UNIT I

INTRODUCTION TO OPERATING SYSTEMS

Introduction: Defining Operating Systems - Operating System objectives and functions - The evolution of Operating Systems - Operating System operations - Operating System structures: Operating System Services - System calls-System programs- Operating System structure-Developments leading to modern Operating Systems - Virtual machines- OS design considerations for multiprocessor and multi core – Operating System generation - System boot.

Introduction of Operating System

An operating system acts as an intermediary between the user of a computer and computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner.

An operating system is software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.

Operating System–Definition:

- An operating system is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware.
- A more common definition is that the operating system is the one program running at all times on the computer (usually called the kernel), with all else being application programs.

An operating system is concerned with the allocation of resources and services, such as memory, processors, devices, and information. The operating system correspondingly includes programs to manage these resources, such as a traffic controller, a scheduler, memory management module, I/O programs, and a file system.

Operating system as User Interface

1. User
2. System and application programs
3. Operating system
4. Hardware

Every computer must have an operating system to run other programs. The operating system manages hardware resources, coordinating their use among system programs, application programs, and multiple users. Essentially, it provides an environment where other programs can execute efficiently.

An operating system consists of specialized programs that enable a computer to function properly. It performs essential tasks such as recognizing input from the keyboard, managing files and directories on the disk, sending output to the display screen, and controlling peripheral devices.

The operating system serves two primary purposes:

1. It controls the allocation and utilization of computing resources among various users and tasks.
2. It provides an interface between the computer hardware and programmers, simplifying coding, program creation, and debugging.

Objectives and Functions of an Operating System

An Operating System (OS) is essential for managing computer hardware and software resources while providing a user-friendly interface. It serves as an intermediary between users and computer hardware, ensuring efficient operation.

Resource management is one of the primary functions of an operating system (OS). It ensures that computer resources such as the CPU, memory, storage, and input/output devices are allocated and used efficiently among multiple users and processes.

Functions of Operating Systems

1.Processor (CPU) Management
- The OS schedules and allocates CPU time among multiple processes.
- Uses process scheduling algorithms like First-Come-First-Serve (FCFS), Round Robin, and Priority Scheduling.
- Ensures fair execution of processes using multitasking and multiprocessing.

2. Memory Management
- Manages allocation and deallocation of RAM to processes.
- Implements virtual memory to extend RAM using disk storage.
- Uses techniques like paging and segmentation to optimize memory use.

3. File System Management
- Controls storage, organization, retrieval, and access to files.
- Implements file access permissions for security.
- Uses file allocation methods like contiguous, linked, and indexed allocation.

4. Device Management
    - Manages hardware devices such as printers, keyboards, and hard drives.
    - Uses device drivers to enable communication between hardware and software.
    - Uses techniques like spooling to queue up tasks (e.g., print jobs).
5. I/O Management
    - Controls input and output operations efficiently.
    - Handles buffering, caching, and spooling for better performance.
    - Coordinates with drivers to interact with hardware devices.
6. Network Resource Management
    - Manages communication between computers over a network.
    - Implements protocols and security to control data transmission.
    - Facilitates resource sharing over networks.
7. Error Detection and Handling
    - Monitors system performance and detects software/hardware failures.
    - Takes corrective actions to prevent system crashes.
    - Logs errors and alerts users when issues occur.

Techniques Used in Resource Management

1. Multiprogramming – Keeps the CPU busy by running multiple programs simultaneously.
2. Multitasking – Allows multiple processes to run concurrently, giving the illusion of parallel execution.
3. Time-Sharing – Allocates CPU time slices to multiple processes for fair execution.
4. Paging & Segmentation – Efficient memory allocation techniques to optimize RAM usage.
5. Spooling – Stores tasks temporarily in memory (e.g., print jobs) for efficient execution.

<u>The evolution of Operating Systems - Generations of Operating Systems</u>

First Generation (1940 – Early 1950s)

The first generation of operating systems emerged during the 1940s and early 1950s, a period when computers were in their infancy. These early systems were primarily used for scientific calculations and lacked the sophisticated operating systems we have today.

Characteristics of First-Generation Operating Systems:

- No Operating Systems: Computers operated without dedicated OS software. Instead, users manually controlled the hardware using plugboards and switches.

- Vacuum Tube Technology: Early computers relied on vacuum tubes, which were large, slow, and prone to failure.
- Batch Processing Beginnings: Programs were loaded manually onto the machine, executed, and then manually removed before running the next task.
- Machine Language Programming: Users wrote programs in machine language or assembly, making programming a complex and time-consuming task.
- Limited User Interaction: There was no direct user interface or multitasking—only one program could run at a time.

Notable First-Generation Computers:

- ENIAC (Electronic Numerical Integrator and Computer) – One of the first general-purpose electronic computers.
- EDSAC (Electronic Delay Storage Automatic Calculator) – Introduced the concept of stored-program computing.
- UNIVAC I (Universal Automatic Computer I) – One of the first commercially available computers.

Second Generation (1955 – 1965)

The second generation of operating systems emerged between 1955 and 1965, coinciding with the transition from vacuum tube-based computers to transistor-based systems. This era marked significant advancements in computing, particularly in automation and processing efficiency.

Characteristics of Second-Generation Operating Systems:

- Batch Processing Systems**:** Programs were collected in batches and executed sequentially, reducing manual intervention.
- Introduction of Transistors: Computers became smaller, faster, and more reliable compared to vacuum tube-based machines.
- Use of Assembly Language**:** Programming became slightly easier with the shift from pure machine language to assembly language.
- Job Control Language (JCL): Early forms of JCL were used to manage job execution in batch processing.
- Improved I/O Operations: Card readers, tape storage, and printers were integrated for better data input and output.
- No Interactive User Interface: Systems were still operated using punch cards, and there was no direct user interaction.

Notable Computers of This Era:

- IBM 1401 – One of the most widely used second-generation computers.
- IBM 7094 – A powerful system that introduced more advanced batch processing techniques.
- UNIVAC 1107 – One of the early machines to implement transistor technology.

Third Generation (1965 – 1980)

The third generation of operating systems, spanning from 1965 to 1980, introduced significant advancements in computing. This era was marked by the transition from batch processing **to** multiprogramming and time-sharing systems**,** allowing multiple users to interact with the computer simultaneously.

Key Characteristics of Third Generation Operating Systems:

- Multiprogramming: Multiple programs could be loaded into memory and executed concurrently, improving CPU utilization.
- Time-Sharing Systems (TSS): Enabled multiple users to access the system simultaneously via terminals, leading to the development of interactive computing.
- Introduction of Integrated Circuits (ICs): Computers became more powerful, reliable, and cost-effective.
- Disk-Based Operating Systems: Faster and more efficient data storage and retrieval replaced earlier tape-based storage systems.
- Development of Standardized Operating Systems**:** IBM introduced **OS/360**, one of the first operating systems designed for different hardware models, setting the stage for compatibility across systems.
- Real-Time and Multi-User Systems: Real-time OS designs emerged, particularly for military, industrial, and scientific applications.
- Command-Line Interfaces (CLI): Users could interact with the system using text-based commands, leading to more user-friendly computing.

Notable Operating Systems and Computers of This Era:

- IBM OS/360 – One of the first multipurpose operating systems.
- UNIX (1969) – A groundbreaking OS developed by AT&T Bell Labs, forming the foundation for modern operating systems.
- Multics (Multiplexed Information and Computing Service) – A pioneering OS that influenced UNIX.

- PDP-11 & VAX Systems – Minicomputers that introduced more advanced OS capabilities.

Fourth Generation (1980 – Present Day)

The fourth generation of operating systems began in the 1980s and continues to evolve today. This era is characterized by the rise of personal computing, graphical user interfaces (GUIs), networking, and mobile computing**.** With advancements in microprocessors, distributed computing, and artificial intelligence**,** operating systems have become more powerful, efficient, and user-friendly.

Key Characteristics of Fourth Generation Operating Systems:

- Graphical User Interface (GUI): Replaced traditional command-line interfaces, making computers easier to use.
- Personal Computing: Operating systems were designed for home users and businesses, leading to widespread adoption.
- Networking & Internet Integration: Support for LAN (Local Area Networks), WAN (Wide Area Networks), and the Internet became a standard feature.
- Multitasking & Multithreading: Allowed multiple applications and processes to run simultaneously.
- Distributed Computing: Enabled computers to work together across networks, improving performance and resource utilization.
- Security Enhancements: Introduction of firewalls, encryption, and access controls to protect user data and systems.
- Real-Time Operating Systems (RTOS): Used in embedded systems, medical devices, and industrial automation for time-sensitive operations.
- Mobile Operating Systems: The emergence of Android, iOS, and Windows Mobile revolutionized smartphones and tablets.
- Cloud Computing: Modern OS platforms integrate cloud storage, virtualization, and remote computing capabilities.
- Artificial Intelligence & Machine Learning Integration: AI-powered features enhance performance, security, and automation.

Notable Operating Systems of the Fourth Generation:

- Microsoft Windows (1985 – Present) – Dominant OS for personal computers.
- macOS (1984 – Present) – Apple's OS, known for its user-friendly design and security.

- Linux (1991 – Present) **–** Open-source OS widely used in servers, cloud computing, and supercomputers.
- Android (2008 – Present) – Leading mobile OS for smartphones and tablets.
- iOS (2007 – Present) – Apple's mobile OS, optimized for iPhones and iPads.
- UNIX & Its Variants – Still widely used in servers, enterprise systems, and cybersecurity.

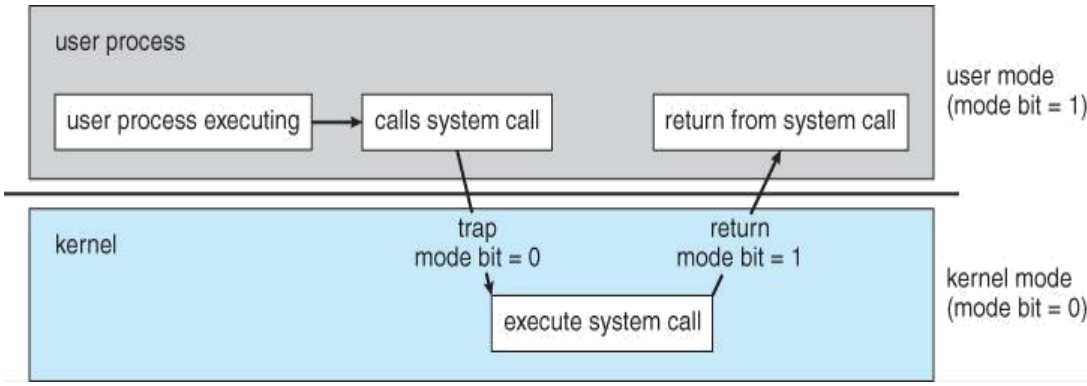
Operating System operations

- Interrupt driven by hardware.
- Software error or request creates exception or trap.
- Division by zero, request for operating system service.
- Other process problems include infinite loop, processes modifying each Other or the operating system.
- Dual-mode operation allows OS to protect itself and other system Components.
- User mode and kernel mode.
- Mode bit provided by hardware.
- Provides ability to distinguish when system is running user code or kernel code.
- System calls changes mode to kernel, return from call resets it to user.

Dual-Mode Operation

In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user defined code. The approach taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution. When the computer system is executing on behalf of a user application, the system is in user mode.

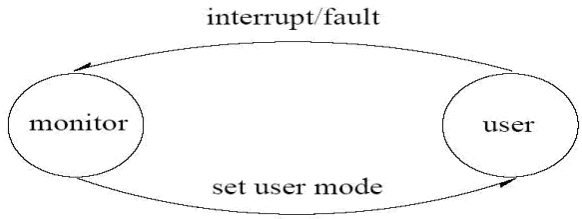
- However, when a user application requests a service from the operating system (via a system call), it must transition from user to kernel mode to fulfill the request.
- At system boot time, the hardware starts in kernel mode.
- The operating system is the n loaded and starts user applications in **user mode**.
- Whenever a trap or interrupt occurs, the hardware switches from **user mode to kernel mode** (that is, changes the state of the mode bit to 0).
- Thus, whenever the operating system gains control of the computer, it is in **kernel mode**.
- The system always **switches to user mode** (by setting the mode bit to1) before passing control to a user program.



At system boot time, the hardware starts in kernel mode. The Operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode. Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode before passing control to a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as privileged instructions. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functional it provided by the underlying processor.



When a system call is executed, it is treated by the hardware as software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit

is set to kernel mode. The system call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call. The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system.

Timer

The operating system maintains control over the CPU. We must prevent a user program from getting stuck in an infinite loop or not calling system services and never returning control to the operating system. To accomplish this goal, we can use a timer. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A variable timer is generally implemented by a fixed-rate clock and a counter.

The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond. Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Thus, we can use the timer to prevent a user program from running too long. A simple technique is to initialize a counter with the amount of time that a program is allowed to run.

Operating System structures: Operating System Services

An Operating System (OS) provides an environment for program execution and offers various services to ensure the system operates efficiently. These services can be categorized into different types, depending on their purpose. These services include process management, which ensures smooth execution of multiple processes through scheduling and synchronization. Memory management allocates and deallocates memory to processes, preventing conflicts and optimizing resource utilization. File system management enables users and applications to store, retrieve, and manage data efficiently. Device management controls hardware components by interacting with device drivers to ensure proper communication between the OS and peripherals. Security and access control protect system resources by implementing authentication, authorization, and encryption mechanisms. User interface services provide interaction methods, including command-line interfaces (CLI) and graphical user interfaces (GUI). Additionally, error detection and handling help identify and respond to system

failures, ensuring reliability. These essential services enable the OS to create a stable, efficient, and user-friendly computing environment.

Operating system services are essential for efficient system operation and user interaction. These services can be categorized into User-Oriented Services and System-Oriented Services. User-Oriented Services focus on providing a seamless user experience and facilitating program execution. Program execution allows programs to be loaded into memory and ensures they run smoothly without interfering with other processes. I/O operations manage input and output activities, handling user interactions with devices such as keyboards, printers, and disk drives. File system manipulation enables programs to read, write, create, delete, and manage files while enforcing file permissions and access control. Communication services facilitate inter-process communication (IPC) by allowing processes running on the same or different machines to exchange data through methods such as message passing and shared memory. Error detection and handling continuously monitor the system for hardware and software errors, taking necessary actions to recover from failures and maintain system reliability.

To facilitate user interaction with these services, operating systems provide different types of interfaces. The Command-Line Interface (CLI) enables users to interact with the system by typing commands directly, commonly seen in Linux Terminal and Windows Command Prompt. The Graphical User Interface (GUI) offers a visually intuitive environment using windows, icons, and menus, as seen in operating systems like Windows and macOS. The Batch Interface allows the execution of predefined scripts or batch files without requiring real-time user input, making it useful for automating repetitive tasks. Additionally, system calls serve as a bridge between applications and the operating system, enabling programs to request various OS services such as file operations, memory management, and process control. These services collectively ensure that the operating system functions efficiently, managing resources, executing user commands, and maintaining security to provide a stable and user-friendly computing experience.

Operating system services are broadly classified into User-Oriented Services and System-Oriented Services, both of which play a crucial role in ensuring a smooth and efficient computing experience.

Types of User Interfaces for OS Services

Operating systems provide different types of user interfaces to allow users and applications to interact with system services efficiently. These interfaces serve as a medium between users, software, and hardware, enabling effective control and management of system resources.

One of the most fundamental interfaces is the Command-Line Interface (CLI), which allows users to type text-based commands to perform various operations such as file manipulation, process control, and system configuration. CLI is widely used in operating systems like Linux, UNIX, and Windows Command Prompt. It is preferred by advanced users and system administrators due to its efficiency, flexibility, and powerful scripting capabilities. However, it has a steep learning curve as users must memorize commands and syntax.

The Graphical User Interface (GUI) is a more user-friendly interface that utilizes visual elements such as windows, icons, menus, and buttons for interaction. Operating systems like Windows, macOS, and many Linux distributions provide GUI-based environments that allow users to interact with applications using a mouse, keyboard, or touchscreen. GUI simplifies navigation and system management, making it more accessible for general users. It also supports multitasking and drag-and-drop functionality, enhancing productivity. However, GUI-based systems consume more system resources compared to CLI.

Another important interface is the Batch Interface, which allows users to execute a series of pre-written commands in a script or batch file without requiring direct user interaction. This is particularly useful for automating repetitive tasks such as system backups, log analysis, and software updates. Batch processing is widely used in enterprise environments where scheduled operations must run without manual intervention, improving efficiency and consistency. However, debugging errors in batch scripts can be challenging as they execute without real-time feedback.

System Calls serve as a low-level interface that enables applications to request services from the operating system. These are essential for performing operations such as process management, memory allocation, file handling, and device communication. System calls act as a bridge between user applications and system hardware, ensuring controlled access to resources while maintaining security and stability. They are categorized into different types, including process control calls (e.g., fork, exec), file management calls (e.g., open, read, write), device management calls, and communication-related calls. System calls are crucial for application development and are commonly used in programming languages such as C and C++.

Each of these interfaces serves a specific purpose and is suited to different types of users and applications. While CLI is preferred by power users for its speed and flexibility, GUI is widely adopted for its ease of use. Batch processing is ideal for automating tasks, and system calls provide a structured way for applications to interact with the OS. By offering multiple interfaces, operating systems cater to a diverse range of users and computing environments, ensuring efficiency, usability, and robust system control.

System calls

System Calls in Operating Systems

A system call is a mechanism that allows user-level applications to request services from the operating system's kernel. Since user applications run in user mode (with restricted access to system resources), they rely on system calls to interact with hardware and perform privileged operations. System calls act as an interface between user programs and the OS, enabling applications to perform essential tasks such as process creation, memory management, file operations, and network communication. These calls are necessary because user applications cannot directly access critical system resources, ensuring security, stability, and controlled access to hardware components such as the CPU, disk, and memory.

System calls can be categorized into several types based on their functionality. Process Control System Calls include commands such as fork(), exec(), exit(), and wait(), which are used for creating, executing, terminating, and synchronizing processes. These calls allow applications to manage processes efficiently, enabling multitasking and parallel execution of tasks. File Management System Calls, such as open(), read(), write(), and close(), enable applications to create, modify, and delete files. They provide a structured way for programs to handle file I/O operations, ensuring proper access control and data integrity. Device Management System Calls include functions like ioctl(), read(), and write(), which allow applications to interact with input and output devices such as keyboards, printers, and disk drives. These calls ensure that multiple processes can share hardware resources without conflicts.

In addition, Information Maintenance System Calls, such as getpid(), gettimeofday(), and set priority(), allow applications to retrieve or modify system-related information, such as process IDs, system time, and scheduling priorities. Communication System Calls, including send(), recv(), socket(), and connect(), enable inter-process communication (IPC) and network interactions, facilitating data exchange between processes running on the same machine or different systems over a network. These calls are essential for distributed computing and client-server applications.

Types of System Calls

System calls serve as a critical interface between user applications and the operating system, enabling programs to interact with hardware and system resources. These calls can be broadly categorized based on their functionality, which includes **process control, file management, device management, inter process communication (IPC), and information maintenance**. Each category plays a vital role in managing system operations, ensuring efficient execution, security, and resource management.

1. Process Control

Process control system calls manage the lifecycle of processes, including creation, execution, synchronization, and termination. These system calls enable multitasking by allowing multiple processes to run concurrently.

- **fork()** – Creates a new child process that is a duplicate of the parent process. This is essential for multitasking and parallel execution of tasks.
- **exec()** – Replaces the current process with a new program, allowing process execution to switch dynamically.
- **exit()** – Terminates a process, releasing resources allocated to it.
- **wait()** – Suspends execution of a process until a child process completes, ensuring synchronization.
- **kill()** – Sends a signal to terminate or control a process, useful for handling unresponsive or unnecessary processes.

2. File Management

File management system calls enable programs to interact with the file system by creating, reading, writing, and deleting files. These calls ensure data persistence and provide structured file access.

- **open()** – Opens a file for reading, writing, or both.
- **read()** – Reads data from a file into memory.
- **write()** – Writes data from memory to a file.
- **close()** – Closes an open file, freeing resources.
- **unlink()** – Deletes a file from the system.

3. Device Management

These system calls enable programs to interact with hardware devices such as disk drives, printers, and network interfaces. The OS manages devices using device drivers, and applications can communicate with devices through system calls.

- **ioctl()** – Sends control commands to a device, used for low-level hardware control.
- **read() / write()** – Reads or writes data from/to a device.
- **open() / close()** – Opens or closes a device file for interaction.

- **pipe()** – Creates a unidirectional communication channel between related processes.

- **shmget() / shmat()** – Creates and attaches shared memory segments, enabling fast data exchange.
- **msgget() / msgsnd()** – Enables message-based communication through message queues.
- **socket() / connect()** – Establishes network communication between processes on different machines.

5. Information Maintenance

These system calls retrieve or modify system-related information, such as process details, system time, and performance statistics.

- **getpid()** – Retrieves the process ID of the calling process.
- **getppid()** – Retrieves the parent process ID.
- **gettimeofday()** – Gets the current system time.
- **sysinfo()** – Provides system statistics such as memory usage and CPU load.

These system calls are useful for debugging, performance monitoring, and resource optimization in operating systems.

System Call Execution Process

The execution of a system call follows a structured process to transition from user mode to kernel mode and back:

1. A **user application** calls a **library function** (e.g., printf(), which internally calls write()).
2. The library function **triggers a system call** by executing a **software interrupt** (trap instruction).
3. The OS **switches to kernel mode**, validates the request, and executes the required operation.
4. Once completed, the OS **returns control to user mode**, and the application resumes execution.

This transition mechanism ensures that applications can access system services without directly interacting with kernel resources, maintaining system stability and security.

User-Level vs. Kernel-Level Mode

To ensure security and system integrity, modern operating systems operate in two distinct modes:

- **User Mode:** Applications run in this restricted mode with limited access to system resources. Any request requiring privileged operations must go through system calls.
- **Kernel Mode:** The OS runs in this mode, granting full access to system resources. System calls execute in kernel mode to perform privileged operations securely.

By enforcing this separation, operating systems prevent unauthorized access to critical resources and ensure smooth, controlled execution of processes.

System programs

System Programs in Operating Systems

System programs are essential software utilities that provide an interface between users and the operating system, helping in system management, file handling, process control, and security enforcement. These programs enhance system usability and efficiency by offering command-line tools, graphical utilities, and background services. They ensure smooth interaction between hardware and software while automating routine tasks.

Types of System Programs

File Management Programs
These programs assist users in managing files and directories by providing commands to create, delete, modify, and organize data efficiently. They play a crucial role in file system navigation and maintenance.

Examples:

1. ls (Linux) / dir (Windows) – Lists files in a directory.
2. cp (Linux) / copy (Windows) – Copies files.

Example Command:

```
mkdir new_folder  # Creates a directory named "new_folder"
```

Status Information Programs

These programs provide real-time data about system performance, hardware resource utilization, and active processes. They help users and administrators monitor the health of the system.

Examples:

1. top – Displays active processes and CPU/memory usage.
2. ps – Shows currently running processes.

Process Management Programs

These programs help manage system processes by allowing users to start, stop, and monitor applications. They enable prioritization, scheduling, and termination of processes.

Examples:

1. kill – Terminates a process forcefully.
2. nice – Adjusts the priority of a running process.

Example Command:

```
kill -9 1234  # Kills process with PID 1234
```

Communication Programs

These programs facilitate communication between users and systems by enabling messaging, file transfers, and network connectivity testing.

Examples:

1. ping – Tests network connectivity.
2. ftp – Transfers files over the internet.

```
ping google.com # Checks connectivity with Google servers
```

Program Development Tools

These tools support software development by providing compilers, debuggers, and text editors to write, compile, and debug programs efficiently.

Examples:

1. gcc – C compiler.

2. gdb – Debugger for debugging programs.

```
gcc program.c -o output  # Compiles a C program
```

System Configuration and Utility Programs

These programs enable users to configure system settings, manage services, and automate tasks, ensuring smooth system operation.

Examples:

1. sysctl – Modifies kernel parameters.
2. service / systemctl – Manages system services (start, stop, restart).

Security and Authentication Programs

These programs enhance system security by enforcing authentication, managing permissions, and restricting unauthorized access.

Examples:

1. passwd – Changes user passwords.
2. chmod – Modifies file permissions.

```
chmod 755 script.sh  # Grants execute permission to a script
```

Importance of System Programs

System programs are essential for the efficient functioning of an operating system. They simplify system administration, enhance security, optimize performance, and automate routine tasks. By providing an interface for users to interact with system resources, they enable smooth execution of processes, maintenance of files, and communication across networks. Whether used in personal computing, enterprise environments, or cloud-based systems, system programs play a fundamental role in ensuring an organized and secure computing experience.

Operating System Structure

The structure of an operating system (OS) refers to how its components are organized, how they interact, and how system resources are managed to ensure the efficient operation of a computer system. The architecture of an OS is critical in determining its overall performance, stability, scalability, and security. Various OS structures offer different advantages and challenges, and

understanding these trade-offs can help in selecting or designing an OS that best meets the needs of a specific environment or application.

Types of Operating System Structures

Monolithic Structure:

The monolithic OS structure is a single, large program that integrates all OS components into a unified kernel. This design allows direct communication between all components, such as process management, memory management, device drivers, and file systems, all running in kernel mode.

Advantages:

High performance: Since all components communicate directly within the kernel, this leads to faster execution and fewer overheads associated with inter-process communication.
Simplicity in design: A monolithic kernel can be simpler to design and implement initially, especially for smaller systems that don't require modularity.

Layered Structure:

In a layered OS, the system is divided into layers that each perform specific functions. Each layer only communicates with adjacent layers, helping to abstract the system's complexity and enabling the implementation of high-level services over basic hardware.

Advantages:

Modularity: The layered approach simplifies the OS, allowing for easier modification and updates. New layers or services can be added without disturbing the existing ones.
Improved security and stability: Since each layer is independent, failure in one layer generally doesn't affect others, improving system reliability.

Microkernel Structure

A microkernel OS keeps only the most essential services within the kernel, such as process scheduling and inter-process communication, while other services (like device drivers and file systems) run in user space. This design promotes stability and security.

Advantages:

Stability and security: Since the kernel is minimal, a failure in user-space services does not affect the kernel or other services, leading to improved overall stability.
Easier updates: With minimal services in the kernel, adding or modifying services is simpler and does not require altering the core system.

Hybrid Structure

A hybrid OS combines the features of both monolithic and microkernel architectures. It integrates essential system services into the kernel for performance but allows other services to run in user space for flexibility.

Advantages:

Balanced performance and modularity: The system can leverage the speed of monolithic kernels while maintaining the flexibility and security benefits of a microkernel approach.
Enhanced reliability and flexibility: Critical services are integrated for performance, while other components can be modified or extended independently.

Developments leading to modern Operating Systems - Modular Structure

The modular OS approach is similar to the monolithic design but emphasizes the use of modules, where each system component is an independent, loadable unit. Modules can be added or removed dynamically as needed, providing flexibility.

Advantages:

Flexibility and extensibility: New modules can be introduced without disrupting the core system. This makes it easier to maintain and update specific parts of the OS.
Improved security: Modules can be isolated from each other, allowing the system to operate more securely by limiting the risk of widespread failures.

Virtual Machine Structure

A virtual machine OS creates an abstraction layer over the hardware, allowing multiple OS instances (virtual machines) to run on the same physical machine. This structure provides a high degree of flexibility and isolation between different operating systems.

Advantages:

Isolation and flexibility: Each virtual machine operates as though it is running on its own hardware, which improves security and allows different OSes to run simultaneously on the same machine.
Resource optimization: Multiple virtual machines can share the same hardware resources, making it easier to allocate resources efficiently and run different applications.

Virtual Machines

Virtualization creates a virtual layer that allows operating systems or applications to function independently of the physical hardware.

A **virtual machine setup** consists of several key components:

Host: The physical hardware that runs virtual machines.

Virtual Machine Manager (VMM) / Hypervisor: Manages and creates virtual machines, providing an interface similar to the host system.

Paravirtualization: A variation where the virtual interface differs from the actual host.

Each **guest process** operates within a virtualized copy of the host system, usually as a separate operating system.

A **single physical machine** can run multiple operating systems simultaneously within their respective virtual machines.

Virtualization blurs the definition of an operating system:

Example: VMware ESX

- Installed directly on hardware and runs from boot.
- Provides essential services such as memory management and scheduling.

The way **Virtual Machine Managers (VMMs)** are implemented can differ significantly:

1. One approach involves **hardware-based solutions**, where virtualization support is built directly into the system's firmware. These types of VMMs are typically used in **mainframes** and **medium to large-scale servers** and are commonly referred to as **Type 0 hypervisors**.

(a)Non virtual machine. (b)Virtual machine.

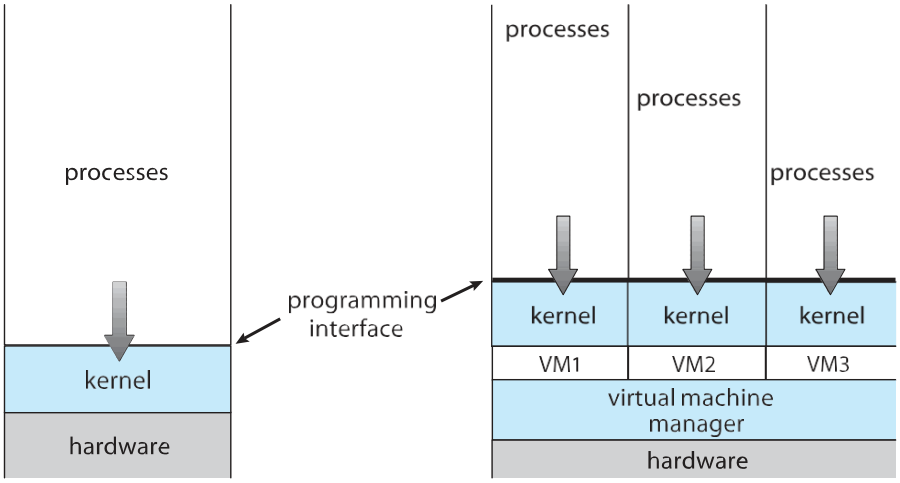


Figure: System models.

2. Operating System-Like Virtualization Software: These systems are designed specifically for virtualization, functioning similarly to an operating system. Examples include VMware ESX, Joyent SmartOS, and Citrix XenServer. These are classified as Type 1 hypervisors.

3. General-Purpose Operating Systems with Virtualization Capabilities: Some operating systems incorporate virtualization features alongside their standard functions. Examples include Microsoft Windows Server with Hyper-V and Red Hat Linux with KVM. Since they provide similar functionality to Type 1 hypervisors, they also fall under the Type 1 category.

4. Application-Based Virtualization (Type 2 Hypervisors): These VMMs function as software applications that run on top of a standard operating system while enabling virtualization for guest OS instances. Examples include VMware Workstation, VMware Fusion, Parallels Desktop, and Oracle VirtualBox.

5. Paravirtualization: This technique requires modifications to the guest operating system, allowing it to collaborate with the hypervisor for improved performance.

6. Programming-Environment Virtualization: Instead of simulating actual hardware, this method creates a tailored virtual environment optimized for specific applications. Examples include Oracle Java and Microsoft .NET.
7. Emulation: This method enables software written for one hardware platform to run on a completely different type of hardware, such as a different CPU architecture.

8. Application Containment: While not true virtualization, this approach isolates applications from the underlying operating system, enhancing security and manageability. Examples include Oracle Solaris Zones, BSD Jails, and IBM AIX WPARs.

The diversity of virtualization techniques highlights its significance in modern computing. Virtualization plays a crucial role in data centre management, software development, testing, and application deployment, making it an essential technology across various industries.

Benefits and Features

A key benefit of virtualization is the strong isolation it provides between the host system and virtual machines, as well as among the virtual machines themselves. If a virus infects a guest operating system, it may harm that specific OS but is unlikely to impact the host or other virtual machines. Since each virtual machine operates in a nearly independent environment, security risks and potential interference between them are minimized.

A common capability in most virtualization systems is the ability to pause or suspend an active virtual machine. While many operating systems offer similar functionality for individual processes, Virtual

Machine Managers (VMMs) extend this feature by enabling the creation of copies or snapshots of the entire guest system.

These snapshots allow for seamless cloning or migration of a virtual machine to a different system while preserving its current state. Once transferred, the virtual machine can resume operation exactly as if it were still running on the original hardware.

The requirements for virtualization specify that:

1. A Virtual Machine Monitor (VMM) must create an environment for programs that closely mirrors the original system.
2. Software running in this environment should experience only slight performance reductions.
3. The VMM must have full authority over system resources.

Types of VMs and Their Implementations

1.**The Virtual Machine Life Cycle**

Virtual Machine Creation

- The life cycle of a virtual machine begins when it is created.
- The system creating the VM provides the Virtual Machine Monitor (VMM) with specific parameters.
    - These parameters include:
    - Number of virtual CPUs
    - Amount of memory (RAM)
    - Networking configuration
    - Storage details
- Example: A user may create a VM with 2 virtual CPUs, 4GB RAM, 10GB disk space, a single network interface using DHCP, and DVD drive access.
- The VMM then uses these parameters to set up the virtual machine.

2. Resource Allocation and Hypervisor Types

- **Type 0 Hypervisor:**
    - Resources are **dedicated** and must be available at the time of VM creation.
    - If required resources (e.g., two virtual CPUs) are not free, the creation request will fail.
- **Other Hypervisor Types:**
    - Resources may be either **dedicated or virtualized**, depending on the hypervisor's design.
    - IP addresses **cannot** be shared.
    - Virtual CPUs are often **multiplexed** across physical CPUs.

- Memory management may allow **more virtual memory than physically available**; a process explained in Section 18.6.2.

3. Virtual Machine Deletion

- When a VM is no longer needed, it can be removed.
- The VMM first **reclaims** any allocated disk space.
- It then **removes the configuration**, effectively erasing the virtual machine.
- This process is far simpler than decommissioning a physical machine.

4. Cloning and Virtual Machine Sprawl

- A VM can be **cloned** from an existing one by simply selecting a "clone" option.
- The user assigns a **new name and IP address** for the cloned VM.
- While convenient, easy VM creation can lead to **virtual machine sprawl**:
  - Too many VMs accumulate.
  - Tracking their usage, history, and status becomes difficult.

Type 0 Hypervisor

Type 0 hypervisors, also known as **"partitions"** or **"domains,"** have been in use for many years. They are built into hardware, which provides both advantages and limitations. Unlike other hypervisors, operating systems do not require special modifications to use them.

The Virtual Machine Monitor (VMM) is embedded in the system firmware and loads automatically during boot-up. It then launches guest operating systems in separate partitions.

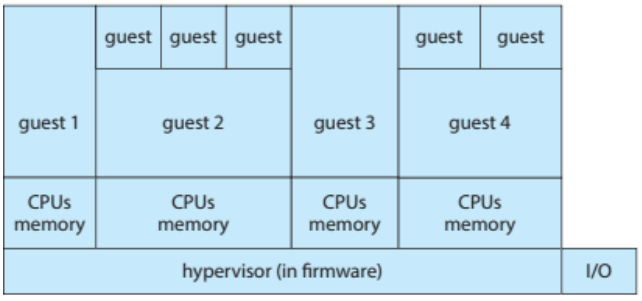


Figure: Type 0 hypervisor

- **Hardware-Based:** Since Type 0 hypervisors are implemented in hardware, they typically offer fewer features than software-based alternatives.
- **Dedicated Resources:** A system using a Type 0 hypervisor can be divided into multiple virtual systems, each with its own CPUs, memory, and I/O devices. Each guest system perceives these resources as physical hardware, simplifying implementation.

Challenges with I/O Management

- **Limited I/O Devices:** If the number of guests exceeds available I/O devices, resource allocation becomes a challenge. For example, if a system has only two Ethernet ports but multiple guests, each guest cannot have a dedicated port.

- **Shared Access:** To resolve this, the hypervisor can manage shared access or assign all devices to a **control partition.** In this setup, one guest OS handles I/O operations, such as networking, and provides services to other guests.
- **Dynamic Resource Allocation:** Some advanced Type 0 hypervisors can reassign CPUs and memory between running guests. In these cases, **Para virtualized** guests can detect hardware changes and adjust their allocated resources accordingly.

