macOS. The OS acts as an intermediary between users and computer hardware, ensuring efficient task execution and resource utilization. It also provides security, manages memory, and handles input/output operations. Without an OS, users would need to interact directly with hardware, making computing complex and inefficient.

1.2 System Structures

System structures define how different components of an OS work together, including the kernel, shell, file system, and device management. The kernel is the core part of the OS responsible for managing system resources and communication between hardware and software. The shell provides an interface for users to interact with the OS. Other components include system calls, user interfaces, and utility programs that support operations like file management, process control, and security.

1.3 The Abstract Model of Computing

This model simplifies computing by dividing tasks into input, processing, and output. The OS plays a crucial role in managing these operations efficiently. The abstraction hides complex hardware details from users and applications, allowing seamless execution of programs. Key layers of abstraction include hardware, OS kernel, system libraries, and user applications. The model ensures efficient resource allocation and multitasking while maintaining security and system stability.

1.4 Resources: Files

Files store data permanently on storage devices. The OS manages file organization, access, and security. A file system organizes files in directories and provides mechanisms for reading, writing, and modifying data. Different file access methods include sequential and direct access. File attributes include name, type, size, and permissions. The OS also implements file protection mechanisms to prevent unauthorized access, ensuring data integrity and security.

1.5 Processes: Creating Processes

Processes are instances of executing programs. In C, processes can be created using:

- `fork()`: Creates a new child process, duplicating the parent process.

- `join()`: Synchronizes processes, ensuring a child process completes before the parent resumes.
- `quit()`: Terminates a process, freeing system resources. Processes go through different states: new, ready, running, waiting, and terminated. The OS schedules processes using various algorithms to optimize CPU usage and performance.

1.6 Threads: C Threads

Threads are lightweight processes that share memory space. C provides threading using the `pthread` library. Threads allow concurrent execution within a process, improving efficiency. Types of threads include user-level and kernel-level threads. The OS manages threads using thread scheduling algorithms. Multithreading improves system responsiveness and resource utilization. Thread synchronization mechanisms, such as mutexes and semaphores, prevent race conditions and ensure data consistency.

Unit 2: Process Management

2.1 The System View of Processes and Resources

- A process is an executing instance of a program, which requires resources such as CPU time, memory, files, and I/O devices.
- The operating system (OS) is responsible for managing processes and allocating resources efficiently.
- Each process has a **Process Control Block (PCB)** that contains information like process ID, state, program counter, registers, and memory allocation.
- **Key system components**:
    - **Process Scheduler**: Determines which process gets CPU time.
    - **Memory Manager**: Allocates and deallocates memory.
    - **I/O Manager**: Handles input and output operations.

2.2 Initializing the Operating System

- The OS initializes during system boot-up, loading the kernel into memory.
- **Steps involved in OS initialization:**
    1. **BIOS Execution**: Basic Input/Output System (BIOS) initializes hardware.
    2. **Bootloader Execution**: Loads the OS kernel into memory.
    3. **Kernel Initialization**: Sets up essential data structures for process and memory management.
    4. **System Daemons Start**: Background services like logging and networking are initialized.

2.3 Process Address Spaces

- A process address space defines the memory structure allocated to a process.
- Address space typically consists of:
    1. **Text Segment**: Stores executable code.
    2. **Data Segment**: Stores global/static variables.
    3. **Heap**: Dynamically allocated memory.
    4. **Stack**: Stores function calls and local variables.

2.3.1 Creating the Address Space

- The OS allocates a unique address space for each process using **Virtual Memory**.
- **Methods of address space creation:**
    - **Forking**: Copies parent process space to a new child process.
    - **Exec System Call**: Replaces the current process address space with a new program.

2.3.2 Loading the Program

- The program must be loaded from disk into memory before execution.
- **Steps in program loading:**
    1. **Locate the program** in secondary storage.

2. **Allocate memory** for the process.
3. **Load the executable file** into memory.
4. **Set up the stack and heap.**
5. **Transfer control to the program.**

2.3.3 Maintaining Consistency in the Address Space

- OS ensures that each process accesses only its allocated memory using:
  - **Memory Protection**: Prevents processes from accessing unauthorized memory.
  - **Virtual Address Translation**: Maps virtual addresses to physical memory using page tables.
  - **Demand Paging**: Loads pages into memory only when needed to optimize resource use.

2.4 The Process Abstraction

- A process is an abstraction that represents an executing program.
- It provides an interface between the OS and running applications.

