Fundamentals Operating System (DI03000041)

GTU Course Presentation

Table of Contents I

Table of Contents II

Table of Contents III

Table of Contents IV

Table of Contents V

Table of Contents VI

Table of Contents VII

Table of Contents VIII

Title Slide I

Course: Fundamentals Operating System

- Unit: Introduction of Operating System
- Topic: Fundamental Goals of Operating system, Operating System services

What is an Operating System? I

An Operating System (OS) is a program that acts as an intermediary between a user of a computer and the computer hardware. Its fundamental purpose is to provide an environment in which a user can execute programs in a convenient and efficient manner.

- It manages the computer's hardware resources, including the Central Processing Unit (CPU), memory, and Input/Output (I/O) devices.
- It acts as a Control Program, heavily supervising the execution of user programs to prevent errors and improper use of the underlying computer hardware.
- It provides a logical basis for application programs and abstractions, shielding developers from complex, low-level hardware specificities.

Fundamental Goal 1: Convenience (User View) I

From the user's perspective, the primary goal of an operating system is to maximize convenience and ease of use. The OS hides the complex and cumbersome details of the underlying hardware from the user.

- The OS abstracts complex hardware concepts, such as representing a physical magnetic hard drive as a hierarchical file system, making it convenient for users to store and retrieve data.
- It provides abstract user interfaces (like Command Line Interfaces or Graphical User Interfaces) to interact with hardware effortlessly.
- In Personal Computers (PCs), the OS is designed primarily for ease of use, with some attention paid to performance and less to hardware resource utilization, as the hardware is dedicated to a single user.

Fundamental Goal 2: Efficiency (System View) I

From the system's perspective, the operating system is heavily involved in resource management. The fundamental goal here is the efficient and fair allocation of hardware resources among competing programs and users.

- Resource Allocator: The OS manages all resources (CPU time, memory space, file-storage space, I/O devices) and dynamically decides between conflicting requests for efficient resource use.
- In mainframe or minicomputer environments, resource utilization is paramount because hardware is extremely expensive and must be maximally shared among multiple users without degradation of service.
- Efficiency encompasses maximizing CPU throughput, minimizing response time for interactive tasks, and implementing algorithms to avoid resource deadlocks or process starvation.

Diagram: The Operating System as a Resource Manager I

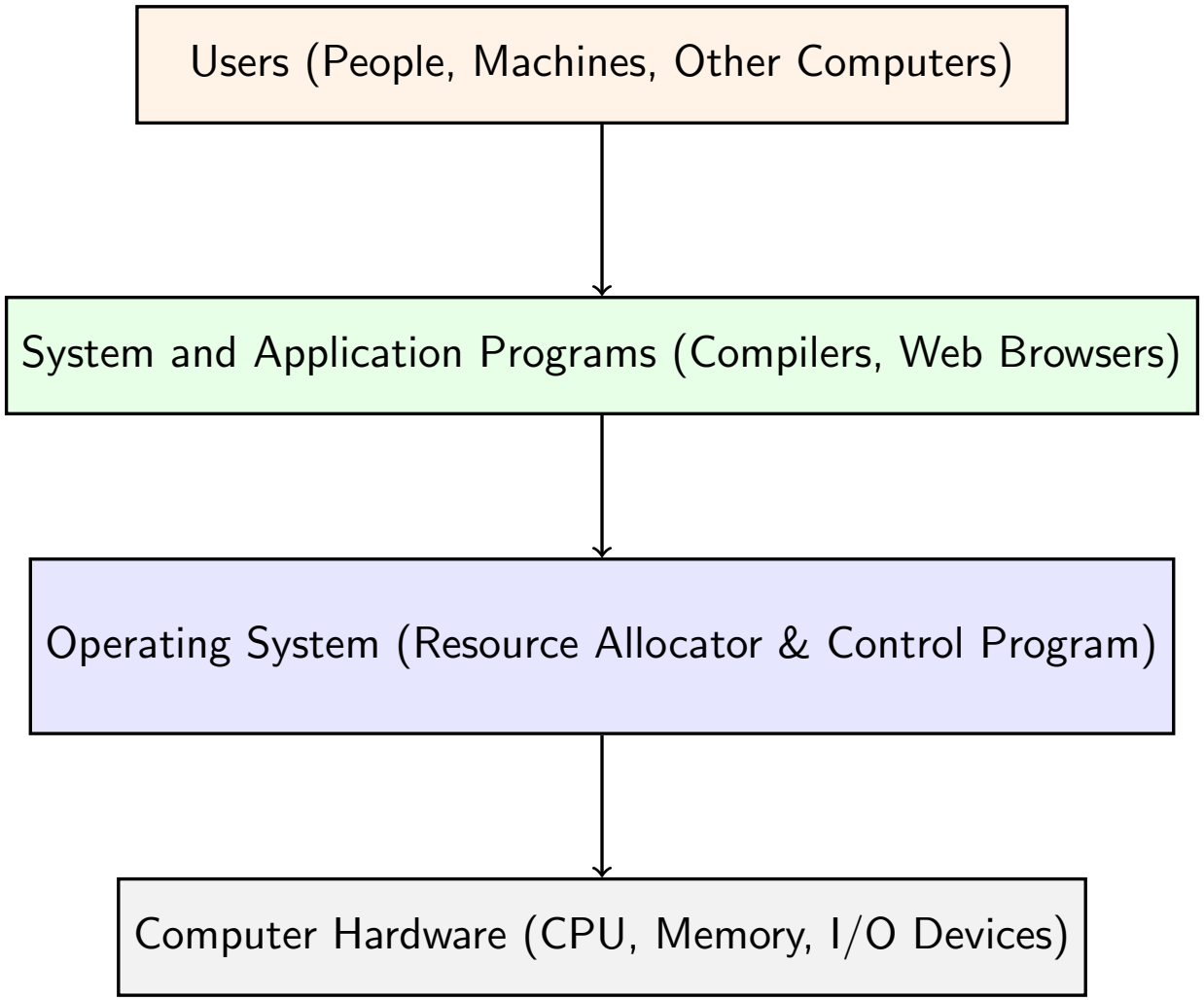


Diagram: The Operating System as a Resource Manager II

- The hardware provides the basic computing resources and physical capabilities for the system.
- The application programs define the logical ways in which these hardware resources are used to solve users' specific computing problems.
- The OS acts as the central coordinator, sitting between applications and hardware, dynamically allocating resources as required.

Other Key OS Goals: Protection, Security, and Robustness I

Beyond convenience and efficiency, modern operating systems must ensure that the computing environment is safe, reliable, and fundamentally resistant to malicious attacks, improper access, or hardware failures.

- Protection: Ensuring that processes and users are isolated; one process should not be able to inadvertently corrupt the memory address space or resources of another process.
- Security: Defending the system against internal and external attacks, including unauthorized access, privilege escalation, viruses, and denial-of-service (DoS) attacks.
- Robustness and Fault Tolerance: The OS kernel should ideally handle hardware or software errors gracefully, preventing a complete system crash or kernel panic when a single peripheral component fails.

Overview of Operating System Services I

An OS provides a safe environment for the execution of programs. It provides critical services to both programs and the users of those programs. These services are typically categorized into user-centric services and system-centric services.

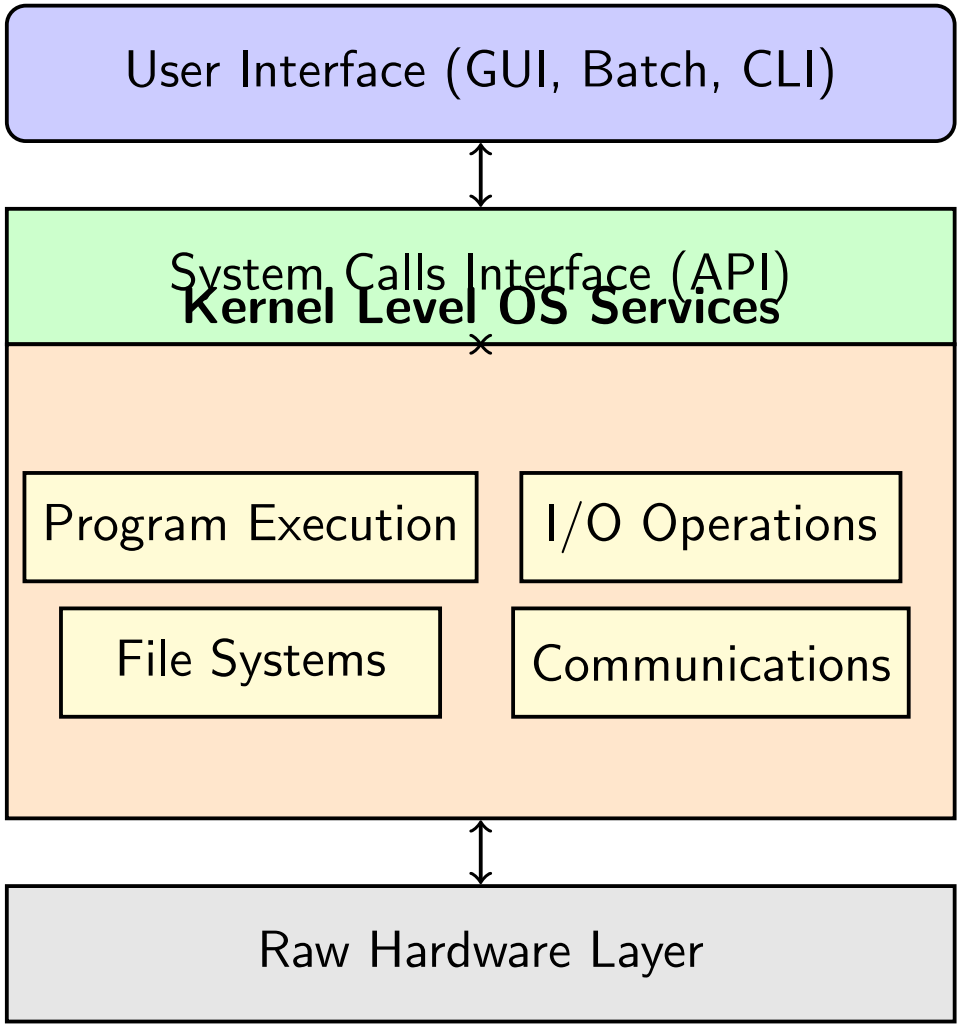
- User-Centric Services are designed to help the user execute computing tasks easily. Key examples include Program Execution, I/O Operations, and File-system manipulation.
- System-Centric Services are designed to ensure the efficient and stable operation of the system itself, particularly in a shared, multi-user environment. Examples include Resource Allocation, Accounting, and Error Detection.
- System calls provide the programmatic Application Programming Interface (API) to these services, acting as the secure bridge between user-space applications and the privileged OS kernel.

Key OS Services: Program Execution & I/O Operations I

The fundamental role of an OS kernel is to run user applications and manage their interactions with hardware components safely, securely, and efficiently without user intervention.

- Program Execution: The OS must be able to load a compiled program into Main Memory (RAM), execute it by assigning CPU time, and gracefully handle its termination (either normal exit or abnormal abort).
- I/O Operations: A running program may require I/O, involving a file or an I/O device. For security and protection, users cannot control physical I/O devices directly; the OS provides the secure abstraction to do so.
- The OS abstracts the complexities of device drivers, hardware interrupts, and Direct Memory Access (DMA) behind a unified, simple interface (e.g., standard POSIX read(), write() calls).

Diagram: Layered Architecture of OS Services I



- The User Interface is the topmost application layer where all human-computer interaction occurs.

Diagram: Layered Architecture of OS Services II

- The System Call interface acts as a rigid, protective boundary, allowing untrusted user programs to request privileged kernel services via software interrupts (traps).
- Kernel Level Services perform the actual heavy lifting (process scheduling, memory paging, file management) in privileged mode before issuing instructions to the Raw Hardware Layer.

Key OS Services: File-System Manipulation & Communications I

Data persistence across reboots and inter-process data exchange are critical services provided by the operating system kernel to enable complex applications.

- File-System Manipulation: The OS provides services to read, write, create, and delete files and hierarchical directories. It enforces file permissions (ACLs), maps files to physical storage sectors, and maintains metadata.
- Communications: Processes often need to exchange information to function collaboratively. The OS facilitates Inter-Process Communication (IPC) via shared memory segments or message passing queues.

Key OS Services: File-System Manipulation & Communications II

- Error Detection: The OS must constantly monitor for hardware errors (e.g., memory parity errors) or software faults (e.g., division by zero), trapping them and taking appropriate action (like killing the faulting process) to maintain system stability.

Title Slide I

Course: Fundamentals Operating System

- Unit: Introduction of Operating System
- Lecture 2: Fundamental Goals and Services
- Focus on convenience, efficiency, and robustness.

Fundamental Goals of an Operating System I

The primary goals of an operating system define its architecture and design philosophy.