Nested Virtualization Capability

- Since Type 0 hypervisors run directly on hardware, they differ from other virtualization methods.
- They allow **nested virtualization,** meaning each guest OS can act as a VMM itself and run additional virtual machines.
- This feature is unique to Type 0 hypervisors, as most other hypervisors do not support virtualization within virtualization.

Type 1 Hypervisor

Type 1 Hypervisors as Data-Centre Operating Systems

Type 1 hypervisors are commonly found in company data centres, often becoming "the data-centre operating system." These hypervisors are special-purpose operating systems that run directly on the hardware, without an underlying host OS. Instead of providing system calls for running programs, they focus on creating, running, and managing guest operating systems. They can operate on standard hardware or on Type 0 hypervisors, but not on other Type 1 hypervisors. Regardless of the platform, guest operating systems typically do not realize they are running on anything other than native hardware.

Functionality and Operation

Type 1 hypervisors run in kernel mode and utilize hardware protection features when available. They may also take advantage of multiple modes on the host CPU to give guest operating systems their own control and enhanced performance. These hypervisors implement device drivers for the hardware they operate on since no other component could provide this function. As full-fledged operating systems, Type 1 hypervisors manage essential functions such as CPU scheduling, memory management, I/O management, protection, and security. They often offer APIs to support applications within guest systems or external applications providing additional features like backups, monitoring, and security.

Commercial and Open-Source Variations

Many Types 1 hypervisors are closed-source commercial products, like VMware ESX. However, some are open-source or hybrid, such as Citrix Xen Server and its open-source Xen counterpart. These hypervisors are valuable tools for data-center managers, enabling more efficient control and management of operating systems and applications.

Benefits of Type 1 Hypervisors in Data Centers

Type 1 hypervisors enable data-center managers to consolidate multiple operating systems and applications onto fewer systems, improving resource utilization. For example, instead of running ten

systems at 10% utilization each, a data center might use one server to handle the entire load. When system utilization increases, guest systems and applications can be moved to less-loaded servers without interrupting service. Additionally, the use of snapshots and cloning allows the states of guest operating systems to be saved and duplicated easily, making recovery simpler than restoring from backups or performing manual installations.

Costs and Complexity

The main trade-off for these increased benefits is the cost of the VMM (if it's a commercial product), the need to learn new management tools, and the added complexity of the system.

General-Purpose Operating Systems with VMM Functionality

Another type of Type 1 hypervisor includes general-purpose operating systems that also provide VMM functionality. Operating systems like RedHat Enterprise Linux, Windows, or Oracle Solaris can run as both the host OS and provide virtualization capabilities for guest operating systems. These hypervisors typically offer fewer virtualization features compared to other Type 1 hypervisors because they perform additional duties. They tend to treat guest operating systems as regular processes, with special handling for when guests attempt to execute specific instructions.

Type 2 hypervisors are less engaging for those studying operating systems because they involve minimal interaction with the host OS.

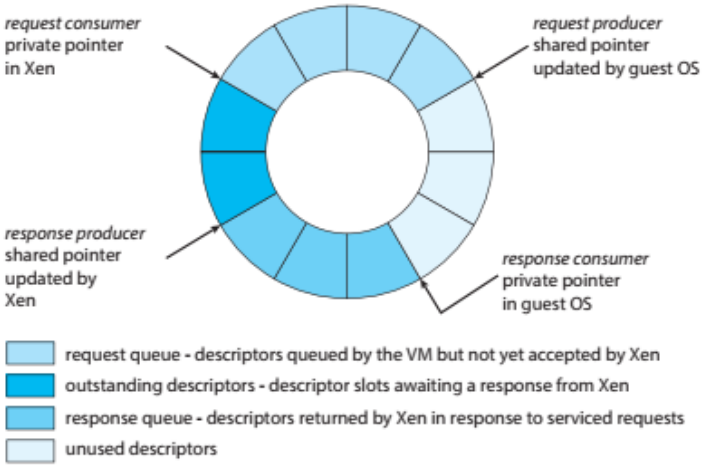


Figure: Xen I/O via shared circular buffer.

A key limitation of Type 2 hypervisors is that many of the advanced hardware features, such as those provided by modern CPUs, require administrative privileges. Without these privileges, the VMM cannot access or utilize those features fully. Additionally, since Type 2 hypervisors rely on running both a general-purpose operating system and guest systems, they experience higher overhead, resulting in lower performance compared to Type 0 or Type 1 hypervisors.

Despite these drawbacks, Type 2 hypervisors come with some advantages. They are compatible with a wide range of general-purpose operating systems, and their use doesn't require altering the host system. This makes them ideal for students who want to explore non-native operating systems without changing their primary OS. For example, a student with an Apple laptop can easily run versions of Windows, Linux, Unix, and other operating systems for learning and experimentation.

Para virtualization

Paravirtualization differs from other virtualization methods in that it doesn't attempt to fool the guest operating system into thinking it has exclusive access to a system. Instead, it presents the guest with a system that resembles, but is not identical to, its ideal environment. This requires modifying the guest to operate on the Para virtualized virtual hardware. The benefit of this extra effort is more efficient resource utilization and a leaner virtualization layer.

Xen became a leader in paravirtualization by employing various techniques to enhance the performance of both guest and host systems. Unlike some VMMs that create virtual devices which appear identical to real hardware, Xen opted for simpler and more efficient device abstractions. This approach allowed better I/O performance and smoother communication between the guest and the VMM. For each device used by the guest, a circular buffer was shared between the guest and the VMM via shared memory, facilitating read and write operations as shown in Figure 18.6.

For memory management, Xen didn't use nested page tables. Instead, each guest had its own set of page tables, which were read-only. When a page-table modification was necessary, the guest had to use a specific mechanism: a hyper call to the hypervisor VMM. This required modifications to the guest operating system's kernel code, adapting it to Xen's unique methods. To boost performance, Xen allowed the guest to queue multiple page-table changes asynchronously through hyper calls, checking that the changes were complete before continuing.

Xen enabled x86 CPU virtualization without relying on binary translation, instead requiring changes to the guest operating systems as described. Over time, Xen has leveraged hardware virtualization features, reducing the need for modified guests and making the paravirtualization method unnecessary. However, paravirtualization is still utilized in some solutions, such as Type 0 hypervisors.

Programming-Environment Virtualization

Another form of virtualization operates under a different execution model, focusing on virtualized programming environments. In this approach, a programming language is designed to function within a specialized virtual environment. A prime example is Oracle's Java, which relies on the Java Virtual Machine (JVM) for essential features such as security protocols and memory management.

If virtualization is strictly defined as replicating hardware, this concept might not qualify. However, a broader perspective allows for the creation of virtual environments based on APIs, offering specific functionalities tailored to a language and the programs written in it. Java programs, for instance, execute within the JVM, which itself is compiled as a native application on various operating systems.

This setup enables Java programs to be written once and executed on any system where a JVM is present. Similarly, interpreted languages operate within programs that process each instruction and translate it into native commands.

Emulation

Virtualization is one of the most common techniques for running applications designed for a specific operating system on a different one, provided they share the same CPU. This approach is generally efficient because the applications are compiled for the instruction set used by the target system.

However, when an application or operating system needs to function on a different type of CPU, the process becomes more complex. In such cases, all instructions from the original CPU must be translated into their equivalents for the new CPU. This process goes beyond virtualization and falls under full emulation.

Emulation becomes essential when the host and guest systems are built on different architectures. For instance, if a company upgrades to a new computing system but still needs to run older software that was developed for the previous hardware, an emulator can be used. This emulator translates instructions from the outdated system into a format the new system can understand. Emulation extends the usability of legacy software and allows users to experience older computing environments without needing the original hardware.

The biggest challenge with emulation is performance. Because each instruction from the older system must be interpreted and converted into multiple instructions for the new system, the process can be significantly slower. If it takes ten instructions on the new hardware to execute a single instruction from the old system, performance may suffer unless the new system is substantially faster. Additionally, developing an accurate emulator is a complex task, as it essentially requires recreating an entire CPU in software.

Despite these difficulties, emulation remains widely used, especially in gaming. Many classic video games were designed for discontinued platforms, but enthusiasts can still play them using emulators. Modern hardware is powerful enough that even devices like the Apple iPhone can run game emulators smoothly.

Application Containment

In some cases, the purpose of virtualization is to isolate applications, manage their performance and resource consumption, and provide a convenient way to start, stop, relocate, and oversee them. However, full-scale virtualization may not always be necessary. When all applications are built for the

same operating system, alternative methods such as application containment can achieve these goals without requiring complete virtualization.

A notable example of application containment is found in Oracle Solaris, starting from version 10. This system introduced containers, or "zones," which create a virtual layer between the operating system and the applications.

Unlike traditional virtualization, this approach does not involve virtualizing hardware. Instead, the operating system and its associated resources are virtualized, giving each zone the illusion that it is operating independently. Each zone can have its own applications, network configurations, user accounts, and allocated CPU and memory resources. Additionally, individual zones can run their own schedulers to enhance performance based on their assigned resources.

Compared to full virtualization, containers are much more lightweight. They require fewer system resources, are quicker to create and remove, and function similarly to processes rather than virtual machines. Due to these advantages, containers have become increasingly popular, particularly in cloud computing. The FreeBSD operating system was one of the first to introduce a container-like system called "jails," while IBM's AIX platform also offers a similar feature. Linux incorporated container technology in 2014 with the introduction of LXC (Linux Containers), which is now a standard feature in many Linux distributions through the clone() system call.

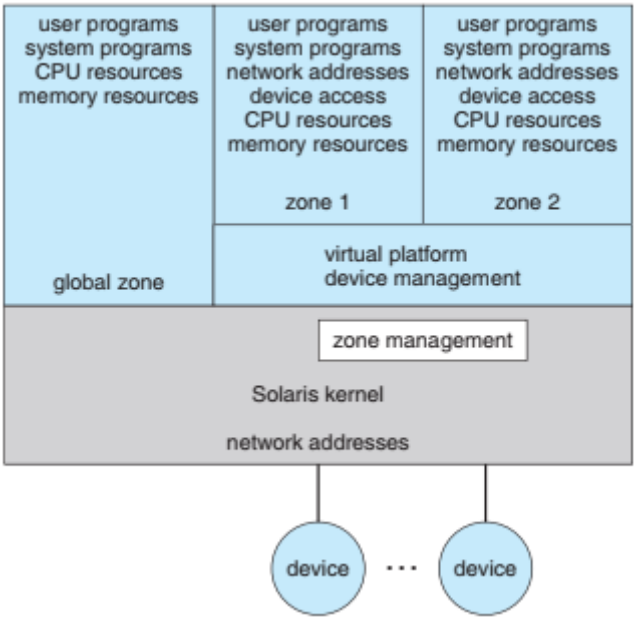


Figure: Solaris 10 with two zones.

Another key benefit of containers is their ease of automation and management. This has led to the development of orchestration tools such as Docker and Kubernetes. These tools simplify the deployment and coordination of complex applications composed of multiple containers, much like

how an operating system manages individual programs. They allow for rapid deployment, monitoring, and administration of distributed applications, making them essential in modern cloud environments.

Examples

VMware

VMware Workstation is a widely used commercial software that enables the virtualization of Intel x86 and compatible hardware, allowing multiple operating systems to run as separate virtual machines. It is a prime example of a Type 2 hypervisor, meaning it operates as an application within a host operating system, such as Windows or Linux, while managing multiple guest operating systems simultaneously.

The structure of this system can be visualized with Linux as the host operating system, running various guest operating systems like FreeBSD, Windows NT, and Windows XP. The core of VMware Workstation is its virtualization layer, which abstracts physical hardware to create independent virtual machines. Each virtual machine has its own virtualized components, including a CPU, memory, storage, and network interfaces, making it function as if it were a separate physical system.

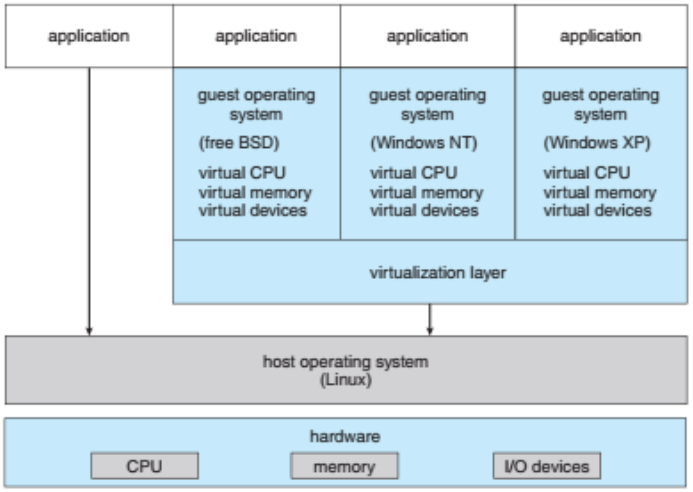


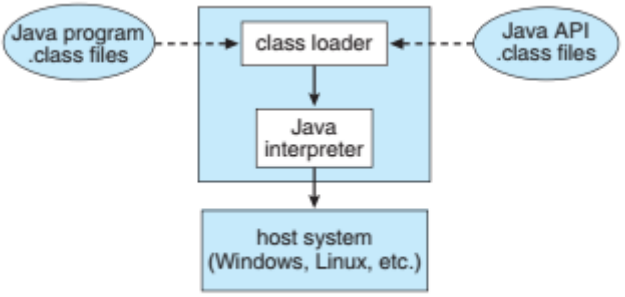
Figure: VMware Workstation architecture.

One of the key features of VMware Workstation is how it handles virtual storage. The storage assigned to a guest operating system is actually a file within the host's file system. This setup makes it easy to create copies of virtual machines by duplicating the file, ensuring quick recovery in case of system failure. Additionally, moving the file to another location effectively transfers the entire guest system, enhancing flexibility in system administration and resource management.

The Java Virtual Machine

Java is a widely used object-oriented programming language that was first introduced by Sun Microsystems in 1995. Along with a detailed language specification and an extensive API library, Java includes a specification for the Java Virtual Machine (JVM).

In Java, objects are defined using the class construct, and a Java program is typically composed of one or more classes. When a Java program is compiled, it generates architecture-neutral bytecode files (.class), which can be executed on any system that has a JVM implementation.
The JVM acts as a virtual computing environment, consisting of a class loader and a Java interpreter that executes the platform-independent bytecode. The process begins when the class loader loads the compiled .class files, including those from both the Java program and the Java API. Once loaded, the verifier checks the validity of the bytecode, ensuring it does not cause stack overflows, underflows, or perform pointer arithmetic that could lead to unauthorized memory access. If the bytecode passes verification, it is executed by the Java interpreter.
Another key feature of the JVM is its automatic memory management through garbage collection. This process reclaims memory from objects that are no longer in use and returns it to the system, preventing memory leaks. Significant research has been dedicated to optimizing garbage collection techniques to improve the performance of Java applications running within the JVM.



The Java virtual machine.

The Java Virtual Machine (JVM) can be implemented in different ways. It may run as software on top of an operating system such as Windows, Linux, or macOS, or it can be integrated into a web browser. Alternatively, the JVM can be built directly into hardware using a specialized chip designed specifically for running Java programs.
When implemented in software, the Java interpreter processes bytecode instructions one at a time, which can be slow. A more efficient approach is to use a just-in-time (JIT) compiler. With JIT compilation, the first time a Java method is called, its bytecode is converted into native machine code for the host system. This compiled code is then stored in memory, allowing future calls to the method to execute directly using the native instructions instead of reinterpreting the bytecode each time.
A hardware-based JVM offers even greater performance advantages. In this setup, a dedicated Java chip executes bytecode instructions as native machine code, eliminating the need for both interpretation and just-in-time compilation, leading to faster execution of Java programs.

OS design considerations for multiprocessor and multicore

Symmetric Multiprocessor OS Considerations

Modern computing, spanning from mobile devices to large-scale servers, is now dominated by multiprocessor systems. Traditionally, these systems consist of two or more processors, each with a single-core CPU. These processors typically share resources such as the computer bus, memory, clock, and peripheral devices. The key advantage of multiprocessor systems is improved processing efficiency—adding more processors allows more tasks to be completed in less time. However, the performance boost is not directly proportional to the number of processors due to the overhead involved in coordinating tasks and managing shared resources.

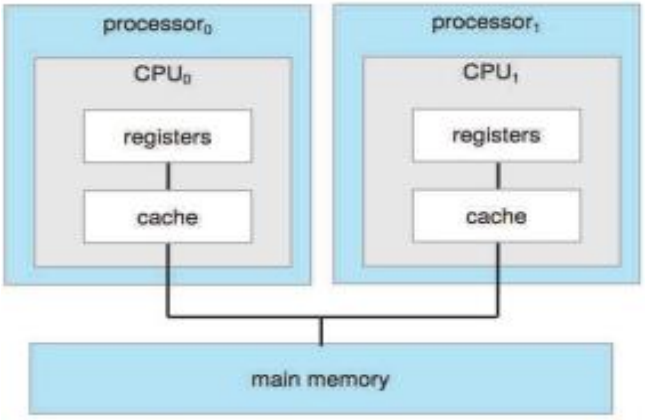


Figure: Symmetric multiprocessing architecture.

The most common type of multiprocessor system is symmetric multiprocessing (SMP), where each processor operates independently, handling both operating system functions and user processes. In a typical SMP setup, each processor has its own registers and a private cache while sharing the system's main memory through a common bus.

This architecture enables multiple processes to run simultaneously without significant performance degradation. However, if workload distribution is uneven, one processor may remain idle while another is overburdened. Efficient resource sharing can help balance workloads and minimize inefficiencies, though this requires careful system design.

Over time, the concept of multiprocessor systems has expanded to include multicore processors, where multiple computing cores are integrated into a single chip. These multicore systems often outperform traditional multi-chip single-core setups by optimizing resource sharing and reducing communication overhead between cores.

Multicore OS Considerations

A dual-core processor integrates two processing cores onto a single chip, as shown in Figure. Each core operates independently, equipped with its own set of registers and a dedicated level 1 (L1) cache. Additionally, both cores share a level 2 (L2) cache located on the chip. This architecture, commonly used in modern processors, balances performance and efficiency by combining fast, private caches with larger, shared caches. Typically, lower-level caches are smaller and quicker, while higher-level caches store more data but operate at slower speeds.

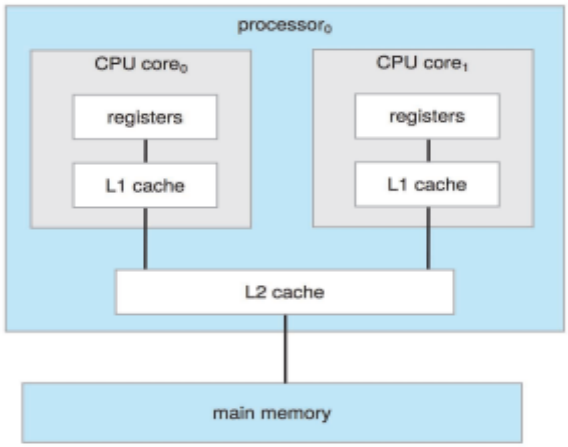


Figure: A dual-core design with two cores on the same chip

From the perspective of an operating system, a multicore processor with *N* cores functions as *N* separate CPUs. This presents a challenge for both operating system developers and application programmers, who must optimize software to take full advantage of available processing power. Virtually all modern operating systems, including Windows, macOS, Linux, Android, and iOS, support symmetric multiprocessing (SMP) with multicore architectures.

While adding more CPUs to a system can boost computational performance, it does not scale indefinitely. As more processors are introduced, competition for shared resources, such as the system bus, can cause bottlenecks and degrade overall efficiency. To address this issue, an alternative approach called non-uniform memory access (NUMA) is used.

In a NUMA system, each CPU—or a group of CPUs—has its own local memory, accessed through a high-speed bus. These CPUs are interconnected, allowing them to share a unified physical memory space.

The advantage of this design is that when a CPU accesses its local memory, performance remains high without interference from other processors. However, if a CPU needs to retrieve data from another processor's memory, access times increase, potentially slowing performance. To mitigate these delays, operating systems employ sophisticated scheduling and memory management techniques. Due to their scalability, NUMA systems are increasingly common in high-performance computing and enterprise server environments.

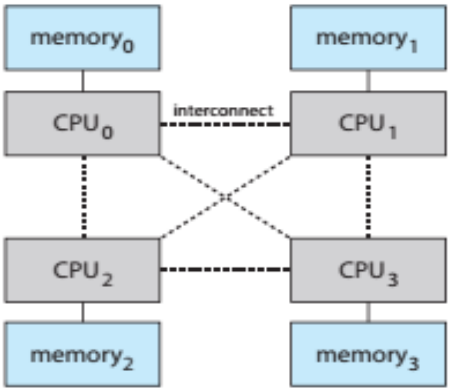


Figure: NUMA multiprocessing architecture

Another approach to multi-processor computing is the blade server, which houses multiple processor boards, I/O boards, and networking components within a single chassis. Unlike traditional multiprocessor setups, each blade operates as an independent unit with its own operating system.

Some blade servers also integrate multiprocessor configurations within individual blades, making the distinction between different computing models less clear. In essence, blade servers function as collections of independent multiprocessor systems within a shared physical infrastructure.

Operating System Generation

When you purchase a computer, it typically comes with a preinstalled operating system, such as Windows or macOS. However, if you want to replace the existing operating system, install additional ones, or set up a computer that doesn't have an OS, you'll need to go through a process to install and configure the appropriate system.

Building an Operating System from Scratch

If you're developing or assembling an operating system from the ground up, the process generally includes the following steps:

1. Writing or obtaining the source code for the operating system.
2. Configuring the OS to suit the hardware it will run on.
3. Compiling the source code into an executable system.
4. Installing the compiled OS onto the computer.
5. Booting the machine with the new operating system.

The configuration phase involves defining which features and components the OS should include. Different systems handle this step in various ways. Configuration details are typically stored in a file that can be used to modify the OS before installation.

There are three main approaches to system generation:

- **Full Compilation:** The system administrator modifies the OS source code, then compiles and builds it entirely, tailoring it to the hardware.

- **Precompiled Modules:** Instead of compiling from scratch, the system selects and links prebuilt modules from a library, making installation faster but potentially less optimized for specific hardware.
- **Modular Execution:** Some systems allow configuration changes dynamically at runtime, rather than during compilation or installation, making them highly flexible.

For embedded systems with fixed hardware, a fully compiled OS is often preferred. However, modern operating systems for desktops, laptops, and mobile devices typically use modular approaches, such as loadable kernel modules, to support hardware changes dynamically.

Installing a Linux System from Scratch

To build a Linux system from scratch, follow these steps:

Download the Linux source code from kernel.org.

1. Configure the kernel using make menu config, which creates a .config file with the selected options.
2. Compile the kernel using make, generating the kernel image.
3. Compile additional kernel modules using make modules.
4. Install the compiled modules with make modules_install.
5. Install the new kernel using make install.

Once the system reboots, it will start running the newly installed Linux OS.

Running Linux as a Virtual Machine

Instead of replacing an existing OS, you can install Linux as a virtual machine. This allows Linux to run within another operating system, such as Windows or macOS, using virtualization software.

There are two main ways to set up a Linux virtual machine:

- **Building from Scratch:** This is similar to installing Linux directly on a computer, but without the need to compile the OS.
- **Using a Preconfigured Virtual Machine Appliance:** This involves downloading a ready-made Linux system and installing it with virtualization tools like VirtualBox or VMware.

For example, to create a Linux virtual machine:

1. Download an Ubuntu ISO file from ubuntu.com.
2. Use VirtualBox to boot the virtual machine using the ISO as the installation medium.
3. Follow the installation prompts to complete the setup and start using the virtual machine.

System Boot

After an operating system has been created, it must be properly loaded for the hardware to use. But how does the system locate and start the operating system's kernel? The process of loading the kernel and starting the system is called **booting**. Typically, this follows these key steps:

1. A small program known as the **bootloader** or **bootstrap program** locates the operating system kernel.
2. The kernel is loaded into memory and executed.
3. The hardware is initialized.
4. The root file system is mounted.

Multi-Stage Booting Process

Many systems use a multi-stage boot process. When a computer is powered on, it runs a small bootloader stored in firmware, such as the **BIOS** (Basic Input/Output System). The BIOS contains minimal instructions, mainly to load a second-stage bootloader from a specific disk location known as the **boot block**. This bootloader then loads the full operating system into memory.

Newer computers have replaced BIOS with **UEFI** (Unified Extensible Firmware Interface), which offers several advantages, such as improved support for 64-bit processors, larger storage devices, and a faster boot process. Unlike BIOS, UEFI is a **complete boot manager**, making the startup process more efficient.

Regardless of whether a system uses BIOS or UEFI, the bootloader performs several tasks, such as:

- Loading the kernel file into memory.
- Running system diagnostics to check hardware functionality.
- Initializing system components like CPU registers, device controllers, and memory.
- Mounting the root file system and launching the operating system.

Booting Linux with GRUB

Linux and UNIX-based systems often use **GRUB** (Grand Unified Bootloader) as their boot manager. GRUB allows users to configure boot parameters, modify kernel settings, and choose between multiple kernels at startup. These configurations are stored in a GRUB file and loaded during boot.

For example, a Linux system's boot parameters might look like this:

plaintext

Copy Edit

BOOT_IMAGE=/boot/vmlinuz-4.4.0-59-generic

root=UUID=5f2e2232-4e47-4fe8-ae94-45ea749a5c92

Here, BOOT_IMAGE specifies the kernel to load, while root defines the unique identifier of the system's root file system.

To optimize boot time and conserve storage, Linux uses a **compressed kernel image**. During startup, the bootloader loads the compressed kernel and extracts it into memory. Additionally, a **temporary RAM-based file system** called **initramfs** is created, containing essential drivers and modules. Once the necessary components are loaded, the system transitions from initramfs to the actual root file system.

Finally, Linux launches the **system d** process, which initializes system services, such as web servers and databases, before presenting a login prompt.

Booting on Mobile Devices

The boot process for mobile operating systems differs from traditional PCs. For example, although **Android** uses a Linux-based kernel, it does not rely on GRUB. Instead, different hardware manufacturers provide their own bootloaders, with **LK (Little Kernel)** being the most commonly used for Android devices.

Android also follows a similar compressed kernel approach, but unlike Linux, it **does not discard initramfs**. Instead, Android keeps it as the **permanent root file system** for the device. Once the kernel is loaded, the init process starts, launching essential services and preparing the system to display the home screen.

Recovery Mode and Troubleshooting

Most operating systems—including Windows, Linux, macOS, iOS, and Android—offer a **recovery mode** or **single-user mode**. These options allow users to:

- Diagnose hardware issues.
- Repair corrupted file systems.
- Reinstall or restore the operating system.

UNIT II
PROCESSES AND THREADS

Processes: Process Concept-Process Scheduling-Operations on processes–Inter-process communication, Threads: Multi core programming - Multithreading models - Threading issues, CPU Scheduling: Basic concepts - Scheduling criteria - Scheduling algorithms - Thread scheduling.

Processes : Process concept

A process is a systematic series of actions or steps taken to achieve a particular goal or result. It involves a structured sequence of activities that transform inputs into outputs through a set of predefined procedures. Processes are fundamental in various fields, including business, manufacturing, computing, and daily life. They ensure efficiency, consistency, and quality by following a well-defined workflow. For example, in manufacturing, a process may involve sourcing raw materials, assembling parts, quality checks, and final packaging. In computing, a process refers to the execution of a program, where the CPU allocates resources to run tasks efficiently. In business, processes streamline operations, such as customer service or supply chain management, ensuring smooth and systematic functioning. Each process consists of multiple stages, including initiation, planning, execution, monitoring, and completion, to ensure desired outcomes. Effective process management improves productivity, reduces errors, and enhances overall performance in any domain.

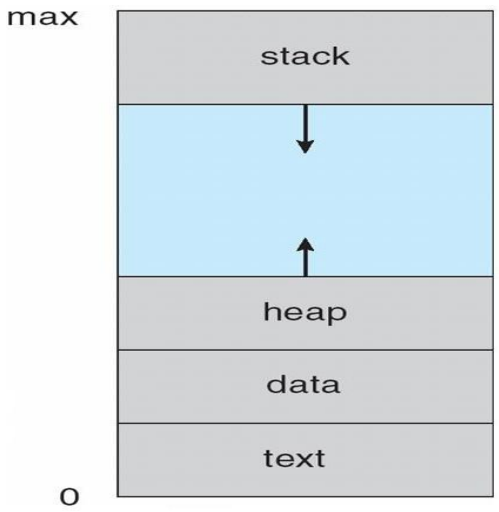
A process is a structured sequence of actions or steps designed to achieve a specific goal or output. It is a fundamental concept that applies to various fields, including business, technology, manufacturing, and daily activities. Processes help ensure efficiency, consistency, and accuracy by providing a standardized approach to completing tasks. A process can be defined as a systematic series of actions performed to convert inputs into desired outputs. It provides a framework for organizing tasks in a logical and sequential manner, aiming to improve efficiency, reduce errors, maintain quality, and ensure repeatability in achieving results.

Every process follows a logical sequence of activities that must be completed in a specific order. It starts with inputs such as raw materials, data, or information and transforms them into outputs like finished products, reports, or results. A well-defined process is repeatable, ensuring consistent outcomes. It also focuses on optimization and efficiency by streamlining operations, reducing waste, and improving productivity. Moreover, processes are subject to monitoring and control, where their performance can be measured, analyzed, and improved based on data.

Processes can be categorized based on their application and purpose. Business processes include operational activities like sales, production, and customer service, as well as supporting functions like HR management and IT support. Manufacturing processes involve transforming raw materials into finished goods through material procurement, production, assembly, quality control, and distribution. In computer and software systems, a process refers to the execution of a program, which includes

stages like creation, execution, resource allocation, scheduling, and termination. Apart from man-made processes, scientific and natural processes occur in the environment, such as photosynthesis in plants, the water cycle, and human digestion.

Multiple parts:



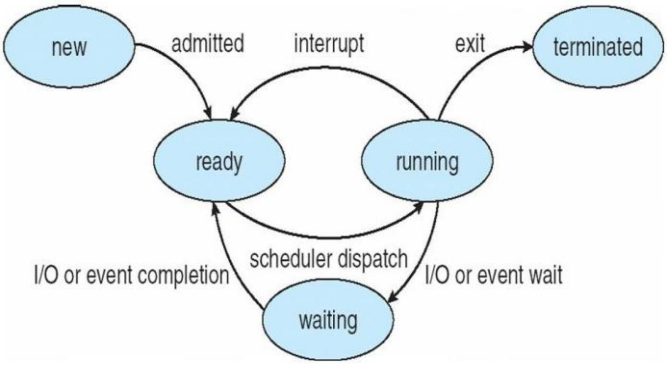
Ultimately, the process concept is about transforming inputs into desired outputs through an organized series of steps, allowing organizations and individuals to achieve their goals efficiently and effectively.

Process States:

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

1. **New:** The process i s being created.
2. **Running:** Instructions are being executed.
3. **Waiting:** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
4. **Ready:** The process is waiting to be assigned to a processor.

Terminated: The process has finished execution.



Process Control Block

- Each process is represented in the operating system by a process control block(PCB)- also called a task control block.
- APCB defines a process to the operating system.
- It contains the entire information about a process.
- Some of the information a PCB contains are:

1. Process state: The state may be new, ready, running, waiting or terminated.

2. Program counter: The counter indicates the address of the next instruction to be executed for this process.

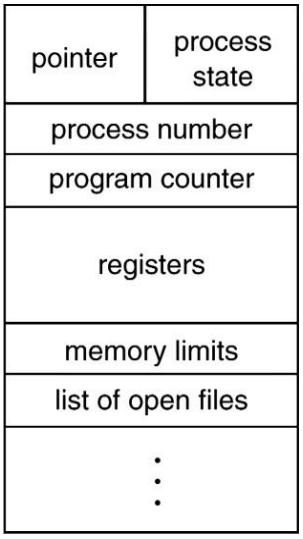
3. CPU registers: The registers vary in number and type, depending on the computer architecture.

4. CPU-scheduling information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

5. Memory-management information: This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

6. Accounting information: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

7. I/O Status information: The information includes the list of I/O devices allocated to this process, a list of open files, and so on.

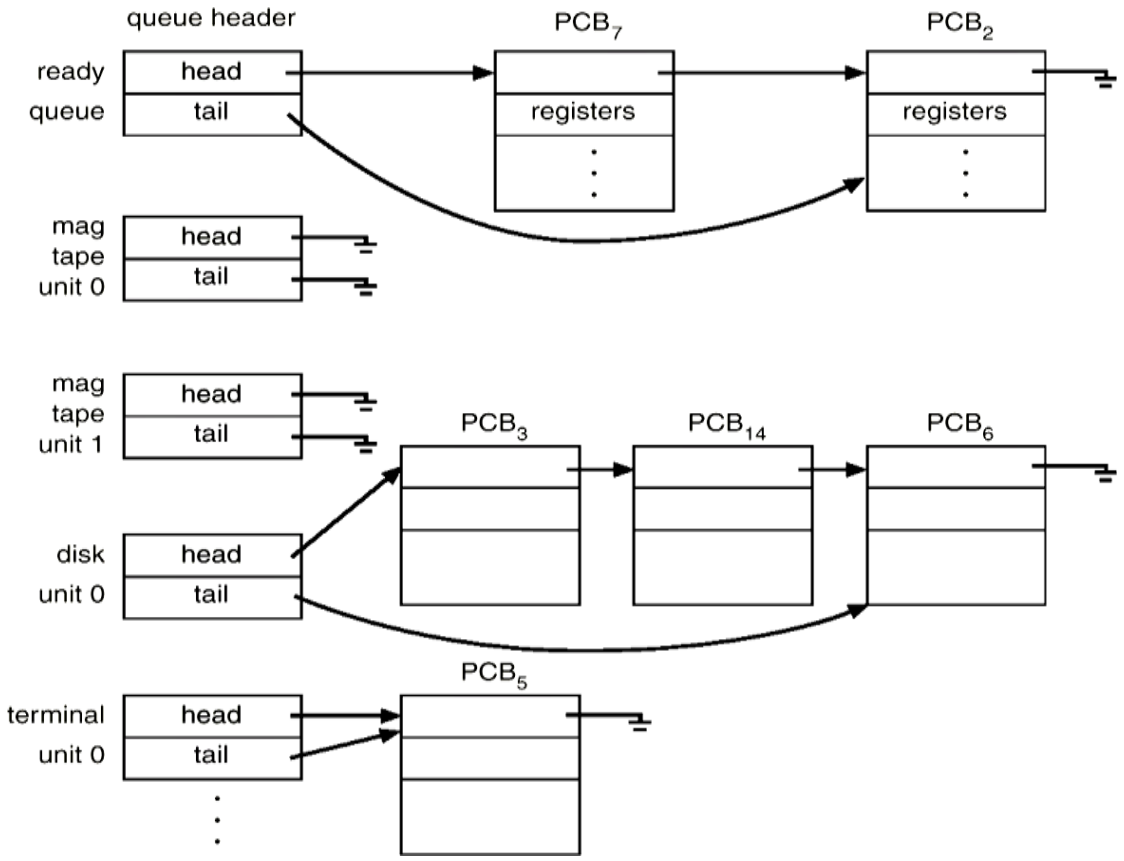


Process Control Block

Process scheduling

Process Scheduling: An Essential Component of Operating Systems

Process scheduling is a crucial function of an operating system (OS) that manages the execution of multiple processes efficiently by allocating CPU time and system resources. In a multitasking environment, multiple processes compete for CPU execution, and the OS must decide the order in which these processes will be executed. Process scheduling ensures that system resources are used optimally, reducing idle time and improving overall system performance. It plays a vital role in ensuring fairness among processes, preventing deadlocks, and maintaining system responsiveness. Process scheduling is classified into three main types: long-term scheduling, which determines which processes should be admitted into the system for execution and controls the degree of multiprogramming; short-term scheduling, which decides which process from the ready queue gets CPU time for execution and occurs frequently using algorithms like First-Come-First-Serve (FCFS), Shortest Job Next (SJN), Round Robin (RR), and Priority Scheduling; and medium-term scheduling, which involves process swapping by temporarily removing processes from memory and reloading them when necessary to optimize CPU and memory utilization. The OS maintains different process scheduling queues to manage execution efficiently, including the job queue, which holds all processes in the system; the ready queue, which contains processes that are ready for execution; the waiting queue, which stores processes waiting for I/O operations or external events; and the device queue, which holds processes waiting for access to system devices such as printers or disks.