2.4.1 Process Descriptors

- A **Process Descriptor** (PCB) is a data structure storing process attributes such as:
  - **Process ID (PID)**
  - **Program Counter (PC)**
  - **CPU Registers**
  - **Memory Allocation**
  - **Process State (Ready, Running, Blocked, etc.)**
  - **Priority**

2.4.2 Process State Diagram

- A process transitions through multiple states:
  1. **New**: Process is being created.
  2. **Ready**: Process is waiting for CPU allocation.
  3. **Running**: Process is currently executing.
  4. **Blocked (Waiting)**: Process is waiting for an I/O operation to complete.
  5. **Terminated**: Process has finished execution.

2.5 The Resource Abstraction

- Resources like CPU time, memory, and I/O devices are required for process execution.
- OS provides **resource abstraction** to manage and allocate these resources efficiently.
- **Types of resources:**
  - **Preemptable**: Can be taken away from a process (e.g., CPU).
  - **Non-preemptable**: Cannot be taken away once allocated (e.g., printer).
- **Resource Allocation Table**: Tracks which resources are assigned to which processes.

2.6 Process Hierarchy

- Processes can be organized in a hierarchy where parent processes create child processes.
- **Example:** Unix-based systems use the `fork()` system call to create a child process.

2.6.1 Refining the Process Manager

- The process manager must handle:
  - **Process Scheduling**: Determines which process runs next.
  - **Inter-process Communication (IPC)**: Allows processes to share data.
  - **Synchronization**: Ensures processes execute in a controlled manner.

2.6.2 Specializing Resource Allocation Strategies

- Different strategies are used to allocate resources effectively:
  - **First-Come-First-Served (FCFS)**: Allocates resources to the first requesting process.
  - **Priority-Based Allocation**: Higher priority processes get resources first.
  - **Round Robin Scheduling**: Allocates CPU time in fixed time slices.
  - **Banker's Algorithm**: Prevents deadlock by ensuring resource requests are safe.

Unit 4: Basic Synchronization Principles

Synchronization in operating systems refers to the coordination of concurrent processes to ensure that they execute in a way that preserves consistency, avoids conflicts, and ensures correctness. This unit covers the key concepts like **interacting processes**, **critical sections**, **deadlock**, and synchronization techniques like **semaphores**.

4.1 Interacting Processes

In an operating system, **interacting processes** refer to processes that can communicate or share resources with each other. When multiple processes interact, especially when they share resources, it becomes crucial to manage synchronization to avoid issues such as **race conditions**, **deadlocks**, and **inconsistent data**.

4.1.1 Critical Sections

A **Critical Section** is a part of a program that accesses shared resources (such as variables, memory, or files) and must not be executed by more than one process at the same time. This is to prevent **race conditions**, where the outcome of a process depends on the non-deterministic ordering of events.

- **Example**: Suppose two processes, P1 and P2, are updating the same bank account balance. If both processes read the balance at the same time and then both write back their updated balance, the final balance might be incorrect, depending on the order of execution.

To prevent this issue, critical sections must be protected so that only one process can execute the section at a time.

Key properties of Critical Sections:

- **Mutual Exclusion**: Only one process can be in the critical section at any time.
- **Progress**: If no process is in the critical section, and a process is waiting to enter, one of the waiting processes must eventually be allowed to enter.
- **Bounded Waiting**: A process should not have to wait forever to enter the critical section (i.e., there should be a limit to the number of times other processes can enter the critical section before a waiting process is allowed).

4.1.2 Deadlock

Deadlock is a situation in a multiprocessor system where two or more processes are unable to proceed because each is waiting for the other to release resources. In other words, a set of processes are in a circular wait, preventing any process from making progress.

A **deadlock** occurs if the following four conditions hold simultaneously:

1. **Mutual Exclusion**: At least one resource must be held in a non-shareable mode (i.e., only one process can use a resource at a time).
2. **Hold and Wait**: A process holding a resource is waiting to acquire additional resources that are currently being held by other processes.
3. **No Preemption**: Resources cannot be forcibly taken from a process; they must be released voluntarily.
4. **Circular Wait**: A set of processes are waiting for each other in a circular chain, forming a loop.

Example:

- Process P1 holds resource R1 and waits for R2.
- Process P2 holds resource R2 and waits for R1.
- Both processes are waiting for each other and cannot proceed.

How to handle deadlock:

- **Deadlock Prevention**: Modify the system to prevent one of the deadlock conditions from occurring.
- **Deadlock Avoidance**: Dynamically assess the system state to decide if it is safe to allocate resources.
- **Deadlock Detection**: Allow deadlock to occur but have mechanisms to detect it and recover from it.
- **Deadlock Recovery**: Once deadlock is detected, terminate or restart processes involved in the deadlock.