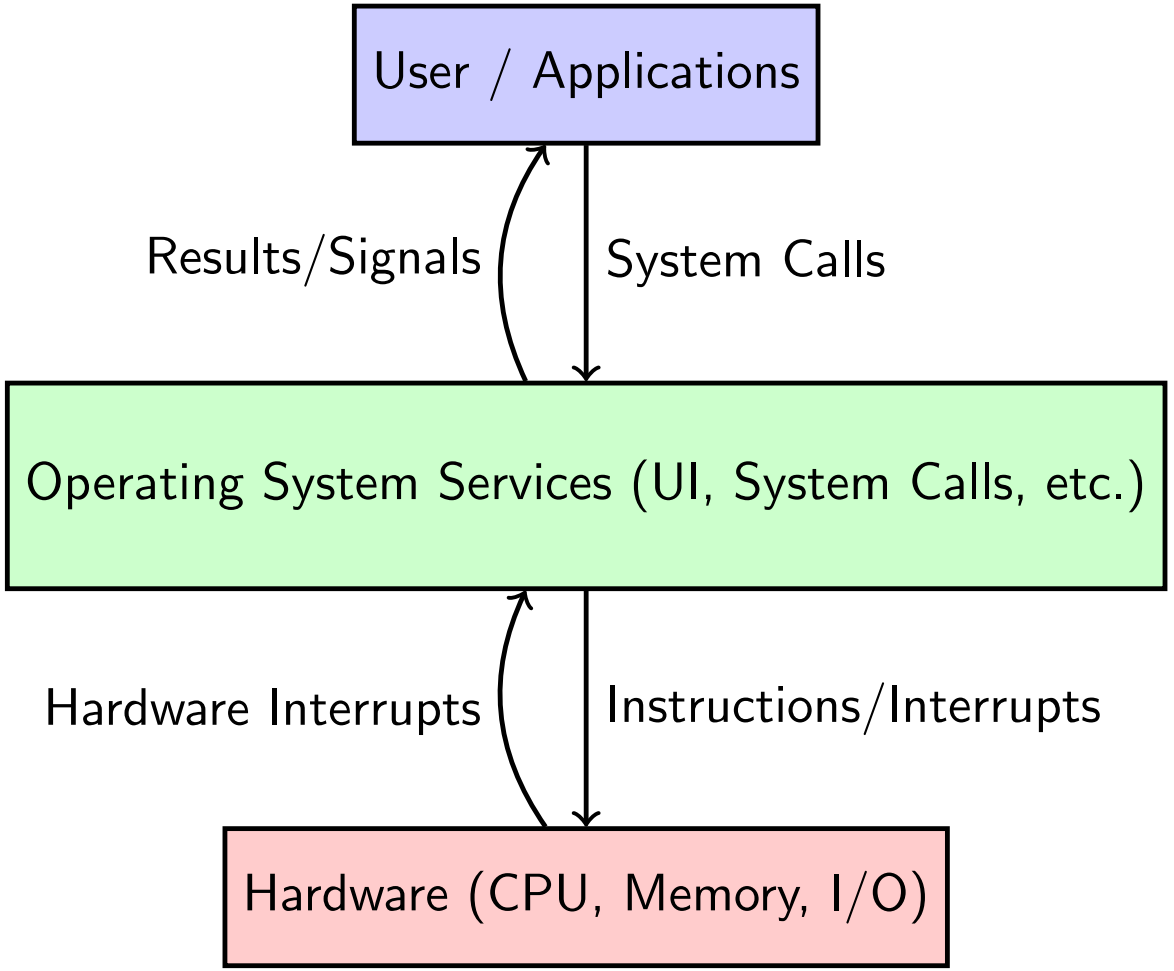
- Convenience: Make the computer easier to use for the end-user by providing a friendly interface and abstracting hardware complexities.
- Efficiency: Allow computer system resources (hardware and software) to be used in an efficient manner, maximizing throughput and minimizing response time.
- Robustness and Security: Ensure the system operates reliably under various conditions and protects user data and resources from unauthorized access or malicious software.
- Evolution: Permit effective development, testing, and introduction of new system functions without interfering with existing services.

The Operating System: Two Perspectives I

Operating systems can be analyzed from the perspective of the user and the system itself.

- User View: The focus is strictly on ease of use, performance, and resource availability, with minimal concern for underlying hardware details or resource utilization.
- System View: The OS is viewed as a resource allocator, managing CPU time, memory space, file-storage space, and I/O devices fairly and efficiently.
- Control Program: The OS controls the execution of user programs to prevent errors and improper use of the computer, acting as the ultimate authority over hardware.
- Compromise: Designing an OS involves balancing these two views, creating a system that is both user-friendly and efficiently managed at the hardware level.

Operating System Services Overview I



- An Operating System provides an environment for the execution of programs.

Operating System Services Overview II

- Services are provided for the convenience of the programmer, to make the programming task easier and standard across the platform.
- These services can be broadly categorized into user-oriented services (e.g., UI, program execution) and system-oriented services (e.g., resource allocation, protection).
- The fundamental interface between user applications and these OS services is provided through System Calls.

User-Oriented Services: Interface and Execution I

These services directly help the user interact with the computer and execute programs seamlessly.

- User Interface (UI): Almost all operating systems have a UI. This can range from a Command-Line Interface (CLI), a graphical user interface (GUI), to a batch interface.
- Program Execution: The system must be able to load a program into memory and run that program. It must be able to end its execution, either normally or abnormally.
- Process Management: Involves creating, scheduling, and terminating processes, allowing multiple tasks to run concurrently without interference.
- The OS abstracts away the low-level details of CPU instruction sets and memory management required for program execution.

User-Oriented Services: I/O and File Systems I

Managing external devices and persistent data are crucial user-oriented services.

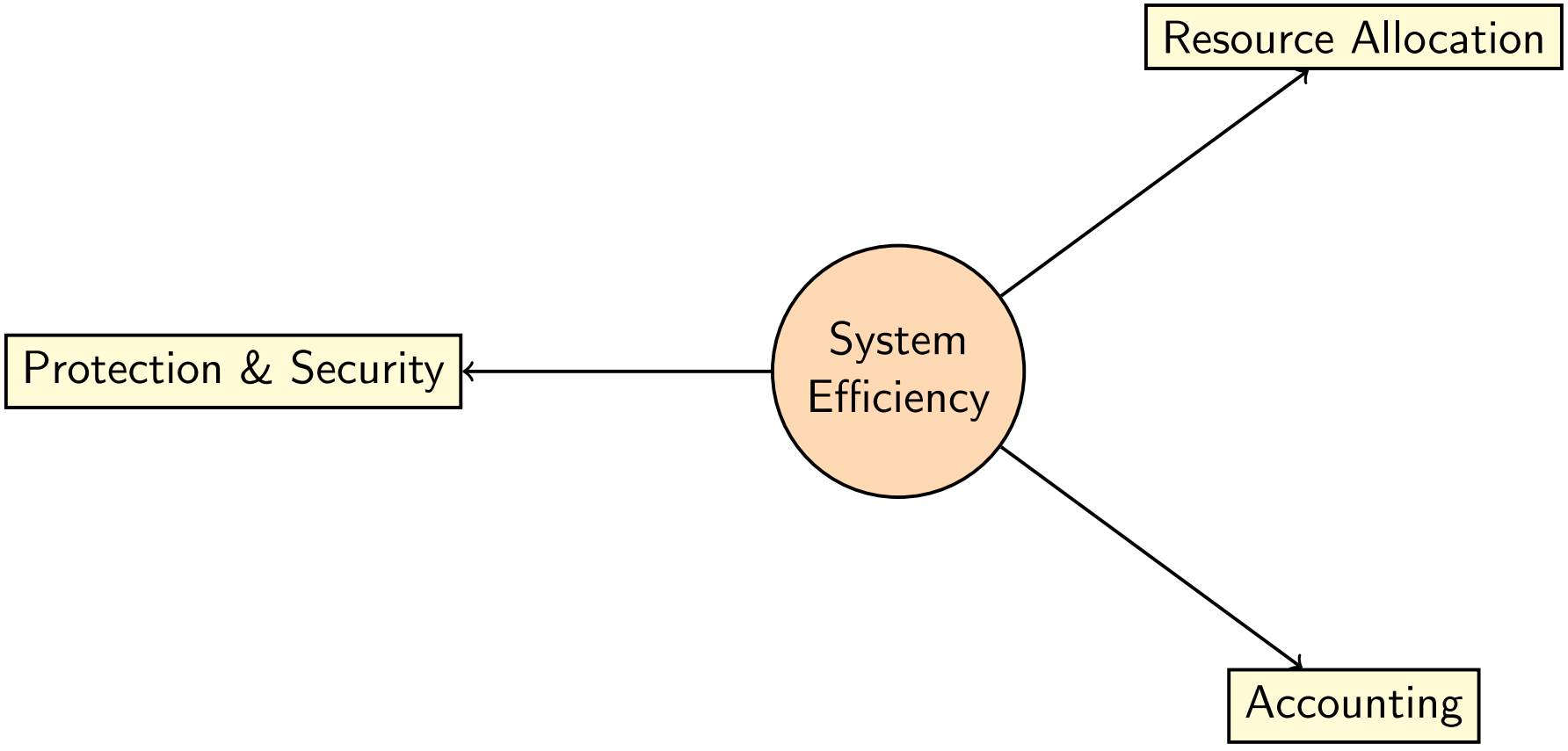
- I/O Operations: A running program may require I/O, which may involve a file or an I/O device. For efficiency and protection, users cannot control I/O devices directly.
- File-System Manipulation: Programs need to read, write, create, and delete files and directories by name, search for a given file, and list file information.
- File Permissions: The OS must provide a robust mechanism for file ownership and permissions to ensure data security and privacy among multiple users.
- Caching and Buffering: The OS optimizes I/O operations through caching frequently accessed data in RAM and buffering I/O streams to match varying device speeds.

Services: Communications and Error Detection I

The OS facilitates inter-process communication and ensures system stability through continuous error handling.

- Communications: Processes may exchange information on the same computer or between computers over a network. This is implemented via shared memory or message passing.
- Message Passing: Packets of information are moved between processes by the operating system, providing a structured, synchronized way to communicate.
- Error Detection: The OS needs to be constantly aware of possible errors occurring in the CPU, memory hardware, I/O devices, or in the user program.
- Recovery: For each type of error, the OS takes the appropriate action to ensure correct and consistent computing (e.g., terminating the process, retrying the operation, or halting the system).

System-Oriented Services for Efficient Operation I



- Resource Allocation: When multiple users or jobs are running concurrently, limited resources (CPU cycles, main memory, file storage) must be allocated to each fair and squarely.

System-Oriented Services for Efficient Operation II

- CPU Scheduling: Complex algorithms determine which process runs next, optimizing for specific metrics like CPU utilization, overall throughput, or user response time.
- Accounting: The OS keeps track of which users utilize how much and what kinds of computer resources, crucial for billing or accumulating usage statistics for tuning.
- System Performance Monitoring: The OS continuously observes system behavior and load to identify hardware or software bottlenecks and optimize overall throughput.

Protection and Security Services I

Safeguarding the system, user data, and the OS itself is a paramount responsibility of modern operating systems.

- Protection: Involves ensuring that all access to system resources is rigidly controlled. It provides a mechanism for controlling access by programs, processes, or users to both system and user resources.
- Security: Starts with requiring each user to authenticate themselves to the system (usually by means of a password or biometrics), extending to defending external I/O devices from invalid access attempts.
- Privilege Levels: Implementing hardware-enforced user mode and kernel mode to prevent user applications from executing privileged instructions directly.

Protection and Security Services II

- Threat Mitigation: Defense mechanisms against viruses, worms, and denial-of-service attacks are increasingly integrated directly into the OS layer.

Summary and Conclusion I

The operating system acts as the essential intermediary between the user and the underlying hardware.

- Goal: Balance convenience for the user with efficiency for the system hardware.
- Service Layer: Provides a unified set of services (UI, Execution, I/O, FS, Communication) that abstract away hardware complexity.
- System Manager: Acts as the ultimate resource allocator and protector, ensuring stable, secure, and fair operation.
- Next Lecture: We will explore the structure of operating systems and how system calls are implemented to access these services.