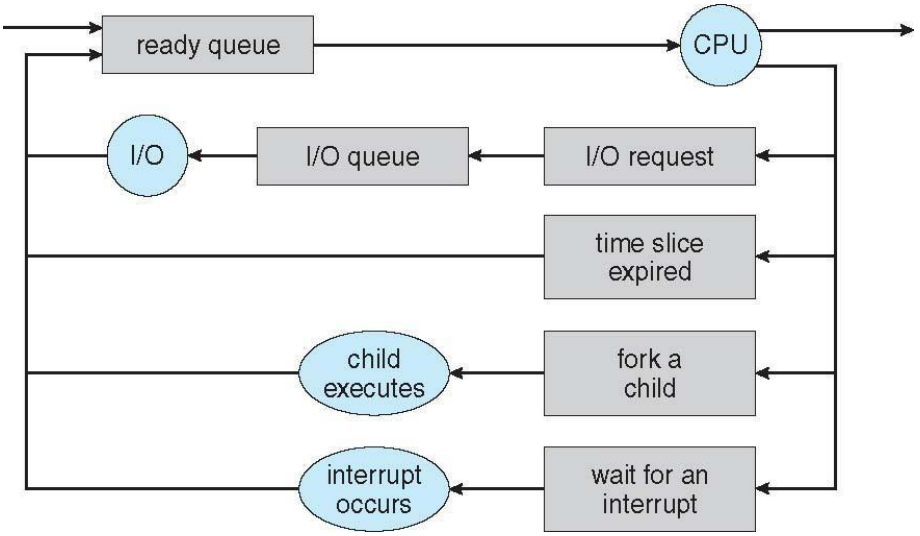


The ready queue and various I/O device queues.

To optimize execution and resource allocation, various process scheduling algorithms are implemented. First-Come-First-Serve (FCFS) executes processes in the order they arrive, ensuring simplicity but potentially leading to long waiting times due to the convoy effect. Shortest Job Next (SJN) or Shortest Job First (SJF) schedules the process with the shortest burst time first, minimizing waiting time but requiring prior knowledge of burst times. Round Robin (RR) assigns each process a fixed time slice (quantum), ensuring fairness but causing higher context-switching overhead. Priority Scheduling executes processes based on priority levels, which can lead to starvation if lower-priority processes are ignored, though aging techniques help mitigate this issue. Multilevel Queue Scheduling divides processes into multiple priority queues, each managed by different scheduling algorithms, and is commonly used in real-time systems where different priority levels are necessary.

Process scheduling is essential for optimizing CPU utilization, improving system responsiveness, ensuring fairness among processes, and preventing issues such as deadlocks and starvation. Efficient scheduling enhances multitasking, allowing multiple users and applications to run smoothly while balancing resource allocation. By implementing different scheduling algorithms and strategies,

modern operating systems maintain high performance, efficiency, and user satisfaction in both single-user and multi-user environments.



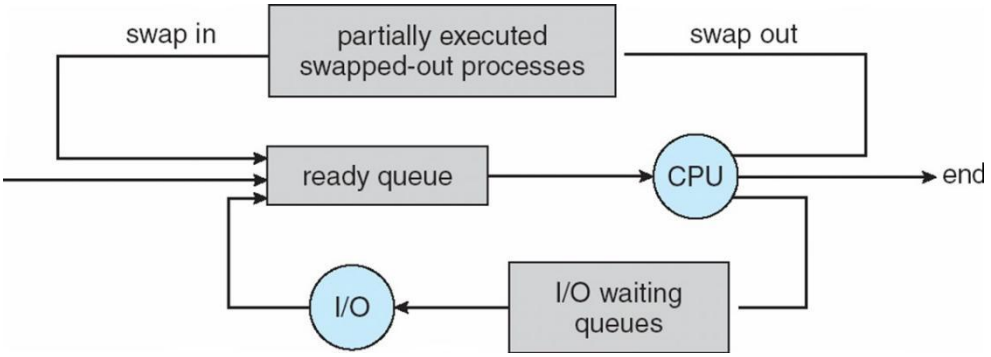
Queueing – diagram representation of process scheduling.

Process scheduling is a fundamental aspect of an operating system (OS) that manages the execution of processes by allocating system resources efficiently. The primary goal of process scheduling is to ensure optimal CPU utilization, minimize response time, and provide fairness among processes. Since modern computer systems often handle multiple processes simultaneously, an efficient scheduling mechanism is crucial to maintaining system performance and responsiveness.

At its core, process scheduling involves selecting which process will execute on the CPU at any given time. The OS maintains different types of process queues, such as the job queue (containing all processes), the ready queue (holding processes ready for execution), and various device queues (managing processes waiting for I/O operations). The scheduling process ensures that these queues are managed effectively to prevent bottlenecks and enhance multitasking capabilities.

There are three main types of scheduling: long-term scheduling, short-term scheduling, and medium-term scheduling. Long-term scheduling determines which processes are admitted into the system for execution, controlling the degree of multiprogramming. Short-term scheduling, also known as CPU scheduling, selects which process from the ready queue will be assigned to the CPU next, making decisions in milliseconds. Medium-term scheduling involves temporarily suspending processes (swapping) to free up resources, improving overall system efficiency.

Different scheduling algorithms govern how processes are selected for execution. Common algorithms include First-Come, First-Served (FCFS), which schedules processes in the order they arrive; Shortest Job Next (SJN), which prioritizes processes with the shortest execution time; Round Robin (RR), which allocates fixed time slices to each process in a cyclic manner; Priority Scheduling, which assigns priority levels to processes; and Multilevel Queue Scheduling, which categorizes processes into different queues based on their characteristics. Each algorithm has its advantages and is chosen based on system requirements and workload characteristics.



Medium-term scheduling to the queueing diagram.

Efficient process scheduling enhances system performance by ensuring fair resource distribution, reducing wait times, and improving overall responsiveness. It is particularly crucial in real-time systems, where tasks must meet strict deadlines, and in multiprocessor environments, where multiple CPUs coordinate process execution.

Operations on processes:

Processes in an operating system undergo various operations to ensure efficient execution, management, and coordination of multiple tasks. The key operations on processes include process creation, process termination, process execution, process suspension and resumption, and process communication. Process creation occurs when a new process is initiated by a parent process, leading to the formation of a hierarchical process structure. The new process, known as the child process, inherits certain attributes from the parent, such as memory space, open files, and execution rights. Operating systems use system calls like fork() in Unix/Linux to create a new process, while in Windows, functions like Create Process() are used. Process termination occurs when a process completes execution or is forcibly terminated by the system or user. Termination can happen voluntarily, such as when a process finishes its task and calls the exit() system call, or involuntarily due to errors, resource constraints, or system failures.

Another essential operation is process execution, which involves transitioning between different states—new, ready, running, waiting, and terminated—as the process interacts with the CPU and I/O devices. The scheduler manages the allocation of CPU time to processes based on scheduling policies, ensuring smooth execution. In multitasking environments, processes may be suspended and resumed, where a process is temporarily removed from execution (swapped out) to free system resources and later reloaded into memory for continuation. This is particularly useful in memory management and when high-priority tasks require immediate execution. Interprocess Communication (IPC) is another crucial operation that enables processes to exchange data and synchronize their execution. IPC mechanisms include shared memory, where multiple processes access common memory space, and message passing, where processes communicate through system-provided channels. These communication methods help coordinate complex operations in distributed systems and multitasking environments.

Efficient handling of process operations ensures optimal CPU utilization, seamless multitasking, and smooth system performance. By managing process creation, execution, termination, suspension, and communication effectively, operating systems maintain stability, responsiveness, and efficient resource allocation, contributing to a robust and reliable computing environment.

Processes are fundamental units of execution in an operating system (OS), and various operations can be performed on them to ensure efficient process management. The OS is responsible for handling processes throughout their lifecycle, from creation to termination, while also enabling inter-process communication and synchronization. These operations include process creation, termination, execution, suspension, and resumption, each playing a critical role in managing system resources and ensuring smooth multitasking.

Process creation is the first and most crucial operation in process management. When a new process is initiated, either by a user or by the system itself, the OS allocates necessary resources such as memory, CPU time, and I/O devices. A parent process creates child processes using system calls like fork() in Unix-based systems. This leads to a hierarchy where child processes inherit attributes from their parent and may execute independently or require coordination. Process creation is essential in modern computing, where multiple processes run concurrently to enhance system performance.

Once created, a process undergoes execution, where it transitions through different states such as new, ready, running, waiting, and terminated. The OS scheduler determines which process gets CPU time based on scheduling algorithms. If a process requires input/output operations, it may move to the waiting state until the required resources are available. Context switching, which involves saving and restoring process states, allows the CPU to switch between multiple processes efficiently, ensuring a seamless multitasking environment.

At times, processes may need to be suspended or resumed to optimize resource allocation. The OS can temporarily suspend a process using process suspension, either due to user intervention, priority adjustments, or system requirements. This operation is crucial in virtual memory management, where processes may be swapped in and out of memory to accommodate new tasks. Process resumption

occurs when a suspended process is reactivated, allowing it to continue execution from where it was paused. These operations help maintain system responsiveness and prevent resource starvation.

Another essential operation is process termination, which occurs when a process completes execution or is forcibly terminated due to errors, system failures, or external commands. A process can terminate voluntarily by issuing an exit system call, signaling that it has finished its task. However, in cases of abnormal termination, the OS may need to forcibly terminate a process that is causing system instability or consuming excessive resources. Upon termination, the OS deallocates all resources associated with the process and removes it from the process table, ensuring optimal resource management.

In multi-process environments, processes often need to communicate and synchronize with each other, leading to the operation of inter-process communication (IPC). IPC mechanisms, such as message passing, shared memory, and signaling, enable processes to exchange data and coordinate actions efficiently. Synchronization techniques, including semaphores and mutexes, prevent race conditions and ensure that multiple processes access shared resources in a controlled manner. These operations are critical in distributed computing, parallel processing, and multi-threaded applications, where multiple processes work together to achieve a common goal.

Overall, operations on processes form the backbone of process management in operating systems. The ability to create, execute, suspend, resume, terminate, and communicate ensures efficient resource utilization and smooth system performance. As computing systems continue to evolve, advancements in process management techniques will further enhance multitasking, responsiveness, and overall user experience.

Inter-process communication, Threads: Multi core programming

Inter-Process Communication (IPC), Threads, and Multi-Core Programming

Inter-Process Communication (IPC) refers to the mechanisms that enable processes to exchange data and synchronize their execution in a multitasking operating system. Since processes operate in separate memory spaces, IPC allows them to communicate efficiently, ensuring coordination in distributed systems, parallel computing, and client-server applications. IPC methods include shared memory, where multiple processes access a common memory region for faster communication, and message passing, where processes send and receive messages using system calls like pipes, message queues, and sockets. Other IPC techniques include semaphores, used for synchronization by controlling access to shared resources, and signals, which notify processes about specific events or interruptions.

Threads are lightweight units of execution within a process, sharing the same memory space but executing independently. Unlike processes, which require separate memory allocation, multiple threads within a single process can efficiently share data, making them faster and more resource-efficient. Threads are categorized into user-level threads, managed by user applications, and kernel-level threads, managed by the operating system. Multi-threading improves application responsiveness,

allowing concurrent execution of tasks such as UI updates and background processing. Thread models include many-to-one, where multiple user threads map to a single kernel thread; one-to-one, where each user thread corresponds to a kernel thread; and many-to-many, which balances multiple user and kernel threads for optimized performance.

Multi-Core Programming leverages modern multi-core processors to enhance performance through parallel execution. Unlike traditional single-core systems, where tasks execute sequentially, multi-core systems distribute workloads across multiple processing cores, improving speed and efficiency. Parallelism in multi-core programming can be task parallelism, where different tasks execute simultaneously on different cores, or data parallelism, where the same operation is performed on different data sets across multiple cores. Programming models such as OpenMP, MPI (Message Passing Interface), and CUDA (for GPU parallelism) enable efficient multi-core and parallel computing. However, challenges such as race conditions, deadlocks, and load balancing require careful synchronization and efficient thread management using techniques like mutexes, locks, and atomic operations.

Efficient use of IPC, threads, and multi-core programming enhances system performance, scalability, and responsiveness in modern computing. As multi-core processors become more prevalent, optimizing applications for parallel execution is essential to fully utilize hardware capabilities, reduce processing time, and improve overall computational efficiency.

Modern computing systems rely on efficient process and thread management to ensure smooth execution of applications. Inter-Process Communication (IPC), threads, and multi-core programming are essential concepts that enhance performance, enable parallelism, and facilitate resource sharing. These mechanisms help optimize system responsiveness, improve multitasking, and allow applications to leverage the full power of modern multi-core processors.

Inter-Process Communication (IPC)

Inter-Process Communication (IPC) refers to the techniques used by processes to exchange data and synchronize their execution. Since processes operate in isolated memory spaces, they require dedicated mechanisms to share information effectively. IPC is crucial in multitasking environments where multiple processes collaborate to perform complex tasks.

There are several IPC mechanisms, each suited for different use cases:

Pipes: A unidirectional communication channel allowing one process to send data to another. Commonly used in Unix-based systems for chaining commands.

Message Passing: Involves sending and receiving messages between processes, often managed by the operating system through message queues. This ensures structured and reliable communication.

Shared Memory: Provides a memory segment accessible by multiple processes, allowing fast data exchange. However, proper synchronization mechanisms like semaphores or mutexes are needed to prevent data corruption.

Sockets: Used for communication between processes over a network, enabling distributed computing and client-server models.

Signals: Lightweight notifications sent by the OS or processes to signal events such as termination or interruption.

IPC is vital for applications requiring modular design, real-time collaboration, and distributed processing. It ensures that processes can work together efficiently without violating data integrity or causing race conditions. utilization by allowing tasks like data processing, user input handling, and background computations to run simultaneously.

Multi-Core Programming

With the advent of multi-core processors, modern software must be designed to leverage multiple CPU cores for enhanced performance. Multi-core programming focuses on distributing computational tasks across multiple cores to achieve parallel execution.

Key approaches in multi-core programming include:

Parallel Processing: Dividing a task into multiple smaller tasks that execute simultaneously across different cores, improving computation speed.

Thread-Level Parallelism (TLP): Utilizing multiple threads to execute different parts of a program concurrently, reducing execution time.

Task Scheduling: The OS assigns processes and threads to available cores, balancing workload and preventing bottlenecks.

Load Balancing: Ensures that all CPU cores are utilized efficiently, preventing some cores from being overloaded while others remain idle.

Programming languages and frameworks like OpenMP, MPI, CUDA, and multi-threading libraries in Java and Python provide tools to efficiently implement multi-core processing. These technologies help optimize performance for applications such as scientific computing, gaming, artificial intelligence, and big data analytics.

Inter-Process Communication (IPC), threads, and multi-core programming are essential for modern computing, enabling efficient multitasking, improved performance, and better resource utilization. IPC ensures smooth data exchange between processes, threads enhance parallel execution

within applications, and multi-core programming maximizes CPU efficiency. As technology advances, these concepts continue to evolve, driving innovations in high-performance computing, real-time systems, and cloud-based applications. Understanding and effectively implementing these mechanisms is crucial for developing scalable and efficient software in today's multi-core era.

Multithreading models

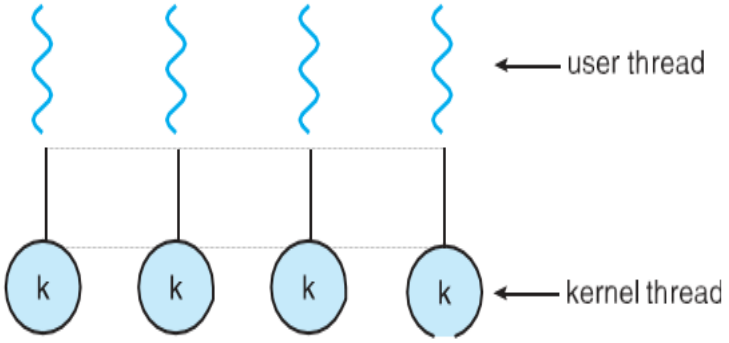
Multithreading models define how threads are managed and mapped between user threads and kernel threads within an operating system. Since user applications generate multiple threads, and the operating system schedules kernel threads for execution, efficient mapping ensures optimal performance, resource utilization, and system responsiveness. There are three primary multithreading models: Many-to-One, One-to-One, and Many-to-Many.

Many-to-One Model

In this model, multiple user threads are mapped to a single kernel thread. Since the OS does not manage user threads directly, thread switching is fast and efficient as it does not require kernel intervention. However, the major drawback is that if one thread makes a blocking system call, the entire process is blocked because only one kernel thread is handling multiple user threads. Additionally, this model does not support parallel execution on multi-core processors since only one kernel thread is available for execution at a time. This model is implemented in Green Threads (used in early Java Virtual Machines).

One-to-One Model

In this model, each user thread is mapped to a separate kernel thread, allowing true parallel execution on multi-core processors. Since the operating system manages each thread individually, blocking one thread does not affect others, making it more efficient for high-performance applications. However, the downside is that creating too many threads can overwhelm system resources, as each kernel thread requires memory and CPU scheduling overhead. This model is used in Windows, Linux, and POSIX Pthreads (Portable Operating System Interface threads).



Many-to-Many Model

This model provides a balanced approach, where multiple user threads are mapped to a smaller or equal number of kernel threads. It allows greater flexibility in managing threads, as the OS can efficiently schedule kernel threads while minimizing overhead. It supports parallel execution while preventing excessive resource consumption. A variant of this model, known as the Two-Level Model, allows some user threads to be bound directly to kernel threads, providing further optimization. This model is used in modern UNIX systems like Solaris and IRIX.

Choosing the Right Model

The choice of multithreading model depends on system requirements and application needs. Many-to-One is suitable for simple applications that do not require parallel execution, while One-to-One is ideal for multi-core systems needing high concurrency. The Many-to-Many model offers flexibility, making it a preferred choice for scalable applications requiring efficient thread management.

Efficient multithreading improves application responsiveness, performance, and resource utilization, making it a fundamental concept in modern operating systems and concurrent programming.

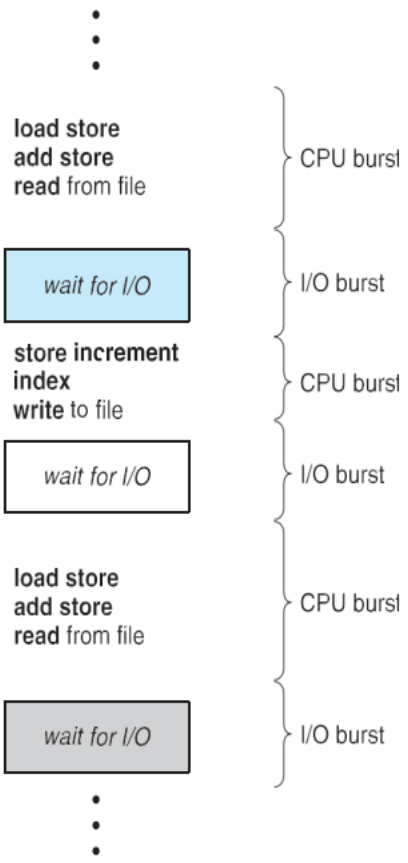
Multithreading models define how threads are managed and mapped between user space and the operating system kernel. Since threads improve performance by enabling concurrent execution, the operating system must efficiently handle thread creation, scheduling, and execution. Multithreading models fall into three primary categories: Many-to-One, One-to-One, and Many-to-Many models, each offering different trade-offs in terms of efficiency, flexibility, and resource utilization.

The Many-to-One model maps multiple user threads to a single kernel thread. In this model, thread management is handled in user space, meaning that the OS is unaware of individual threads. While this approach allows for lightweight thread creation and context switching, it has a major drawback: if one thread makes a blocking system call, all threads in the process are blocked. Additionally, it does not take full advantage of multi-core processors, as only one thread can execute at a time in the kernel. This model is primarily used in early threading libraries but is not widely adopted in modern systems due to its limitations.

The One-to-One model maps each user thread to a separate kernel thread. This approach provides better concurrency, as multiple threads can run in parallel on multi-core systems, and blocking calls affect only the thread that issued them, not the entire process. However, creating a kernel thread for each user thread increases overhead, as the OS must manage more threads, which can lead to performance issues if too many threads are created. Popular operating systems such as Windows and Linux use this model in their threading implementations, making it a common choice for applications that require high responsiveness and parallel execution.

The Many-to-Many model allows multiple user threads to be mapped to multiple kernel threads, providing a balance between efficiency and flexibility. The OS can schedule threads

dynamically, allowing for optimal resource utilization. This model enables greater scalability by limiting the number of kernel threads while still supporting high levels of concurrency. If a user thread is blocked, other threads can continue execution, improving overall performance. The Many-to-Many model is used in hybrid threading implementations, such as the two-level model, where the system can either assign user threads to kernel threads dynamically or follow a fixed mapping strategy.



Each multithreading model has its advantages and trade-offs, and the choice of model depends on the system architecture, workload requirements, and desired performance characteristics. Modern operating systems often employ variations of the One-to-One or Many-to-Many models to maximize efficiency, reduce thread management overhead, and leverage multi-core processing power effectively. As multi-threaded applications continue to evolve, advancements in thread management techniques and scheduling algorithms will further optimize system performance and responsiveness.

Threading issues, CPU Scheduling: Basic concepts:

Threading Issues: Multithreading improves performance by enabling concurrent execution, but it also introduces several challenges that must be managed for efficient execution. Some key threading issues include:

Race Conditions – Occur when multiple threads access shared resources simultaneously, leading to unpredictable results due to unsynchronized execution. Proper synchronization techniques like mutexes and semaphores help prevent this issue.

Deadlocks – Happen when two or more threads are waiting for each other to release resources, causing an infinite wait state. Deadlock prevention techniques include resource ordering, avoiding circular wait, and using timeouts.

Starvation – Occurs when a thread is indefinitely delayed because higher-priority threads keep executing. This can be mitigated using aging techniques, which gradually increase a low-priority thread's priority over time.

Livelock – Similar to deadlock but involves continuously changing states without making progress. Threads repeatedly respond to each other's actions but never proceed with execution.

Context Switching Overhead – Frequent switching between threads incurs CPU and memory overhead, reducing performance. Optimizing scheduling and minimizing unnecessary thread creation can help reduce context switching overhead.

Synchronization Overhead – While locking mechanisms prevent race conditions, excessive use of locks can slow down execution. Efficient locking strategies like lock-free programming and read-write locks help minimize this overhead.

CPU Scheduling: Basic Concepts

CPU scheduling is a fundamental aspect of an operating system that determines the order in which processes are executed by the CPU. It ensures efficient resource utilization, system responsiveness, and fairness among processes.

CPU-I/O Burst Cycle – Processes alternate between CPU execution and I/O operations. CPU-bound processes require more CPU time, while I/O-bound processes spend more time waiting for I/O. Effective scheduling balances these workloads.

Preemptive vs. Non-Preemptive Scheduling

1. Preemptive Scheduling – The OS can interrupt a running process and allocate CPU time to another process (e.g., Round Robin, Priority Scheduling). This ensures better responsiveness but increases context switching overhead.
2. Non-Preemptive Scheduling – Once a process starts execution, it runs until completion or voluntarily releases the CPU (e.g., First-Come-First-Serve, Shortest Job Next). This reduces overhead but may lead to long waiting times.

Scheduling Criteria – Different scheduling algorithms aim to optimize performance based on several criteria:

1. CPU Utilization – Keeping the CPU as busy as possible.
2. Throughput – Number of processes completed per unit time.
3. Turnaround Time – Time from process submission to completion.
4. Waiting Time – Time a process spends in the ready queue.
5. Response Time – Time from process submission to the first CPU execution.

Types of CPU Scheduling Algorithms

1. First-Come-First-Serve (FCFS) – Executes processes in arrival order but may lead to the convoy effect, where short processes wait behind long ones.
2. Shortest Job Next (SJN) / Shortest Job First (SJF) – Prioritizes shortest processes, reducing waiting time but requiring knowledge of burst times.
3. Round Robin (RR) – Assigns a fixed time slice to each process, ensuring fairness in multitasking systems.
4. Priority Scheduling – Executes higher-priority processes first, which can cause starvation for lower-priority processes.
5. Multilevel Queue Scheduling – Divides processes into different priority queues, allowing optimized scheduling for each category.

Efficient CPU scheduling enhances system performance, reduces waiting times, and ensures fair resource allocation among processes. Proper management of threading issues and scheduling policies is crucial for smooth and responsive system operation.

Threading enhances performance by allowing concurrent execution of tasks, but it also introduces several challenges that need careful management. One common issue is race conditions, which occur when multiple threads access shared data or resources simultaneously without proper synchronization, leading to unpredictable results. To prevent this, synchronization mechanisms like mutexes, semaphores, and critical sections are used to ensure that only one thread can access shared resources at a time. Deadlocks are another major problem, occurring when two or more threads are blocked indefinitely because each one is waiting for the other to release resources. Deadlock can be avoided

using strategies such as resource ordering, where resources are always requested in a specific order, and timeout mechanisms, which allow a thread to give up waiting after a certain period. Similarly, starvation happens when a low-priority thread is perpetually denied access to resources because higher-priority threads continue to execute. This issue can be addressed using aging techniques, which gradually increase the priority of a thread the longer it waits, ensuring that it eventually gets CPU time. Livelock, another threading issue, is similar to deadlock but involves threads continuously changing states without progressing, often due to each thread trying to avoid conflict with others. Avoiding livelock involves ensuring threads can make progress even when they share resources. Context switching is necessary when the CPU switches from one thread to another, but frequent context switches incur overhead, reducing system performance. Optimizing thread management and reducing unnecessary switches helps mitigate this problem. Additionally, synchronization overhead arises when multiple threads contend for access to shared resources, and excessive locking can reduce performance. Using efficient locking techniques, such as read-write locks or lock-free programming, can minimize this overhead and improve thread execution efficiency.

In terms of CPU scheduling, it is a critical part of operating system management, responsible for determining the order in which processes are assigned to the CPU. Efficient CPU scheduling ensures optimal CPU utilization, fair resource distribution, and system responsiveness. Processes generally alternate between CPU bursts (where they perform calculations) and I/O bursts (waiting for input or output operations), and the scheduler must prioritize based on these behaviors. Scheduling policies are typically divided into preemptive and non-preemptive types. Preemptive scheduling allows the operating system to interrupt a running process and allocate CPU time to other processes, which is especially important in multitasking environments where responsiveness is crucial. Algorithms such as Round Robin or Priority Scheduling are preemptive, ensuring that higher-priority tasks or tasks that have been waiting the longest are executed first. On the other hand, non-preemptive scheduling does not interrupt running processes, meaning they continue until completion or voluntarily release the CPU. This is seen in algorithms like First-Come-First-Serve (FCFS), where processes are executed in the order of their arrival.

The key scheduling criteria for determining the most appropriate scheduling algorithm include CPU utilization, which aims to keep the CPU as busy as possible; throughput, which measures the number of processes completed in a given period; turnaround time, the total time a process takes from submission to completion; waiting time, which tracks how long a process waits in the ready queue before getting CPU time; and response time, the time from submitting a process to receiving the first response from the CPU. Different scheduling algorithms seek to optimize these criteria, but there is often a trade-off between them. For example, while Shortest Job First (SJF) minimizes waiting time and is optimal in terms of throughput, it requires prior knowledge of the process's burst time, which may not always be available. Similarly, Round Robin (RR) ensures fairness by giving each process a fixed time slice, but it can cause higher context switching overhead. Other algorithms like Priority Scheduling attempt to prioritize certain processes, but they may cause starvation if lower-priority processes are perpetually delayed. To manage complex systems with varying needs, operating systems use Multilevel Queue Scheduling, where processes are divided into different priority queues, each managed by its own algorithm.

By addressing threading issues and implementing an efficient CPU scheduling strategy, operating systems can ensure smooth and optimal operation, improving system responsiveness, reducing delays, and maintaining fair resource allocation. Effective management of these processes is essential for maintaining system stability, improving application performance, and ensuring that resources are utilized in the most efficient manner possible.

While multithreading enhances performance by enabling parallel execution, it introduces several challenges that must be managed to ensure efficient and error-free operation. Common threading issues include race conditions, deadlocks, starvation, and context switching overhead.

A race condition occurs when multiple threads access shared resources concurrently, leading to unpredictable outcomes. If two or more threads attempt to modify a shared variable at the same time without proper synchronization, it can result in inconsistent data. This issue is commonly addressed using synchronization mechanisms like mutexes, semaphores, and monitors, which ensure that only one thread accesses a critical section at a time.

A deadlock arises when two or more threads are waiting for resources held by each other, causing an indefinite stall in execution. Deadlocks occur due to poor synchronization design and can be prevented using techniques such as resource ordering, deadlock detection, and timeout mechanisms.

Starvation happens when a low-priority thread is perpetually denied access to resources because higher-priority threads dominate CPU time. This can be mitigated using priority aging, where the priority of a waiting thread increases over time, ensuring it eventually gets CPU access.

Context switching overhead is another challenge in multithreading. When the CPU switches from one thread to another, it must save and restore the execution context, which introduces processing delays. While necessary for multitasking, excessive context switching can degrade performance. Optimizing thread scheduling and reducing unnecessary thread creation can help minimize this overhead.

CPU Scheduling: Basic Concepts

CPU scheduling is a core function of the operating system that determines which process or thread gets CPU time at any given moment. Since multiple processes and threads compete for CPU resources, scheduling ensures efficient execution while balancing responsiveness, fairness, and performance.

The CPU scheduling process relies on a scheduler, which selects processes from the ready queue based on a predefined algorithm. Scheduling decisions occur when a process transitions between states, such as when it moves from running to waiting, from waiting to ready, or when a process terminates.

CPU scheduling is classified into two types: preemptive and non-preemptive. Preemptive scheduling allows the OS to interrupt a running process and assign the CPU to another process, improving multitasking and responsiveness. In contrast, non-preemptive scheduling ensures that once a process starts executing, it runs to completion or until it voluntarily releases the CPU.

Several CPU scheduling algorithms are commonly used:

First-Come, First-Served (FCFS): The simplest scheduling algorithm where processes execute in the order they arrive. While easy to implement, it suffers from the convoy effect, where shorter processes get delayed behind longer ones.

Shortest Job Next (SJN): Prioritizes the process with the shortest burst time, minimizing average wait time. However, predicting execution times accurately can be difficult.

Round Robin (RR): Assigns a fixed time slice (quantum) to each process, cycling through the ready queue. This approach ensures fairness and responsiveness but may introduce higher context switching overhead.

Priority Scheduling: Assigns a priority value to each process, with higher-priority processes executing first. However, it can lead to starvation, where low-priority processes are indefinitely delayed.

Multilevel Queue Scheduling: Divides processes into separate queues based on priority or process type (e.g., system processes, interactive processes), ensuring efficient resource allocation.

Multilevel Feedback Queue: A dynamic scheduling approach that allows processes to move between queues based on execution characteristics, balancing responsiveness and efficiency.

Effective CPU scheduling enhances system performance by maximizing CPU utilization, minimizing response time, and ensuring fair distribution of resources among processes. As modern computing evolves, scheduling techniques continue to advance, incorporating machine learning and predictive algorithms to optimize performance in real-time applications and multi-core environments.

Scheduling criteria

Scheduling Criteria in CPU Scheduling

CPU scheduling is a critical function of an operating system, and various criteria are used to evaluate and determine the effectiveness of different scheduling algorithms. These criteria ensure that the CPU is allocated efficiently, system performance is optimized, and the system behaves fairly and responsively. The primary **scheduling criteria** are:

CPU Utilization

CPU utilization refers to the percentage of time the CPU is actively executing processes. The goal of any scheduling algorithm is to keep the CPU as busy as possible to ensure maximum system performance. High CPU utilization indicates that the system is efficiently handling processes and minimizing idle time. In ideal scenarios, the CPU utilization should remain close to 100%, though slight variations may occur based on process needs.

Throughput

Throughput refers to the number of processes completed within a specific period of time. A higher throughput indicates that the system is able to execute and finish more processes in a given time, contributing to greater productivity. Optimizing throughput is especially important in systems that require the execution of many processes, such as batch processing or high-performance computing tasks. Scheduling algorithms that minimize waiting times and maximize resource allocation tend to improve throughput.

Turnaround Time

Turnaround time is the total time taken from the submission of a process to its completion. It includes both the waiting time (the time a process spends in the ready queue) and the execution time (the actual CPU time). Minimizing turnaround time is crucial for improving the responsiveness of a system, especially in real-time applications or user-facing tasks where quick results are expected. The goal is to ensure that processes are executed in a timely manner without unnecessary delays.

Waiting Time

Waiting time is the total amount of time a process spends in the ready queue waiting for CPU time. A lower waiting time indicates that processes are being scheduled promptly, reducing delays and improving overall system responsiveness. Waiting time is influenced by factors like scheduling algorithms, the number of processes in the queue, and the priority assigned to each process. Minimizing waiting time is particularly important in systems with high process volumes or real-time applications, where delays can significantly impact performance.

Response Time

Response time is the time from when a process is submitted to when it produces its first output or response. It is particularly critical in interactive systems where users expect quick feedback after initiating a task. A low response time enhances user satisfaction and system interactivity. Response time is a critical factor for systems like web servers, video games, or any application that requires real-time feedback. Scheduling algorithms that balance CPU time fairly among processes while prioritizing user-facing tasks tend to minimize response time.

Trade-offs Between Scheduling Criteria

While all of these criteria aim to improve system performance, there are often trade-offs when optimizing for one criterion over another. For example, a scheduling algorithm that focuses on maximizing CPU utilization may cause increased waiting times for processes in the ready queue. Similarly, algorithms designed to minimize waiting times may result in high response times for processes that require immediate execution. Hence, selecting the appropriate scheduling algorithm depends on the specific needs of the system, whether it's prioritizing fairness, response time, or throughput.

Optimal Scheduling Criteria

In practice, no single scheduling algorithm can optimize all the criteria simultaneously. Different systems have different requirements, and choosing the best algorithm involves considering the most important criteria based on the application. For instance, **Round Robin (RR)** is designed to ensure fairness and reduce response time by giving each process a fixed time slice, while **Shortest Job First (SJF)** minimizes waiting time and maximizes throughput, but it may increase the risk of starvation for longer processes. **Priority Scheduling** prioritizes high-priority processes but can lead to starvation of low-priority processes.In summary, understanding the various **scheduling criteria** helps in selecting or designing scheduling algorithms that balance these factors based on the specific needs of the operating system and the applications running on it.

Scheduling Algorithms in CPU Scheduling

CPU scheduling algorithms determine the order in which processes are assigned to the CPU for execution. The choice of scheduling algorithm affects system performance, efficiency, and responsiveness. Scheduling algorithms can be broadly classified into preemptive (where a process can be interrupted and moved to the ready queue) and non-preemptive (where a process runs until it completes or voluntarily releases the CPU). Below are the most commonly used CPU scheduling algorithms:

1. First-Come, First-Served (FCFS) Scheduling

Type: Non-Preemptive
Concept: Processes are executed in the order they arrive in the ready queue.

Advantages:

- Simple to implement.
- Works well for batch processing.

Disadvantages:

- Can lead to the convoy effect, where short processes wait for long processes to finish.
- Poor response time for interactive systems.

Example:
If three processes arrive in this order:

- P1 (Burst time = 10ms)
- P2 (Burst time = 5ms)
- P3 (Burst time = 2ms)
  Then P1 will execute first, followed by P2 and P3, causing P2 and P3 to experience long waiting times.

2. Shortest Job Next (SJN) / Shortest Job First (SJF) Scheduling

Type: Non-Preemptive / Preemptive (Shortest Remaining Time First, SRTF)
Concept: The process with the shortest CPU burst time is executed first.

Advantages:

- Minimizes average waiting time and turnaround time.

Efficient for batch systems.

Disadvantages:

- Requires knowledge of process burst times, which is often unavailable.

Can cause starvation for longer processes if many short processes keep arriving.