4.2 Coordinating Processes

To ensure safe and correct coordination between interacting processes, synchronization mechanisms are employed to manage access to shared resources and prevent issues like race conditions or deadlocks.

Semaphores

A **semaphore** is a synchronization primitive used to control access to shared resources in a concurrent system. It helps manage mutual exclusion and coordinate processes in an efficient manner.

- **Types of Semaphores**:
    1. **Binary Semaphore** (also called a mutex): Takes values 0 or 1 and is used to manage exclusive access to a resource.
    2. **Counting Semaphore**: Can take any integer value and is used to manage access to a pool of resources.

Operations on Semaphores:

1. **Wait (P operation or down)**: This operation is used to acquire the semaphore. If the semaphore value is greater than 0, it is decremented, and the process proceeds. If the value is 0, the process is blocked until the semaphore becomes available.
2. **Signal (V operation or up)**: This operation is used to release the semaphore. It increments the semaphore value and potentially wakes up any processes that were waiting for the semaphore.

Principles of Operation

- **Mutual Exclusion**: When a process enters the critical section (protected by a semaphore), other processes must wait for the semaphore to be released before they can enter.
- **Signaling**: A process can signal (release) the semaphore once it is done with the resource, allowing another process to enter the critical section.

Example:

Consider two processes, **P1** and **P2**, that need to access a shared printer resource. A binary semaphore `S` can be used to ensure that only one process can access the printer at a time.

- Initially, the semaphore `S` is set to 1.
- **P1** executes `Wait(S)`, and the semaphore is decremented to 0. **P1** can now access the printer.
- While **P1** is using the printer, **P2** tries to execute `Wait(S)` but is blocked because the semaphore is 0.
- After **P1** finishes and calls `Signal(S)`, the semaphore is incremented to 1, and **P2** can now access the printer.

Practical Considerations

When using semaphores, it's essential to consider the following practical concerns:

1. **Race Conditions**: Ensure that the `wait` and `signal` operations are atomic. If multiple processes simultaneously attempt to access the semaphore, it might result in race conditions if the operations are not implemented properly.
2. **Starvation**: Ensure that no process is left waiting indefinitely. This can happen if lower-priority processes are perpetually preempted by higher-priority ones.
3. **Deadlock Prevention**: When using semaphores, deadlock can occur if there are circular dependencies among processes. To prevent this, proper semaphore ordering and resource allocation strategies must be followed.
4. **Priority Inversion**: In some cases, a higher-priority process may be blocked by a lower-priority process holding a semaphore. This is known as **priority inversion**, and it can be addressed using priority inheritance protocols.

Unit 5: Memory Management

Memory management is a critical aspect of operating systems that ensures efficient utilization of the primary memory (RAM). It manages the allocation and deallocation of memory, tracking which parts of memory are in use and which are free. Effective memory management helps prevent fragmentation, improves system performance, and ensures that processes run without interference.

5.1 The Basics

Memory management is responsible for allocating, tracking, and managing the primary memory of the computer. The OS ensures that each process gets the memory it needs and that processes do not interfere with each other's memory.

5.1.1 Requirements on the Primary Memory

The primary memory (RAM) is the fastest form of memory in a computer system, but it is limited in size. The operating system must efficiently manage this memory and ensure the following:

- **Isolation**: Processes should not interfere with each other's memory to maintain security and stability.
- **Efficiency**: The system should use memory efficiently, ensuring minimal wasted space (memory fragmentation).
- **Protection**: Each process should have its own memory space, and the OS must protect processes from accessing each other's memory (ensuring data integrity).
- **Relocation**: Memory locations should not be fixed, allowing for processes to be loaded anywhere in memory.

5.1.2 Mapping the Address Space to Primary Memory

The **address space** refers to the set of memory addresses that a process can use. This address space is mapped to the actual physical memory in the system by the operating system. The mapping can be done in several ways, such as:

- **Contiguous Allocation**: The entire address space of a process is mapped to a contiguous block of physical memory.
- **Paged or Segmented Memory**: The address space is divided into small chunks (pages or segments), which are mapped to non-contiguous blocks of physical memory.

5.1.3 Dynamic Memory for Data Structures

Operating systems manage **dynamic memory allocation** for processes during their execution. The memory required by processes for their data structures (like arrays, linked lists, and buffers)

may not be known in advance, and the system must allocate memory dynamically as the process runs. Dynamic memory management is essential for efficient resource utilization.