Title Slide I

Course: Fundamentals Operating System

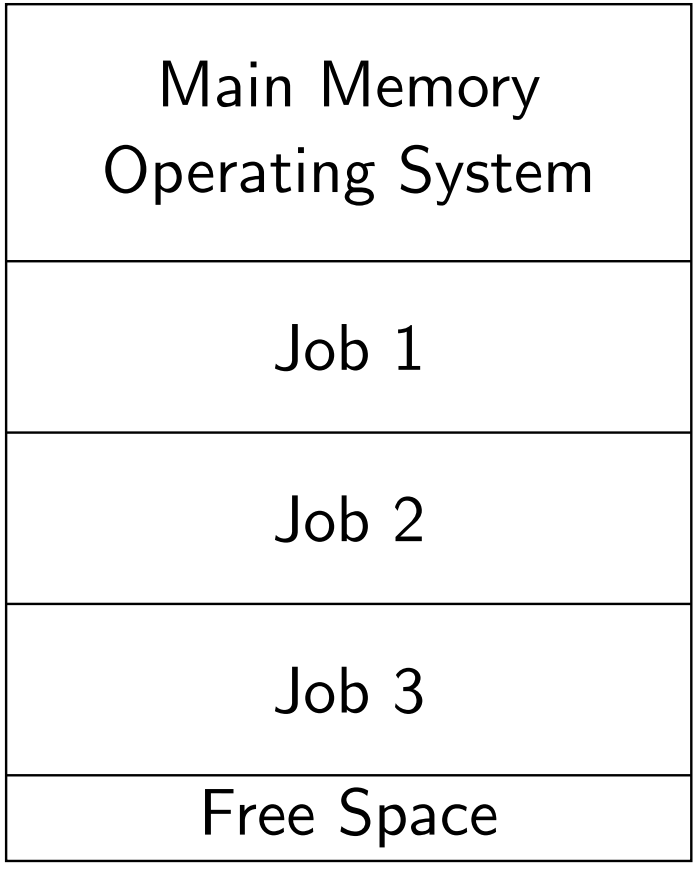
- Unit: Introduction of Operating System
- Lecture 3: Types of Operating Systems
- Overview of diverse OS architectures

Multiprogramming Operating Systems I

Maximizing CPU utilization by keeping multiple jobs in main memory simultaneously.

- In a multiprogramming system, the OS keeps several jobs in memory at the same time.
- The OS selects one job and begins executing it. When that job needs to wait (e.g., for I/O operations), the OS switches to another job.
- This ensures that the CPU is never idle as long as there is at least one job ready to execute.
- Requires job scheduling to choose which jobs enter memory, and CPU scheduling to choose which job to execute next.

Memory Layout in Multiprogramming I



- Memory is partitioned to hold the OS and multiple user jobs simultaneously.
- Requires memory management mechanisms to allocate and deallocate partitions.

Memory Layout in Multiprogramming II

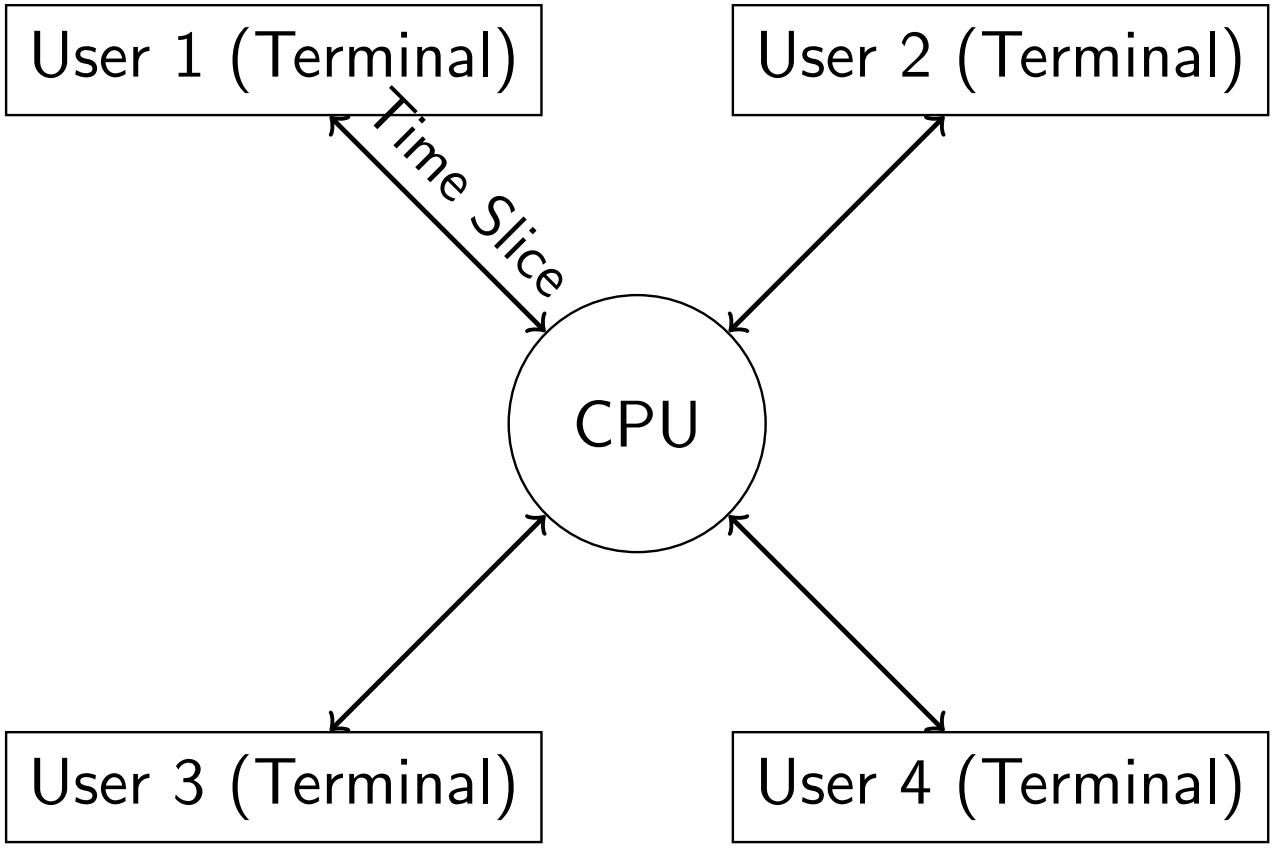
- Memory protection is crucial to prevent one job from modifying the memory of another job or the OS.

Time-Sharing (Multitasking) Operating Systems I

A logical extension of multiprogramming that provides interactive computing for multiple users.

- The CPU executes multiple jobs by switching among them so frequently that users can interact with each program while it is running.
- Each user has at least one separate program in memory. A program loaded into memory and executing is called a process.
- CPU scheduling algorithms provide a brief 'time slice' (or quantum) to each user's process.
- Provides the illusion that each user has their own dedicated computer, maintaining response times typically under a second.

Time-Sharing Architecture I



- Multiple terminals connect to a single central computer system.
- The OS rapidly multiplexes the CPU across all active terminals.

Time-Sharing Architecture II

- Requires advanced memory management and protection, such as virtual memory, to handle many processes larger than physical RAM.

Real-Time Operating Systems (RTOS) I

Systems designed for applications where data processing must happen within a strict time boundary.

- An RTOS has well-defined, fixed time constraints; processing must occur within these constraints or the system will fail.
- Hard real-time systems guarantee that critical tasks complete on time (e.g., medical imaging, industrial control, robotics).
- Soft real-time systems prioritize critical processes but do not provide absolute time guarantees (e.g., multimedia streaming, virtual reality).
- Characterized by minimal interrupt latency and deterministic scheduling algorithms.

Multithreading Operating Systems I

Operating systems that support the execution of multiple threads within a single process context.

- A thread (lightweight process) is a fundamental unit of CPU utilization, comprising a thread ID, program counter, register set, and stack.
- Threads within the same process share code, data, and OS resources like open files and signals.
- Advantages include increased responsiveness, easier resource sharing, economy (cheaper to create and switch than processes), and scalability on multiprocessor architectures.
- Concurrency interleaves threads on a single core; parallelism executes threads simultaneously on multiple cores.

Distributed Operating Systems I

A system managing a collection of independent, networked computers to appear as a single coherent system.

- Users are unaware of the location where their programs run or their files reside, achieving system transparency.
- Facilitates resource sharing across a network, including hardware (printers), files, and compute power.
- Enhances computation speedup through load balancing and increases data availability and system reliability (fault tolerance).
- Relies heavily on networking protocols and distributed file systems (DFS) for communication and data consistency.

Embedded Operating Systems I

Specialized OS designed to perform dedicated tasks within hardware-constrained devices.

- Highly resource-constrained, operating with limited RAM, ROM, and processing power.
- Unlike general-purpose computing, they are typically designed for a single specific application.
- Frequently overlap with RTOS requirements, as many embedded systems (like automotive engine controllers) require real-time responsiveness.
- Common examples include FreeRTOS, VxWorks, and specialized, stripped-down versions of Linux used in IoT devices and appliances.

Summary I

Comparing Operating System Architectures and their Primary Objectives.

- Multiprogramming: Maximizes CPU throughput by keeping jobs ready in memory.
- Time-sharing: Maximizes user response time and interactivity via time slicing.
- Real-Time: Meets strict deterministic timing constraints for critical operations.
- Distributed: Pools networked hardware to create a transparent, fault-tolerant unified system.
- Embedded: Runs dedicated, resource-efficient tasks on specialized hardware.

Title Slide I

Course: Fundamentals Operating System

- Unit: Introduction of Operating System
- Topic: OS Structure: Layered, Monolithic, Microkernel

Introduction to OS Structures I

An operating system's structure dictates how its core components are organized and how they communicate with each other.

- A well-designed OS structure is crucial for maintainability, reliability, scalability, and security.
- Early operating systems lacked well-defined structures, leading to complex and error-prone systems.
- Modern operating systems employ specific architectural models to manage the immense complexity of hardware management and software execution.
- The primary structures we will examine in this lecture are Monolithic, Layered, and Microkernel architectures.
- Each architectural approach represents a different set of tradeoffs between raw execution performance, system modularity, and kernel security.

Monolithic Operating Systems I

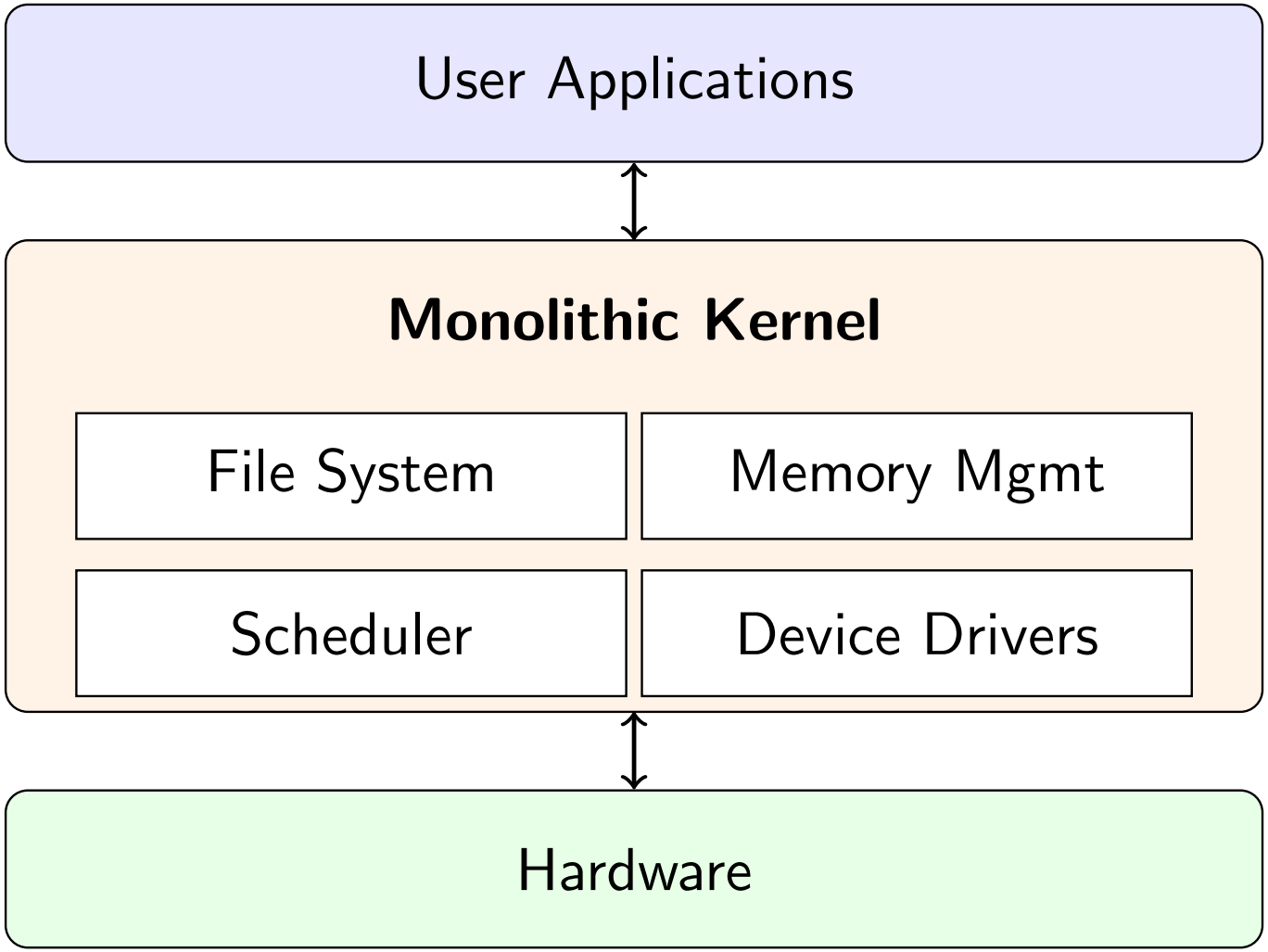
A monolithic operating system places all system services within a single, unified kernel space running in privileged supervisor mode.