Example:
If processes arrive as follows:

- P1 (Burst time = 10ms)
- P2 (Burst time = 2ms)
- P3 (Burst time = 4ms)
- P4 (Burst time = 1ms)
  The execution order will be P4 → P2 → P3 → P1.

3. Round Robin (RR) Scheduling

Type: Preemptive
Concept: Each process gets a fixed time slice (quantum). If a process does not finish in its time slice, it is moved to the end of the queue.

Advantages:

- Provides fairness and ensures no process is left waiting indefinitely.
- Reduces response time, making it suitable for interactive systems.
- Disadvantages:
- Frequent context switching leads to overhead.
- The performance depends on the time quantum chosen—too small increases context switching, too large makes it behave like FCFS.

Example:
With a time quantum of 4ms, and processes:

- P1 (Burst time = 10ms)
- P2 (Burst time = 5ms)
- P3 (Burst time = 8ms)
  Execution will occur as:

P1 (4ms) → P2 (4ms) → P3 (4ms) → P1 (6ms) → P2 (1ms) → P3 (4ms).

4. Priority Scheduling

Type: Preemptive or non-preemptive
Concept: Each process is assigned a priority, and the CPU is allocated to the highest-priority process.

Advantages:

- Allows differentiation of process importance (e.g., system processes vs. user processes).
- Can be optimized to execute critical tasks first.

Disadvantages:

- Can lead to starvation, where low-priority processes may never execute.
- Requires priority assignment, which may not always be optimal.
- Solution to Starvation: Aging, where a process's priority increases the longer it waits.

Example:
If processes have priorities as follows (lower number = higher priority):

- P1 (Priority = 3, Burst time = 10ms)
- P2 (Priority = 1, Burst time = 5ms)
- P3 (Priority = 2, Burst time = 2ms)
  Execution order: P2 → P3 → P1.

5. Multilevel Queue Scheduling

Type: Preemptive
Concept: Processes are divided into different priority queues, such as foreground (interactive tasks) and background (batch tasks), each with its own scheduling algorithm.

Advantages:

- Provides better control over different process types.
- Allows priority-based scheduling.
- Disadvantages:
- Complex to implement.
- Requires proper tuning to prevent starvation in lower-priority queues.

Example:

- Queue 1: Foreground processes (Round Robin).
- Queue 2: Background processes (FCFS).
- CPU always serves the higher queue first.

6. Multilevel Feedback Queue Scheduling

Type: Preemptive

Concept: Similar to Multilevel Queue Scheduling, but processes can move between queues based on their behavior and execution time.

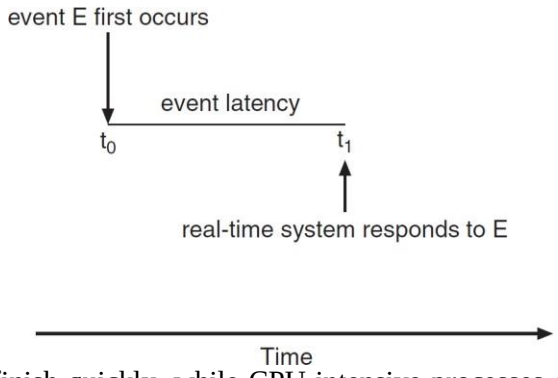
Advantages:

- Dynamically adjusts process priority based on execution history.
- Prevents starvation by allowing long-waiting processes to move to higher-priority queues.
- Disadvantages:
- Complex to configure correctly.
- Requires careful tuning of queue movement rules.

Example:

A process starts in the highest-priority queue (short time quantum).

If it does not finish, it moves to a lower-priority queue (longer quantum).



Interactive processes finish quickly, while CPU-intensive processes gradually move to lower queues.

Event latency.

Choosing the Right Scheduling Algorithm

Algorithm	Best for	Disadvantages
FCFS	Simple batch systems	Convoy effect, poor response time
SJF (SJN)	Optimizing turnaround time	Requires prior knowledge of burst times, starvation issue
Round Robin (RR)	Interactive systems	High context switching overhead
Priority Scheduling	Real-time systems with task prioritization	Starvation of low-priority tasks
Multilevel Queue	Systems with different process types	Requires careful tuning
Multilevel Feedback Queue	Dynamic and adaptive scheduling	Complex implementation

Each scheduling algorithm has its advantages and trade-offs. In real-world operating systems, a combination of these algorithms is often used to balance fairness, efficiency, and responsiveness.

Thread scheduling

Thread scheduling is the process of determining which thread will execute on the CPU at any given time. Since modern operating systems support multithreading, where multiple threads run concurrently, an efficient scheduling mechanism is essential to optimize CPU utilization, system responsiveness, and resource management. Thread scheduling can be broadly classified into user-level

scheduling and kernel-level scheduling, depending on whether it is managed by the user-space thread library or the operating system kernel.

In user-level thread scheduling, the scheduling decisions are made entirely in user space without kernel intervention. This approach is implemented by the threading library in a process and is faster than kernel scheduling because it does not involve system calls. However, since the kernel is unaware of user-level threads, if one thread blocks (e.g., waiting for I/O), the entire process may be blocked, leading to inefficiency. Kernel-level thread scheduling, on the other hand, is managed by the operating system, which has full control over thread execution. It ensures better CPU utilization, as the kernel can schedule another thread of the same process if one thread blocks. However, it incurs additional overhead due to frequent context switching between threads.

Thread scheduling is based on different scheduling policies used by operating systems. These include preemptive scheduling, where the OS can interrupt a running thread to assign the CPU to another thread, and non-preemptive scheduling, where a thread runs until it voluntarily yields control. Most modern operating systems use preemptive scheduling to ensure fair resource allocation and responsiveness. Common thread scheduling algorithms include First-Come, First-Served (FCFS), where threads execute in the order they arrive, Round Robin (RR), which assigns each thread a fixed time slice to prevent monopolization, and Priority Scheduling, where higher-priority threads are executed before lower-priority ones. In multilevel queue scheduling, threads are grouped into different priority levels, such as system threads and user threads, each managed by its own scheduling algorithm.

In multi-core processors, multithreading models like many-to-one, one-to-one, and many-to-many determine how user threads are mapped to kernel threads, impacting scheduling efficiency. Load balancing techniques are used to distribute threads across multiple CPU cores, improving parallelism and system performance. Advanced scheduling techniques, such as work-stealing algorithms, dynamically redistribute tasks among processor cores to enhance efficiency. Thread affinity, which binds threads to specific CPU cores, can also improve cache utilization and reduce performance overhead.

Overall, thread scheduling plays a crucial role in maximizing system throughput, ensuring fair CPU allocation, and enhancing application performance. Modern operating systems implement sophisticated scheduling strategies to balance efficiency, responsiveness, and resource management in both single-core and multi-core environments.

Thread scheduling is a crucial aspect of modern operating systems that determines how threads are assigned to the CPU for execution. With the advent of multithreading, multiple threads can execute concurrently within a process, improving system efficiency and responsiveness. However, efficient scheduling is required to optimize CPU utilization and ensure fair execution among threads. Thread scheduling can be categorized into user-level thread scheduling and kernel-level thread scheduling. In user-level thread scheduling, the scheduling decisions are managed entirely in user space by a thread library without kernel intervention. This approach allows for faster context switching and lower overhead since system calls are not required. However, the primary drawback is that if one

thread performs a blocking operation, the entire process may be blocked because the kernel is unaware of the individual threads. On the other hand, kernel-level thread scheduling is managed by the operating system, where the kernel is responsible for scheduling individual threads independently. This approach ensures that if one thread blocks, the OS can schedule another thread of the same or a different process, leading to better CPU utilization and efficiency. The trade-off is that kernel-level scheduling incurs additional overhead due to frequent context switches and system calls.

Thread scheduling policies vary depending on the operating system and its design. Preemptive scheduling allows the OS to interrupt a running thread to allocate CPU time to another thread, ensuring that no single thread monopolizes CPU resources. This is commonly used in real-time and multitasking systems to maintain fairness and system responsiveness. Non-preemptive scheduling, in contrast, allows a thread to run until it either voluntarily yields the CPU or completes execution. Although this reduces context switching overhead, it can lead to issues where lower-priority threads experience starvation if a higher-priority thread runs continuously. Various scheduling algorithms are used to implement thread scheduling, including First-Come, First-Served (FCFS), where threads execute in the order they arrive, Round Robin (RR), which assigns a fixed time quantum to each thread before moving to the next, and Priority Scheduling, where threads with higher priority values execute first. In systems that handle diverse workloads, Multilevel Queue Scheduling is often employed, where threads are categorized into multiple queues based on priority or type, such as real-time threads, system threads, and user threads, with different scheduling algorithms governing each queue.

In multi-core processors, thread scheduling becomes more complex as multiple threads must be efficiently distributed across available cores. Different multithreading models, such as many-to-one, one-to-one, and many-to-many, define how user threads map to kernel threads, influencing performance and resource allocation. The one-to-one model, where each user thread corresponds to a kernel thread, is widely used in modern operating systems due to its ability to leverage multiple CPU cores for parallel execution. Load balancing techniques help distribute threads across processor cores to avoid overloading a single core while underutilizing others, thereby improving system performance and efficiency. Advanced scheduling strategies, such as work-stealing algorithms, allow idle cores to "steal" tasks from overloaded cores, ensuring optimal utilization of computing resources. Additionally, thread affinity, where threads are bound to specific CPU cores, can improve performance by reducing cache misses and context-switching overhead.

Efficient thread scheduling plays a critical role in ensuring smooth execution of applications, maximizing system throughput, and maintaining fairness among threads. Modern operating systems implement sophisticated scheduling mechanisms to balance efficiency, responsiveness, and resource management. Factors such as workload distribution, CPU core availability, priority levels, and real-time constraints all influence scheduling decisions. Furthermore, in real-time operating systems (RTOS), specialized thread scheduling techniques, such as Rate-Monotonic Scheduling (RMS) and Earliest Deadline First (EDF), are employed to meet strict timing constraints for time-sensitive applications like embedded systems, telecommunications, and autonomous systems. As computing architectures continue to evolve, with increasing reliance on hyper-threading, parallel processing, and

UNIT III

CONCURRENCY

Process Synchronization: Background - The Critical Section problem - Peterson's solution –Synchronization hardware - Mutex Locks-Semaphores-Classic problems of synchronization, Deadlocks: System model -Deadlock characterization - Methods for handling deadlocks: Deadlock prevention, Deadlock avoidance , Deadlock detection, Recovery from deadlock.

Process Synchronization: Background

In a multitasking environment, multiple processes often run concurrently, sharing system resources such as CPU, memory, and I/O devices. While this improves efficiency, it also introduces challenges when processes need to access shared resources simultaneously. Without proper coordination, issues like race conditions, data inconsistency, and deadlocks can arise.

- Race Conditions: This occurs when two or more processes attempt to modify shared data at the same time, leading to unpredictable outcomes. For example, if two threads try to update a counter simultaneously without synchronization, the final value may not be as expected.
- Critical Section Problem: Top revent race conditions, processes must coordinate access to shared resources. This is managed through a concept called the critical section—a part of the program where shared resources are accessed. A proper synchronization mechan is mensures that only one process executes its critical section at a time.
- Synchronization Mechanisms: Various techniques are used to achieve process synchronization. These include:
- Mutex Locks: A mutual exclusion lock ensures that only one process can access a shared resource at a given time.
- Semaphores: These are signaling mechanisms that control access to resources using counters, preventing multiple processes from entering the critical section simultaneously.
- Monitors: A high-level synchronization construct that allows controlled access to shared resources whiles implifying code complexity.

Effective synchronization ensures system stability, prevents deadlocks, and maintains

data consistency. Operating systems implement synchronization techniques to manage concurrent processes efficiently and provide a seamless user experience.

Example: Producer- Consumer Problem

The best example to see the cooperation between two processes is Producer Consumer problem. It is also known as the bounded-buffer problem.

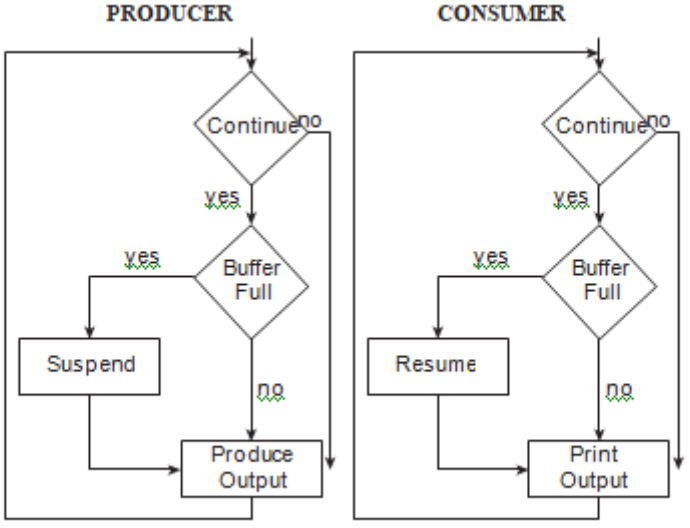
The problem contains two processes: the producer and the consumer. Both processes share a common, fixed-size buffer used as a queue. Producer produces and consumer consumes. Producer puts its production in a buffer and consumer picks it from the buffer. Producer and Consumer must synchronize.

Types

The following buffer is needed for Producer – Consumer Problem

Unbounded-buffer

- Places no practical limitations in the size of the buffer
- Consumer may wait, producer never waits



Bounded-buffer

- Assumes that the re is a fixed buffer size

- Consumer waits for new item ,producer waits if buffer is full

Illustration

Shared memory solution to the bounded- buffer problem allows at most(BUFFERSIZE-1) items in the buffer at the same time. A solution, where all *N* buffers are used is not simple.

- Suppose that the producer-consumer code is modified by adding a variable counter, initialized to 0 and incremented each time a new item is added to the buffer
- Shared data

  #defineBUFFER_SIZE10typedefstruct{

  ...

  }item;

  Item buffer[BUFFER_SIZE];int in=0;

  int out = 0;

  Int counter=0;

Producer

The Producer code is modified as follows while(true)

{

/*produce an item and put in next Produced

*/while(counter==BUFFER_SIZE);

//do nothing

buffer[in]=next Produced; in =(in +1)%BUFFER_SIZE;

counter++;

}

Consumer

The Consumer code is modified as follows while(true)

{

while(counter==0)

//do nothing

Next Consumed=buffer[out];

```
out=(out+1)%BUFFER_SIZ
E;
counter—;
/*consume the item in next Consumed
}
```

Race Conditions

The situation where several processes access and manipulate shared data concurrently and the outcome of execution depends on the particular order in which the access takes place are called race *conditions*

How can processes coordinate(or synchronies)in

Order to guard again race conditions?

The final value of the shared data depends upon which process finishes last. To prevent race conditions, concurrent processes must be synchronized.

Critical Section Problem

Consider a system consisting of n processes$\{P_0,P_1,P_2,\ldots,P_{n-1}\}$all competing to use shared data.

Critical Section(CS)

Each process has a code segment, called critical section, in which the shared data is accessed Levels. The critical section problem is to design a protocol that the process can use to cooperate.

When one process is executing in the critical section, no other process is to be allowed in its critical section. This is the problem of Critical Section Problem.

Types of Section

Each process must request permission to enter its critical section.

1. **Entry Section (ES):** Section of code implementing the request permission to enter the critical section.
2. **Exit Section (LS):** Section of code to leave the critical section which follows Entry Section.
3. **Remainder Section (RS):** Remaining code of the process after the critical section.

Solution to Critical Section Problem–Requirements

Mutual Exclusion

If process P ¡is executing in its critical section, then no other processes can be executing in their critical sections

Implications

- Critical sections better be focused and shor to Better not get into an infinite loop in there
- If a process somehow halts / waits in its critical section, it must not interfere with other processes

Progress

- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed in definitely
  - If only one process wants to enter, its should be able to
  - If two or more want to enter, one of them should succeed

Bounded Waiting

A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made are quest to enter its critical section and before that request is granted.

- Assume that each process executes at a non-zero speed
- No assumption concerning the relative speed of the n processes

Types of Solutions

- Software solutions Algorithms whose correctness does not rely on any other assumptions.
- Hardware solutions Rely on some special machine instructions.
- OperatingSystemsolutionsProvidesomefunctionsanddatastructur estotheprogrammer through system / library calls.

- ProgrammingLanguagesolutionsLinguisticconstructsprovidedaspartofalanguage.

Two-Process Solutions

Consider an algorithm that is applicable to only two processes at a time with the following notation.

Initial notation

- Only 2 processes ,P_0 and P_1
- When Usually just presenting process P_i, always denotes other processP_j(i!=j)

Structure of Process P_i

```
do{
entry section/*enter critical section*/Critical section/* access shared
variables */ exit section/* leave critical section
*/Remainder section/*do other work*/
}while(1);
```

Processes may share some common variables to synchronize/ coordinate their actions.

Algorithm1

Shared variables

```
Int turn;
//initially turn=0
Turn=i,          Pi can enter its critical section
```

Structure of Process P_i do

```
{

while(turn!=i);
critical section turn=j;
remainder section
}while(1);
```

This solution guarantees mutual exclusion, but not progress.

Drawback

- Processes must strictly alternate

Pace of execution of one process is determined by pace of execution of other processes if one processes fails other process is permanently locked.

Problem

Does not retain information about the state of each progress, it remembers only which process is allowed to enter its critical section.

Sharedvariablesofalgorithms1and2arecombinedtoobtainacorrectsolution tothecriti- cal-section problem with all three requirements are met.

Peterson's Solution

- It is a Two process solution
- **Assume that the LOAD and STORE instructions are atomic ;that is, cannot be inter- rupted. The two processes share two variables:**
  - Int turn;
  - Boolean flag[2]
- The variable **turn** indicates who set urn it is to enter the critical section.
- The **flag** array is used to indicate if a process is ready to enter the critical section.
- Flag[i]=true implies that process P_i is ready

Shared variables Boolean flag[2]; int turn;

Initially flag[0]=flag[1]=false and turn=0(or1)

Structure of Process P_i

```
do{
flag [i] =true ; turn = j; while(flag[j]&&turn==j); critical section
flag[i] =false; remainder section
}while(1);
```

Properties

Thefollowingpropertiesshouldbesatisfiedtoprovethatthesolutioniscorrect

1. Mutual exclusion is preserved
2. The progress requirement is satisfied
3. The bounded-waiting requirement is met.

1. **Mutual exclusion is preserved Proof:**
   - If flag[j]=false or turn=i, then $P_i$enters its CS.

- If flag[0]=flag [1]=true, then both processes can execute in its CS.
- Turn must be 0or1(not both)=>only one thread can be in CS
- Assume P_i and P_j in CS at the same time
- flag[i]&flag[j]==True
- (I)flag [j]& turn ==j False turn=i
- (II)flag[i]&turn==i False turn=j
- Turn can not be I and j at the same time

2. **The progress requirement is satisfied.**
3. **The bounded-waiting requirementismet.**

Proof:

- $P_i$stuck at—flag[j]==true & turn ==j||while loop
- (case1)$P_j$is not ready to enter
  - flag[j]==false can enter
  - (case2)$P_j$is ready to enter
  - **P_j:**set flag[j] to true and at its while
  - **Observation :**turn==I XOR turn==j
  - (case2.1)turn==i$P_i$will enter
  - (case2.2)turn==j$P_j$will enter
- (case2.2.1)$P_j$leaves CS; sets flag[j] to false
- (case2.2.2)P_j leaves CS;sets flag [j]to true ;
- Then sets turn toi $P_i$enters
- (P_i at while can not change turn)
- P_i(P_j)will enter the CS(progress)after at most one entry by P_j(P_i)(bounded waiting)

Multiple-Process Solutions(Software Solutions)

Bakery Algorithm

BakeryAlgorithmisthealgorithmwhichsolvesthecritical-sectionproblemfornprocesses.
Before entering its critical section, process receives a number Holder of the smallest number enters the critical section.

- If processes $P_i$and $P_j$receive the same number, if i<j, then $P_i$is served first; else $P_j$is served first.
- Thenumberingschemealwaysgeneratesnumbersinincreasingorderof

enumeration; i.e.,1,2,3,3,3,3,4,5...

Notation

Lexico graphical order(ticket#, processid#)

- (a, b)<(c, d)if a<c or if a =c and b <d
- max(a_0,…,a_{n-1})is a number, k, such that k$a_i$fori=0,…,n–1

Shared Data Structures

The command at structures are

- Boolean choosing[n];
- Int number[n];

Data structures are initialized to **false** and **0** respectively.

Structure of Process P_i

```
do{choosing [i]=true;
number[i]=max(number[0],number[1],…,number[n–1])+1choosing
[i]=false;
for (j=0; j<n; j++) { while(choosing[j]);
while((number[j]!=0)
&&(number[j, j]<number[i,i]));
}

Critical section number[i]=0;remainder section
}while(1);
```

Proof

To prove that the Bakery algorithm works, we have to show that if $P_i$is in its critical Section and Pk (ki) has already chosen its number [k]0 then

(number[i],i)<(number[k],k)

Consider that P_i is already in its critical-section and P_k trying to enter the $P_{k'_s}$critical-section When process P_k executes the second while loop for j=i, it finds that:

number[i]0(number[i],i)<(number[k],k)

Thus it continues looping in the while loop until P_i leaves the critical section.

Note: Processesenterintheircritical-sectiononafirst-come,first-servebasis.

Handling Process Failures

- If all three conditions—Mutual Exclusion (ME), Progress, and Bounded Waiting—are met, the solution will be robust against a process failure in its remainder section (RS). This is because a failure in the RS is equivalent to the process staying there indefinitely, which does not affect other processes.
- However, no solution can guarantee robustness if a process fails while inside its critical section (CS).
- If a process P_i fails in its CS, other processes have no way of knowing that it has failed. To them, P_i still appears to be holding the lock, leading to indefinite blocking.

Draw back of Software Solutions

- Processes that are requesting to enter their critical section are busy waiting (consuming processor time needlessly).
- If critical sections are long, it would be more efficient to block processes that are waiting.

Mutex Locks

- **Mutex** is short for **Mutual ex**clusion
- Mutex lock is used to solve the critical-section problem
  - Used to ensure exclusive access to data shared between threads

Illustration

- Aprocessmustacquirethelockbeforeenteringacriticalsectionusingacquire()function releases the lock when it exits the critical section using release() function
- Definition of acquire()

```
acquire()
{

while(!available);/*busy wait*/
available= false;;
}
```

- Definition of release()

```
release()
{
```

available=true;
}

- Structure

```
do{
    acquire lock
        critical section
    acquire lock
        remainder section
}while(true);
```

- Disadvantage
  - Requires busy waiting

Spinlock

- A spinlock is a type of mutex lock where a process continuously checks for lock availa bility (sP_ins) instead of sleeping when the lock is held by another process.
- When a process enters its critical section ,it acquires the spin lock.
- Anyotherprocesstryingtoenteritscriticalsectionmustkeeploopinguntilthelockis released.

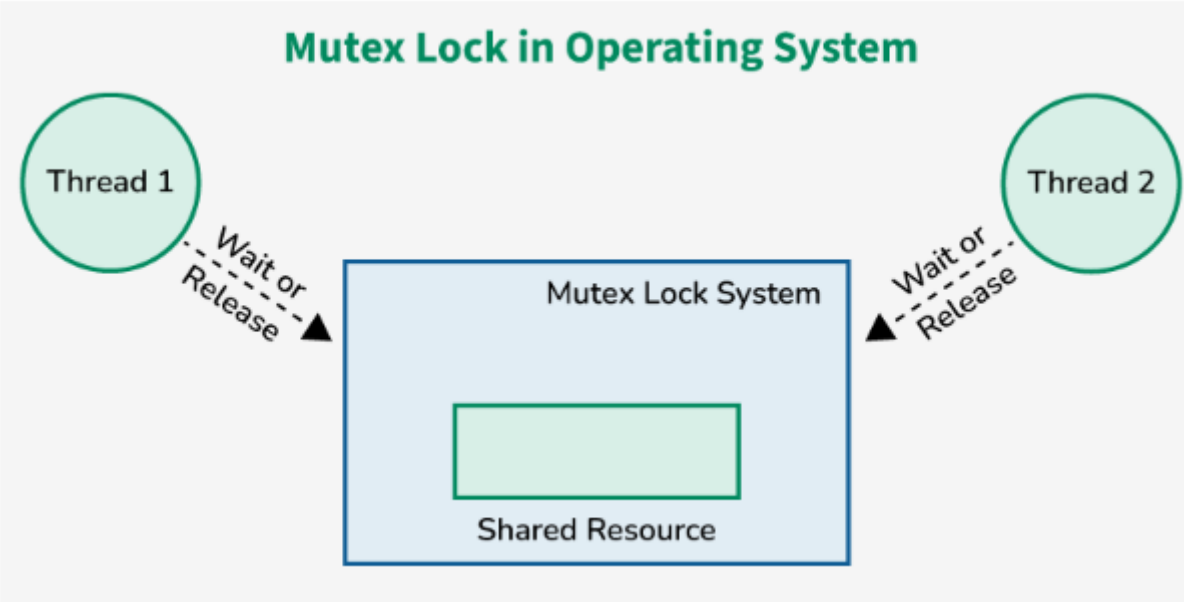
Advantage:

NoContextSwitchOverhead–Sincetheprocessdoesnotgetsuspended,itavoidstheover-headofswitchingbetweenprocesses,makingitefficientforshortwaittimes.

Synchronization Hardware

Mutex Lock

A mutex lock, or simply mutex, is a synchronization primitive used in concurrent programming to prevent simultaneous access to shared resources by multiple threads. It ensures that only one thread can acquire the lock (enter a critical section) at a time, preventing race conditions and data inconsistencies.



The producer-consumer problem: Consider the standard producer-consumer problem. Assume, we have a buffer of 4096-byte length. A producer thread collects the data and writes it to the buffer. A consumer thread processes the collected data from the buffer. The objective is, that both the threads should not run at the same time.

Solution: A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by the producer, the consumer needs to wait, and vice versa. At any point in time, only one thread can work with the entire buffer. The concept can be generalized using semaphore.

Semaphores

Semaphore is a synchronization tool(provided by the OS)that does not require busy waiting. Semaphore is a variable that has an integer value

- May be initialized to an on negative number
- Wait operation decrements the semaphore value
- Signal operation increments semaphore value

Logical Definition

A semaphore S is an integer variable that, a part from initialization, can only be accessed through 2 atomic and mutually exclusive operations:

- **wait(S):**Originally called P(),from Dutch proberen—to test||
- **signal(S):**Originally called V(),from Dutch verhogen—to increment||

Wait and Signal Operations

- **Busy-waiting implementation of these indivisible(atomic)operations:**

```
wait(S)
{
    while(S<=0do);//no-op S--;
}

signal(S)
{
    S++;
}
```

- Modification to the value of semaphore(S)in the wait and signal is executed individually.
- In wait, the testing and possible modification of S mustal so be executed with out interruption.

Usage of Semaphore

The fundamental principle of semaphore is this: Two or more processes can cooperate by meansofsimplesignalssuchthataprocesscanbeforcedtostopataspecifiedplaceuntilit has received a specified signal.

Usage 1: Critical Section of n Processes Problem

Shared data

Semaphore mutex;//initially mutex=1

Structure of Process P_i do{wait(mutex); critical section signal (mutex);

Remainder section
}while(1);

Usage 2: Synchronization of 2 processes

Suppose that we have two processes, P_1 with statement s_1 and P_2 with statement s_2. We require that s_2 be executed only after s_1 has completed. This can be done with a shared semaphore variable, mutex, initialized to 0, and by inserting the following statements in P_1

s_1; signal (mutex);

and the following statements in P_2 wait(mutex) s_2;

Advantages

- Applicable to any number of processes on either a single processor or multiple processors. It is simple and therefore easy to verify.
- It can be used to support multiple critical sections.

Disadvantages

- Busy-waiting consumes processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting

To avoid busy waiting, when a process has to wait, it will be put in a blocked queue of processes waiting for the same event. Hence a semaphore is defined as a record

```
Type defstruct
{

Int value;
Struct process*L;
}semaphore;
```

Assume two simple operations:

- **Block** suspends the process that invoke sit.
- **Wakeup(P)** resumes the execution of a blocked process P.

Semaphore Operations

Semaphore operations now defined as

- **wait(S):** When a process must wait for a semaphore S, it is blocked and put on the semaphore's queue.

```
S. value—; if(S. value<0)
{

    Add this process toSL; block();
}
```

- **signal (S):** The signal operation removes (assume a fair policy like FIFO) one

process from the queue and puts it in the list of ready processes.

```
S. value++;if(S. value<=0)
{
    Remove a process P from S.L;
    wakeup(P);
}
```

Deadlocks and Starvation

Deadlock–two or more processes are waiting indefinitely for an event that can be caused by only one of waiting processes.

Let S and Q be two semaphores initialized to1

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
signal(S);	signal(Q);
signal(Q)	signal(S);

Starvation–in definite blocking. A process may never be removed from the semaphore queue (say, if LIFO) in which it is suspended.

Priority Inversion–scheduling problem when lower- priority process holds a lock needed by higher – priority process

Types of Semaphores

1. **Counting Semaphore**
   - Integer value can range over an unrestricted domain.
   - Useful fork-exclusion scenarios of replicated resources
2. **Binary Semaphore**
   - Integer value can range only between 0 and 1(really a Boolean);
   - Can be simpler to implement(use wait Band signal B operations).
   - Also known as mutex locks

A counting semaphore S can be implemented using 2 binary

semaphores (that protect its counter).

Implementing S as a Binary Semaphore Data Structures

```
Binary  semaphoreS1,S2;
int C;
```

Initialization

```
S1=1
S2=0
C=initial value of counting semaphore S
```

Implementing S Wait operation

```
wait(S1);
C—;
        if(C<0)
{

signal(S1);wait(S2);
}

signal(S1);
```

Signal Operation

```
wait(S1);C++;if(C<=0)
signal(S2);
else
signal(S1);
```

Semaphore as a General Synchronization Tool

- Execute B in P_j only after A executed in P_i
- Use semaphore flag initialized to 0
- **Code:**

P_i	P_j
A	**wait (flag)**
Signal (flag)	**B**

Classical Problem of Synchronization

Problems with Semaphores

- Semaphores are an effective mechanism for ensuring mutual exclusion and process coordination.
- However, the wait(S) and signal(S)operations are distributed across multiple processes, making it challenging to track their overall impact.
- Proper usage is essential across all processes, requiring the correct sequence, appropriate variables, and no omissions.
- A single faulty or malicious process can disrupt the entire system of processes.

Critical Region

Monitors

A monitor is high-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes. It can be implemented using semaphores.

Syntax

Only one process may be active with in the monitor

{

Shared variable declarations procedure body P_1(…)
{}

ProcedurebodyP_2(…)
{}

Procedure body Pn(…){
}{

Initialization code
}}

Monitor is a software module containing : one or more procedures an initialization

sequence local data variables

Characteristics

- Localdatavariablesareaccessibleonlybythemonitorprocessentersmonitor by invoking one of its procedures.
- Only one process may be executing in the monitor at a time

Monitor Conditions

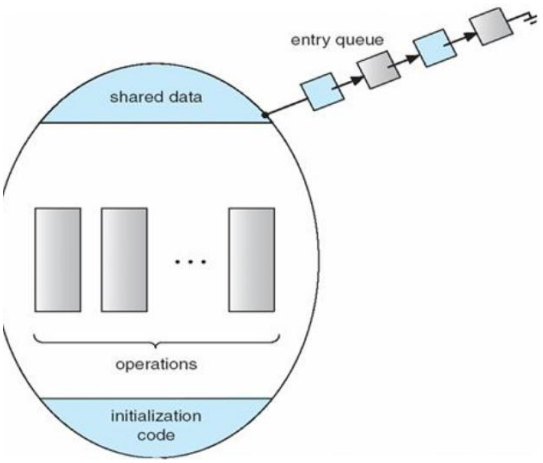
To allow a process to wait with in the monitor, a **condition** variable must be declared ,for example:

Condition x, y;

Condition Variables

Condition variables are local to the monitor(accessible only with in the monitor).Condition variables can only be used with the operations **c wait** and **c signal** executed on an associated queue.

Schematic View of a Monitor



Schematic View of a Monitor

Example

Now illustrate the monitor concepts by presenting a deadlock-free solution to the dinning-philosophers problem.

Dining-Philosophers Problem

Five philosophers are seated around a circular table with Two states: eating or thinking. There is a shared bowl of rice. In front of each one is a plate. Between each pair of people there is a chopstick, so there are five chop sticks. It takes two chop sticks to take/eat rice, so while n is eating neither n+1 nor n-1can beating.

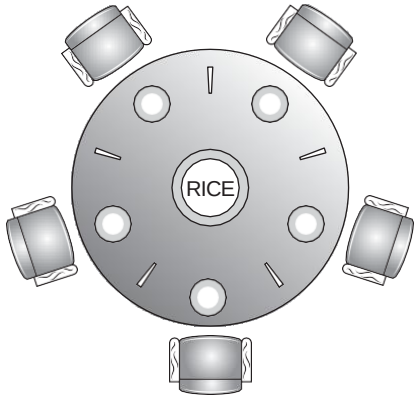
Each one thinks for a while, gets the chop sticks/forks needed, eats, and puts the chopsticks

/forks down again, in an endless cycle.

Problem Definition

Illustrate the difficulty of allocating resources among process without deadlock and starvation.

- The challenge is to granter quests for chopsticks while avoiding deadlock and starvation.
- Dead lock can occur if every one tries to get their chopstick sat once. Each gets a left chopstick, and is stuck, because each right chopstick is someone else's left chopstick.



The situation of the dinning philosophers

Solution to Dining Philosophers

Monitor DP

{

```
Enum {THINKING; HUNGRY,EATING} state[5];
condition self[5];
void Pickup (int i)
{state[i]=HUNGRY;
test(i); if(state[i]!=EATING)
self[i].wait;
}

Void test(inti)
{

if((state[(i+4)%5]!=EATING)&&(state[i]==HUNGRY)&&(state[(i
+1)%5]!=EATING))
{

state[i]=EATING;
self[i].signal() ;
}

}

Void put down(inti){
state[i] = THINKING; // test left and right neighbors test((i + 4) % 5);
test((i+1)%5);
}

Void init(){
for(int i=0;i<5;i++) state [i]= THINKING;
}

}
```

Each philosopher I invokes the operations Pickup() and put down() in the following sequence:

dp. pickup
(i)EAT
dp. putdown(i)

Solution assures that no two neighbors are simultaneously eating. This is a deadlock-free solution, but starvation is possible.

Monitor Implementation Using Semaphores Variables Semaphore

Deadlock

Deadlock is a situation that occurs when two processes are waiting for the other to complete before proceeding. The result is that both processes hang.

Types of Deadlock

- Process Deadlock
  - A process is deadlocked when it is waiting on an event which will never happen
- System Deadlock
- A system is deadlocked when one or more processes are deadlocked

System Model

A system consists of a number of resources (R_1,R_2,...,R_m) to be distributed among a number of competing processes (P_1,P_2,...,P_n).

- A process must request are source before using it
  - Can request any number of resources
  - Number of resources requested<=total number of available resources
- A process must release the resource after using it

Resource Utilization Steps

A process may utilize are source in only the following sequence:

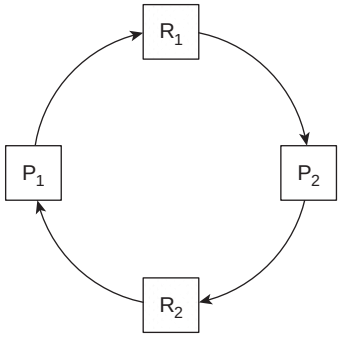
1. **Request**
   - The process request here source
   - If the request cannot be granted immediately(resource not available),then the Requesting process must wait until it can acquire there source
2. **Use:** The process can operate on the resource
3. **Release:** The process releases the resource

Example

System has two resources named as R1 and R2. Each of the two processes, P_1 and P_2, hold one resource and each process needs another one resource.

- Both processes need resources to continue executing
- P_1 requires additional resource R1 and is in possession of resource R2

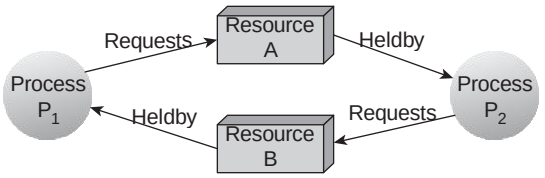
- P_2 requires additional resource R2 and is in possession of R1;
  - Neither process can continue.