- **Heap**: The area of memory used for dynamic memory allocation during the execution of a program.
- The system must provide mechanisms like **malloc**, **free**, or **new** to allocate and deallocate memory dynamically in high-level programming languages.

5.2 Memory Allocation

Memory allocation strategies are key to managing memory efficiently. These strategies determine how memory is divided among processes and how fragmentation is handled.

5.2.1 Fixed-Partition Memory Strategies

In fixed-partition memory management, the physical memory is divided into a fixed number of partitions, each assigned to a different process. The size of the partitions is fixed and is determined at the time the system starts.

- **Advantages**:
  - Simple to implement.
  - No overhead for managing memory allocation.
- **Disadvantages**:
  - Wasted memory due to fixed partition sizes, leading to internal fragmentation.
  - Processes may not be able to use the entire partition if they don't need all of the allocated memory.

Example: If there are 4 partitions of 1GB, 2GB, 3GB, and 4GB, and a process that needs 2.5GB, it would still be assigned to the 3GB partition, leading to 0.5GB of wasted memory.

5.2.2 Variable-Partition Memory Strategies

In variable-partition memory management, memory is dynamically divided into partitions as needed. The size of the partitions depends on the size of the processes.

- **Advantages**:
  - Memory is allocated more efficiently based on the process size.
  - No wasted memory as with fixed partitions.
- **Disadvantages**:
  - **External fragmentation**: Over time, small free blocks of memory scattered throughout can prevent the allocation of larger blocks, even though total free memory is sufficient.
  - Requires complex bookkeeping to track free and used memory blocks.

Example:

- A system starts with 16GB of memory.
- Process A needs 4GB, so it is allocated a 4GB partition.
- Process B needs 6GB, so it gets the next available 6GB partition.
- Later, process C requests 5GB. There is enough total free memory (6GB + 6GB), but due to fragmentation, no large contiguous space is available for process C.

5.2.3 Contemporary Allocation Strategies

Modern operating systems use more sophisticated memory allocation strategies to minimize fragmentation and efficiently allocate memory.

1. **Buddy System**: The system divides memory into partitions that are powers of two (e.g., 1MB, 2MB, 4MB, etc.). If a process needs 3MB, the system will allocate a 4MB block, and the extra memory (1MB) is returned to the free pool.
2. **Slab Allocator**: This strategy is commonly used for kernel memory allocation. It organizes memory into slabs for storing objects of the same type, reducing internal fragmentation and improving memory utilization.
3. **First Fit, Best Fit, Worst Fit**: These are strategies used for allocating memory blocks to processes. The system can choose the first available block that fits, the best fit (smallest block that fits), or the worst fit (largest block available).

5.3 Dynamic Address Resolution

Dynamic Address Resolution refers to the process of converting logical addresses (virtual addresses) into physical addresses during program execution. It is crucial for managing the memory address space of processes.

5.3.1 Runtime Bound Checking

Runtime Bound Checking ensures that a process does not access memory outside its allocated range during execution. This is important to prevent **out-of-bounds errors**, which can lead to data corruption or crashes.

- **Bounds checking** is done during the execution of a program to ensure that the program accesses only valid memory regions. If a process tries to access memory outside of its allocated space, an exception or error is raised.

Example: If a process with an array of 10 elements tries to access the 11th element, a runtime error will be triggered due to out-of-bounds access.

5.4 Memory Manager Strategies

The memory manager in an operating system ensures that memory is allocated and deallocated efficiently and handles more complex strategies for process management.

5.4.1 Swapping

Swapping is a technique where entire processes are temporarily moved from the main memory (RAM) to secondary storage (e.g., hard disk) and then back to memory when needed. This is done to free up memory for other processes.

- **Advantages**:
  - Allows the system to handle more processes than can fit in memory at once.
  - It can provide the illusion of having more memory than is physically available.
- **Disadvantages**:
  - **Disk I/O overhead**: Swapping processes in and out of memory can slow down the system.
  - **Thrashing**: Occurs when the system spends more time swapping processes than executing them.

Example:

- If there are three processes, but the system only has memory for two, one process will be swapped out to the disk, allowing the other to execute. The swapped-out process will be swapped back when needed.

5.4.2 Virtual Memory

Virtual Memory allows processes to access more memory than is physically available by using a combination of **main memory** and **secondary storage** (e.g., disk). The OS uses a **page table** to map virtual addresses to physical addresses.