- The entire operating system operates as a single executable program, sharing a unified memory space.
- This approach yields extremely high performance due to low overhead in inter-process communication (IPC) and fast system call execution.
- Components like the file system, CPU scheduler, device drivers, and memory management are tightly intertwined.
- Drawbacks: A bug or vulnerability in any single component (like a faulty device driver) can crash the entire system.
- It becomes increasingly difficult to maintain, debug, and extend as the codebase grows large and complex.

Monolithic Operating Systems II

- Examples include early UNIX implementations, Linux (though modern Linux incorporates loadable kernel modules to mitigate drawbacks), and MS-DOS.

Monolithic Architecture Diagram I



- User applications reside in user space and communicate via the system call interface.

Monolithic Architecture Diagram II

- The monolithic kernel is a single, large block containing all OS services operating in kernel mode.
- Hardware resources sit directly below the monolithic kernel.
- All internal kernel components can communicate directly using standard C function calls.

Layered Operating Systems I

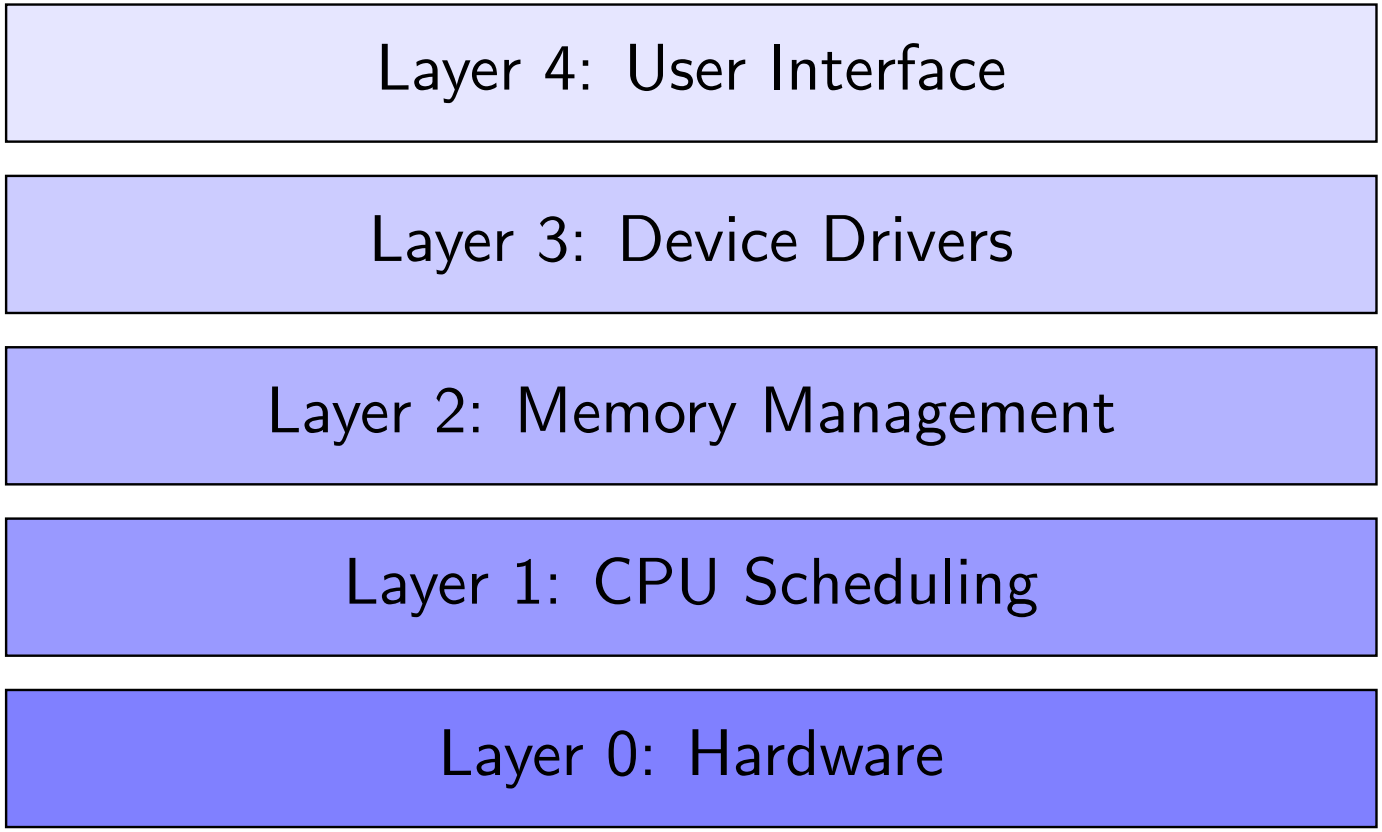
The layered approach structures the operating system as a hierarchy of well-defined, distinct layers.

- Layer 0 is the lowest level and directly interfaces with the physical hardware, while the highest layer (Layer N) represents the user interface.
- A critical design rule: Each layer is constructed using only the operations, services, and data structures provided by the layers immediately below it.
- This design significantly simplifies debugging and system verification, as layers can be developed and tested sequentially from the bottom up.
- Information hiding is strictly enforced; inner layers encapsulate implementation details from outer layers.

Layered Operating Systems II

- A major challenge is appropriately defining the specific functionalities that belong in each layer to prevent circular dependencies.
- Performance can suffer compared to monolithic designs because a single user program's request may incur the overhead of passing through multiple layers.

Layered Architecture Diagram I



- The system is decomposed into sequential, independent layers.
- Higher layers rely exclusively on the services of lower layers.
- The hardware is abstracted entirely from the uppermost software layers.
- Debugging can proceed systematically from Layer 0 upwards.

Microkernel Architecture I

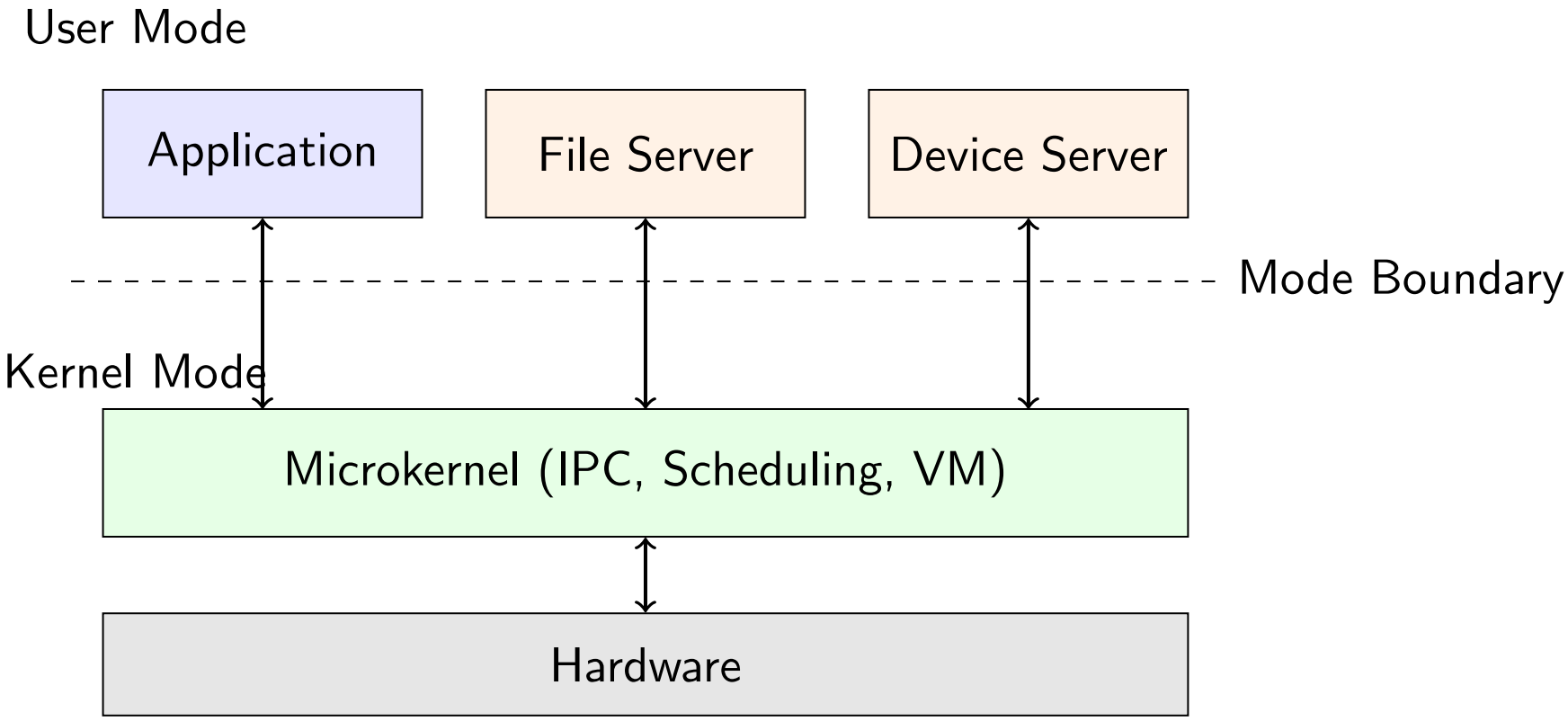
The microkernel approach minimizes the size of the kernel by moving non-essential components out of supervisor mode and into user space.

- Only the most essential core OS functions (IPC, basic thread scheduling, minimal memory management) remain in the privileged kernel.
- Other services, such as file systems, device drivers, and networking protocols, run as independent user-mode processes known as servers.
- Communication between user applications and these OS services is handled almost entirely via message passing mechanisms managed by the microkernel.
- Provides high reliability and security: if an OS service (e.g., a network driver) crashes, it does not bring down the core kernel or the entire system.

Microkernel Architecture II

- Highly extensible and easily portable to new hardware architectures, as only the small microkernel requires significant modification.
- The primary drawback is performance overhead due to the high volume of message passing and frequent context switches between user and kernel space.
- Notable examples include the Mach kernel (the foundation of macOS and iOS), QNX (used heavily in automotive and embedded systems), and the L4 microkernel family.

Microkernel Architecture Diagram I



- A strict boundary separates User Mode processes from the minimal Kernel Mode core.
- Traditional OS services operate as standard user-level processes (servers).

Microkernel Architecture Diagram II

- The Microkernel's primary role is to act as a message router and hardware abstraction layer.
- Direct interactions between OS components are replaced by indirect IPC.

Comparison of OS Structures I

Comparing Monolithic, Layered, and Microkernel architectures highlights their distinct engineering tradeoffs and optimal use cases.

- Performance: Monolithic is generally the fastest due to direct C function calls; Microkernel is the slowest due to heavy IPC and context switching overhead; Layered falls in between.
- Reliability & Stability: Microkernels are highly reliable since a service crash is localized and doesn't halt the system; Monolithic architectures are highly vulnerable to single-point driver failures.
- Maintainability & Extensibility: Layered and Microkernel structures are far easier to maintain, debug, and extend than tightly-coupled, massive Monolithic codebases.
- Security: The Microkernel approach offers the smallest Trusted Computing Base (TCB), significantly reducing the system's attack surface and enhancing overall security.

Comparison of OS Structures II

- Modern systems often utilize Hybrid approaches (e.g., Windows NT, macOS XNU) attempting to combine the performance of monolithic kernels with the stability and modularity of microkernels.

Summary and Conclusion I

Understanding the fundamental OS structures is essential for analyzing system behavior, performance bottlenecks, and security vulnerabilities.

- Monolithic kernels prioritize absolute raw performance over internal modularity and fault isolation.
- Layered kernels prioritize structured software development and systematic debugging but can suffer from performance degradation due to layering overhead.
- Microkernels prioritize extreme reliability, strong security isolation, and system extensibility at the cost of IPC-induced performance penalties.
- Hybrid kernels are the most prevalent in modern general-purpose computing, seeking the optimal pragmatic balance of these theoretical traits.

Summary and Conclusion II

- Next Lecture Topic: We will dive into Processes, Process State Models, and the mechanisms of Process Management.

Title Slide I

Course: Fundamentals Operating System

- Unit: Introduction of Operating System
- Lecture 5: OS structure: Layered, Monolithic, Microkernel Operating Systems

Introduction to OS Structures I

An operating system is a large and complex system that must be engineered carefully to function correctly, be easily modified, and maintain performance. Structuring an OS defines how its components are interconnected and communicate.