Dead Lock Characterization

Dead lock can arise if four conditions hold simultaneously in a system

1. **Mutual Exclusion Condition**
- Only one process at a time can use are source(non-shareable resource)
- Each resource is assigned to a processor is available
2. **Hold and wait condition:** A process holding at least one resource can request for additional resources
3. **No preemption condition:** A resource can be released only voluntarily by the process holding it.That is previously granted resources cannot be forcibly taken away
4. **Circular wait condition:** The reexists a set$\{P_0,P_1,\ldots,P_n\}$of waiting processes such that P_0 is waiting for a resource that is held by P_1,P_1 is waiting for a resource that is held by $P_2,\ldots,P_{n-1}$is waiting for a resource that is held by P_n,and $P_0$is waiting for a resource that is held by P_0



Circular Wait

- These four conditions are not independent. The first three are necessary condition, and the fourth is necessary and sufficient. The fourth condition incorporate the first three.

Resource- Allocation Graph

Resource-allocation graph consists of a set of vertices V and a set of edges E.V is partitioned into two types:

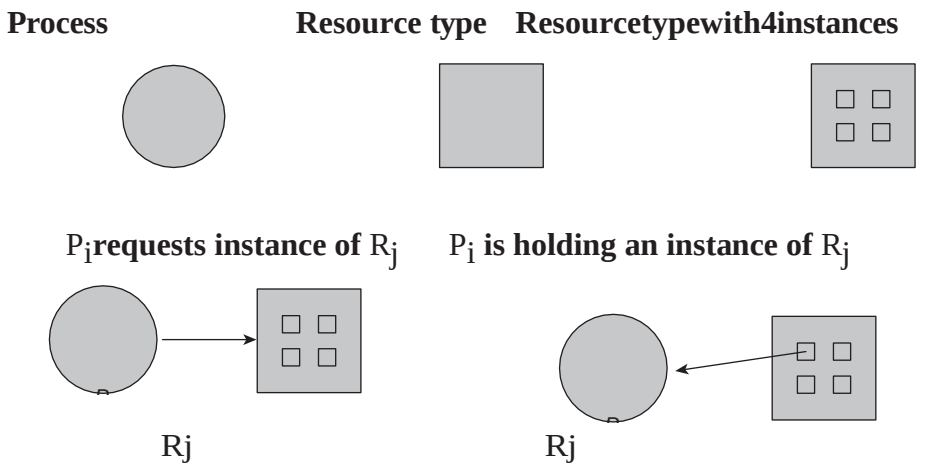
- $P=\{P_1,P_2,\ldots,P_n\}$ set consisting of all the processes in the system
- $R=\{R1,R2,\ldots,Rm\}$ set consisting of all resource types in the system

Types of Edges

1. Request edge
   - Directed edge P_iRj
   - Process P_i requested an instance of resource type R_j and is waiting for that resource
2. Assignment edge
   - Directed edge R_jP_i
   - An instance of resource type R_j has been allocated Process P_i

Representation

The symbols that are used to represent the processes and resources as given below

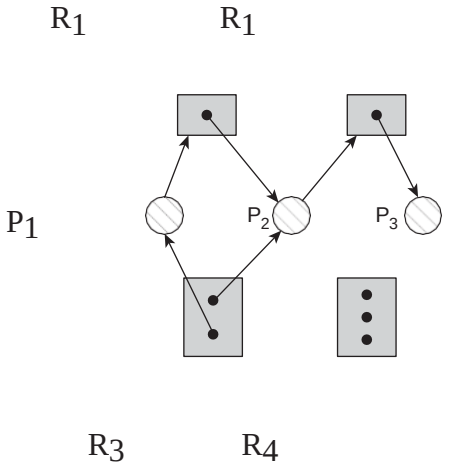


Example

Consider the following situation

- Set P,R andE
  - P={P_1,P_2,P3}
  - R={R1,R2,R3,R4}
  - E={P_1R1,P_2R3,R1P_2,R2P_2,R2P_1,R3P3}
- Resource instances
  - One instance of resource type R1
  - Two instance of resource type R2
  - One instance of resource type R3
  - Three instance of resource type R4
- Process States
  - Process P_2 holds an instance of R1 and R2, and is waiting for an instance of R3.
  - Process P3 holds an instance of R3.
- Basic Facts
  - If graph contains no cycles NO DEADLOCK
  - If graph contains a cycle
- If only one instance per resource type, then deadlock
- If several instances per resource type, possibility of deadlock.

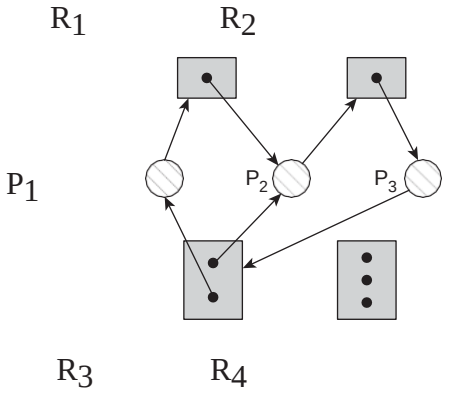
Graph with No Cycles



Resource- allocation graph

Graph with Cycles and Deadlock

When process P_3 requests an instance of resource type R_2. But there is no resource instance available at R_2.So a request edge P_3'! R_2 is added to the graph as follows.



Resource-allocation graph with a deadlock

Here two minimal cycles exist in the system

P_1 R$P_2R_3P_3R_2P_1P_2R_3P_3R_2P_2$

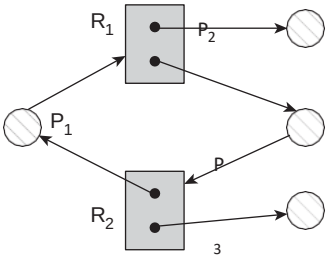
Deadlock occurs at processes P_1,$P_2$and P_3since $P_2$is waiting for R_3which is held by $P_3P_3$is waiting for either P_1or $P_2$to release $R_2P_1$is waiting for $P_2$to release R_1.

Graph with Cycles

Consider the following resource-allocation graph with following graph $P_1R_1P_3R_2P_1$. No deadlock occurs since

P_4 release its instance of resource type R_2.

Released resource may be allocated to P_3breaking the cycle.



Methods for Handling Deadlocks

There are three primary approaches to managing deadlocks:

- **Prevention:** Implement protocols that ensure the system never enters a deadlock state.
- **Detection and Recovery:** Allow deadlocks to occur but provide mechanisms to detect and resolve them.
- **Ignoring Deadlocks:** Some systems, like UNIX, choose to ignore deadlocks entirely, assuming they are rare and can be managed manually if they occur.

Deadlock Management Approaches

- **Deadlock Prevention:** Prevents deadlocks by enforcing restriction son how processes request and allocate resources.
- **Deadlock Avoidance:** Carefully evaluate search resource request and only grants it if doing so will not lead to a deadlock.
- **Deadlock Detection:** Always grants resource requests when possible but eriodically checks for deadlocks. If a deadlock is detected, the system takes steps to resolve it.

Recovery from Deadlock

Once a deadlock is detected, the system must take action to recover:

- It may allow processes to complete naturally so that resources become available.
- In some cases, processes may need to be terminated and restarted to free up resources and restore normal system operation.

Deadlock Prevention

Deadlock prevention is to prevent the deadlocks occurrence by ensuring that at least one of the conditions cannot hold.

Types of Prevention Method

The dead lock prevention methods consists of the following two types

1. **Indirect Method**
   - Prevent the occurrence of the three following conditions

- Mutual Exclusion
- Hold and Wait
- No-Preemption

2. **Direct Method**
   - Prevent the occurrence of the condition circular wait.

Mutual Exclusion Condition

- Must hold for at least one non-sharable resource
- Not for sharable resource
- Example
- Non-sharable resource
- Tape drive, printer
- Sharable resource
- Read-only files

Hold and Wait Condition

The Hold and Wait condition must guarantee that when ever a process requests are source, it does not hold any other resources.

Protocol

Require each process to request and be allocated all its resources before it begins execution Allow process to request resources only when the process has n one.

Disadvantages of above protocol

- Low resource utilization
  - Many resources may be allocated but not used for long time. Starvation possible Process may have to wait indefinitely for popular resources.

No Preemption Condition

The third condition is that there is no preemption of resources that have already been allocated. If a process that is holding some resources requests another resource that cannot be immediatelyallocatedtoit,thenallresourcescurrentlybeingheldarereleased.

- Preempted resources are added to the list of resources for which the process is waiting.

- Process will be restarted only when it can regain its old resources, as well as then new ones that it is requesting.
- The required resource(s) is / are taken back from the process(s) holding it/ them and given to the requesting process

Disadvantages

- Some resources (e.g. printer, tap drives) cannot be preempted without detrimental implications. May require the job to restart

Circular Wait Condition

- Impose a total ordering of all resource types
- Requirethateachprocessrequestsresourcesinanincreasingorderofenumeration; if are source of type N is held, process can only request resources of types>N.

Disadvantages

- Adding a new resource that up sets ordering requires all code ever written to be modified Resource numbering affects efficiency
- A process may have to request are source well before it needs it, just because of the requirement that it must request resources in ascending order

Dead lock Avoidance

Deadlock avoidance involves making real-time decisions about whether granting are source request could lead to a potential deadlock. This approach requires prior knowledge of future resource requests from processes.

To implement deadlock avoidance, each process must provide some additional information about its resource needs. A commonly used model requires every process to declare in advance the maximum number of resources of each type that it may require.

The deadlock-avoidance algorithm continuously monitors the system's resource allocation state to ensure that circular wait conditions never occur. By carefully evaluating each request before granting it, the system can prevent deadlocks before they happen.

Resource allocation state

- The Resource allocation state is defined by the number of available and

allocated resources, and the maximum demands of the processes.

Two Approaches to Deadlock Avoidance

- Do not start a process if its demands might lead to deadlock
- Do not grant an increment al resource request to a process if this allocation might lead to deadlock.

Safe State

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

Definition

A resource allocation state is safe if the system can allocate resources to each process(upto its maximum) in some order and still avoid deadlock.

Safe Sequence

A sequence of processes<P_1,P_2,…P_n>is a safe sequence for the current allocation state if, for each P_i,the resources that $P_i$can still request can be satisfied by currently available resources + resources held by $P_j$with j <i.

- If $P_i$resource needs are not available, $P_i$can wait until all $P_j$have finished.
  - When $P_j$is finished, $P_i$can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1}can obtain its needed resources

Un safe State - If no such safe sequence exists, then the system state is said to be *unsafe*.

Facts

- If a system is in a safe state node a d locks.
- If a system is in unsafe state possibility of deadlock.
- Avoid ad ensure that a system will never reach an un safe state.

Resource-Allocation Graph Algorithm

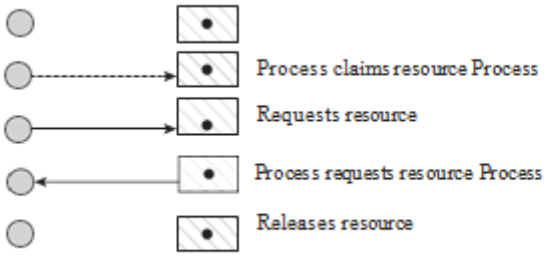
Resource-Allocation Graph Algorithm is used for deadlock avoidance when there is only

one instance of each resource type.

Claim edge

- Claim edge is a edge in which $P_iR_j$indicates that process $P_i$may request resource R_j represented by a dashed line. Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to claim edge. Resources must be claimed a priori in the system.
- If request assignment does not result in the formation of a cycle in the resource allocation graph safe state, else unsafe state.

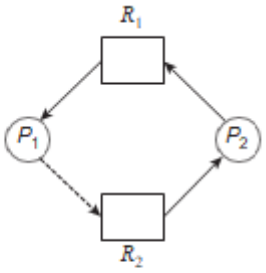
Claim Graph



Illustration

Suppose that process P_i requests a resource R_j. The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph using cycle-detection algorithm which requires an order of n^2operations (n-number of processes in the system).

- Cycle exists System allocation in an un safe state
- No Cycle exists System allocation in an safe state Consider the following esource-allocation graph Avoidance using resource allocation graphs –for one instance resources
- Add an extra type of arc-the claim arc to indicate future requests



Resource- allocation graphs for dead lock avoidance

Then check for loops:
In the above graph, suppose that P_2 requests R2 which is currently free cannot be allocated to P_2, since a cycle will be created in the graph. If P_1 requests R2 and P_2 requests R1 then a deadlock will occur.

Banker's Algorithm

- Banker's Algorithm is the deadlock-avoidance algorithm which is applicable to system with multiple instances of resource types.
- Each process must a prior I claim maximum use.
  - When a process requests are source it may have to wait.
  - When a process gets all its resources item return them in a finite amount of

time. A process requests resource – Is it safe ?

- A process terminates–Can I allocate released resources to a suspended process waiting for them?
- Anewstateissafeifandonlyifeveryprocesscancompleteafterallocationismade make allocation and then check system state and deal locate if unsafe

Data Structures for Banker's algorithm

Let n=number of processes, and m = number of resources types.

- **Available:** Vector of length m. If available [j] =k, there are k instances of resource type R_j available.
- **Max:** nxm matrix.Max [i,j] =k mean that process $P_i$may request at most k instances of R_j.

- **Allocation:** nxmmatrix. If Allocation[i,j]= k then P_i is currently allocated k Instances of R_j.

Safety Algorithm

The Safety algorithm is a algorithm used to find whether a system is in safe state or not which s given below:

Let Work and Finish be vectors of length m and n, respectively.
Initialize: Work=Available Finish[i] =false
For i=1,3,…,n
Find and i such that both: Finish [i] =false
Need I Work If no such i exists, go to step
4.
Work=Work + Allocation I Finish[i]=true goto step2.
If Finish[i]==true for all i, then the system is in a safe state.

Time Taken

It requires $m \times n^2$operations.

Resource-Request algorithm for Process P_i

Let $Request_i$= request vector for process P_i .If Request i[j]= k then process P_i wants k Instances of resource type R_j.

- If Request i Need i go to ste P_2. Otherwise, raise error condition, since process has exceed edits maximum claim.
- If Request i Available, go to step 3.Otherwise P_i must wait, since resources are not available.

Pretend to allocate requested resources to P_i by modifying the state as follows:

Available =Available =Request i; Allocation i =Allocation i + Request I;

Need i=Need i–Request i

If safe the resources are allocated to P_i.
If unsafe P_i must wait, and the old resource- allocation state is restored

Example of Banker's Algorithm

Consider a system with

- 5 processes P_0 through4;

- 3resourcetypes
- A(10instan c),
- B(5instances),and
- C(7instances)

Snap shot at time T0:

	Allocation	Max	Available
	ABC	ABC	ABC
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix **Need = Max–Allocation.**

Process	Need
	ABC
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Deadlock Detection

Let deadlock occur, then detect and recover some how. Under the environment of a system employs neither a deadlock prevention nor a deadlock avoidance, it should provide:

- **Find:** An algorithm that examines the state of the system to determine whether a dead- lock has occurred
- **Recover:** an algorithm to recover from the deadlock

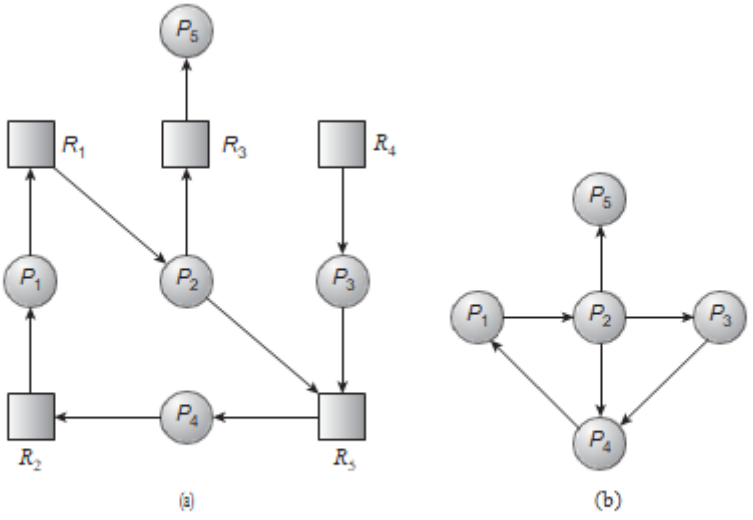
The overhead of the detection includes not only the run-time cost of maintaining the necessary information and execute the detection algorithm but also the recovery part

Detection Algorithms

- **Single** instance of are source type. Use a wait-for graph
- **Multiple** instances of a resource type. Use the algorithm similar to banker's algorithm

Deadlock Detection

- Maintain wait-for graph Nodes are processes. P_i P_j if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.



(a) Resource- allocation graph (b) Wait-for graph Several Instance sofa Resource Type

Recovery from Deadlock

When a system detects a deadlock, it must take action to resolve the issue. One

approach is to notify the system operator, allowing them to handle the situation manually. Alternatively, the system can automatically recover from the deadlock using one of two strategies: terminating processes to break the circular wait or preempting resources from the deadlocked processes.

Process and Thread Termination

One way to resolve a deadlock is byte reminating one or more processes involved. This can be done using two methods:

1. **Terminate All Deadlocked Processes:** This approach guarantees the deadlock is resolved but comes at a significant cost. Any progress made by these processes will be lost, requiring them to restart from scratch, which may lead to wasted computation time.
2. **Terminate Processes One at a Time:** Instead of stop Ping all processes at once, the sys-tem can terminate them one by one, running a deadlock detection algorithm after each termination to check if the deadlock is resolved. While this method prevent sun necessary terminations, it introduces additional overhead due to repeated checks.

Process termination is not always straight forward. If a process Is in the middle of updating a file or modifying shared data while holding a mutex lock, abruptly stopping it may leave the system in an inconsistent state. The operating system must carefully restore resources, but there is no guarantee that data integrity will be maintained.

Whendecidingwhichprocessestoterminate,thesystemconsidersfactorssuchas:

- Process priority
- Time already spent computing and estimated time to completion
- Resources the process has used and how easily they can be reclaimed
- Additional resources required for the process to finish
- The total number of processes that need to be terminated

Resource Preemption

Another method to resolve deadlocks is resource preemption, where the system forcefully takes resources away from some processes and reallocates them too there until the deadlock is resolved. However, preemption introduces its own challenges:

1. **Selecting a Victim:** The system must decide which processes and resources to preempt while minimizing the overall cost. Factors such as the number of resources held and the process's runtime are considered in making this decision.

2. **Rollback:** When a process loses are source, it may not be able to continue execution. In such cases, the system rolls the process back to a previous state where it can safely restart. The simple stroll back strategy is to completely restart the process, thoughroll- in git back only as much as necessary would be more efficient. However, this requires the system to maintain detailed records of each process's state.
3. **Starvation Prevention:** If the same process is repeatedly chosen as a victim for resource preemption, it may never complete its task, leading to starvation. To prevent this, the system limit show many times a process can be preempted before it must be allowed to proceed. One common solution is to factor in the number of rollbacks when determining which process to preempt next.

UNIT IV

MEMORY MANAGEMENT

Main Memory: Background – Swapping - Contiguous Memory Allocation – Segmentation – Paging - Structure of the Page Table – Virtual Memory: Background - Demand Paging - Copy-on-Write – Page Replacement - Allocation of Frames – Thrashing.

Memory Management : Background

- Storage management is the process of Controlling and coordinating computer memory Assigning portions called blocks to various running programs to optimize overall system performance (Allocating Process)
- Freeing it for reuse when no longer needed(Deallocating process)
- Memory management resides in hardware, in the OS(operating system),and in programs and applications.

Instruction execution cycle

Allcomputershaveaninstructionexecutioncycle.Abasicinstructionexecution cycle can be broken down into the following steps:

- Fetch cycle
- Execute cycle

Steps in Fetch/Execute Cycle

1) The instruction is fetched from memory addressed by PC(Program Counter)
2) PC is advanced to address the next instruction
3) Instruction Decoder decodes the instruction held in IR
4) Instruction is then executed
5) Results are stored back to the memory

The above process is repeated until the complete set of instructions is fetched.

Basic Hardware

- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock, but, main memory can take many cycles

Example

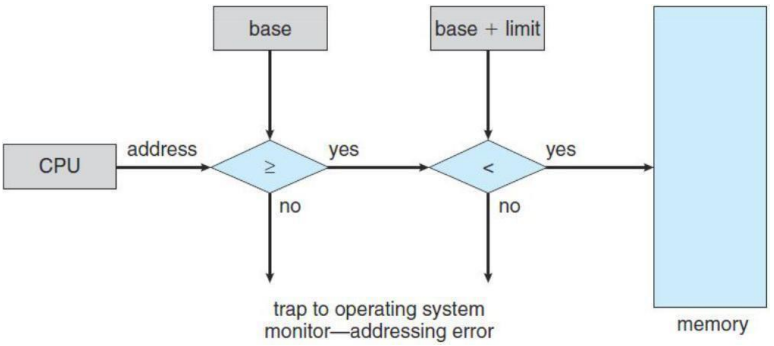
Assume

- Base registers= 300040
- Limit register=120900

  The program can legally access all addresses from 300040 through 420939.

Memory Space Protection

Memory protection is a way to control memory access rights on a computer.



Hardware address protection with base and limit registers

CPU hardware compare sever address generated in user mode with the registers

- A program executing in user mode attempt to access OS memory or other users' memory results in a trap to the OS, which treats the attempt as a fatal error
- Thispreventsauserprogramfromaccidentallyordeliberatelymodifyingthe code or data structures of either the OS or other users

Address Binding

- Address Binding is the process of converting the (relative or symbolic) address used in a program to an actual physical (absolute) RAM address.
- Program must be brought into memory and placed within a process for it to be executed.

Multistep Processing of a User Program

- User programs go through several steps before being run

- Program components do not necessarily know where in RAM they will be loaded
- RAM deals with absolute addresses
- Logical addresses need to be bound to physical addresses at some point.

Binding Steps

The stages involved in the binding of instructions and data to memory addresses is listed below

- Compile time
  - If memory location is known a priori, absolute code can be generated; must recompile code if starting location changes.

- Load time
  - If memory location is not known at compile time, then the relocatable code must be generated.

- Execution time
  - Binding delayed until runtime if the process can be moved during its execution from one memory segment to another.
  - Need hardware support for address maps.

Logical address

Logical address is the address generated by the CPU. It is also refer red to as virtual address.

Physical address

Physical address is the address seen by the memory unit.

Virtual Address

Virtual Address is the logical address results in the execution-time address- binding scheme.

Logical address Space

These to fall logical address generated by the program is called Logical address Space.

Physical address Space

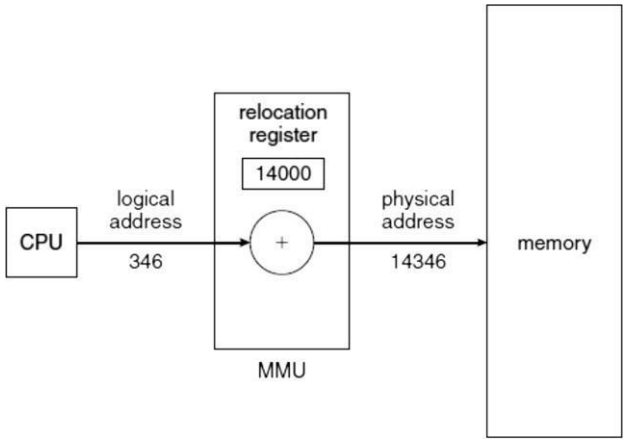
These to fall physical address corresponding to the logical addresses is called Physical address Space.

Note:

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes;
- Logical (virtual)and physical addresses differ in execution-time address-binding scheme.

Memory-Management Unit (MMU)

- MMU is a hardware device that maps virtual to physical address at run-time.
- In a simple MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with logical addresses; it never sees the real physical addresses.



Dynamic relocation using a relocation register

Dynamic loading

Assume that the whole program and data for the program to be in memory in order for a process to execute.

Problem

Even if you code is small, the memory needed will be large. Why?

- Each program must be linked with the whole standard library.
- The whole library must be in memory even though most functions in that library are not used.

Solution (Dynamic Loading)

Dynamic loading means loading the library(or any other binary for that matter) into the memory during load or run-time

The steps involved in the dynamic loading are listed below

- Don't load the whole program into memory before run a program
- Just load the main routine at the beginning and execute
- When a new routine is called, it is loaded into the memory
- The executable will still be large, but the memory used will be smaller

  - Routines that are not used will not be in the memory
  - Dynamic loading can introduce sign if I cant time penalty
  - Space/time trade-off

Advantage

- Better memory-space utilization
- Used routine is never loaded
- Useful when large amounts of code are needed to handle in frequently occurring cases
- No special support from the operating system is required implemented through program design.

Program vs. Memory sizes

What to do when program size is larger than the amount of memory/partition (that exists or can be) allocated to it?

There are two basic solutions with in real memory management:

1) Dynamic Linking(Libraries–DLLs)
2) Overlays

Dynamic Linking and Shared Libraries Static Linking Libraries

The basic idea of static linking is to resolve all addresses before running program. Here Compiler compiles each file, but leaves external references unresolved. But Static linker resolves all previously unresolved references.

In static linking, the code segments will be copy to each executable.

Advantages

- Changing libraries does not affect previously compiled programs.
- No address resolution necessary at runtime

Disadvantages

- The executable size will be larger.
- Multiple copies of code might be in memory
  - Wastes both disk space and main memory
- New version of library=>must recompile

Dynamic Linking Libraries

Dynamic linking is the process that links the external shared libraries at runtime. It is also called as late linking.

Stub

With dynamic linking, a stub is included in the image for each library-routine reference. A Stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine, or how to load the library if the routine is not already present.

How it works

- Use as tub to replace the whole library routine
- Enough information to locate the library routine
- When the stub is executed, it replaces itself with the address of the routine and executes the routine.
  - Only one copy of library code is needed for all programs.
  - Such library is also known as a shared library.

Note: The executable is much smaller with such libraries.

Problem

One problem with dynamically linked executable is that it is less portable: assume the shared library is available and the same Statically linked executablesare more self-contained.

SWAPPING

Swapping refers to moving entire processes in and out of main memory.

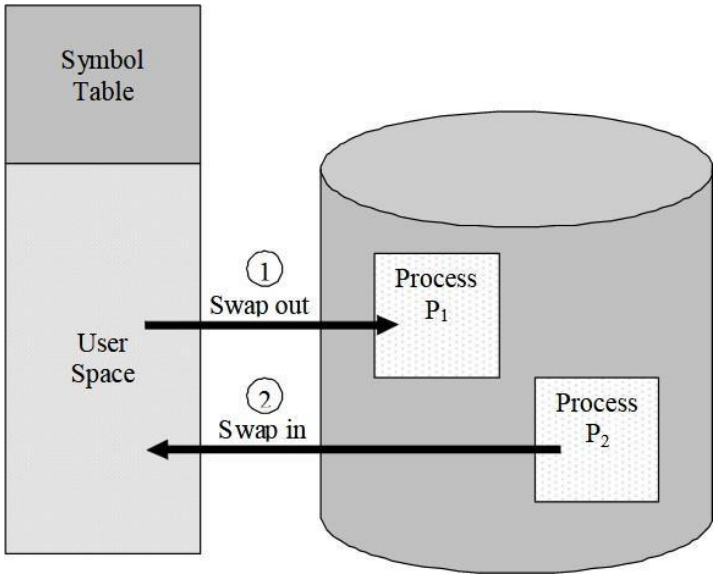
Purpose - A process to be executed should be available in the main memory for execution. All the process are stored in the backing store temporarily. The purpose of swapping is to move the process between main memory and backing store.

Round Robin CPU-Scheduling Algorithm

- Each process is provided a fix time to execute called times lice or quantum
- CPU scheduler picks the process from the circular/ready queue, set a timer to interrupt It after the given time slice /quantum and dispatche sit
- If process has burst time less than defined time slice/quantum
  - Process will leave the CPU after the completion
  - CPU will proceed with the next process in the ready queue/circular queue

  Else If process has burst time longer than defined time slice/quantum.

Schematic view of swapping



Swapping of two processes using disk as a balance store

Operating System

- Timer will be stopped. It causes interruption to the OS
- Executed process is swapped out of the memory by the memory manager and it is placed at the tail of the circular / ready queue using context switch
- CPU scheduler then proceed by selecting then ext process in there add queue by swapping into the main memory.

Advantages

- Starvation free
  - Every process will be executed by CPU for fixed interval of time (which is set as time slice). So, in this way no process left waiting for its turn to be executed by the CPU.
- Simple and easy to implement

Priority-based scheduling algorithms

- Each process is assigned a priority. Process with highest priority is to be executed first and so on
- Processes with same priority are executed on first come first serve basis
- Priority can be decided based on memory requirements, timer requirements or any other resource requirement.

- Follows Rollout, Rollin
  - When higher-priority processor requests service
- Lower-priority process is swapped out so higher-priority process can be loaded and executed.

Address Binding

If the address binding is do neat

- Assembly or load time–swapping of process cannot be moved to different memory locations
- Execution time–swapping of process moved to different memory locations.

Backing store

Swapping needs a backing store which is a fast disk large enough to accommodate copies of all memory images for all users. It must provide direct access to these memory images.

System maintains a ready queue of ready-to-run processes which have memory images on disk

Transfer Time

The total transfer time is directly proportional to the amount of memory swapped. Note that context switching time in this scheme is quite high.

Example

Assume that

- User process is 100MB in size and
- Backing store is a standard hard disk Transfer rate of 50MB per second

The actual transfer of the100-MBprocessto or from main memory

=100 MB /50 MB per second

=2 seconds

CONTIGUOUS MEMORY ALLOCATION

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. The memory is usually divided into two partitions: one for the resident operating system and one for the user processes.

We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

Memory Allocation

Memory Allocation is the process of assigning blocks of memory on request for the program execution.

Memory Allocation mechanism

Operating system uses the following memory allocation mechanism.

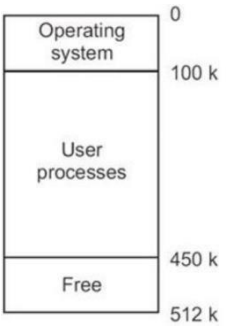
- Single-partition allocation
- Multiple-partition allocation

Single-partition allocation

In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-systemcode and data.

Main memory usually has two partitions

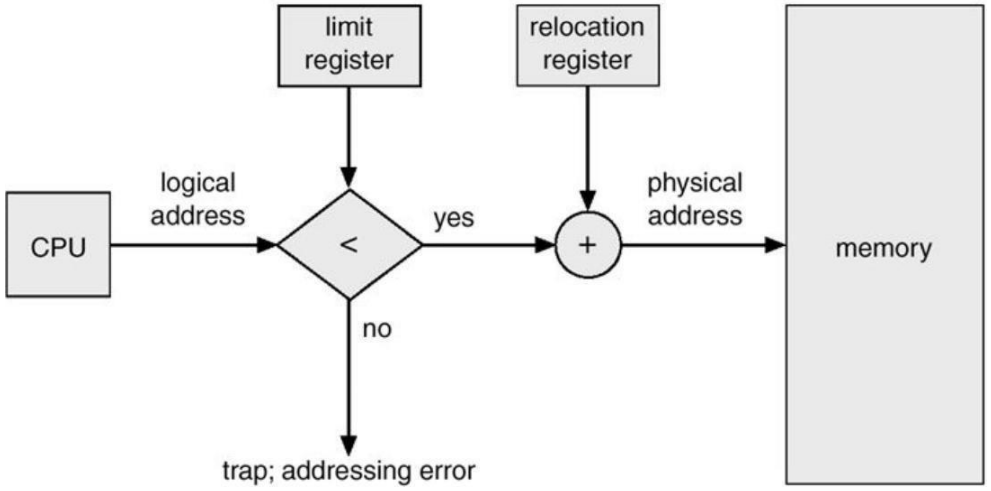
- **Low Memory**—Operating system resides in this memory
- High Memory — User processes then held in high memory



Memory Protection

- Memory protection is to prevent a process from accessing memory it does not own. Memory protection can be achieved by combining relocation register with the limit register. Relocation-register scheme provides an effective way to allow the operating-system size to change dynamically.

- Relocation-register scheme used to provide the memory protection by
  - Allows protection against user programs accessing are as that they should not allows programs to be relocated to different memory starting addresses as needed
  - Allows the memory space devoted to the OS to grow or shrink dynamically as needs change



Hardware support for relocation and limit registers

- Relocation register contain value of smallest physical address
- Limit register contains range of logical addresses
- The dispatcher loads the relocation and limit registers with the correct values as part of the context switch
- Each logical address must be less than the limit register.
- Then, MMU maps the logical address dynamically by adding the value in their location register which is sent to memory.

Advantages

- Simple, easy to understand and use

Disadvantages

- Leads to pre utilization of processor and memory
- Users job is limited to the size of available memory

Memory Allocation Fixed
Partitions Scheme

In this type of allocation, main memory is divided into an umber of fixed-sized partitions where each partition should contain only one process. When a partition is free

,a process is selected from the input queue and is loaded in to the free partition. When the process terminates, the partition becomes available for another process.

Advantages

- Any process whose size is less than or equal to the partition size can be loaded into any available partition.
- Supports multi programming.

Disadvantages

- If a program is too big to fit into a partition use over layer technique.
- Memory use is in efficient, i.e., block of data loaded into memory may be smaller than the partition. It is known as internal fragmentation.

Variable Partition Scheme

- Partitions are of variable length and number.
- Process is allocated exactly as much memory as required

Dynamic Allocation Placement Algorithms

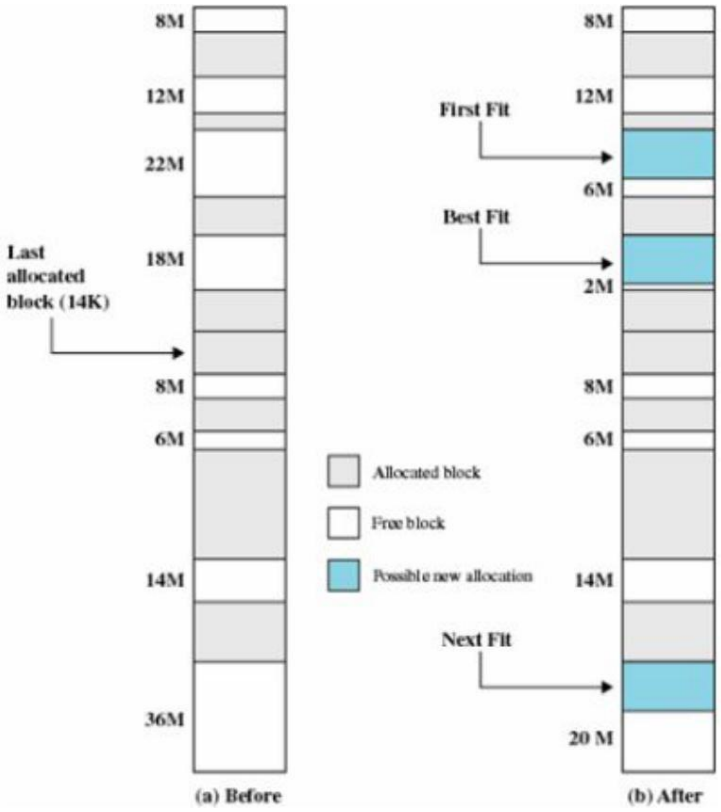
The following algorithm is used to decide which free block to allocate to a program

First fit

- Allocate the first hole that is big enough;
- Process is placed in the first hole it can fit in.

Best fit

- Allocate the smallest hole that is big enough; must search entire list, unless ordered by size.
- Produces the smallest left-over hole.



Performance Issues

- Best Fit usually per forms the worst, since it leaves behind the smallest possible hole
- First Fit is simplest to implement and usually fastest and best
- Next fit is usually a little worse than First Fit

Fragmentation

Fragmentation occurs when a system contains total free memory to satisfy any request, but that can't be utilized.

Internal fragmentation

Internal fragmentation is that the allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Solution

- Blocks are usually split to prevent internal fragmentation

External fragmentation

External fragmentation is done when the total memory space exists to satisfy a request, but it is not contiguous.

- Many small holes.