- **Advantages**:
  - Provides an illusion of a larger memory space, improving system efficiency and allowing processes to run as though they have access to more memory than physically available.
  - Enables processes to be isolated from each other, improving security and stability.
- **Disadvantages**:
  - Can lead to slower performance due to page faults and swapping.
  - Requires significant management overhead, such as maintaining page tables and handling page faults.

Example:

- A process might request 10GB of memory, but only 4GB of physical RAM is available. Virtual memory allows the OS to load parts of the process into physical memory while swapping out other parts to disk storage.

5.4.3 Shared-Memory Multiprocessors

In **shared-memory multiprocessors**, multiple processors access the same physical memory space. This allows processes to share data directly without needing to communicate through inter-process communication (IPC) mechanisms like message passing.

- **Challenges**:
  - **Synchronization**: Multiple processes accessing the same memory can cause race conditions or inconsistent data if synchronization mechanisms are not properly used.
  - **Cache Coherence**: Ensuring that multiple processors have consistent views of memory when each processor has its own cache.
- **Solution**: Operating systems use specialized hardware mechanisms (like **cache coherence protocols**) and software techniques (like semaphores or locks) to coordinate access to shared memory.

Unit 6: File Management LH 5

Unit 6 delves into **File Management**, which is essential for organizing, storing, and accessing data on storage devices. The concepts covered include file system structures, storage management, disk scheduling, I/O systems, and their implementation. Below is a detailed breakdown:

6.1 File System

A **file system** is a method for storing and organizing files on storage devices, ensuring efficient data management and retrieval.

6.1.1 File Concept

A **file** is a collection of data stored on a storage device (e.g., hard disk or SSD). Files are categorized into:

- **Text files**: Contain human-readable characters (e.g., `.txt` files).
- **Binary files**: Contain non-human-readable data, such as executable programs (`.exe`, `.bin`).
- **Special files**: Files used by the system, such as device files.

Each file is identified by its **filename** and has associated **attributes**, including:

- **Name**: A string used to identify the file.
- **Type**: Defines the format of the file (e.g., text, binary).
- **Location**: The physical location of the file on disk.
- **Size**: The amount of storage space it occupies.
- **Access Control Information**: Permissions for reading, writing, or executing.

6.1.2 Access Methods

There are different ways of accessing data from a file:

- **Sequential Access**: Data is read in order, one byte or block after another. Ideal for text files.
- **Random Access**: Data can be read from or written to any part of the file, regardless of the order. Used for databases and media files.
- **Indexed Access**: Uses an index to quickly locate data within a file, reducing the need to search through it sequentially.
- **Hashed Access**: Uses a hash function to directly access data at a particular location.

6.1.3 Directory Structure

A **directory** is a collection of files. The **directory structure** defines how these files are organized. Common structures include:

- **Flat Structure**: All files are stored in one single directory, without subdirectories.
- **Hierarchical (Tree) Structure**: Files are organized in a tree, with directories and subdirectories. This is the most commonly used structure.
- **Acyclic Graph (ACG)**: Extends the tree structure by allowing multiple directories to point to the same file.

Directories may contain the following:

- **File names**: Each file has a unique name within a directory.
- **Pointers**: Each file entry contains a pointer to its data blocks on the storage device.

6.1.4 File System Mounting

File system **mounting** refers to attaching a file system to an existing directory structure, making it accessible for file operations.

- **Mount Point**: A directory where the new file system is mounted.
- **Mounting Process**: The operating system reads the file system's superblock and structures, allowing users to access the file system as though it is part of the system.

6.1.5 File Sharing

File sharing refers to multiple users or processes accessing a file simultaneously.

- **Read-Write Sharing**: Multiple processes can read and write to the file at the same time.
- **Exclusive Locking**: Only one process can access a file at a time, often used for critical or sensitive data.
- **Network File Sharing**: Files can be shared over a network using protocols like **NFS** (Network File System) or **SMB** (Server Message Block).

6.1.6 Protection

File protection controls who can access or modify a file. Protection mechanisms include:

- **File Permissions**: Users can be granted different levels of access (e.g., read, write, execute) based on their role (owner, group, or others).
- **Access Control Lists (ACLs)**: Provide more fine-grained control, specifying which users or groups can access the file and what operations they can perform.
- **Encryption**: Files can be encrypted to ensure confidentiality and prevent unauthorized access.

6.2 Implementing File Systems

This section discusses how file systems are structured and implemented, including methods for organizing files and managing disk space.