- Without structure, an OS becomes an unmanageable collection of procedures.
- Proper structuring allows for modularity, easier debugging, and maintenance.
- Common structural models include: Monolithic, Layered, and Microkernel approaches.

The Monolithic Structure I

In a monolithic operating system, the entire OS runs as a single program in kernel space. All OS services—such as process management, memory management, and file systems—are bundled together.

- There is no strict separation between components; any procedure can call any other procedure.
- Often referred to as 'The Big Mess' if poorly organized, though modern monolithic OSes (like Linux) use modular designs.
- Execution is highly efficient due to low overhead in inter-process communication (IPC) within the kernel.

Monolithic OS: Advantages and Disadvantages I

While monolithic kernels are fast, their lack of isolation poses significant engineering challenges as the system grows in size and complexity.

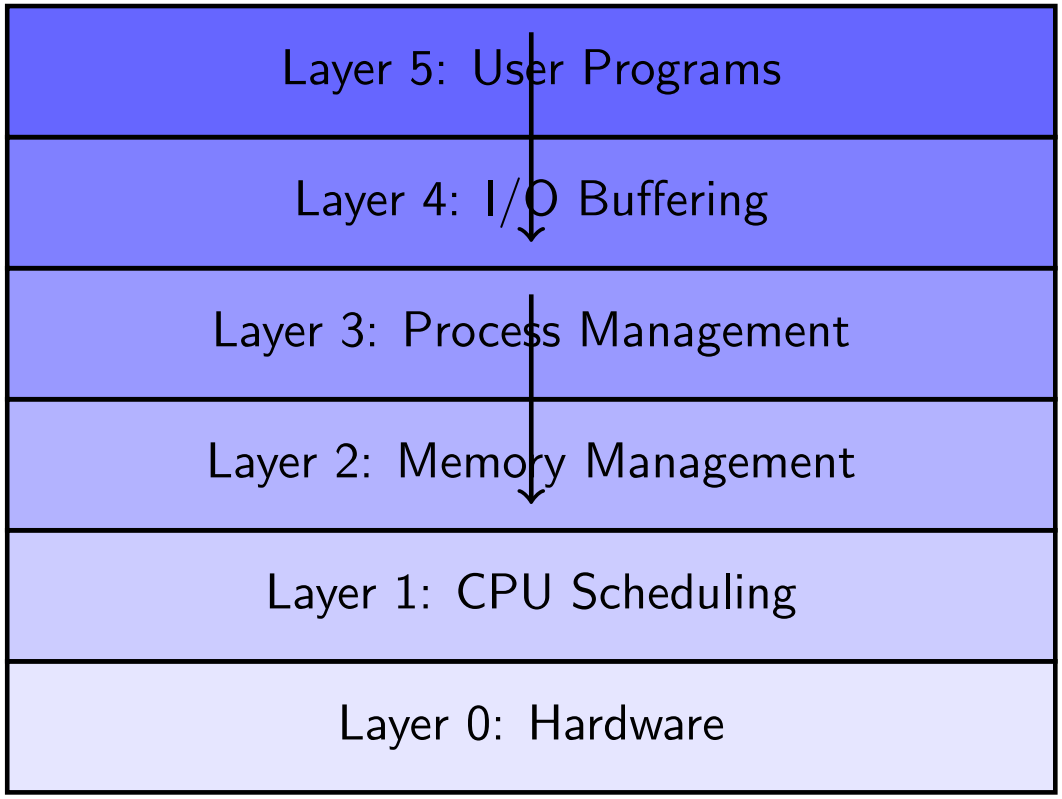
- Advantage: High performance due to direct procedure calls and shared memory space.
- Disadvantage: A bug in one component (e.g., a device driver) can easily crash the entire system.
- Disadvantage: Hard to maintain, extend, and understand due to tightly coupled components.
- Examples: MS-DOS (early monolithic), UNIX, Linux (modular monolithic).

The Layered Approach I

To address the complexity of monolithic systems, the layered approach divides the operating system into a number of distinct layers (levels). Each layer is built directly on top of lower layers.

- The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.
- A layer is an implementation of an abstract object made up of data and the operations that can manipulate those data.
- Crucial rule: Layer M can only use the services of layers below it (M-1, M-2, etc.).

Layered Architecture Diagram I



Strict Top-Down Dependency

- This diagram illustrates a generic conceptual layering similar to the THE operating system designed by Dijkstra.
- Each layer encapsulates specific state and functions, isolating them from higher layers.

Layered Architecture Diagram II

- Information hiding: Upper layers do not need to know how lower layers are implemented, only what they do.

Layered Approach: Pros & Cons I

Layering provides a significant advantage in system construction and debugging, although it can introduce noticeable performance overhead.

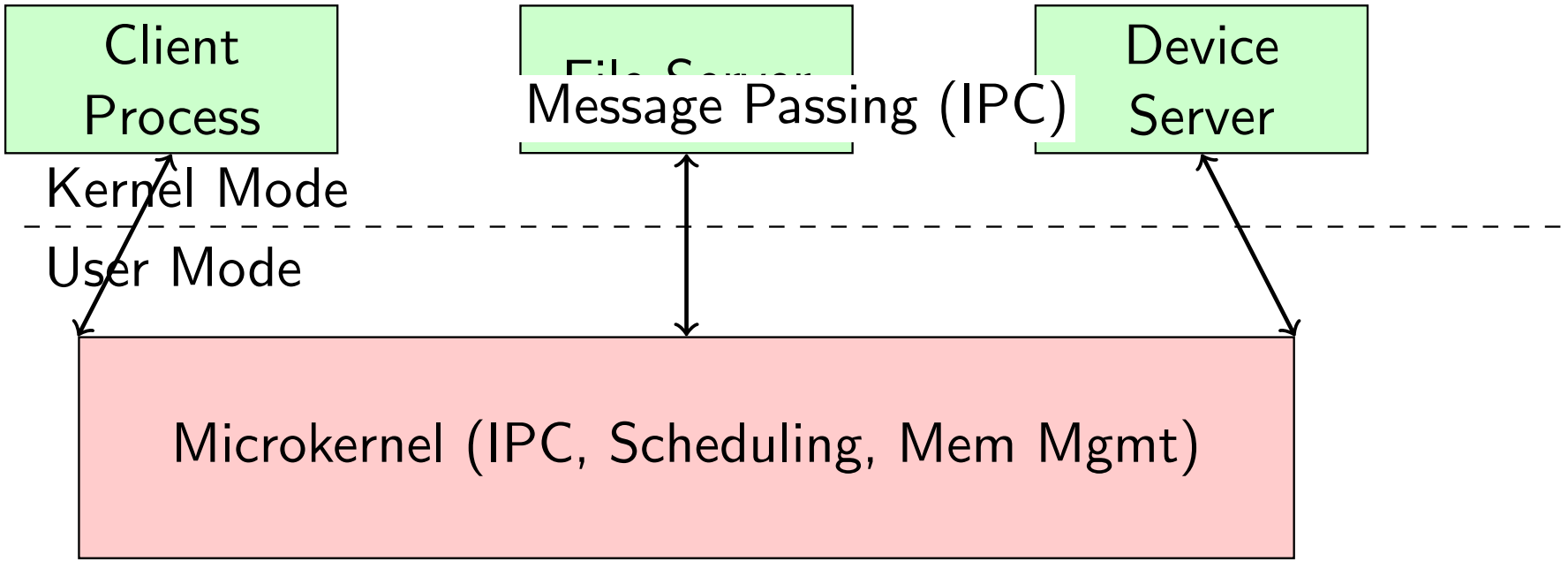
- Advantage: Simplicity of construction and debugging. Layers are built and tested sequentially from the bottom up.
- Advantage: Clear interfaces and information hiding make the system significantly easier to modify.
- Disadvantage: Performance degradation. A system call from a user program must pass through multiple layers to reach hardware.
- Disadvantage: Difficulty in appropriately defining the layers (e.g., does the backing-store driver go above or below the CPU scheduler?).

The Microkernel Structure I

The microkernel approach minimizes the kernel by moving as much functionality as possible out of the kernel space into user space. These functions run as independent, user-level processes called servers.

- The kernel only provides minimal mechanisms: IPC (Inter-Process Communication), basic scheduling, and low-level memory management.
- Services like file systems, device drivers, and networking run completely in user mode.
- Communication between client programs and services occurs via message passing, facilitated securely by the microkernel.

Microkernel Architecture Diagram I



- User mode securely hosts both the standard client processes and system servers.
- Kernel mode is strictly reserved solely for the microkernel itself.
- The microkernel's primary role is to act as a highly secure message bus routing IPC between user-mode components.

Microkernel: Advantages and Disadvantages I

Microkernels heavily prioritize system security, stability, and reliability, usually at the cost of overall system execution performance.

- Advantage: High reliability and security. If a service (like a device driver) crashes, the rest of the OS remains completely unaffected.
- Advantage: Extensibility and portability. New services can be added in user space without needing to modify or recompile the kernel.
- Disadvantage: Performance overhead due to frequent user-kernel space context switching and the necessity of message passing.
- Examples: Mach (the basis for macOS/iOS XNU kernel), QNX (real-time OS), and Minix.

Title Slide I

Course: Fundamentals Operating System

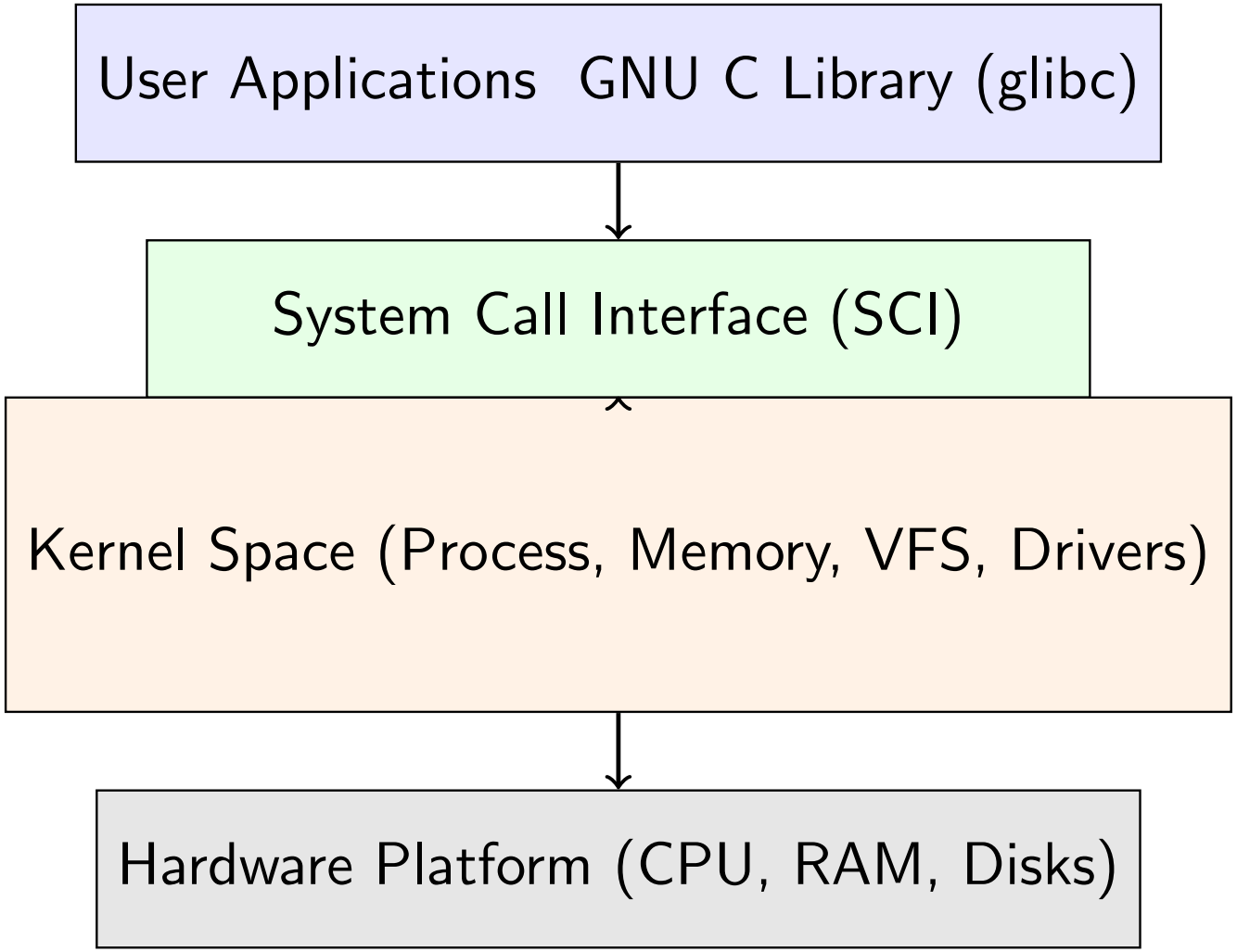
- Unit: Introduction of Operating System

Introduction to Operating System Case Studies I

This lecture provides an in-depth comparative study of the three most widely used modern operating systems: Linux, Windows, and macOS, analyzing their architectural designs and core subsystems.