Solution

- Compaction
- Non contiguous logical-address space
  - Paging
  - Segmentation

Compaction

The goal is to shuffle memory content stop place all free memory to get there in one large block.

Compaction is

- Possible only if relocation is dynamic, and is done at execution time.
  - Determine its cost
- Impossible only if relocation is static ,and is done at assembly or load time.

Simple Compaction algorithm

- Move all processes toward one end of memory.
- All holes move in the other direction which produces large hole of memory.
- Drawback
  - Very expensive scheme.

I/O problem

1) Latch job in memory while it is in I/O
2) Do I/O only into OS buffers.

PAGING

Paging is a memory-management scheme that permits the logical address space of a process can be non-contiguous; process is allocated physical memory whenever the latter is available.

Basic Method

Frames

Physical memory is divided into fixed-sized blocks called frames(size is power of 2, between 512 bytes and 8192 bytes).

Pages

Logical memory is also divided into blocks of same size called pages. The page size is Defined by the hard ware and is typically a power of 2 varying between 512 by test and 16 megabytes per page. The selection of a power of 2 as a page size makes the translation of logical address in to a page number and off set easy.

Torun a program of size n pages, need to find n free frames and load program.

Illustration of Paging

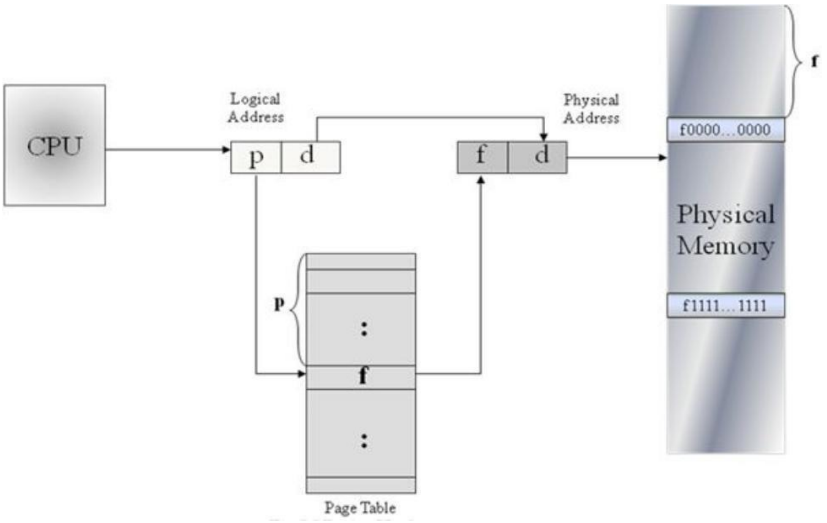
The hardware support for paging is illustrated below.

Address Translation Scheme

Address generated by CPU is divided into:

- Page number(p)
  - Used as an index into page table which contain base address of each page in physical memory
- Page offset(d)
  - Combined with base address to red fine the physical memory address that is sent to the memory unit.
- Page Table
  - The page table contains the base address of each page in physical memory. Setup a page table(for each process)to translate logical to physical addresses.

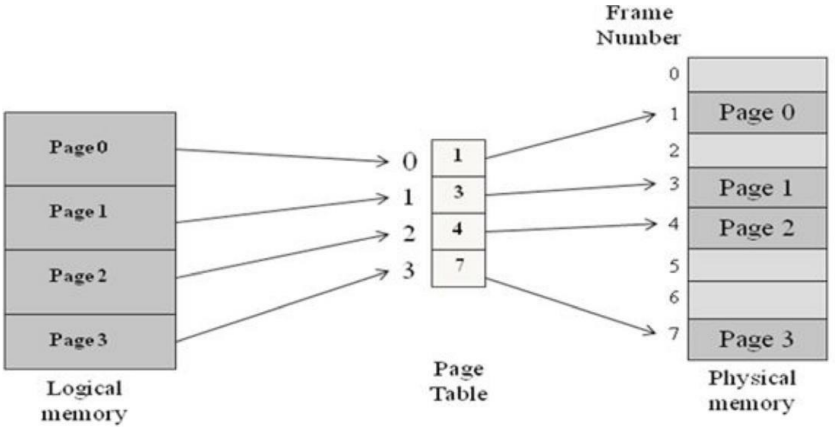
Note: Internal fragmentation is possible.



Paging Hardware

Paging Model of Memory

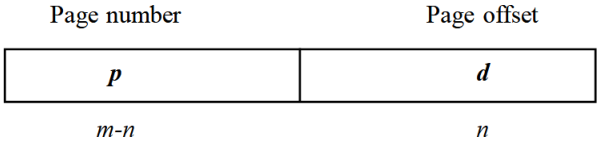
The paging model is logical and physical memory is given below then



Paging Model of logical and physical memory

If size of logical-address space is 2^m and page size is 2^n addressing units(bytes or words),

- High-order m-n bits of a logical address–page number
- Low-order n bits of a logical address–page offset

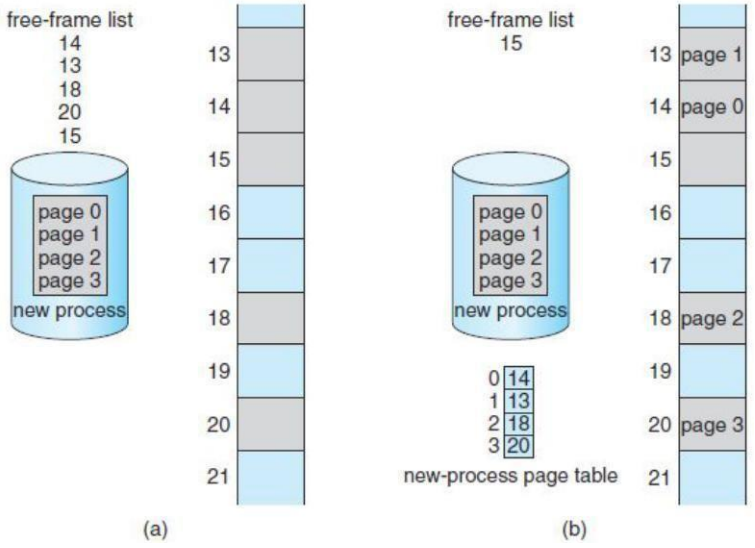


Paging Example

Consider the memory of
Page Size =4 bytes
Physical Memory=32 Bytes (32/4=8Pages)

Paging Examples

Memory Allocation in Paging



Free Frames(a)Before allocation(b)After allocation

When a new process arrives, its size, in pages, is determined. If the process has n pages then n frames must be available in the physical memory.

The first page of the process is then loaded into the first available frame, the next into next available frame and so on.

Frame table

Since OS is managing the physical memory, it must be aware of the allocation details of physical memory (allocated and free frames). This information is generally kept in a data structure known as **frame table** which has one entry for each frame and indicates whether it is free or not and if allocated to which page of which process.

Aspect of paging

- An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory
- User program views memory as one single space containing only this one program
    - Actually, the user program is scattered throughout physical memory
    - Also holds other programs
- The difference between the user's view of memory and the actual physical memory is reconciled by the **address-translation hardware.**

Frame Table

OSmustbeawareoftheallocationdetailstomanagephysicalmemorywhichframes are allocated, which frames are available, how many total frames there are kept in a data structure called a **frame table**

- Has one entry for each physical page frame
    - Indicting whether the latter is free or allocated
    - If it is allocated, to which page of which processor processes

Hardware Support

Each OS has its own methods for storing page tables

1) A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block
2) When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table

Implementation of Page Table

1) Implemented as a set of **dedicated registers**

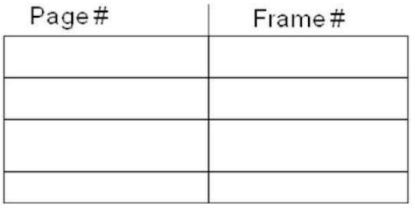
- Built with very high-speed logic to make the paging-address translation efficient
- Every access to memory must go through the paging map
- CPU dispatcher reloads these registers
  - Instructions to load or modify the page-table registers are privileged
- DECPDP-11
  - Address contains of 16bits, and page size is 8KB (13 bits)
  - Page table consists of eight entries that are kept in fast

2) Keep Page Table in main memory:
   - **Page-table base register(PTBR)**points to the page table.
   - **Page-table length register(PTLR)**indicates size of the page table.

   However, in this scheme, every data/instruction access requires won memory accesses one for the page table and one for the data/instruction

3) Keep Page Table in hardware (in MMU)
   - However, page table can be large–too expensive.
4) The two memory accesses problem can be solved by combining mechanisms1 & 2:
   - Use a special fast look up hardware cache called Associative Memory (Registers) or Translation Look-aside Buffer (TLB)
     - Associative, high-speed memory
     - Each entry contains
     - A key (tag)
     - Value
     - Enables fast parallel search:
     - Item is compared with all keys simultaneously

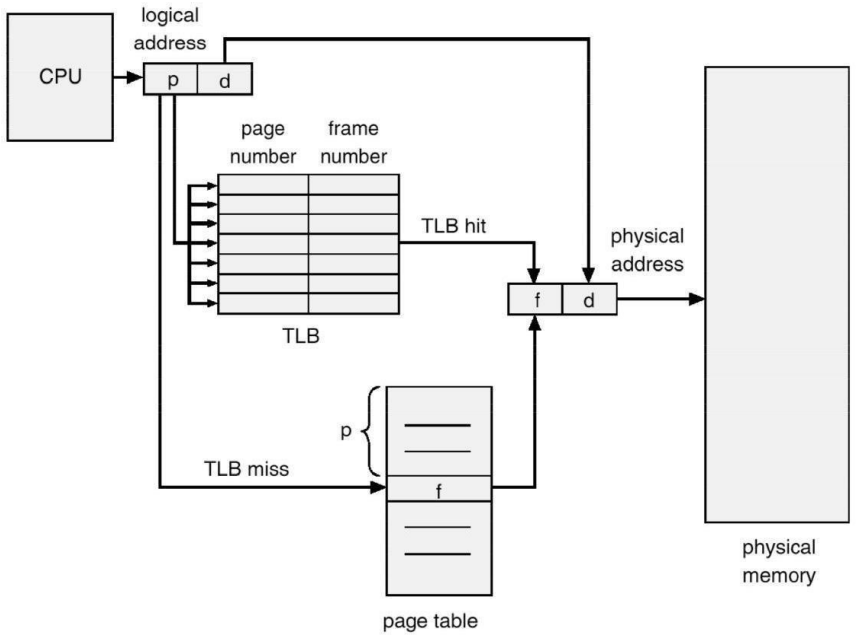
Associative Registers



Associative memory allows parallel search. It contains page table entries that have been most recently used. Functions same way as a memory cache.

Given a virtual address, processor examines theTLB

- If page table entry is present (a hit),the frame number is retrieved and the real address is formed.
- If page table entry is not found in the TLB (a miss),the page number is used to index the process page table
- First checks if page is already in main memory
  - If not in main memory a page fault is issued
- The TLB is updated to include the new page entry
- Note that TLB must be flushed every time a new page table is selected (during context switching, for example).



Paging Hardware with TLB

- Address translation(p, d)
  - If p is in associative register, get frame # out.
  - Otherwise get frame# from page table in memory.

Effective Memory Access Time

- Associative Lookup =20 nano second

SEGMENTATION

Segmentation is the Memory Management Scheme that supports user view of memory.

Logical address space is the collection of segments

Basic Method

Program

A program is a collection of segments.

Segment

A segment is a logical unit of a program such as

- Main program, procedure, function
- Local variables, global variables, common block
- stack, symbol table, arrays

Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment.

Segmentation Architecture

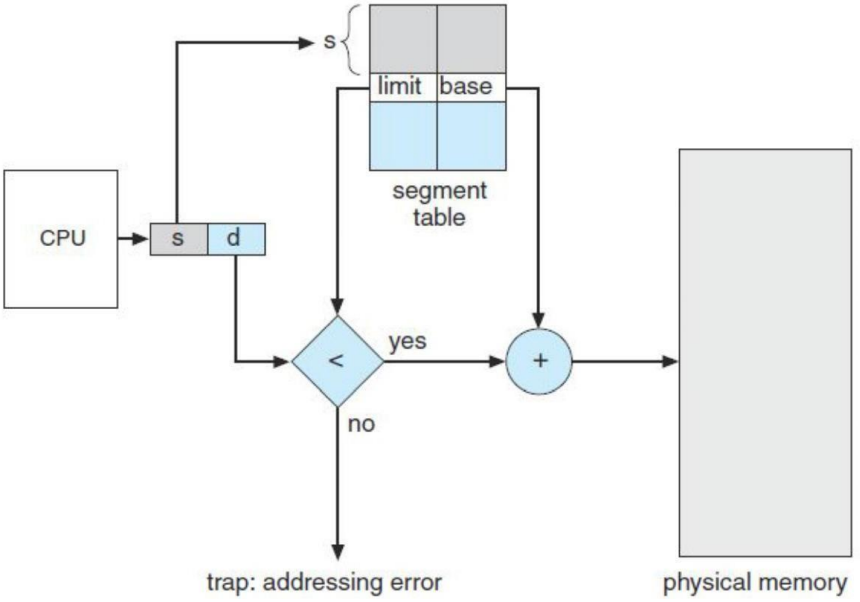
- Segments are numbered and referred to by a segment number, thus a logical address consists of a tuple:

Logical Address

Segment Number	Offset

- Segment table–maps two-dimensional physical addresses; each table entry has:
  - base–contains the starting physical address where the segments reside in memory
  - limit–specifies the length of the segment

- **Segment-table base register(STBR)**points to the segment table's location in memory
- **Segment-table length register(STLR)**indicates number of segments used by a program;
    - Segment numbers is legal ifs<STLR.



Segmentation Hardware

Example

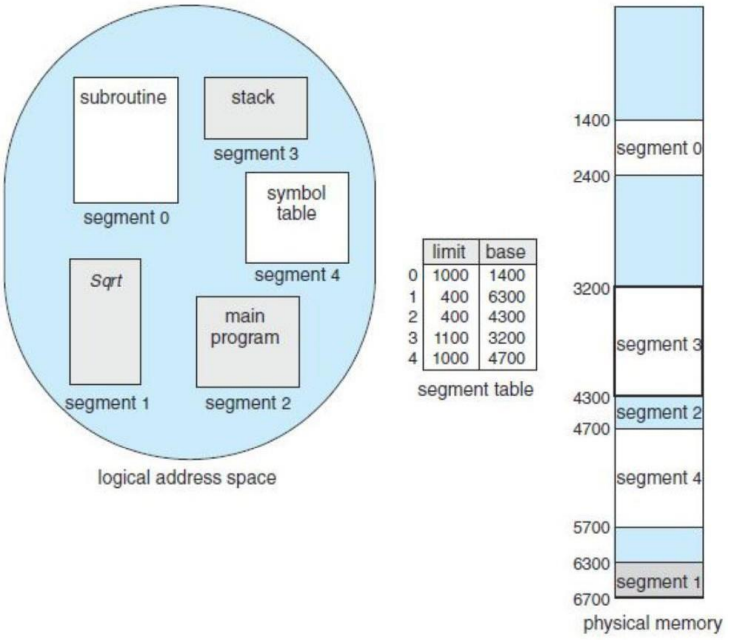
Suppose a program is divided into 5 segments:

- Segment 0:Subroutine
- Segment 1:sqrt function
- The segment table is as

follows:

Segment	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

SEGMENTATION WITH PAGING



Segmentation Examples

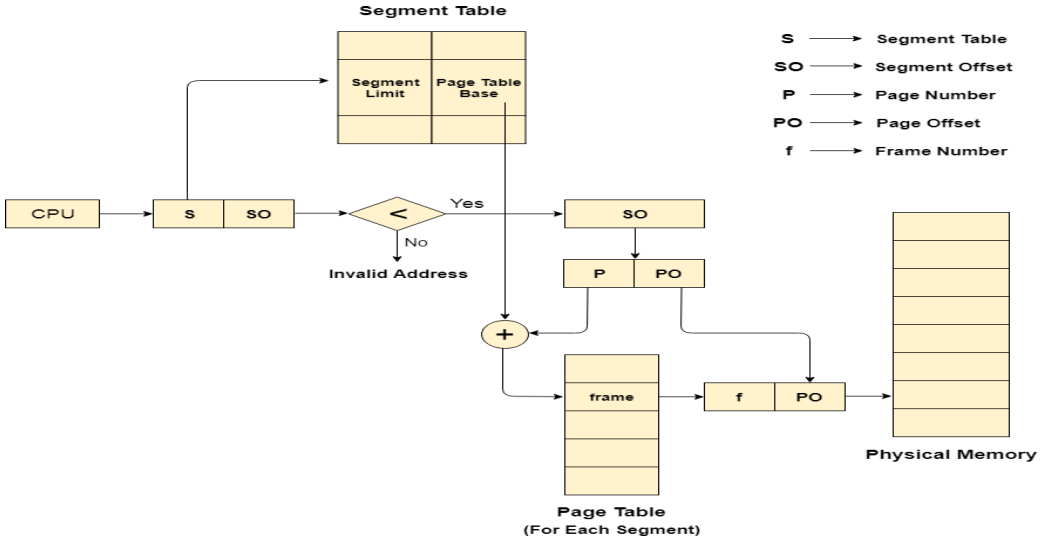
Pure segmentation is not very popular and not being used in many of the operating systems. However, Segmentation can be combined with Paging to get the best features out of both the techniques.

In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages.

1. Pages are smaller than segments.
2. Each Segment has a page table which means every program has multiple page tables.
3. The logical address is represented as Segment Number (base address), Page number and page offset.

Segment Number→ It points to the appropriate Segment Number.
Page Number → It Points to the exact page within the segment
Page Offset → Used as an offset within the page frame

Each Page table contains the various information about every page of the segment. The



Advantages of Segmented Paging

1. It reduces memory usage.
2. Page table size is limited by the segment size.
3. Segment table has only one entry corresponding to one actual segment.
4. External Fragmentation is not there.
5. It simplifies memory allocation.

Disadvantages of Segmented Paging

1. Internal Fragmentation will be there.
2. The complexity level will be much higher as compare to paging.
3. Page Tables need to be contiguously stored in the memory.

VIRTUAL MEMORY

BACKGROUND

Motivation of the memory-management strategies

- Keep several processes in memory to improve a system performance
  - Allows Multi programming

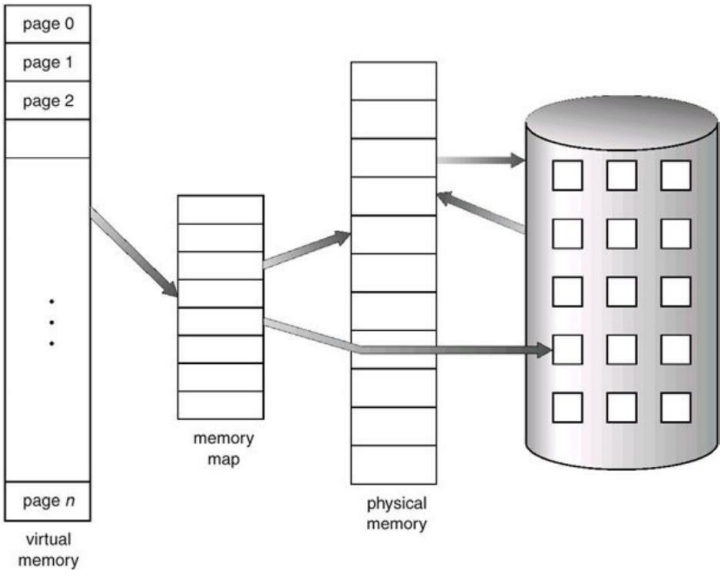
Limitation of the memory-management strategies

- Requires the entire process to be in memory for execution

Virtual memory

Virtual memory is a technique that allows execution of processes that are not completely in the physical memory.

It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory. The process of translating virtual addresses into real addresses is called mapping.



Virtual Memory that is Larger than Physical Memory

Implementation

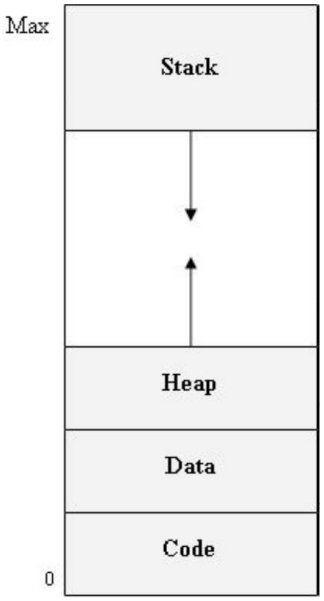
Virtual memory can be implemented via:

1) Demand paging
2) Segmentation System

Virtual Address Space and Memory Space

Virtual Address Space (VAS) or address space is the set of virtual addresses. Memory Space is the set of physical addresses in main memory directly addressable for processing.

- Heap grows upward in memory as it is used for dynamic memory allocation
- Stack grows downward in memory through successive function calls
- Virtual address space is the blank space(or hole)between the heap and the stack.

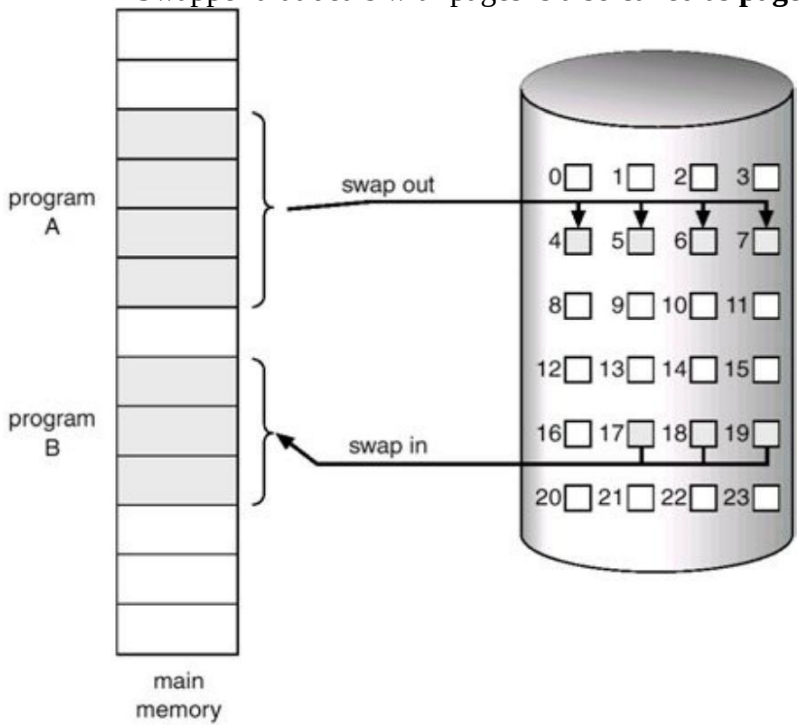


- Holes can be filled as the stack or heap segments grow
  - Shared by two or more processes through page sharing

DEMAND PAGING

- Demand-paging system is a paging system with swapping. It initially loads pages only into memory (not the entire process) as they are needed.
- Page is needed reference to it
  - Invalid reference abort
  - not-in-memory bring to memory Idea
- Lazy swapper
  - Process of swapping a page into memory only when it is needed

- Swapper that deals with pages is also called as **pager.**



Transfer of a Paged Memory to Contiguous Disk Space

Benefits

Demand paging has the many benefits such as

- Less I/O needed Less swaptime
- Less memory needed
- Faster response
- More users

Basic Concepts

Valid-Invalid Bit Scheme

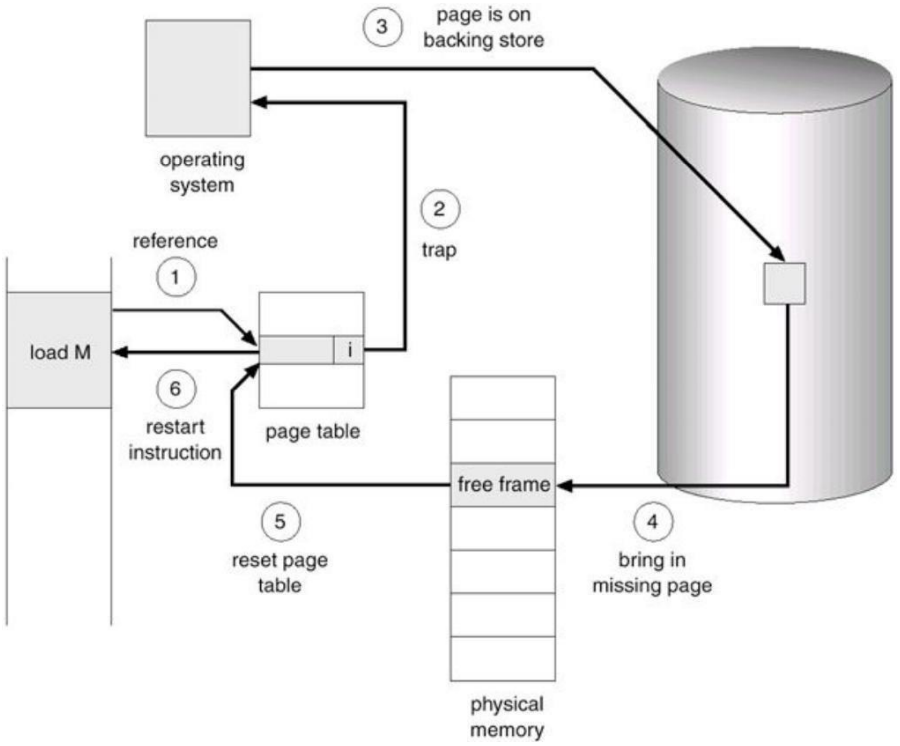
The Valid-Invalid Bit Scheme is used to distinguish between the pages that are in memory and the pages that are on the disk.

With each page table entry, a valid-invalid bit is associated

Handling a Page Fault

The procedure for handling this page fault is straightforward

1) Operatingsystemchecksaninternaltabletodeterminewhetherthereference was a **valid** or an **invalid** memory access.
2) Terminatetheprocessifthereferencewasinvalidifitwasvalid,butwehave not yet brought in that page, we now page it in.
3) Swap page into frame
   - Find a free frame from the free-frame list
4) Schedule a disk operation to read the desired page into the newly allocated frame.
5) When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
   - Set validation bit=**v**
6) Restart the instruction that was interrupted by the trap
   - Process can now access the page as though it had always been in memory



Pure demand paging

Pure demand paging is a scheme that never brings a page into memory until it is required.

Requirements for Demand Paging

The hardware to support demand paging include

1) Page table
   - Table that has ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.
2) Secondary memory
   - Memory that holds the pages that are not present in main memory.

Performance of Demand Paging

The performance of Demand Paging is measured by computing the effective access time for a demand paged memory.

Let p be probability of a Page Fault Rate 0 p1.

- If p =0 no page faults
- If p =1,every reference is a fault

Effective Access Time(EAT)

The Effective Access Time(EAT)is computed as follows

EAT=(1–p)x memory access +px page fault time.

Page fault time=page fault overhead +[swap page out]

+swap page in+ restart overhead)

Procedure for page fault

In order to compute EAT, how much time is needed to service a page fault. A page fault causes the following sequence to occur.

1) Trap to the OS
2) Save the user registers and process state
3) Determine that the interrupt was a page fault
4) Check that the page reference was legal and determine the location of the page on the disk

5) Issue a read from the disk to a free frame
   a) Wait in a queue for this device until the read request is serviced
   b) Wait for the device seek and /or latency time
   c) Begin the transfer of the page to a free frame
6) While waiting, allocate the CPU to some other user
7) Receive an interrupt from the disk I/O subsystem (I/O completed)
8) Save the registers and process state for the other user
9) Determine that the interrupt was from the disk
10) Correct the page table and other tables to show that the desired page is now in memory
11) Wait for the CPU to be allocated to this process again
12) Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Copy on Write:

Page-fault service Time

The three major components of the Page-fault service Time are

1) Service the page-fault interrupt
2) Read in the page
3) Restart the process

The1stand3rd tasks can be reduced to several hundred instructions

4) May take from 1 to100microseconds each

Example

Memory access time=100nanoseconds

Average page-fault service time=25milliseconds EAT =(1–p) □ 100+p(25milli seconds)

=(1–p)□ 100+p□ 25,000,000

=100+p□ 24,999,900

If one access out of 1,000 causes a page fault, Then EAT=25 micro seconds.

Allow fewer than one memory access out of 2,500,000 to page-fault.

PAGE REPLACEMENT

Page replacement algorithm decides which memory pages to be paged out to disk (Swap out) to allocate memory for another page.

It is needed when the operating system uses paging for managing virtual memory.

Types of Page replacement algorithm

1) FIFO Page Replacement
2) Optimal Page Replacement
3) LRU Page Replacement

Basic Steps

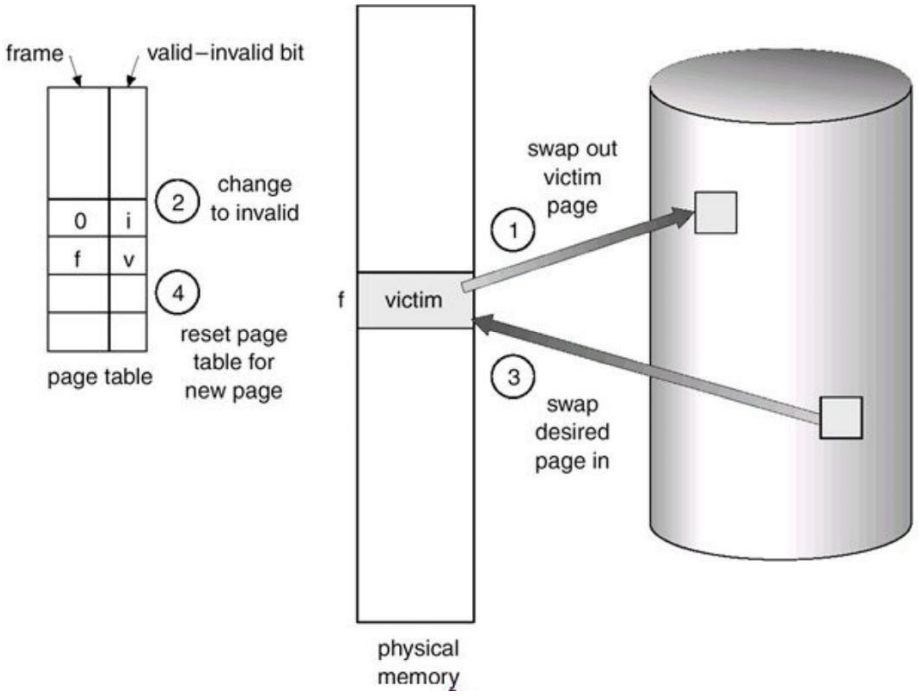
When there is a page fault, the referenced page must be loaded. The steps involved in replacing the referenced page is mentioned below

1) Find the location of the desired page on disk
2) Find the free frame in main memory
   - If there is a free frame available in main memory, use it.
   - If there is no free frame available in main memory,
   - Using page replacement algorithm, select an existing page for replacing with the new referenced page in memory
   - The page being replaced is referred as the victim page

   If the victim page has been modified, it must be copied back to disk(swapped out). Change the page and frame tables accordingly.
3) Read the desired page into the(newly)free frame. Update the page and frame tables.
4) Restart the process.

FIFO Page Replacement

- FIFO stands for First-In First-Out
- Simple page replacement algorithm
- Chooses the—oldest ‖page in the memory



Page Replacement

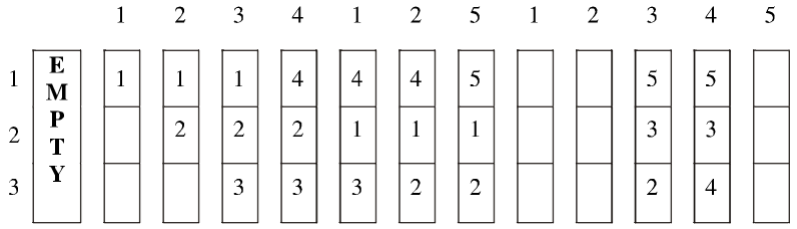
Implementation Steps

- Create a FIFO queue to hold the identifier so fall pages in memory Initially all the frames are empty
- The references are brought into the frames in FIFO manner
  - First n references are inserted at the tail of the queue
  - Next n+1th frames are checked with the existing frames in queue
    - If it is available, then no page fault
    - If it is not available, the n page fault occurs. Replace the reference at the head of the queue
- Repeat the above process until all references is allocated.

Illustration

Referencestring:1, 2,3,4,1,2, 5,1,2,3,4, 5

- Initially all the 3frames are empty
- First3 frames are inserted at the tail of the queue
- When reference 4 comes, it is checked with the existing frames, it is not available and also there is no empty frame sin the queue
  - Thereference4isreplacedwiththeframes at the head of the queue



Number of page fault=9

The page fault occurs for the given reference string is shown below

Reference string:	1	2	3	4	1	2	5	1	2	3	4	5
Misses:	×	×	×	×	×	×	×	√	√	×	×	√

Fault rate=9/12=0.75

Belady's Anomaly

The page fault rate may increase as the number of allocated frames increases

- More frames(can) more page faults(but not generally)

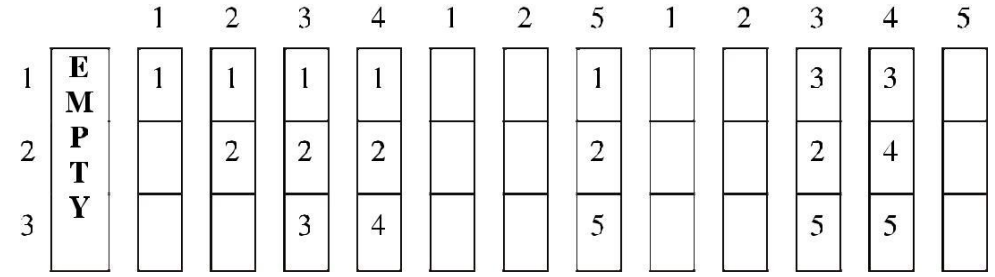
- Easy to understand and program

Optimal Page Replacement

- Algorithm that yields the lowest possible page fault rate
- Replace the page that will not be used for the longest period of time in the future
- Uses the time when a page is to be used
- Also called as OPT or MIN.

Implementation Steps

- Initially all the three frames are empty



Number of page fault=7

The page fault occurs for the given reference string is shown below

Reference string:	1	2	3	4	1	2	5	1	2	3	4	5
Misses:	×	×	×	×	√	√	×	√	√	×	×	√

Fault rate=7/12=0.58

Merits:

- Better than FIFO algorithm since number of page faults is less in optimal
- Never suffer from Belady‘s anomaly.

Demerits:

- Difficult to implement because it requires future knowledge of the reference string.

LRU Page Replacement

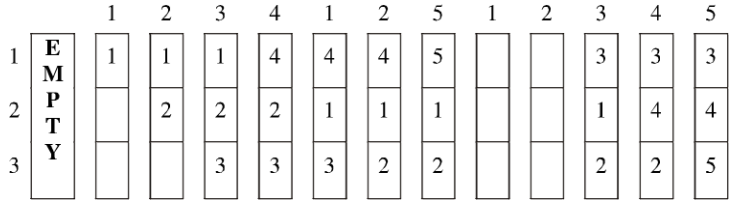
- Use recent past as an approximation of near future.
- Replace the page that has not been used for the longest period of time in past. Implementation Steps
- Initially all the three frames are empty
- The references are brought into the frames one by one
  - First three references are brought into empty frames-Page faults
  - Remaining frames are checked with the existing three frames
    - If it is available, then no page fault

- If it is not available, then page fault occurs. Replace the page that has not been used for the longest period of time with the current reference.
- Repeat the above process until all references is allocated.

Illustration

Reference string: 1, 2,3,4,1,2, 5,1,2,3, 4, 5

- Initially all the 3frames are empty
- First3frames are inserted at the end
- When reference 4 comes, it is checked with the existing frames, it is not available and also there is no empty frames in the queue
  - The reference 4 is replaced with the frames that will not be used for long time
    - Here,reference1,2and3 was referenced in the order
    - As per the order,reference1was referenced first. So it is replaced



Number of page fault=10

The page fault occurs for the given reference string is shown below

Reference string:	1	2	3	4	1	2	5	1	2	3	4	5
Misses:	×	×	×	×	×	×	×	√	√	×	×	×

Fault rate=10/12=0.83

Problem

- How to implement LRU replacement
- Determine an order for the frames defined by the time of last use.