6.2.1 File System Structure

A file system is composed of several components:

- **File Control Block (FCB)**: Stores metadata about a file, including its size, location, and access information.
- **Inode (Index Node)**: Contains information about the file's properties, such as its size, owner, access permissions, and pointers to data blocks.
- **Superblock**: Contains vital information about the file system, such as the total number of blocks, available blocks, and block size.

6.2.2 File System Implementation

File systems are implemented using various methods and structures, including:

- **File Allocation Table (FAT)**: A table that keeps track of the allocation of disk blocks for files. FAT is used in older systems like MS-DOS.
- **NTFS**: The New Technology File System, which uses a **Master File Table (MFT)** to store file metadata and supports advanced features like file compression and encryption.
- **ext3/ext4**: Extensible file systems used in Linux, which support journaling to improve reliability and performance.

6.2.3 Directory Implementation

Directories are implemented using:

- **Linked List**: A simple list of file names and pointers to their inodes.
- **Hash Table**: A more efficient method of implementing directories by hashing the file names to locate file metadata quickly.

6.2.4 Allocation Methods

Files are allocated disk space using various methods:

- **Contiguous Allocation**: Files are stored in contiguous blocks on the disk. This method allows for fast sequential access but suffers from fragmentation.
- **Linked Allocation**: Files are stored in scattered blocks, with each block containing a pointer to the next. This method reduces fragmentation but increases access time due to pointer traversal.
- **Indexed Allocation**: Files have an index that maps to their data blocks. It combines the benefits of both contiguous and linked allocation and allows for efficient access.

6.2.5 Free Space Management

Free space management involves tracking unused blocks of disk space, and methods include:

- **Bit Map**: A simple array of bits where each bit represents a block. A bit of "1" indicates a block is used, while "0" means it is free.

- **Linked List**: Each free block points to the next free block, providing a dynamic list of available space.
- **Group Block**: Groups free blocks together, with an entry for each group, improving the management of large disks.

6.3 Secondary Storage Structure

Secondary storage devices, such as hard drives or SSDs, store data persistently. Efficient secondary storage management is crucial for system performance.

6.3.1 Disk Structure

A disk consists of multiple platters, each divided into concentric circles (tracks), and each track divided into sectors.

- **Cylinder**: A stack of tracks at the same position on all platters.
- **Disk Head**: Reads and writes data to the disk.

6.3.2 Disk Scheduling

Disk scheduling algorithms determine the order in which disk I/O requests are processed, optimizing seek time and improving disk performance. Common algorithms include:

- **First-Come, First-Served (FCFS)**: Processes requests in the order they arrive.
- **Shortest Seek Time First (SSTF)**: Processes the request closest to the current head position.
- **SCAN**: Moves the disk arm in one direction, servicing requests along the way, and then reverses.
- **LOOK**: Similar to SCAN but stops when there are no more requests in the current direction.

6.3.3 Disk Management

Disk management involves handling issues like bad sectors and RAID:

- **Bad Block Management**: Bad sectors are marked and excluded from allocation.
- **RAID (Redundant Array of Independent Disks)**: A technology that uses multiple disks for redundancy and improved performance. Common RAID levels include RAID 0, RAID 1, RAID 5, and RAID 10.

6.3.4 Swap Space Management

Swap space is used when physical memory (RAM) is full, allowing the operating system to move data from RAM to the disk:

- **Swap Partition**: A dedicated disk partition for swap space.
- **Swap File**: A file that acts as swap space, used in systems where a dedicated partition is not available.

6.4 I/O Systems

An **I/O system** handles communication between the CPU and peripheral devices (e.g., disks, printers, network interfaces).

6.4.1 I/O Hardware

The **I/O hardware** includes devices such as:

- **Storage Devices** (Hard drives, SSDs)
- **Input Devices** (Keyboard, Mouse)
- **Output Devices** (Monitor, Printer)

The **I/O controller** manages communication between the CPU and the I/O devices, ensuring data is transferred correctly.

6.4.2 Application I/O Interface

The **I/O interface** provides an abstraction layer between the hardware and application programs. System calls allow applications to request file operations, such as:

- **open()**
- **read()**
- **write()**
- **close()**

The I/O interface ensures efficient and secure communication between applications and devices, providing access to files, devices, and networks.

Unit 7: Protection and Security LH 3

Unit 7 focuses on the **fundamentals of protection and security** in operating systems, emphasizing the mechanisms that safeguard resources, prevent unauthorized access, and maintain the integrity and confidentiality of data.

7.1 Fundamentals

Protection and security are essential in an operating system to ensure that only authorized users can access and modify data. Protection involves controlling access to resources, while security ensures that the system is protected from malicious attacks or unauthorized actions.