- Understanding theoretical OS concepts through real-world implementations.
- Examining different architectural choices (Monolithic vs. Microkernel vs. Hybrid).
- Analyzing design trade-offs in process management, memory allocation, and file systems.

Linux: Architectural Overview I



- Linux uses a monolithic kernel architecture where OS services run directly in kernel space for maximum performance.

Linux: Architectural Overview II

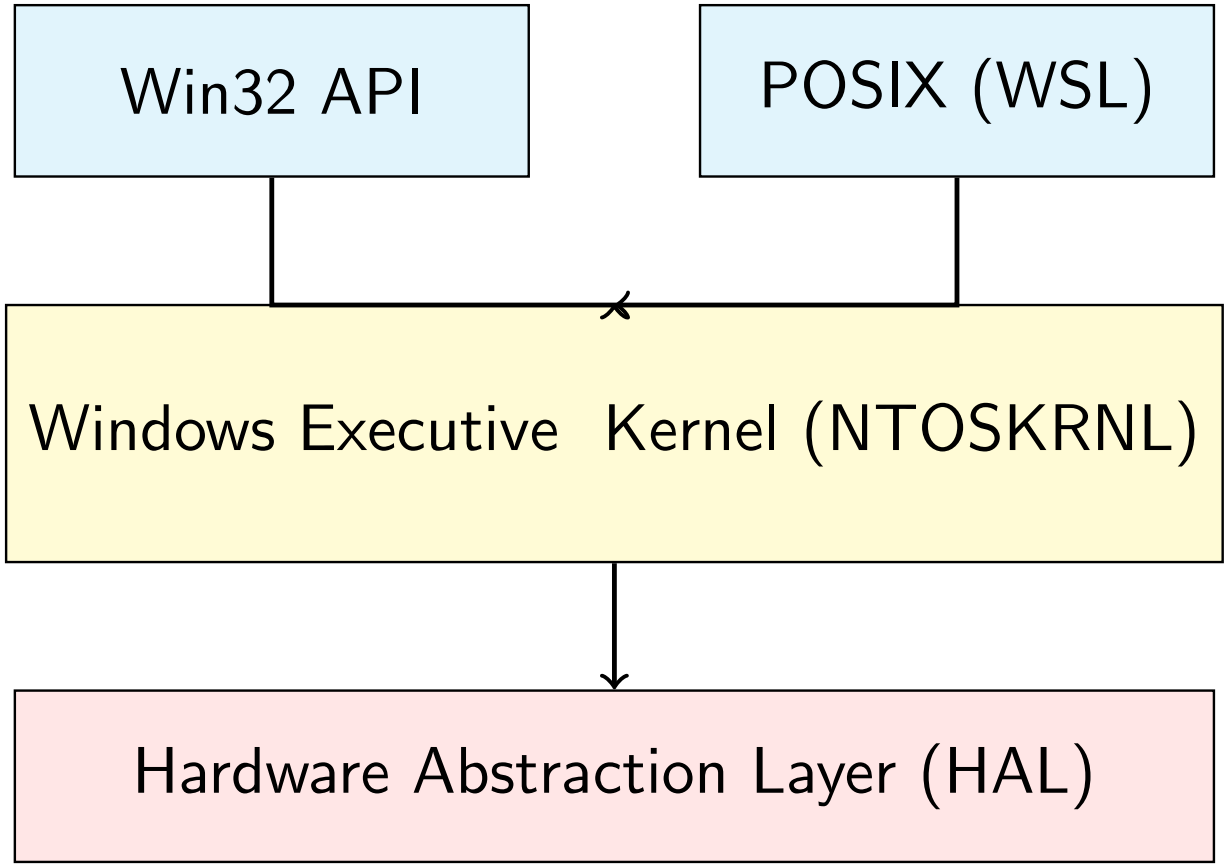
- The System Call Interface (SCI) acts as the bridge, allowing user-space applications to request kernel services.
- Despite being monolithic, Linux features a highly modular design allowing Loadable Kernel Modules (LKMs) to be inserted dynamically.

Linux: Process Management & File System I

Linux simplifies process management by treating both processes and threads similarly and abstracts all hardware via its Virtual File System.

- Process Representation: Both processes and threads are represented by the same 'task_struct' data structure.
- Scheduling: Utilizes the Completely Fair Scheduler (CFS) which provides O(log N) scheduling complexity via a red-black tree.
- Virtual File System (VFS): Abstracts different file systems (e.g., ext4, btrfs, XFS) into a single unified directory tree starting from the root (/).
- File Descriptor Abstraction: 'Everything is a file'—devices, network sockets, and IPC pipes are accessed identically to regular files.

Windows 11: Architectural Overview I



- Windows NT architecture uses a hybrid kernel, combining a microkernel's modularity with a monolithic kernel's performance elements.

Windows 11: Architectural Overview II

- User Mode applications interact through environmental subsystems (like Win32 or WSL) which translate API calls into native system services.
- The Hardware Abstraction Layer (HAL) isolates the kernel and executive modules from platform-specific hardware differences.

Windows 11: Memory Management & Security I

Windows implements a complex virtual memory manager alongside an object-oriented security model natively integrated into the kernel.

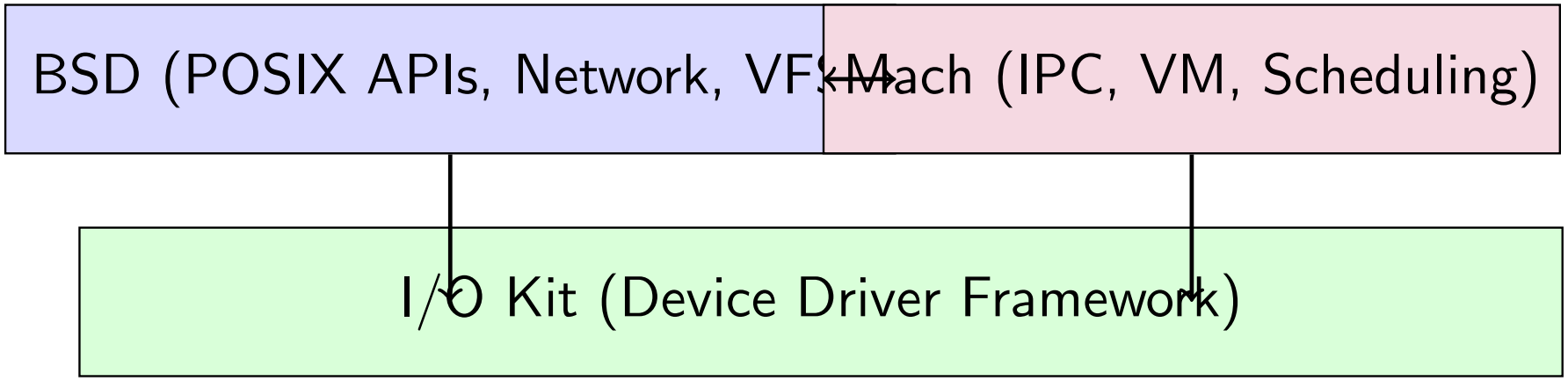
- Memory Management: Employs a working set model with demand paging, keeping active pages in RAM and trimming inactive ones to a pagefile.
- Object Manager: Centralized resource manager where every OS entity (file, thread, event) is treated as an object with attached security descriptors.
- Security Reference Monitor (SRM): Enforces strict access validation and generates audit logs based on Access Control Lists (ACLs).
- NTFS Features: The default file system provides essential enterprise features like journaling, file-level encryption (EFS), and sparse files.

macOS: Architectural Overview (Darwin) I

macOS is built on Darwin, a fully POSIX-compliant UNIX OS, which integrates advanced GUI frameworks with a robust command-line core.

- Darwin Core: An open-source UNIX environment providing fundamental operating system services and stability.
- XNU Kernel: The heart of Darwin, combining concepts from the Mach microkernel with FreeBSD subsystems.
- Core Services: Includes foundational APIs, such as Foundation and Core Data, sitting directly above the kernel layer.
- Application Frameworks: The Cocoa framework provides robust, object-oriented APIs required for native macOS application development.

macOS: The XNU Kernel Architecture I



- XNU stands for 'X is Not Unix' and employs a hybrid kernel architecture to balance speed and modularity.
- Mach Subsystem: Handles low-level OS operations, primarily inter-process communication (IPC), thread scheduling, and virtual memory.
- BSD Subsystem: Provides higher-level abstractions built on top of Mach, including a mature networking stack, VFS, and POSIX compliance.

macOS: The XNU Kernel Architecture II

- I/O Kit: An object-oriented device driver framework written in a subset of C++ that manages hardware hot-plugging and power states natively.

Comparative Analysis: OS Architectures I

A comparison of the core architectural paradigms that drive Linux, Windows, and macOS reveals differing priorities in design and execution.

- Linux (Monolithic): Focuses on raw performance where most OS services run in the same memory space, resulting in highly efficient system calls.
- Windows (Hybrid NT): Balances robust modularity with monolithic execution speeds, heavily utilizing HAL for broad hardware compatibility.
- macOS (Hybrid XNU): Fuses the Mach microkernel's pristine IPC and memory management with BSD's mature networking and API standards.
- Driver Models: Linux incorporates drivers directly or via LKMs, whereas Windows (KMDF) and macOS (I/O Kit) utilize isolated driver frameworks.

Conclusion & Summary I

Modern operating systems are the result of decades of computer science evolution, each tailored strategically to specific target environments and users.

- Linux remains the dominant force in servers, supercomputers, and embedded/mobile (Android) systems due to its open-source adaptability.
- Windows sustains global enterprise and desktop dominance through strong backward compatibility and user-friendly GUI integration.
- macOS delivers a premium, highly secure UNIX-based desktop experience optimized explicitly for Apple's proprietary hardware ecosystems.
- Convergence: Despite architectural differences, all three OSes share common foundational concepts like virtual memory and preemptive multitasking.

Title Slide I

Course: Fundamentals Operating System

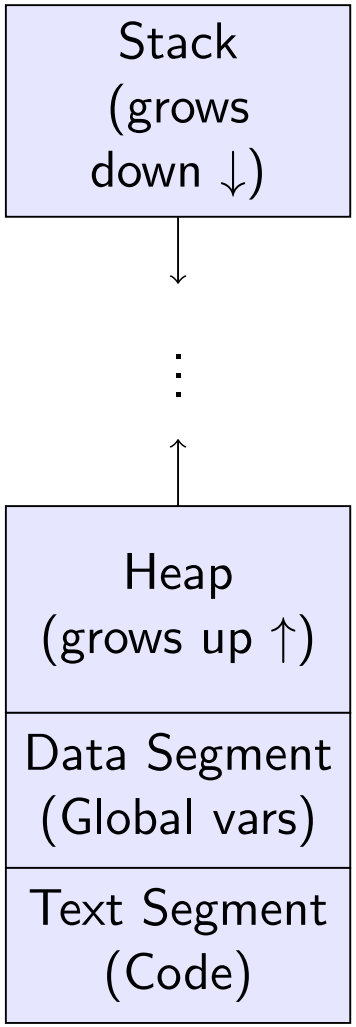
- Unit: Concepts of Process
- Topics: Overview of the Process & threads, Process Life Cycle, PCB

What is a Process? I

A process is a program in execution, forming the basis of all computation in modern operating systems.

- A program is a passive entity (executable file on disk), while a process is an active entity.
- A process requires resources such as CPU time, memory, files, and I/O devices to accomplish its task.
- When an executable file is loaded into memory, a process is created.
- The execution of a process must progress in a sequential fashion.
- A process includes the current activity represented by the value of the program counter and the contents of the processor's registers.

Process Memory Layout I



- Text Section: The executable code.
- Data Section: Global and static variables.

Process Memory Layout II

- Heap Section: Memory dynamically allocated during run time.
- Stack Section: Temporary data such as function parameters, return addresses, and local variables.
- The heap and stack grow towards each other, but the OS prevents them from overlapping.

Threads: The Lightweight Process I

A thread is the basic unit of CPU utilization, comprising a thread ID, a program counter, a register set, and a stack.