Implementation of LRU algorithm

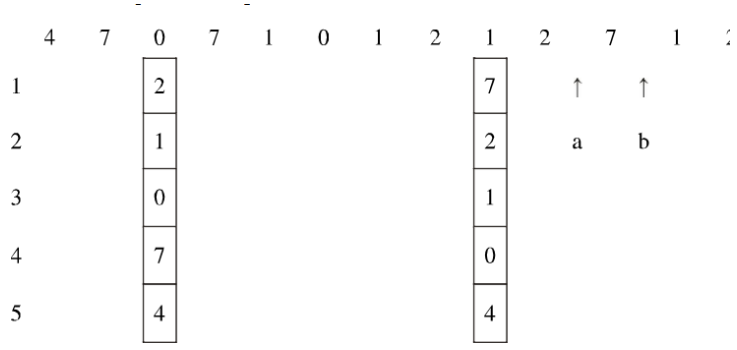
(1) **Counter Implementation**

- Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.

- When a page needs to be changes, look at the counters to determine which page to change (page with smallest time value).

(2) Maintain the time of changes.Stack Implementation

- Keeps as tack of page number in a doubly linked form
- When a Page is referenced
  - Removed from the stack and move it to the top
- Top of the stack–Most recently used page
- Bottom of the stack–Least recently used page.
  - Required 6pointers to be changed
- No search required for replacement

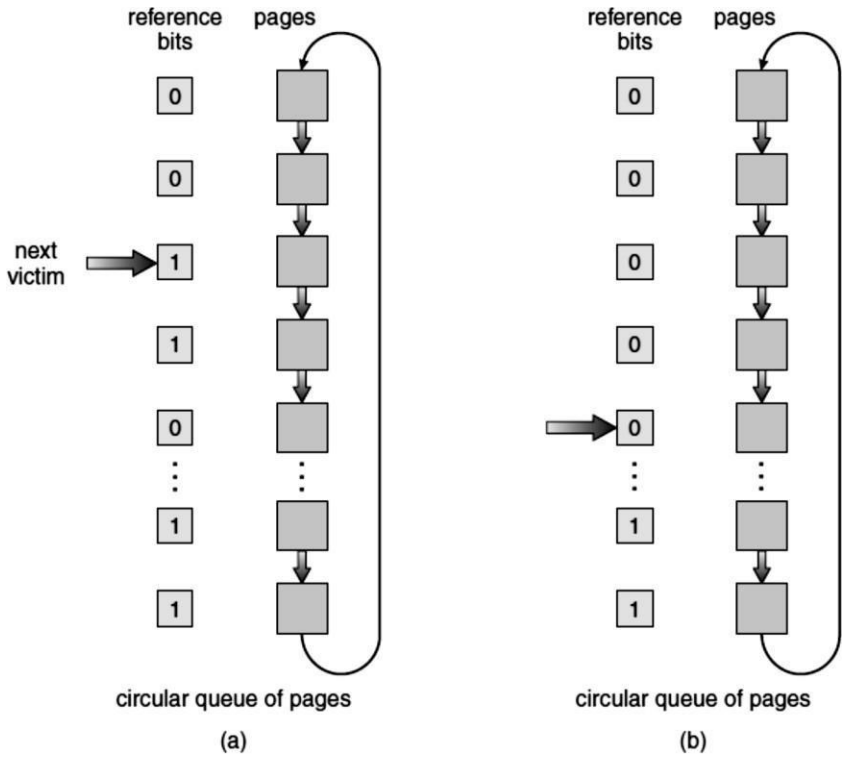


LRU-Approximation Page Replacement

Reference Bit

When the page is referenced, the reference bit for a page is set. Then when a fault occurs, pick a page that hasn‘t been referenced. Reference bits are associated with each entry in the page table.

- Initially all the reference bits=0.
- When page is referenced, bit is set to 1.
- Replace the one which is0 (if one exists).
  - Determine which page has been used and which has not.



Second-Chance Page replacement Algorithm Enhanced

Second-Chance Algorithm

Enhanced Second-Chance Algorithm is an algorithm that need a reference bit and a modify bit as an ordered pair.

The4 possible values for (reference bit, modify bit)is listed below

(1) (0,0) class
- Neither recently used nor modified
- Best page to replace.

(2) (0,1) class

ALLOCATION FRAMES

The fixed amount of free memory among various processes are allocated as follows

Assume that

Number of free frames	: 93
Number of processes	: 5
System type	: Single-user

Consider the single-user system with

Memory Capacity	: 128 KB
Page Size	: 1 KB
Number of frames	: Memory Capacity/Page size
	: 128 KB /1KB
	: 128 frames
OS capacity	: 35 KB
Capacity for user process	: Total Memory Capacity–OS Capacity
	: 128 KB–35KB
	: 93 KB
Number of frames for user process	: User Process Capacity/Page size
	: 93 KB / 1KB
	: 93frames

All 93frames are initially put on the free-frame list in pure demand paging.

Started Execution

Once the process started its execution, a sequenceof93-page faults will occur by getting all free frames from the free-frame list.

During Execution

A page-replacement algorithm is used to select one of the 93 in-memory pages to be replaced with the 94^{th} pages after all the frames in the free-frame list was exhausted.

Architecture

- Maximum number is defined by the amount of available physical memory.

Allocation Algorithms

The Two major allocation schemes are

1) Fixed Allocation
2) Priority allocation

Fixed Allocation

Fixed Allocation scheme is the allocation scheme in which the ratio of frames depends on the size of processes.

Types

1) Equal Allocation
2) Proportional Allocation

(1) Equal Allocation

If we have n processes and m frames, give each process m/n frames.

Example

Number of free frames	93
Number Of processes	5
Number Of free frames for each erprocess	93/5 18

(2) **Proportional Allocation**

Proportional Allocation is an algorithm that allocate according to the size of process.

Let s_i=size of process

piS□ □ $

Let m=total number of available frames

a_i= allocation for p_i

Global Versus Local Allocation

Page replacement algorithm scan be implemented broadly in two ways

1) Global replacement
2) Local replacement

Global replacement

Global replacement allows a process to select a replacement frame from the set of all frames; one process can take a frame from another process.

Under global replacement algorithms, the page-fault rate of a given process depends also on the paging behavior of other processes.

Local replacement

Local replacement requires that each process selects from only its own set of allocated frames. The number of frames allocated to a process does not change.

Under local replacement algorithms, the set of pages in memory for a process is affected by the paging behavior of the processes.

- Less used pages of memory are not made available to a process that may need them.

THRASHING

If a process does not have enough pages, the page-fault rate is very high. This leads to high paging activity called Thrashing.

Thrashing

A process is busy swapping pages in and out. In other words, a process is spending more time paging than executing.

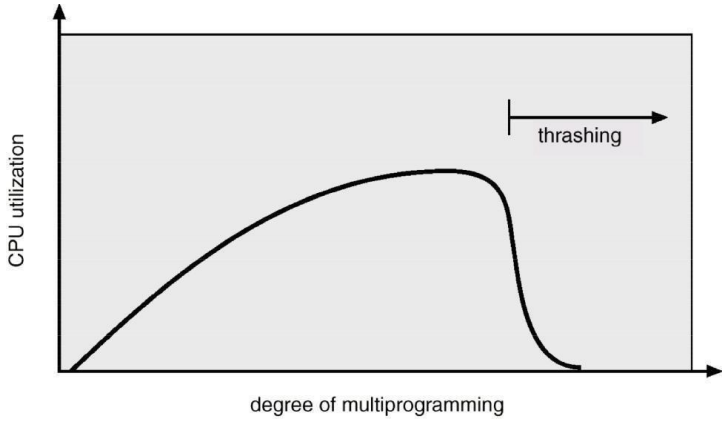
Cause of Thrashing

Consider the followings scenario which is based on the actual behavior of early paging systems.

- Operating System observes low CPU utilization and increases the degree of multi programming. Global page-replacement algorithm is used, it takes away frames belonging to other processes
- But these processes need those pages, they also cause page faults.
- Many processes join the waiting queue for the paging device, CPU utilization further decreases.
- OS introduces new processes, further increasing the paging activity.

The following graph shows the plot of CPU utilization against the degree of multiprogramming

- As the degree of multi programming increases, CPU utilization also increases until a maximum is reached.
- If the degree of multi programming in increased further, thrashing sets in and CPU utilization drops slowly.



Thrashing

Why does paging work?

- Locality Model
  - Computations have locality!
  - Working-set strategy starts by looking a thow many frames a process is actually using.
- A Locality is a set of pages that are actively used together.
- Process migrates from one locality to another.
- Localities may overlap.

UNIT V
STORAGE MANAGEMENT

Mass-Storage Structure: Disk Structure- Disk Scheduling-Disk Management - RAID - Swap-Space Management-I/O Systems Basics –File System Interface: File concept-Access methods-Directory and Disk Structure – File-System Implementation: File-System Structure – File system implementation-Directory implementation-Allocation methods – Free Space management-Case studies: FAT, NTFS File Systems.

Mass-Storage Structure

- Systems that store vast amounts of data are known as mass storage devices. The term "massive storage devices" is sometimes used interchangeably with "peripheral storage," which refers to managing data volumes exceeding a computer or device's built-in storage capacity.
- The fundamental purpose of mass storage is to establish a system for data backup and recovery.
- As computer systems have evolved, so have the definitions and techniques associated with mass storage. Experts trace the earliest forms of mass storage back to the mainframe supercomputer era, where punch cards, Hollerith cards, and other manual storage media were common.
- Today, mass storage encompasses various technologies, including hard disks, solid-state drives, tape drives, and other physical data storage solutions.
- Mass storage media are closely associated with data backup and recovery. Large corporations develop strategies to record, store, and back up extensive amounts of data, requiring far more storage capacity than what standard hardware offers.
- This often involves continuous data storage methods utilizing tape or other media. Additionally, mass storage solutions can support large networks or multiple mobile devices. For instance, a portable tablet with limited internal storage may rely on flash drives or USB storage for backup purposes.

Types of Mass Storage Devices:

- Magnetic Disks
- Solid-State Disks
- Magnetic Tapes

Magnetic Disks

- A magnetic disk is a storage device that uses magnetization to write, rewrite, and access data. It features a magnetically coated surface with tracks, sectors, and spots for data storage.
- IBM introduced the first magnetic hard drive in 1956, featuring 50 large 21-inch (53 cm) platters. Despite its considerable size, it had a storage capacity of only 5 megabytes. Since then, magnetic disks have undergone significant advancements, offered exponentially larger storage capacities while become more compact.

Solid-State Disks (SSDs)

- As technology and economic conditions evolve, older technologies are often adapted for new purposes. One example is the increasing use of solid-state drives (SSDs).
- SSDs function like small, fast hard drives using memory technology. Some models use flash memory or DRAM chips with battery backup to retain data during power cycles.
- Since SSDs have no moving parts, they operate much faster than traditional hard drives and eliminate issues like disk access scheduling. However, they come with some drawbacks—they are more expensive, typically smaller in size, and may have a shorter lifespan than hard drives.
- SSDs are especially useful as high-speed caches for frequently accessed data, such as file system metadata (e.g., directory and inode information). They are also used as boot drives, storing the operating system and key applications but not essential user data. Additionally, SSDs help make laptops thinner, lighter, and faster.
- Because SSDs are much faster than traditional hard drives, the system's bus speed can become a bottleneck. To overcome this, some SSDs are directly connected to the PCI bus for improved performance.

Magnetic Tapes

- Before hard disk drives became common, magnetic tapes were widely used for secondary storage. Today, they are primarily used for data backups.
- Finding specific data on a tape can be slow, but once reading or writing begins, the speed is comparable to disk drives.
- Tape drives typically store between 20 and 200 GB, with compression technology doubling their capacity.

DISK STRUCTURE

In traditional hard drives, the head-sector-cylinder (HSC) numbering system is converted into linear block addresses. The numbering begins at sector 0, which is located on the first head of the outermost track. From there, numbering continues through the remaining sectors on the same track, then through the tracks within the same cylinder, and finally proceeds through all cylinders toward the center of the disk.

Today, linear block addressing (LBA) is used instead of HSC for several reasons:

- The outer tracks of a disk are physically longer than the inner tracks, allowing more sectors to be stored on the outer tracks than on the inner ones.
- Disks naturally have some defective sectors. To compensate, they include spare sectors that can replace faulty ones, a process managed internally by the disk controller.
- Modern hard drives have thousands of cylinders and hundreds of sectors per track, especially on the outermost tracks. These large numbers exceed the limitations of the HSC system in older operating systems. As a result, disks can be configured with any combination of HSC values that fit within the total available sectors.
- Although there is a physical limit to how densely bits can be stored on a disk, advancements in technology continue to increase storage density.
- Sector Allocation in Modern Disks

Modern hard drives allocate more sectors to the outer cylinders than the inner ones, using one of two methods:

Constant Linear Velocity (CLV) – In this approach, the bit density remains consistent across all cylinders. Since outer cylinders contain more sectors, the disk spins slower when reading them. This ensures that the data transfer rate remains stable. CLV is commonly used in CDs and DVDs.

Constant Angular Velocity (CAV) – Here, the disk spins at a fixed speed, meaning that the bit density decreases toward the outer cylinders. This results in a constant number of sectors per track across all cylinders.

DISK SCHEDULING

Disk Scheduling Algorithms is an algorithm used to schedule the servicing of disk I/O requests.

Types

- FCFS Scheduling
- SSTF Scheduling
- SCAN Scheduling
- C-SCAN Scheduling LOOK Scheduling

The disk scheduling algorithms are illustrated with a request queue(0-199)

- Queue=98, 183, 37, 122, 14, 124, 65,67
- Head pointer=53

FCFS Scheduling

FCFS(First-Come First-Served)Scheduling is the simplest form of disk scheduling. It simply servers the first request in the queue.

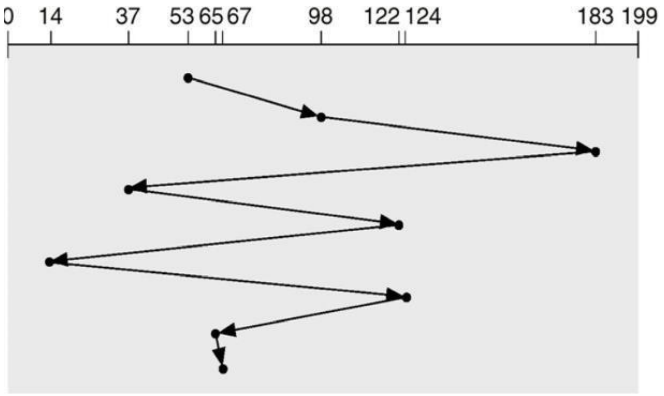
Illustration

Initially the disk head is at cylinder 53.

Head Movement

Current	**Next**	**Total**
53	98	45
98	183	85
183	37	146
37	122	85
122	14	108
14	124	110
124	65	59
65	67	2
	Total	**640**

Total head movement = 640 cylinders.



FCFS Scheduling

Advantage

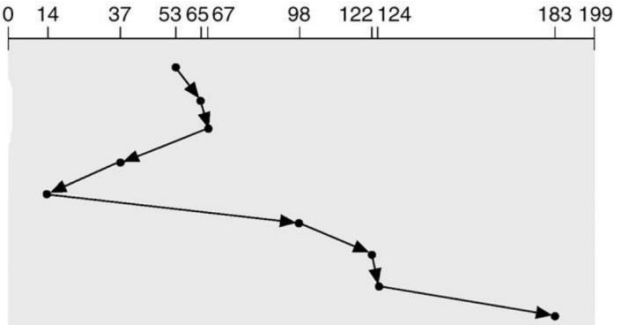
- Simple to write and understand
- Perform operations in order requested
- No reordering of work queue
- No starvation: every request is serviced

Disadvantage

- Large seek time as the disk head may have to jump from one extreme to another.
- Does not provide the fastest service.

SSTF Scheduling

SSTF **(shortest- seek time first algorithm)** scheduling is an algorithm that selects the request with the minimum seek time from the current head position.



SSTF Scheduling

Advantages

- Minimizes seek time
- Maximizes through put

Disadvantages

- Not fair. Starvation is possible.
- More CPU time required to find out the shortest seek time.
- Not always optimal

SCAN Scheduling

In SCAN algorithm, the disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.

The head continuously scans back and forth across the disk. So it is called as the elevator algorithm.

Current	Next	Total
53	37	16
37	14	23
14	65	51
65	67	2
67	98	31
98	122	24
122	124	2
124	183	59
	Total	**208**

Total head movement = 208 cylinders.

Disadvantage

It is un fair a sit results into more wait for the requests at thee nds

LOOK/C-LOOK Scheduling

LOOK / C-LOOK Scheduling is a version of SCAN/C-SCAN in which the arm goes only as far as the final request in each direction. Then it reverses the direction immediately, without going all the way to the end of the disk because they look for a request before continuing to move in a given direction.

Illustration

Initially the disk head is at cylinder 53.Head Movement

Current	**Next**	**Total**
53	65	12
65	67	2
67	98	31
98	122	24
122	124	2
124	183	59
183	14	169
14	37	23
	Total	**322**

Total head movement = 382 cylinders.

Selection of a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal.
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk.
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating system.

DISK MANAGEMENT

The operating system handles several critical tasks related to disk management, including:

- Disk Initialization
- Booting from the Disk
- Bad Block Recovery

Disk Formatting

Low-Level Formatting (Physical Formatting)

- Also known as physical formatting, this process divides the disk into sectors that the disk controller can read and write.
- It fills each sector with a specific data structure, which includes:

  - Header
  - Data
  - Trailer
  - The header and trailer contain essential information such as the sector number and an Error-Correcting Code (ECC).
  - When data is written to a sector, the ECC is updated based on the bytes stored in the data section.
  - During reading, the ECC is recalculated and compared with the stored value to ensure data integrity.

Disk Partitioning

- Before a disk can store files, the operating system must create and manage its own data structures.
- The disk is divided into **partitions**, where each partition consists of groups of cylinders.
  - Each partition is treated as an independent disk.
- **Logical Formatting (File System Creation)**
  - This step installs the file system's initial data structures onto the disk, including:
    - Free space maps and allocated space maps (e.g., FAT or inode tables)
    - An empty root directory

Boot Block

Boot Block is an initial program to run when the systems is powered up or rebooted. This initial boot strap program tends to simple.

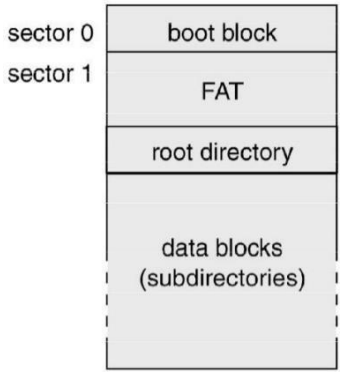
- Boot block initializes system

Boot Strap is stored in Read-only memory (ROM), this location is convenient, because ROM needs no initialization and is at a fixed location that processor canstart executing when powered up or reset.

The full boot strap program is stored in “boot blocks” at a fixed location non the disk.

Boot Disk

A disk that has a boot partition is called boot disk or system disk. The boot ROM instructs the disk controller to read the boot blocks into memory, and then start executing the code to load the entire OS.



MS-DOS Disk Layout

Bad Blocks

Because disks are moving parts and small tolerances they are prone to failures. More frequently, one or more sectors become defective. Most disks come from the factory with bad blocks

SCSI Disk

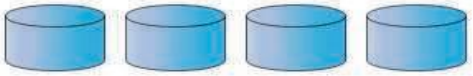
The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level format at the factory and is updated over the life of the disk.

- Low-level formatting spare sectors(OS don't know)
- Sector sparing(or forwarding)
  - Controller replaces each bad sector logically with one of the spare sectors(sectorsparing, or forwarding)
    - Invalidate optimization by OS's disk scheduling
    - Each cylinder has a few spare sectors

- Bad-sector transaction
  - OS tries to read logical block 87
  - Controller calculates EC Cand finds that it is bad.Report to OS
  - Reboot next time, a special command is run to tell the controller to replace the bad sector with a spare.
- When ever the system requests block87,it is translated into the replacement sector's address by the controller
- Sector Slipping
  - As an alternative approach to sector sparing, some controllers can be instructed to replace a bad block by **sector slipping**.

RAID Structure

- As **storage devices** continue to become **smaller and more affordable**, it is now economically viable to connect multiple drives to a computer system. Having **multiple drives** enables faster data read and write speeds by allowing drives to operate in **parallel**.
- Additionally, this setup enhances **data reliability** by storing **redundant information** across multiple drives. As a result, even if one drive fails, data loss can be prevented.
- To optimize both **performance and reliability**, various disk organization methods are used, collectively known as **Redundant Arrays of Independent Disks (RAID)**.
-

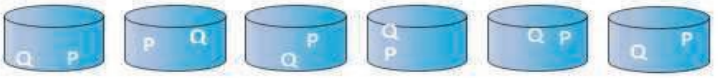


(a) RAID 0: non-redundant striping.

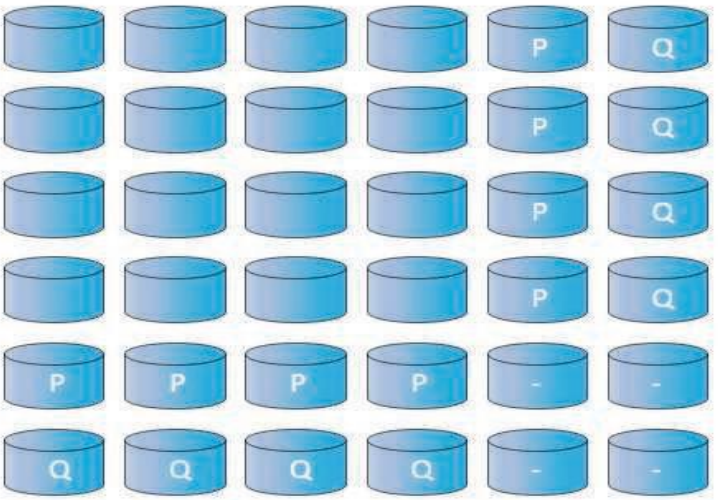
(b) RAID 1: mirrored disks.

(c) RAID 4: block-interleaved parity.

(d) RAID 5: block-interleaved distributed parity.



(e) RAID 6: P + Q redundancy.



(f) Multidimensional RAID 6.

RAID level 0.RAID level 0 refers to drive arrays with striping at the level of blocks but without any redundancy (such as mirroring or parity bits), as shown in Figure (a).

RAID level 1.RAID level 1 refers to drive mirroring. Figure (b) shows a mirrored

RAID level 4.RAID level 4 is also known as memory-style error-correcting code (ECC) organization. ECC is also used in RAID 5 and 6. The idea of ECC can be used directly in storage arrays via striping of blocks across drives. For example, the first data block of a sequence of writes can be stored in drive 1, the second block in drive 2, and so on until the *N*th block is stored in drive *N*; the error-correction calculation result of those blocks is stored on drive *N* + 1. This scheme is shown in Figure (c), where the drive labeled *P* stores the error-correction block. If one of the drives fails, the error-correction code recalculation detects that and prevents the data from being passed to the requesting process, throwing an error.

RAID level 5. RAID level 5, or block-interleaved distributed parity, differs from level 4 in that it spreads data and parity among all N+1 drives, rather than storing data in N drives and parity in one drive. For each set of N blocks, one of the drives stores the parity and the others store data. For example, with an array of five drives, the parity for the nth block is stored in drive (n mod 5) + 1. The nth blocks of the other four drives store actual data for that block. This setup is shown in Figure (d), where the Ps are distributed across all the drives. A parity block cannot store parity for blocks in the same drive, because a drive failure would result in loss of data as well as of parity, and hence the loss would not be recoverable. By spreading the parity across all the drives in the set, RAID 5 avoids potential overuse of a single parity drive, which can occur with RAID 4. RAID 5 is the most common parity RAID.

blocks because they would be identical and would not provide more recovery information. Instead of parity, error-correcting codes such as Galois fiel math are used to calculate Q. In the scheme shown in Figure (e), 2 blocks of redundant data are stored for every 4 blocks of data—compared with 1 parity block in level 5—and the system can tolerate two drive failures.

Multidimensional RAID level 6. Some sophisticated storage arrays amplify RAID level 6. Consider an array containing hundreds of drives. Putting those drives in a RAID level 6 stripe would result in many data drives and only two logical parity drives. Multidimensional RAID level 6 logically arranges drives into rows and columns (two or more dimensional arrays) and implements RAID level 6 both horizontally along the rows and vertically down the columns. The system can recover from any failure —or, indeed, multiple failures—by using parity blocks in any of these locations. This RAID level is shown in Figure(f). For simplicity, the figure shows the RAID parity on dedicated drives, but in reality the RAID blocks are scattered throughout *and columns.*

Swap-Space Management

Swapping occurs when **physical memory becomes critically low**, prompting the system to move processes from memory to swap space to free up RAM.

However, most modern operating systems no longer use this traditional approach. Instead, they integrate **swapping with virtual memory techniques** and swap **individual pages** rather than entire processes. In fact, some systems now use the terms **"swapping" and "paging" interchangeably**, highlighting the convergence of these two concepts.

Managing swap space is a key **low-level function** of the operating system. Virtual memory treats secondary storage as an **extension of main memory**. However, because disk access is much slower than RAM, relying on swap space can **significantly reduce system performance**.

The primary goal of swap-space design and management is to **optimize throughput** for the virtual memory system. This section examines:

Swap-Space Use

- The amount of swap space an operating system needs differs significantly based on its usage patterns. Some systems demand a swap space equivalent to the physical RAM, others require a multiple, and some size it based on the difference between the total virtual memory and the physical RAM. It's also possible for systems to operate with very little or no swap space at all.
- To improve the performance of the virtual memory system, certain operating systems can utilize multiple swap spaces located on separate disk drives.

Swap space can be physically situated in one of two ways:

- As a large file within the standard file system: This method offers ease of implementation but can be inefficient. Accessing swap data requires navigating the file system's directory structure, and the file can become fragmented. While caching block locations can help in finding the physical blocks, it doesn't entirely resolve the performance limitations.

Example

Historically, operating systems swapped entire processes in and out of memory as needed. Modern systems, however, typically employ page-level swapping, moving individual memory pages to swap only when required. For example, if a block of process code hasn't been modified since it was initially loaded, it's often more efficient to simply discard it from virtual memory and reload it from the file system if needed, rather than writing it to the swap space and then reading it back.

Consider the swap management approach used in Linux. It maintains an in-memory map where each entry corresponds to a 4KB block within the swap space. A zero in this map signifies a free block, while a non-zero value indicates the number of processes currently referencing that specific block. A value greater than one points to a memory page being shared among multiple processes.

This in-memory map is a key data structure for managing swap space in Linux systems.

I/O Systems basics

- Operating system designers dedicate significant attention to managing the diverse array of devices connected to a computer.
- Given the vast differences in functionality and speed among I/O devices – think of a mouse versus a hard disk, a USB drive, or a tape robot – a range of control methods is essential.

- These methods constitute the kernel's I/O subsystem, which acts as a crucial layer, abstracting the complexities of device management from the core of the operating system.

Memory-Mapped I/O

How does the CPU tell a device controller what to do for an I/O operation? Essentially, controllers have special storage locations called registers for data and control signals. The CPU communicates with these registers by reading and writing specific patterns of bits.

There are two main ways this communication happens:

- **Special I/O Instructions:** The CPU uses dedicated instructions to send or receive data to specific I/O port addresses, triggering signals on the system bus to select the correct device and move data in or out of its registers.

- **Memory-Mapped I/O:** In this approach, the controller's registers are mapped into the computer's main memory address space. The CPU then performs I/O by using regular data transfer instructions (like load and store) to read and write to these memory locations that are actually the device's control registers.**Data-in:** The CPU reads this register to receive data from the device.
- **Data-out:** The CPU writes to this register to send data to the device.
- **Status:** This register contains bits that the CPU can read to understand the device's current state, such as whether an operation is complete, if new data is available, or if an error has occurred.
- **Control:** The CPU writes to this register to initiate commands or change the device's operating mode (e.g., setting communication speed or enabling error checking).

Data registers are usually small, holding a few bytes. Some controllers use FIFO (First-In, First-Out) buffers to temporarily store more data, allowing for short bursts of input or output.

Polling

The basic communication between the CPU and a controller involves a simple "handshake." Let's imagine two status bits are used: a "busy" bit (set by the controller

1. The CPU sets a "write" command bit and puts the data into the "data-out" register.
2. The CPU sets the "command-ready" bit.
3. The controller sees the "command-ready" bit is set, so it sets its "busy" bit.
4. The controller reads the command and the data from the "data-out" register and performs the I/O operation.
5. The controller clears the "command-ready" bit, clears any error bits, and then clears its "busy" bit, indicating it's finished.

   This process repeats for each piece of data.

   In step 1, the CPU is "polling" or "busy-waiting" – it's stuck in a loop constantly checking the status register. If the device is fast, this is okay. But if the wait is long, the CPU wastes time that could be used for other tasks. Also, for some devices like serial ports or keyboards where data arrives continuously, the controller has a small buffer. If the CPU doesn't check quickly enough, this buffer can overflow, and data will be lost.

   The basic polling operation itself is quite fast, often taking only a few CPU cycles. However, it becomes inefficient when the CPU spends a lot of time repeatedly checking devices that aren't ready, while other important work remains undone. In such cases, it's better if the hardware controller can directly notify the CPU when it's ready for service. This notification mechanism is called an **interrupt**.

   Interrupts

   Instead of constantly asking devices if they need attention (polling), operating systems use **interrupts**. Think of it like a device tapping the CPU on the shoulder when it needs service.

1. **Device Needs Attention:** When an I/O device finishes a task or has data ready, it sends a signal along a special wire called the **interrupt-request line** to the CPU.
2. **CPU Notices:** After finishing each instruction, the CPU checks this interrupt-request line.
3. **State Save:** If an interrupt signal is detected, the CPU pauses its current task and saves its current state (like where it was in the program).
4. **Jump to Handler:** The CPU then jumps to a specific memory address where a special program called an **interrupt handler** routine is located. This address is fixed so the CPU knows where to go.
5. **Identify and Service:** The interrupt handler figures out which device sent the interrupt and performs the necessary actions to deal with it (e.g., reading the data from the device).

6. **Clear Interrupt:** Once the device is serviced, the interrupt handler signals that the interrupt has been dealt with.
7. **State Restore:** The CPU restores its previously saved state.
8. **Resume Task:** The CPU returns to where it left off before the interrupt occurred and continues executing the original program.

Modern operating systems handle a huge number of interrupts every second, even on simple desktop computers. This shows how crucial interrupts are for managing system activities.

Modern systems need more advanced interrupt handling:

1. **Deferring Interrupts:** The OS needs to be able to temporarily ignore interrupts during critical operations that can't be interrupted.
2. **Efficient Dispatching:** The OS needs a quick way to find the correct interrupt handler for a specific device without checking every device.
3. **Interrupt Priorities:** The OS needs to differentiate between important and less important interrupts, handling urgent ones first.
4. **Traps:** The OS needs a way for programs to directly request services from the kernel (like accessing files), which is done using special instructions called "traps."

Modern CPUs and special hardware called **interrupt controllers** provide these advanced features.

Most CPUs have two interrupt lines:

- **Non maskable Interrupt (NMI):** Used for critical errors like unrecoverable memory issues. It can't be ignored.
- **Maskable Interrupt:** Used by device controllers. The CPU can temporarily ignore these.

The interrupt mechanism uses an **interrupt vector**, which is like a table containing the memory addresses of different interrupt handlers. When an interrupt occurs, a number associated with that interrupt is used as an index into this table to find the correct handler. This makes finding the right handler faster.

Sometimes, there are more devices than entries in the interrupt vector. To solve this, **interrupt chaining** is used. Each entry in the vector points to the beginning of a list of interrupt handlers. When an interrupt happens, the handlers in that list are called one by one until the correct one is found.

Interrupts also have **priority levels**. This allows the CPU to handle important interrupts immediately, even if it's currently dealing with a less important one.

Operating systems use interrupts for many things:

- **Device Communication:** Notifying the OS when a device is ready for data or has finished a task.
- **Exceptions:** Handling errors like division by zero or accessing invalid memory.
- **System Calls:** Allowing programs to request services from the OS kernel.
- **Virtual Memory:** Managing the loading of data from disk into memory (page faults).
- **Kernel Management:** Controlling the flow of tasks within the OS itself.

Direct Memory Access (DMA)

Imagine moving a lot of boxes from one room to another. It would be inefficient to have a supervisor (the main CPU) constantly watching and telling you (the device) where to pick up and drop off each box, one by one. This slow, step-by-step process is like programmed

things up, computers use a special helper called a DMA controller. Think of the DMA controller as a dedicated moving crew.

How DMA Works:

1. Start the Transfer Request
    - The device driver is told to move data from drive2 into a buffer in memory located at address "x".

2. Informing the Drive Controller
    - The device driver sends instructions to the drive controller to transfer "c" bytes (amount of data) to memory at address "x".
3. Begin DMA Transfer
    - The drive controller starts a DMA transfer. This allows data to be moved directly to memory without the CPU handling every byte.

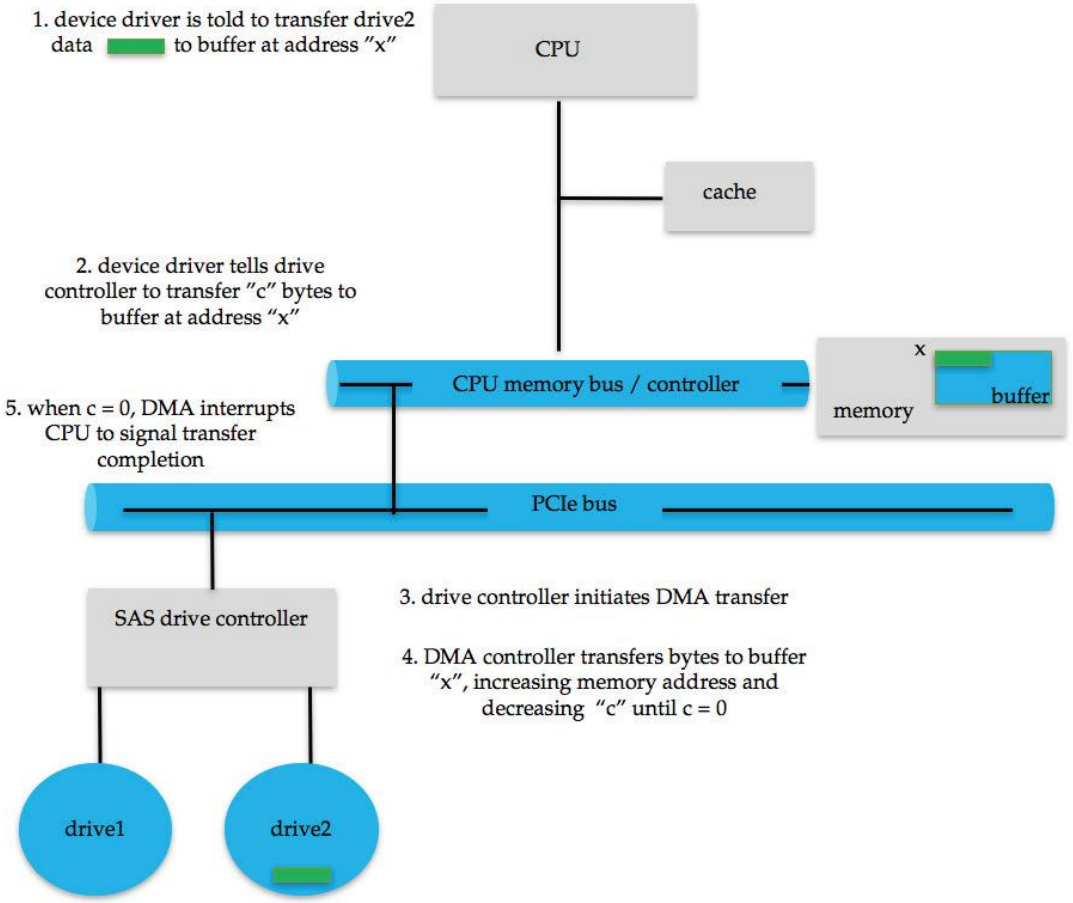


Figure: Steps in a DMA Transfer

4. Data Transfer in Progress
    - The DMA controller transfers the data from drive2 into memory at address "x".
    - As it transfers, it:
        - Increases the memory address (moves forward in the buffer)
        - Decreases the count "c" (tracks how much data is left)
5. Transfer Complete
    - When all bytes are moved (c = 0), the DMA controller sends an interrupt to the CPU to let it know the transfer is done.