7.1.1 Policy and Mechanism

- **Policy** refers to the rules or guidelines that define who can access which resources and what actions they can perform. It answers the **who**, **what**, and **when** questions of resource access. For example, a policy might state that only administrators can access certain files, or that users can only read but not write to specific directories.
- **Mechanism** refers to the actual tools or systems used to implement the policy. Mechanisms enforce policies and typically include **authentication systems**, **authorization controls**, and **encryption methods**. These are the technical means to enforce the policies set by the system administrators.

7.1.2 Implementing Policy and Mechanism

To implement a protection policy, the operating system uses various mechanisms. These include:

- **Access Control Lists (ACLs)**: Lists that specify which users or groups can access a resource and the type of access they have (read, write, execute).
- **Capability Lists**: These are used to describe the access rights granted to a particular user for a specific resource, providing more fine-grained control compared to ACLs.
- **Role-Based Access Control (RBAC)**: Users are assigned roles, and each role has a defined set of permissions.

By combining policy definitions with appropriate mechanisms, the operating system can enforce rules that ensure users only perform authorized actions on resources.

7.1.3 Authentication Mechanisms

Authentication ensures that the identity of a user, process, or system is verified before granting access. Common authentication mechanisms include:

- **Passwords**: The most common form of authentication, though vulnerable to attacks like brute force or phishing.

- **Biometric Authentication**: Uses unique biological traits such as fingerprints or facial recognition to authenticate users.
- **Multi-factor Authentication (MFA)**: Combines two or more authentication factors, such as something the user knows (password), something the user has (security token), and something the user is (biometric data).
- **Public Key Infrastructure (PKI)**: Involves the use of asymmetric cryptography, where users authenticate using public and private key pairs.

These mechanisms ensure that only authorized users can access the system or resources.

7.1.4 Authorization Mechanisms

Authorization determines what an authenticated user can do once they have access. It controls **permissions** to resources based on the user's identity or role.

- **Access Control Lists (ACLs)**: Define permissions for users or groups on specific resources. For example, an ACL may specify that a user can read but not write to a file.
- **Role-Based Access Control (RBAC)**: Controls access based on the roles assigned to users, ensuring that users can only perform actions relevant to their roles.
- **Discretionary Access Control (DAC)**: Gives the resource owner the ability to determine who can access the resource.
- **Mandatory Access Control (MAC)**: A stricter model where the operating system enforces access controls based on system-wide policies.

Authorization ensures that users can only access what they are allowed to based on their rights or roles, minimizing the risk of unauthorized actions.

7.1.5 Encryption

Encryption is the process of converting data into a scrambled form to prevent unauthorized access. Only authorized parties with the correct key can decrypt the data and read it.

- **Symmetric Encryption**: Both the sender and the receiver use the same key to encrypt and decrypt data. The main challenge is securely exchanging the key.
- **Asymmetric Encryption**: Uses a pair of keys: a public key for encryption and a private key for decryption. This is commonly used in systems like **Secure Socket Layer (SSL)** or **Public Key Infrastructure (PKI)**.
- **End-to-End Encryption**: Ensures that data is encrypted at the source and only decrypted at the destination, preventing any intermediaries from accessing the data in transit.

Encryption helps protect sensitive information, ensuring its confidentiality even if the data is intercepted during transmission.

Unit 8: Device Management

Unit 8 focuses on **device management** in operating systems, which is responsible for coordinating and controlling the various hardware devices connected to a computer. Efficient device management ensures that I/O operations are carried out smoothly, and devices are utilized without conflicts.

8.1 Device Management Approaches

8.1.1 I/O System Organization

The **I/O system** is responsible for managing communication between the computer and its external devices, such as printers, disk drives, keyboards, and display monitors. It organizes the flow of input and output (I/O) data between the hardware devices and software applications. The system typically includes:

- **Device Controllers**: Hardware components that manage a specific device.
- **Device Drivers**: Software that enables communication between the operating system and device controllers.
- **I/O Buffers**: Temporary storage for data being transferred between the device and the computer.

I/O system organization focuses on managing device access, scheduling I/O operations, and optimizing communication between software and hardware components.

8.1.2 Direct I/O with Polling

Direct I/O with polling is a method where the CPU continuously checks the status of a device at regular intervals. The CPU queries the device to see if it is ready for an I/O operation (e.g., reading or writing data). The process involves:

1. **Polling Loop**: The CPU repeatedly checks the status of the device (whether it's ready or not).
2. **Delay**: The CPU wastes processing power checking devices, which can result in inefficiency, especially for slow devices.