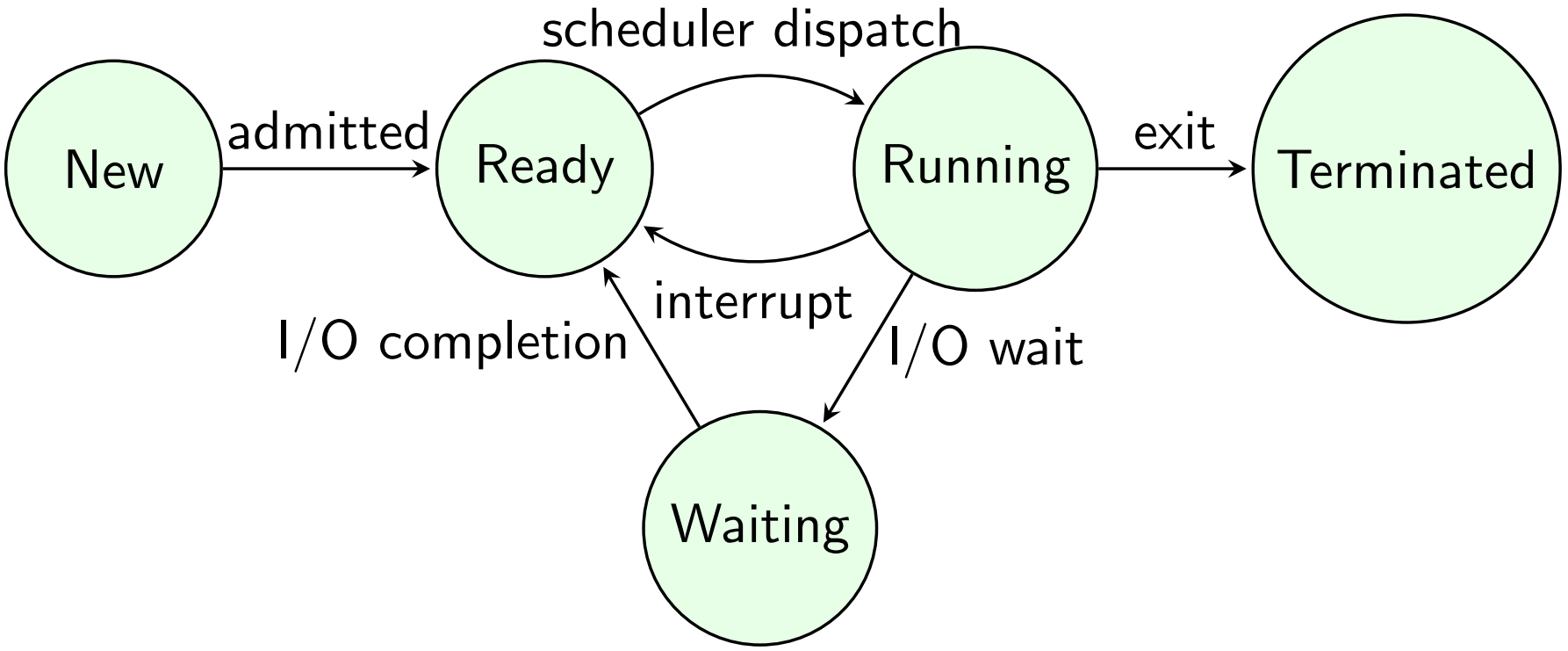
- Threads belonging to the same process share its code section, data section, and other OS resources like open files and signals.
- Multithreading improves responsiveness, allows resource sharing, and is more economical than creating multiple processes.
- Context switching between threads of the same process is significantly faster than process context switching.
- User-level threads are managed without kernel support, while kernel-level threads are supported and managed directly by the OS.
- Many-to-One, One-to-One, and Many-to-Many are standard multithreading models connecting user and kernel threads.

Process Life Cycle and States I

As a process executes, it changes state. The state of a process is defined in part by its current activity.

- New: The process is being created and its PCB is being initialized.
- Ready: The process is in memory, waiting to be assigned to a processor.
- Running: Instructions are being executed by the CPU.
- Waiting (or Blocked): The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- Terminated: The process has finished execution, and the OS reclaims its resources.

Process State Transition Diagram I



- The state diagram illustrates the valid transitions a process can undergo during its lifetime.
- Only one process can be 'Running' on any processor at any instant, although many may be 'Ready' or 'Waiting'.
- Transitions are managed by the Process Scheduler and OS interrupt handlers.

Process State Transition Diagram II

- The transition from Running to Ready typically occurs due to a timer interrupt (time slice expiration) in preemptive scheduling.
- The transition from Waiting to Ready occurs asynchronously when the requested resource becomes available or the I/O operation completes.

Process Control Block (PCB) I

Each process is represented in the operating system by a Process Control Block, also called a task control block.

- The PCB serves as the repository for any information that may vary from process to process.
- It contains all the necessary data for the OS to start, pause, resume, and terminate the process.
- PCBs are typically stored in a kernel data structure like a linked list or hash table for efficient access by the scheduler.
- When a process is not executing, its hardware state (registers, PC) must be safely stored in the PCB.
- The size and complexity of a PCB depend heavily on the specific operating system implementation.

Key Components of the PCB I

The PCB groups together various pieces of information vital for process management.

- Process State: Current state (new, ready, running, waiting, halted).
- Program Counter (PC): The address of the next instruction to be executed for this process.
- CPU Registers: Accumulators, index registers, stack pointers, and general-purpose registers whose states must be saved upon interrupts.
- CPU-Scheduling Information: Process priority, pointers to scheduling queues, and other scheduling parameters.
- Memory-Management Information: Value of base and limit registers, page tables, or segment tables depending on the memory system used.

Extended PCB Information & Resource Tracking I

Beyond CPU and memory state, the PCB tracks I/O and accounting statistics.

- Accounting Information: Amount of CPU and real time used, time limits, account numbers, job or process numbers.
- I/O Status Information: The list of I/O devices allocated to the process, a list of open files, etc.
- Process Identifier (PID): A unique integer assigned to the process upon creation for identification by the OS and other processes.
- Parent/Child Relationships: Pointers linking the process to its creator (parent) and its spawned sub-processes (children).
- Signal Handling: Information regarding which signals the process is ignoring, catching, or blocking.

Context Switching and the PCB I

Context switching is the mechanism of saving the state of the old process and loading the saved state for the new process.

- When an interrupt occurs, the system saves the current context of the process currently running on the CPU into its PCB.
- The OS scheduler then selects another process from the ready queue and loads its saved context from its PCB into the CPU registers.
- Context-switch time is pure overhead because the system does no useful work while switching.
- The speed of context switching varies from machine to machine, depending on the memory speed, the number of registers, and the presence of special instructions.

Context Switching and the PCB II

- Hardware support, such as multiple sets of registers, can significantly reduce the overhead of context switching by simply changing a pointer to the active register set.

Title Slide I

Course: Fundamentals Operating System

- Unit: Concepts of Process
- Topic: Overview of the Process & Threads, Process States, PCB

Introduction: What is a Process? I

A process is fundamentally an instance of a computer program that is being executed.

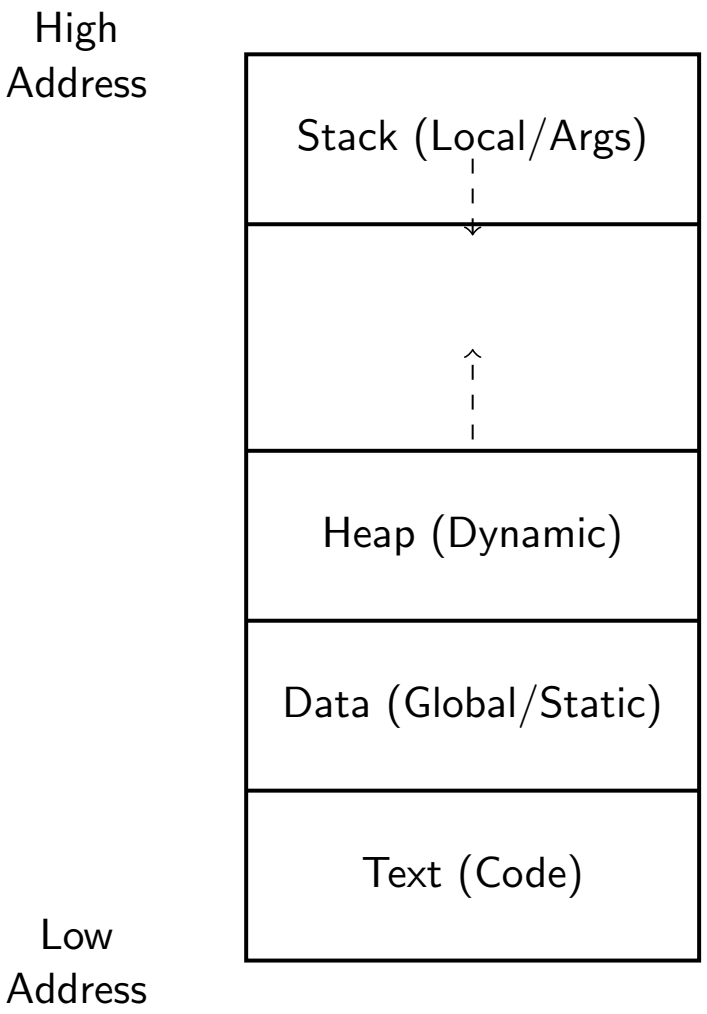
- It contains the program code and its current activity.
- A process is an 'active' entity, unlike a program which is a 'passive' entity residing on disk.
- Depending on the OS, a process may be made up of multiple threads of execution that execute instructions concurrently.
- The OS is responsible for managing processes, including their creation, scheduling, and termination.

Memory Layout of a Process I

When a program is loaded into memory and becomes a process, it is organized into several segments.

- Text Section: The executable instructions of the compiled program.
- Data Section: Contains global and static variables initialized by the programmer.
- Heap Section: Memory dynamically allocated during run time (e.g., using malloc or new).
- Stack Section: Temporary data such as function parameters, return addresses, and local variables.
- The Stack and Heap grow towards each other to maximize memory utilization without overlapping.

Diagram: Process Memory Layout I



- Visual representation of Text, Data, Heap, and Stack segments.
- Note the dynamic growth directions of the Heap and Stack.

Overview of Threads I

A thread is the basic unit of CPU utilization and constitutes the smallest sequence of programmed instructions managed by a scheduler.

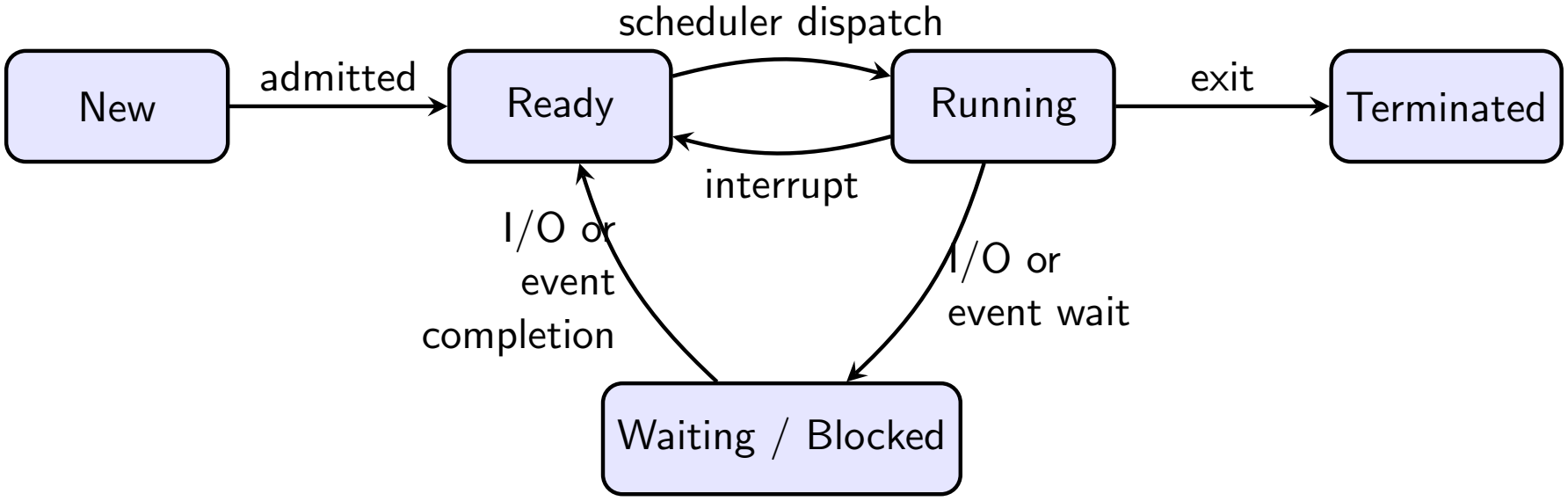
- A thread comprises a thread ID, a program counter (PC), a register set, and a stack.
- It shares with other threads belonging to the same process its code section, data section, and OS resources.
- Multithreading allows a single process to perform multiple tasks concurrently, improving application responsiveness.
- Threads are 'lighter' than processes: context switching between threads of the same process is faster because they share the same memory space.