File System Interface: File concept

A **file system interface** is a crucial component of an operating system that provides an abstraction for storing, organizing, and retrieving data on storage devices. At the core of this interface is the **file concept**, which refers to how data is logically stored and accessed. A file is a collection of related information that resides on secondary storage, such as a hard disk, SSD, or external device, and serves as the basic unit of storage managed by the file system. Files have specific attributes, including a name, type, size, location, and permissions, which help in identification and access control. They can contain various types of data, such as text, images, audio, or executable programs. The file system interface allows users and applications to interact with files using operations like creation, deletion, reading, writing, and modification. To efficiently organize files, the file system employs structures like directories and folders, enabling hierarchical storage. Additionally, file access methods, such as sequential and direct access, determine how data is read or modified. Security mechanisms, including access control lists (ACLs) and encryption, ensure that only authorized users can access or modify specific files. Overall, the file concept within a file system interface plays a vital role in managing data efficiently while ensuring accessibility, security, and organization.

A file in a file system is a structured collection of data stored on a storage device, such as a hard drive, SSD, or flash drive. It serves as the basic unit of storage, allowing users and applications to read, write, and manage information efficiently. Each file has a unique name, metadata (such as size, creation date, and permissions), and a specific format that dictates how its contents are interpreted. Files are organized within directories (or folders) to facilitate easy access and management. The file system provides mechanisms for file operations, including creation, modification, deletion, and access control, ensuring data integrity and security.

Access methods

Access methods in a file system define how data within a file can be retrieved and manipulated. The primary access methods include sequential access, where data is read or written in order from the beginning to the end, and direct (or random) access, which allows retrieval of data from any position within the file without reading previous content. Indexed access enhances direct access by using an index to locate specific data quickly, while hashed access organizes data using a hash function for efficient searching. These methods optimize file operations based on the nature of data and application requirements, ensuring performance, efficiency, and ease of use.

Directory and Disk Structure

The directory and disk structure in a file system play a crucial role in organizing, managing, and accessing files efficiently. A directory structure serves as a hierarchical framework that stores file names and metadata, allowing users to navigate and manage data easily. It can be single-level, where all files are stored in a single directory, or multi-level, which includes subdirectories to organize files systematically. More advanced structures, such as tree-structured, acyclic graph, and general graph directories, provide flexibility by enabling shared files and complex relationships. On the other hand, the disk structure determines how data is physically stored and retrieved on a storage device. It consists of sectors, tracks, and blocks that store file data and metadata, managed by the file system. Allocation methods, such as contiguous, linked, and indexed allocation, define how files are stored efficiently on the disk to minimize fragmentation and improve access speed. Additionally, disk management techniques, including partitioning, disk scheduling, and free space management, optimize storage utilization and system performance. Together, the directory and disk structure ensure organized data storage, fast retrieval, and efficient file management in an operating system.

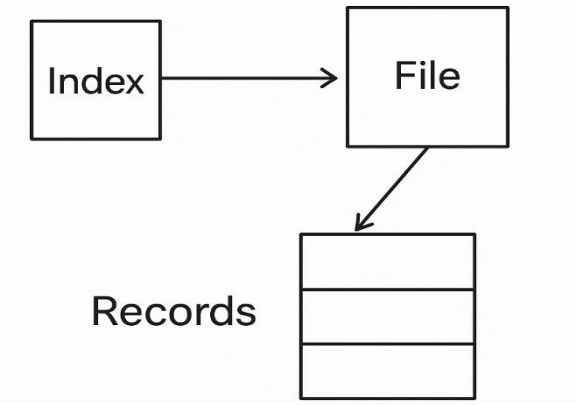
File Access Methods in Operating System

In an operating system, **file access methods** are the techniques used to read from and write to files stored on secondary storage devices. These methods are essential for the efficient management and retrieval of data. The simplest and most commonly used method is **sequential access**, where data is processed in a linear fashion, starting from the beginning of the file and continuing to the end. Each read or write operation moves the file pointer forward, and if access to an earlier part of the file is required, the pointer must be reset. This method is straightforward and efficient for tasks where data naturally flows in order, such as reading logs, streaming media, or handling text files.

For applications requiring more flexibility, **direct access**—also known as random access—is used. In this method, the system can jump directly to any part of the file without reading through the previous data. This is accomplished by specifying the exact location or block number to be accessed. Direct access is particularly useful for large files or databases where users may need to retrieve specific pieces of information without scanning the entire file. It allows faster data retrieval and updating, making it suitable for scenarios like video editing or database management.

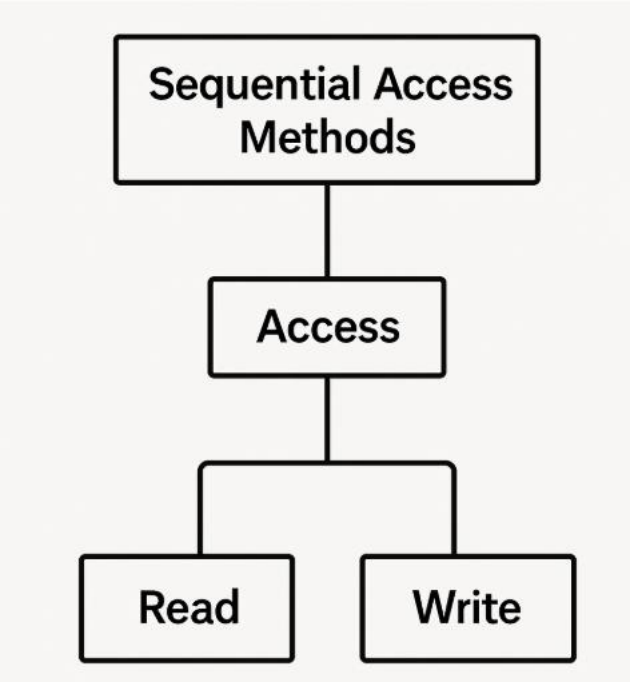
Indexed access provides an even more advanced method of accessing files. It involves creating a separate index file that contains pointers to various blocks or records in the main data file. The operating system uses the index to quickly locate and retrieve the required data. This is similar to the index of a book, where instead of reading every page, you can directly jump to the page number listed in the index. Indexed access is efficient for search-heavy operations and is commonly used in file systems, library databases, and search engines.

Another efficient method is **hashed access**, where a hash function is applied to a key (such as a record identifier) to determine the location of the data block in the file. This method provides constant-time access in the average case and is extremely fast for operations that involve frequent lookups, such as in caching, symbol tables, or password verification systems. However, hash collisions can occur, where two keys hash to the same location, so proper collision handling techniques like chaining or open addressing are needed.



Disadvantages of Sequential Access Method

The sequential access method, despite its simplicity, has several disadvantages that make it less suitable for certain types of applications. One of the primary drawbacks is the lack of direct access to specific data. In this method, to reach a particular block or record, the system must read through all the preceding data in the sequence, which results in slower access times, especially in large files. This makes sequential access highly inefficient for operations that require random or frequent access to different parts of a file, such as searching, updating, or deleting specific records.



Direct Access Method

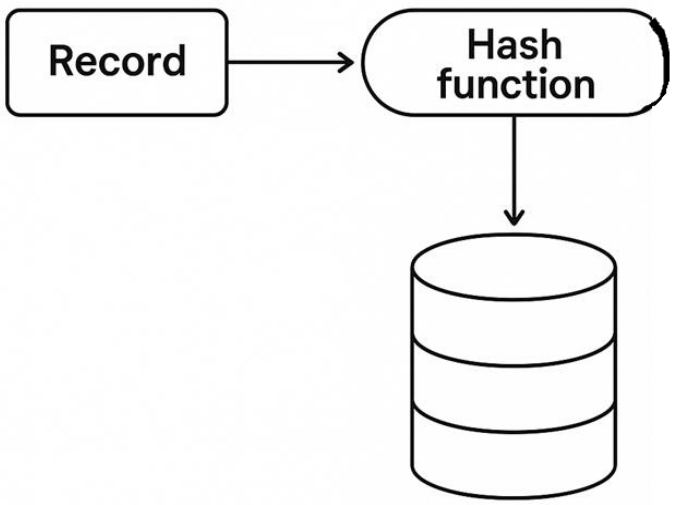
The **Direct Access Method**, also known as **Random Access**, is a file access technique used in operating systems that allows data to be read from or written to a file at **any position**, without the need to read through previous data sequentially. Unlike sequential access, which requires going through data in order, direct access provides the flexibility to jump directly to a desired block or record by specifying its location. Each piece of data in the file is associated with a unique address or block number, enabling the operating system to compute the exact position and access it instantly.

This method is especially useful in situations where speed and flexibility are crucial, such as in databases, multimedia systems, or applications that frequently read and write specific parts of large files. For instance, if a user wants to update or retrieve the 500th record in a file, direct access allows the system to go straight to that record, skipping the previous 499. This makes it highly efficient for search and update operations.

Direct access significantly improves performance in environments that require **frequent, non-linear** access to data. It is commonly used in devices such as **hard disks and solid-state drives**, where accessing data at any position is physically possible and fast. However,

implementing this method requires more complex file management, such as maintaining block addresses and handling file fragmentation. Despite these challenges, the direct access method remains an essential part of modern file systems, offering a balance of speed, flexibility, and control over how data is stored and retrieved.

The Direct Access Method, also known as random access, is a file access technique that enables immediate retrieval or updating of data without the need to read through preceding records. This method significantly reduces average access time, as any block of data can be accessed directly by specifying its address. Unlike sequential access, where each block must be accessed in order, direct access allows users to jump straight to the desired block, making it ideal for large databases and applications that require frequent and rapid access to specific records. Among its key advantages is the ability to instantly access files, which enhances performance and efficiency in data handling. However, the direct access method comes with its own challenges. Its implementation is more complex, as it requires sophisticated algorithms and data structures to effectively manage and locate records. Additionally, it often incurs a higher storage overhead due to the need for maintaining auxiliary data like address tables or pointers.



A file system structure refers to how data is stored and organized on a storage device, such as a hard drive or SSD. It consists of directories, files, and subdirectories arranged hierarchically. At the top of this structure is the root directory, from which all other files and directories branch out. Directories are containers that hold files and other directories, helping to logically organize data. Files themselves contain user data and are stored within these directories, each

having a name and potentially an extension (like .txt, .jpg, etc.). File systems can include various file types, such as regular files, directories, special files (e.g., device files), and links (symbolic or hard). The metadata associated with files, such as their size, permissions, and timestamps, is essential for managing and retrieving the data. Different operating systems use various file system types, such as NTFS for Windows, ext4 for Linux, and APFS for macOS. File systems also include crucial operations, such as mounting a file system to a directory and managing file permissions. Ultimately, the file system structure ensures efficient storage, organization, and access to data across devices.

File System Implementation

File system implementation involves the following components:

- On-disk structures
- In-memory structures

On-Disk Structure Overview

The on-disk portion of a file system typically includes data such as:

- Instructions for booting the operating system stored on the disk
- The total count of blocks available
- Details on the number and location of free blocks
- The layout of the directory structure
- Information about individual files

On-Disk Structure

The on-disk structure consists of the following components:

- **Boot Control Block**
  Stores the information required to boot the operating system from the volume.
  It is usually located in the first block of the volume.
- **Volume Control Block**
  Holds key information about the volume, such as the total number of blocks, block size, the number and location of free blocks, the count of free File Control Blocks (FCBs), and pointers to those FCBs.
- **Directory Structure**
  Responsible for organizing and managing the arrangement of files within the system.
- **File Control Block (FCB)**
  Contains metadata about files, including file permissions, ownership, file size, and the locations of data blocks.

Additional file-related information may include:

- Data blocks that store the file content

- File permissions
- Timestamps (creation, last accessed, last modified)
- Ownership details (user, group, and Access Control List)
- File size

Typical File Control Block (FCB)

In-Memory Structures

Information stored in memory supports both file system management and enhances performance through caching.

The in-memory structure may consist of the following components:

- **In-Memory Mount Table**
  Contains details about all currently mounted volumes.
- **In-Memory Directory Structure**
  Stores directory data for directories that have been accessed recently.
- **System-Wide Open File Table**
  Includes copies of the FCBs for all files that are currently open, along with other related information.
- **Per-Process Open File Table**
  Maintains pointers to the corresponding entries in the system-wide open file table and stores additional data specific to each process.

File Creation Process

When a new file is created:

- An application invokes the logical file system.

The logical file system understands the format of directory structures and allocates a new FCB.

- The relevant directory is loaded into memory.
- The directory is updated to include the new file name and its associated FCB.
- The updated directory information is written back to the disk.

File Input / Output Operations (Opening & Reading)

After a file is created, it can be used for input/output operations.

Before that, it must be opened:

- The open() function is called with the file name as an argument.
- The file system searches the directory structure to locate the file.

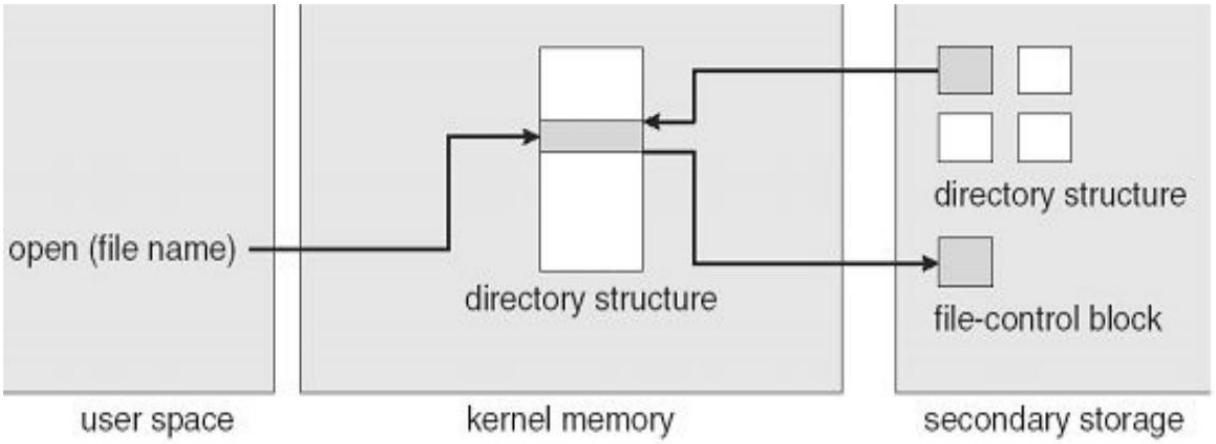


Fig: File Open Process Overview

The diagram illustrates the steps taken when a file is opened in a system:

1. **User Space:**
   - The open(file name) request is initiated by a user application.
2. **Kernel Memory:**
   - The kernel receives the file name and searches through the directory structure in memory.
   - This structure helps locate the file by mapping the name to its corresponding File Control Block (FCB).
3. **Secondary Storage:**
   - If the necessary directory or FCB data isn't already in memory, it is retrieved from secondary storage.
   - The directory structure and file control block stored on disk are accessed to complete the operation.

File System Structure & File System implementation

Once a file is located, its File Control Block (FCB) is duplicated into the system-wide open file table stored in memory.

- An entry is then added to the per-process open file table, which references the corresponding entry in the system-wide table.
- Additional information in these entries may include:
  A pointer indicating the current position within the file.

The access mode (such as read or write) used when opening the file.

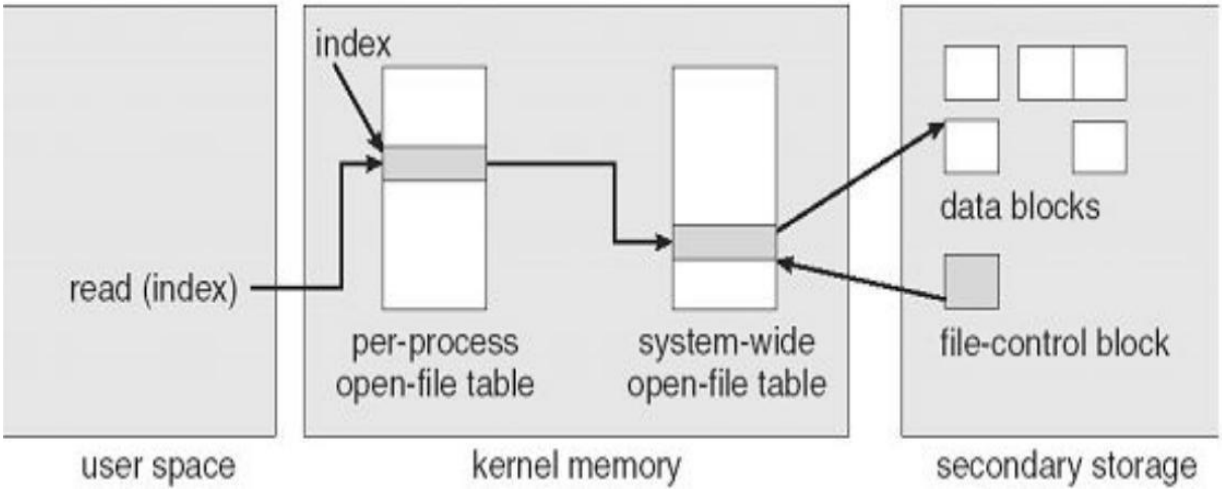


Fig: Memory file system structures

In-Memory File System Structures for File Reading

When a file is opened, the open() system call returns a reference to the relevant entry in the per-process file table. This reference is known as:

- File Descriptor in UNIX
- File Handle in Windows 2000

Partitions and Mounting

A disk can be divided into multiple partitions, or a single partition may span across several disks. Each partition may be classified as either:

- Raw – contains no file system
- Cooked – includes a file system

Directory Implementation

The directory can be implemented in two ways

1) Linear List
2) Hash Table

Linear List

Linear list of file names with pointer to the data blocks is the simplest method of implementing a directory which requires linear search to find a particular entry.

- Simple to program.
- Time-consuming to execute.
- File Creation
  - Search the directory to be sure that no existing file has the same name
  - Add to the end of the directory.
- File Deletion
  - Search the directory for the named file
  - Release the space allocated to it.
- File Reuse
  - Mark the entry as unused by assigning a special name or a blank name
  - Attach to the list of free directory entries.
  - Copy the last entry in the directory into the freed location and to decrease the length of the directory.

Advantages

- A linked list is used to decrease the time required to delete a file.

Disadvantages

- Finding the files in linear search
- Searching is slow so some operating system uses software cache to store more recently used directory information.

Hash Table

A hash table is another data structure which is used for file directory. Hash Table is a linear list with hash data structure.

The hash table takes a value computed from the file name and returns a pointer to the filename in the linear list.

Collisions

A collision is a situation where two file names hash to the same location. Alternatively, achained-overflow hash table can be used to avoid collisions.

Allocation Methods

Allocation methods is a method used to allocate space to the files for

- Utilizing the disk space effectively
- Accessing the files quickly

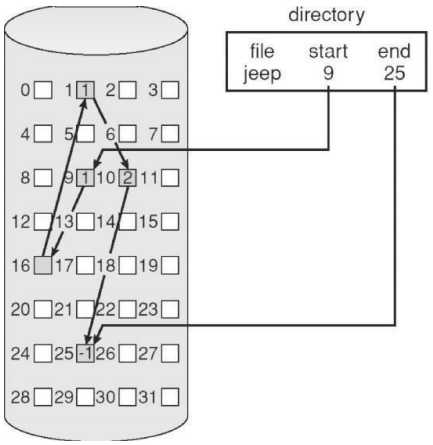
Methods

The three major methods for allocating disk space are

1) Contiguous allocation
2) Linked allocation
3) Indexed allocation

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk.
- File location is defined by the disk address of the first block and its length
  - Example:
    - If the file is n blocks long and starts at location b, then it occupies blocks b,b+1, b+2….b+n-1
- Both sequential access and direct access are supported by the contiguous allocation



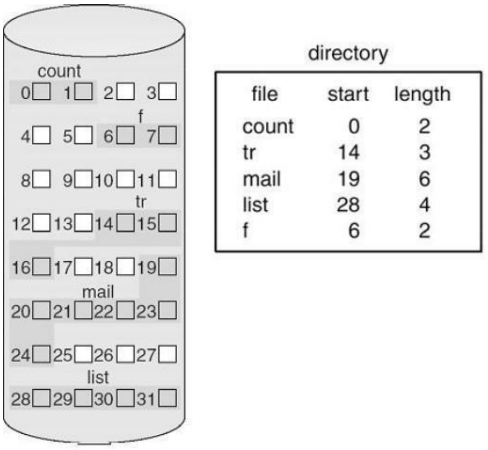
Contiguous allocation of disk space

Disadvantage

- Difficult to find free space for a new file
- Suffer from external fragmentation

Linked Allocation

- Each file is a linked list of disk blocks, the disk blocks may be scattered on the disk.
- Directory contains
  - A pointer to the first and (optionally the last) block of the file
  - Pointer is initialized to null to signify an empty file
- Each block contains
  - a pointer to the next block and the last block contains a NIL pointer



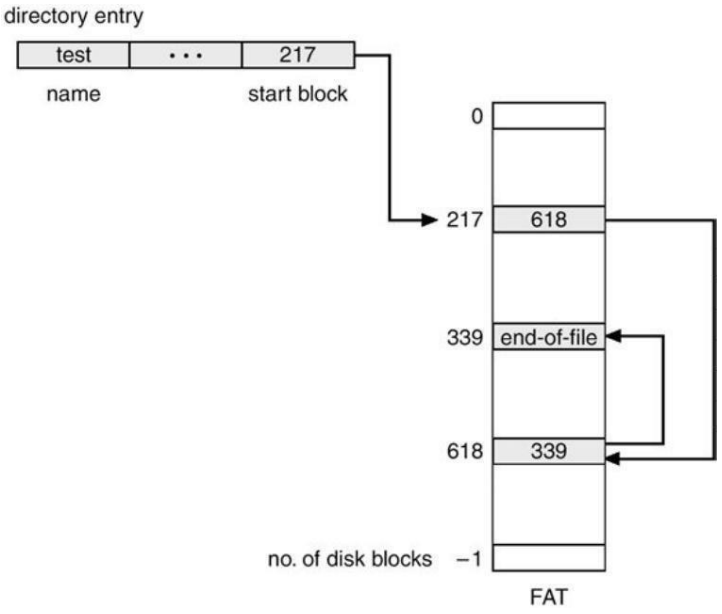
Linked allocation of Disk space

File Operations

- To create a new file,
  - Create a new entry in the directory in the directory
- To write to a file,
  - Removes the first free block and writes to that block
  - New block is then linked to the end of the file
- To read a file,
  - the pointers are just followed from block to block

File allocation table (FAT)

- An important variation on linked allocation
- Simple but efficient method of disk-space allocation
  - Used by the MS-DOS operating system.
- FAT table has one entry for each disk block and is indexed by block number
  - The directory entry
    - contains the block number of the first block of the file
  - The table entry
    - indexed by that block number
    - contains the block number of the next block in the file
  - This chain continues until it reaches the last block



File Allocation Table

Advantages

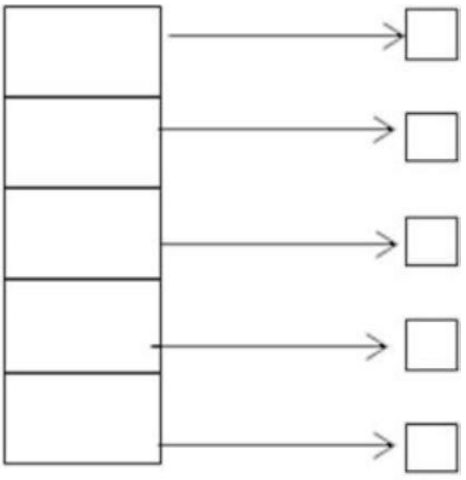
- No external fragmentation
- No need to declare the size of a file when that file is created
- File can continue to grow as long as there are free blocks

Disadvantages

- Inefficient to support direct-access
  - Effective only for sequential-access files
- Space required for the pointers

Indexed Allocation

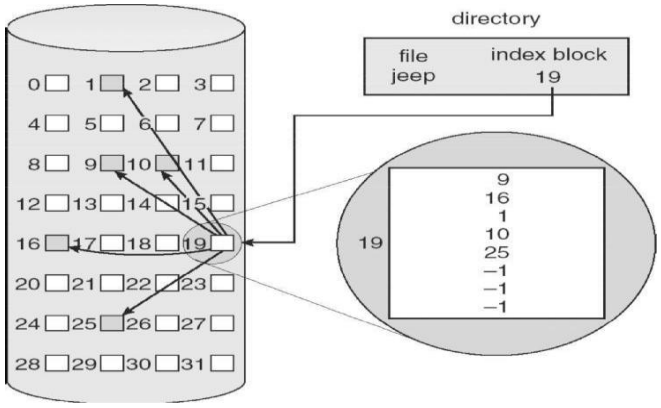
- Solution to the problem of both contiguous and linked allocation
  - By bringing all the pointers together into one location called the index block.
- Each file has its own index block which is an array of disk block addresses. The i^{th} entry in the index block points to the i^{th} block of the file.



Logical View

- The directory contains the address of the index block.
- When the file is created
  - all pointers in the index block are set to NULL
- When the i^{th} block is written

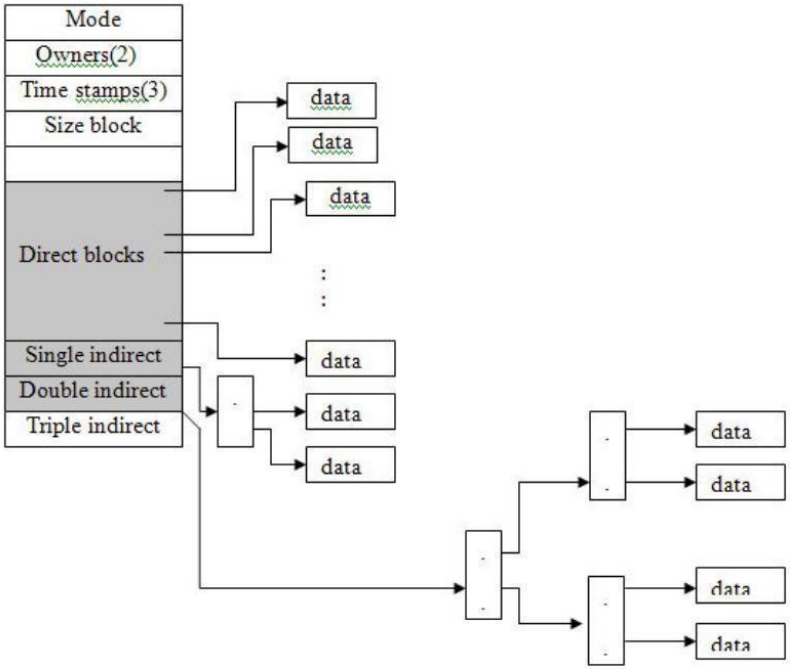
- a block from the free space manager and its address is put in the i^{th} index block entry



The Mechanisms to deal with the index block is too small are

1) **Linked Scheme**
   - An index block is normally one disk block
   - To allow large files, several index blocks are linked together Next address is nil or a pointer to another index block
2) **Multi-Level index**
   - A variant of the linked representation uses a first level index block to point to a set of second-level index blocks
   - This approach could be continued to a third or fourth level, depending on the desired maximum file size
3) **Combined Scheme**
   - Another alternative, used in the UFS is to keep the first say 15 pointers of the index block in the file's I node
   - The first 12 of these pointers point to direct blocks that are they contain addresses of blocks that contain data of the file.

- The next three pointers are an indirect block.
  - The first points to a single indirect block which is an index block containing not data but the addresses of blocks that do contain data.
  - The second points to a double indirect block which contain the address of a block that contains the address of blocks that contain pointers to the actual data blocks.



The Unix I node

- The last pointer contains the addresses of a triple indirect block

FREE SPACE MANAGEMENT

- Free Space Management is the mechanism of maintaining the free-space list to keep track of free disk space.
- File Creation
  - Search the free space list for the required amount of space
  - Space is removed from the free space list and allocate it to the new file
- File Deletion
  - Disk space is added to the free space list

Types of mechanism

The Free space list can be maintained in the following waysBit Vector

- Linked List
- Grouping
- Counting

Bit Vector

- The free space on a disk can be tracked using a bit map or bit vector.
- Each disk block is represented by a single bit:
  - 1 indicates a free block
  - 0 indicates an allocated block
- A drawback of this method is the difficulty in efficiently locating the first available block or a series of free blocks.
- However, many systems offer bit-level operations that can help streamline this search.
- To compute the block number:
  block number = (number of bits per word × number of words with 0) + position of first 1 bit
- This approach is only practical when the entire bit vector can be kept in main memory. It works well with small disks but becomes inefficient for larger storage systems.

Linked List

- Another method for managing free space is to use a linked list of free disk blocks.
- A pointer to the first available block is kept in a specific disk location and cached in memory.
- Each free block contains a pointer to the next available block, forming a chain.
- For example, block 2 might point to block 3, which in turn points to block 4, and so on.

Grouping

- A variation of the linked list method is grouping.
- Instead of storing one pointer per block, the first free block stores the addresses of n free blocks.
- The first n-1 blocks are free, and the nth block holds the addresses of another group of n free blocks.
- This approach speeds up the process of locating multiple free blocks at once.

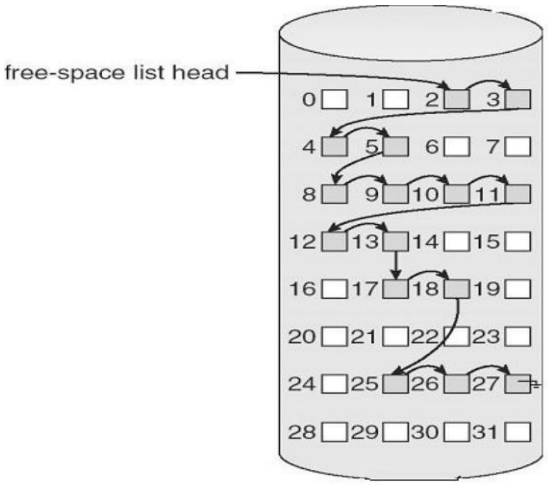
Counting

- This method takes advantage of the fact that multiple contiguous blocks are often allocated or freed together.
- Instead of storing every free block address, the system keeps:
  - The starting address of a free block range
  - The number of contiguous free blocks (n) that follow
- This simplifies management, especially when using contiguous allocation or clustering.

Free Space Management Using Linked List on Disk

This method manages free disk space by linking all unused blocks together in a sequence. Each free block contains a pointer to the next available block, forming a chain of free blocks directly on the disk. A reference to the first free block is stored in a predefined disk location and often cached in memory for quick access.

Each item in the free space list contains a disk block address along with a count of



consecutive free blocks.

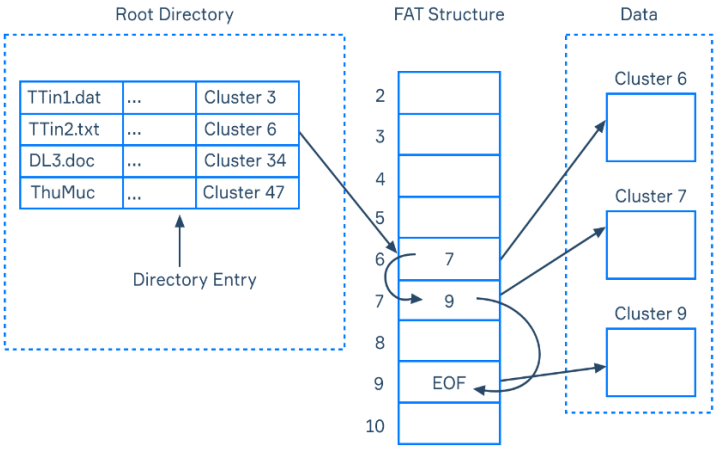
CASE STUDIES: FAT, NTFS FILE SYSTEMS

File systems play a crucial role in managing data storage, organization, and access on various storage devices, like HDD, SSD, and flash drives. They provide a structure that allows the operating system and applications to interact with the storage content in an efficient manner. Different file systems offer various features, performance levels, and compatibility, catering to specific needs and preferences. Three commonly used file systems are FAT32, NTFS, and EXT4.

FAT32

FAT32, one of the oldest file systems, offers wide compatibility across various operating systems and devices. In FAT32, the system divides and stores files between data clusters, which you access when interacting with a file.

A **File Allocation Table (FAT)** stores information about data clusters on the disk and indicates whether a specific cluster is allocated for a file or directory



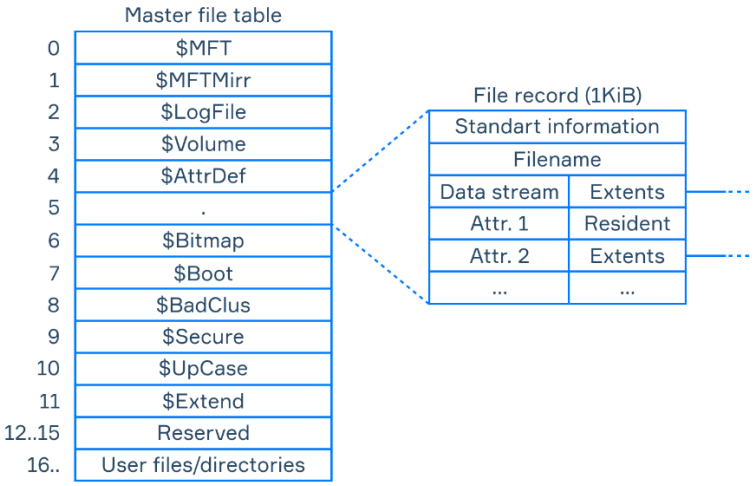
In FAT32, files are divided into multiple data clusters. For each file, FAT32 creates directory entries that contain essential information, such as the first cluster of the file. Then, FAT32 accesses each data cluster in sequence using the information about the next cluster stored in the current one. This process continues until the End of File (EOF) is reached. For instance, let's consider a picture file stored in clusters 6, 7, and 9. FAT32 begins with cluster 6 and obtains the information that the next cluster is 7. It then proceeds to the next cluster based on this information until it ultimately reaches the EOF at cluster 9.

NTFS

The New Technology File System (NTFS) is a modern file system developed by Microsoft as the default file system for its Windows operating systems. NTFS is designed to provide advanced features, enhanced security, and improved performance compared to its predecessor, the FAT file system.

Similar to other file systems, NTFS divides the storage space into clusters, which are the smallest units used for data allocation. However, NTFS clusters can be larger than those in FAT file systems, allowing for more efficient use of space on larger volumes.

The main component of NTFS is a Master File Table (MFT). It acts like an index that keeps track of all files and directories on the storage device. Each entry in the MFT represents a file or directory and includes metadata like file attributes, permissions, timestamps, and pointers to the data's location on the disk.



NTFS supports the concept of data streams, which allows a single file to have multiple streams of data associated with it. This is used for features like file compression, encryption, and alternate data streams, which can store additional information beyond the main file content.

Key features of file systems

	FAT32	NTFS
Max File Size	4GB	256TB for a cluster 64KB size
Journaling	-	X
OS	Windows, Linux, macOS	Windows

FAT32 is the simplest file system of the three described here. It has no journaling, meaning that it is more vulnerable to data corruption in case of sudden system crashes or power failures. It also has a file size limited only to a 4 GB. But it is supported by virtually all major operating systems, including Windows, macOS, and Linux. This compatibility makes it a popular choice for cross-platform data exchange. Nevertheless, NTFS and EXT4 are far more complex with some nuances to them.

NTFS is the default file system for Windows. It includes journaling which keeps it safe from data corruption. It is also more secure because each file record includes a security descriptor that specifies the permissions and access rights for different users and groups. This enables administrators to control who can access, modify, or delete files and directories.

_____**********_____

I/O (PIO).To speed