While simple, polling is not efficient, especially for devices that require little I/O interaction, as the CPU spends a lot of time checking for device readiness.

8.1.3 Interrupt-Driven I/O

In **interrupt-driven I/O**, the CPU does not continuously poll devices. Instead, the device sends an **interrupt** signal to the CPU when it is ready to perform an I/O operation. This approach has several advantages over polling:

- The CPU can perform other tasks until the device is ready, leading to better resource utilization.

- The device can notify the CPU immediately when it needs attention, making I/O operations faster and more efficient.

When the interrupt occurs, the CPU stops its current task, saves its state, and processes the I/O request before returning to the previous task.

8.1.4 Memory-Mapped I/O

Memory-mapped I/O uses the computer's memory address space to access I/O devices. In this approach, device registers are mapped into the system's memory, allowing the CPU to read from or write to the devices as though they were memory locations. This provides:

- **Speed**: Faster access to I/O devices compared to traditional methods because memory access is typically faster than using I/O instructions.
- **Simplicity**: Using memory operations to handle device data eliminates the need for separate I/O instructions.

For example, a specific memory address could be reserved for a device's control register, and the CPU can directly read or write to that memory location to control the device.

8.1.5 Direct Memory Access (DMA)

Direct Memory Access (DMA) is a technique that allows peripherals (such as disk drives or network cards) to directly transfer data to and from memory without involving the CPU. DMA increases efficiency by:

- **Freeing the CPU**: The CPU is not involved in the data transfer process, allowing it to perform other tasks simultaneously.
- **Faster Data Transfer**: DMA is more efficient than using interrupt-driven or polling methods because data transfer occurs directly between memory and I/O devices.

DMA is typically used for large data transfers, such as disk-to-memory or memory-to-network transfers, where the CPU's involvement is minimal.

8.2 Device Drivers

8.2.1 The Device Driver Interface

A **device driver** is a specialized program that allows the operating system to communicate with hardware devices. The **device driver interface (DDI)** is the set of functions provided by the driver that the operating system uses to interact with the device. The interface typically includes:

- **Initialization**: Functions to initialize and configure the device.
- **Control Operations**: Functions to start and stop I/O operations, configure the device, and handle interrupts.
- **Error Handling**: Functions to manage errors related to device operations.

Device drivers abstract the low-level details of hardware devices, making it easier for applications to interact with hardware.

8.2.2 CPU-Device Interactions

The **CPU-device interaction** is a crucial aspect of device management. The CPU communicates with devices via the following:

- **Direct Communication**: The CPU issues commands to the device via memory-mapped I/O or port I/O.
- **Interrupts**: Devices signal the CPU when they need attention, and the CPU processes the interrupt and responds accordingly.
- **Buffering**: Data is often temporarily stored in **buffers** to facilitate smooth data transfers between the CPU and the device.

This interaction involves efficient scheduling, data transfer, and synchronization to ensure smooth communication between the CPU and devices.

8.2.3 I/O Optimization

I/O optimization refers to improving the performance of I/O operations. It involves techniques to minimize the latency and maximize the throughput of I/O devices, such as:

- **Caching**: Frequently used data can be stored in memory for faster access.
- **Buffering**: Temporary storage of data to improve the performance of data transfer operations.
- **Request Merging**: Combining multiple I/O requests into a single request to reduce the overhead of initiating multiple I/O operations.

Optimizing I/O operations is crucial in systems with frequent or large-scale data transfers to improve overall system performance.

8.3 Some Device Management Scenarios

8.3.1 Serial Communications

Serial communication involves transferring data one bit at a time over a single communication channel. It is commonly used in devices such as modems, keyboards, and mice. Device management in serial communication involves:

- **Data Transmission Protocols**: Such as UART (Universal Asynchronous Receiver-Transmitter) to manage data timing and flow.
- **Error Checking**: Ensuring that data is transmitted correctly using techniques like checksums or parity bits.

Serial communication typically uses interrupt-driven I/O to efficiently handle data transmission without wasting CPU cycles.

8.3.2 Sequentially Accessed Storage

Sequentially accessed storage refers to devices where data is accessed in a specific, ordered sequence, such as **tape drives**. Device management for such storage systems focuses on:

- **Efficient Data Access**: Ensuring that data is retrieved in the correct order and optimizing the movement of storage media.
- **Buffering and Caching**: Reducing the time spent waiting for data to be retrieved by pre-fetching or buffering data.

Managing sequential storage involves understanding the physical layout of the device and the most efficient way to access the data stored in sequence.