Process Life Cycle and States I

As a process executes, it changes state. The state indicates the current activity of that process.

- New: The process is being created and resources are being allocated.
- Ready: The process has all necessary resources and is waiting to be assigned to a processor.
- Running: Instructions of the process are being executed by the CPU.
- Waiting/Blocked: The process is waiting for some event to occur (e.g., an I/O completion or reception of a signal).
- Terminated: The process has finished execution and the OS reclaims its resources.

Diagram: Process State Transitions I



- State transition diagram illustrating the life cycle of a standard process.
- Shows transitions like context switches triggered by interrupts and I/O requests.

Process Control Block (PCB) I

The OS maintains a data structure for every process, known as the Process Control Block (PCB) or Task Control Block.

- The PCB serves as the repository for any information that may vary from process to process.
- It enables the OS to manage the concurrent execution of multiple processes.
- When a context switch occurs, the CPU's current state is saved into the PCB of the outgoing process, and state is restored from the PCB of the incoming process.
- PCBs are typically stored in kernel memory, protected from user access, often in a linked list for quick traversal by the scheduler.

Key Components of a PCB I

A standard PCB contains essential metadata required for process management and scheduling.

- Process State: Current state (New, Ready, Running, Waiting, Terminated).
- Program Counter: Address of the next instruction to be executed.
- CPU Registers: General-purpose registers, stack pointers, index registers, and condition codes.
- CPU-Scheduling Information: Process priority, pointers to scheduling queues.
- Memory-Management Information: Value of base and limit registers, page tables, or segment tables.
- Accounting & I/O Status: CPU time used, time limits, list of open files, I/O devices allocated.

Summary & Context Switching I

Context switching ties the concepts of states and PCBs together to allow multitasking.

- Context Switching is the process of saving the state (PCB) of the old process and loading the saved state (PCB) of the new process.
- Context-switch time is pure overhead; the system does no useful work while switching.
- Hardware support (like multiple register sets) can significantly reduce context switch overhead.
- Understanding processes, their memory layout, and the PCB is foundational to grasping how an OS manages concurrency and system resources.

Title Slide I

Course: Fundamentals Operating System

- Unit: Concepts of Process
- Lecture 9: Overview of the Process & threads, Process Life Cycle/ Process States, Process Control Block (PCB)

Overview of a Process I

A process is an instance of an executing program, encompassing the program code and its current activity.

- A process is more than just the program code (often called the text section).
- It includes the current activity represented by the value of the program counter and the contents of the processor's registers.
- It includes the process stack, which contains temporary data such as function parameters, return addresses, and local variables.
- It includes a data section containing global variables.
- It may also include a heap, which is memory dynamically allocated during process run time.

Process vs. Program I

Understanding the fundamental distinction between a passive program and an active process.

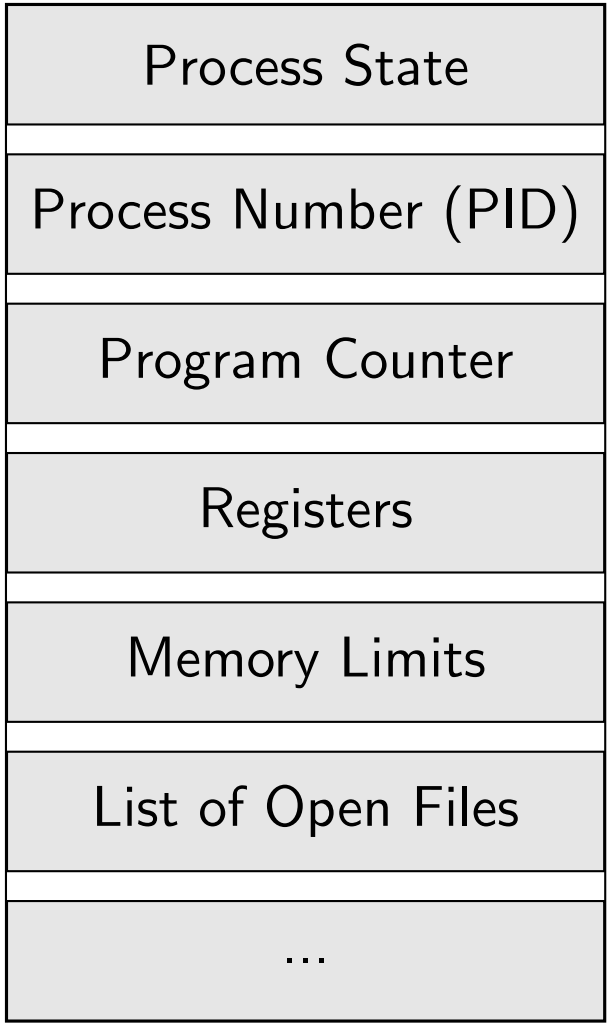
- A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file).
- A process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory.
- Two common techniques for loading are double-clicking an icon or entering the name of the executable on the command line.
- Two processes associated with the same program are considered two separate execution sequences.

Threads vs. Processes I

A thread is a basic unit of CPU utilization, comprising a thread ID, a program counter, a register set, and a stack.

- Traditional (heavyweight) processes have a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.
- Threads belonging to the same process share its code section, data section, and other operating-system resources, such as open files and signals.
- Processes are generally independent, whereas threads exist as subsets of a process.
- Context switching between threads of the same process is faster and less resource-intensive than context switching between processes.
- Multithreading models include Many-to-One, One-to-One, and Many-to-Many.

Process Control Block (PCB) Diagram I



- Each process is represented in the operating system by a Process Control Block (PCB).
- It is also known as a task control block.

Process Control Block (PCB) Diagram II

- The PCB serves as the repository for any information that may vary from process to process.
- It is implemented as a data structure in the operating system kernel.

PCB Components in Detail I

The PCB contains crucial information necessary for the OS to manage and control the process.

- Process State: The state may be new, ready, running, waiting, halted, and so on.
- Program Counter: The counter indicates the address of the next instruction to be executed for this process.
- CPU Registers: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers.
- CPU-Scheduling Information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

PCB Components in Detail II

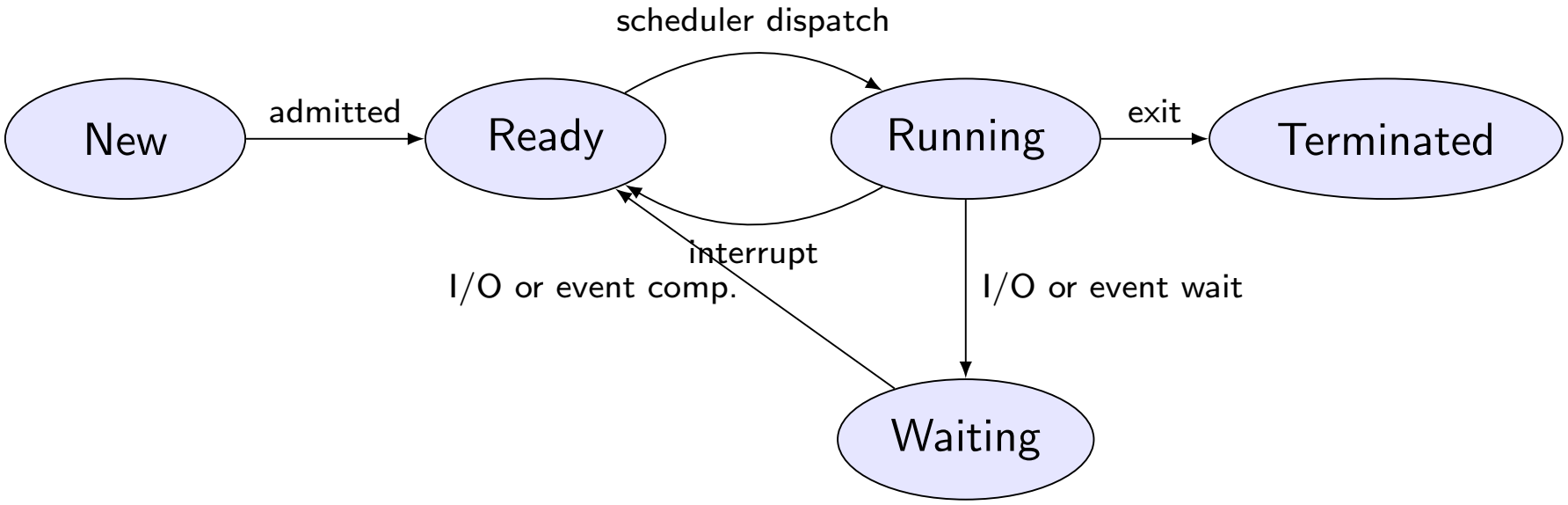
- Memory-Management Information: This information may include such items as the value of the base and limit registers and the page tables, or the segment tables.
- Accounting Information: Includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

Process Life Cycle / Process States I

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.

- New: The process is being created.
- Running: Instructions are being executed.
- Waiting (or Blocked): The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- Ready: The process is waiting to be assigned to a processor.
- Terminated: The process has finished execution.

Process State Transition Diagram I



- Many processes may be in the ready and waiting states at the same time.
- Only one process can be running on any processor at any instant.
- State transitions are triggered by events like interrupts, I/O requests, or the scheduler dispatching a process.
- A process in the 'New' state transitions to 'Ready' once it is admitted to the system.

Process State Transition Diagram II

- The 'Waiting' state prevents a process from utilizing CPU cycles while waiting for an external event.

Context Switching I

When the CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch.

- Context switch time is pure overhead, because the system does no useful work while switching.
- Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions.
- The context of a process is represented in the PCB of the process.
- It includes the value of the CPU registers, the process state, and memory-management information.
- Advanced operating systems employ hardware support (like multiple sets of registers) to accelerate context switching.

Summary and Conclusion I

Overview of processes, threads, PCB, and the process lifecycle.

- A process is a program in execution, having a dynamic lifecycle characterized by state transitions.
- The Process Control Block (PCB) is the OS's data structure for representing and managing a process.
- Threads provide a mechanism for multiple streams of execution within the same process environment, sharing resources.
- The process life cycle involves states like New, Ready, Running, Waiting, and Terminated.
- Context switching is an essential operation for multitasking, allowing the CPU to multiplex between different processes.

Title Slide I

Course: Fundamentals Operating System

- Unit: Concepts of Process

CPU-I/O Burst Cycle I

Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states.

- Execution begins with a CPU burst.
- That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution.
- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.

Preemptive vs Non-Preemptive Scheduling I

CPU scheduling decisions may take place when a process switches states.

- Non-preemptive scheduling: Once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
- Preemptive scheduling: The OS can interrupt a currently executing process and allocate the CPU to another process.
- Windows, macOS, and Linux all use preemptive scheduling algorithms.
- Preemptive scheduling requires special hardware (e.g., a timer) to ensure that a process does not monopolize the CPU.

Scheduling Criteria I

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.

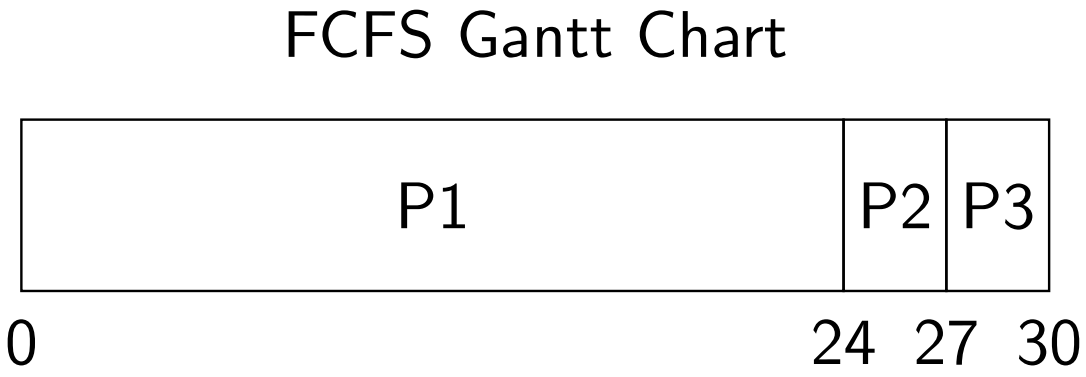
- CPU utilization: Keep the CPU as busy as possible (from 0% to 100%).
- Throughput: The number of processes that are completed per time unit.
- Turnaround time: The interval from the time of submission of a process to the time of completion.
- Waiting time: The sum of the periods spent waiting in the ready queue.
- Response time: The time from the submission of a request until the first response is produced.

First-Come, First-Served (FCFS) Scheduling I

The simplest CPU-scheduling algorithm where the process that requests the CPU first is allocated the CPU first.

- Implemented easily with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- FCFS is a non-preemptive scheduling algorithm.
- The average waiting time under the FCFS policy is often quite long.
- Subject to the 'convoy effect' where all other processes wait for one big process to get off the CPU.

FCFS Gantt Chart Diagram I



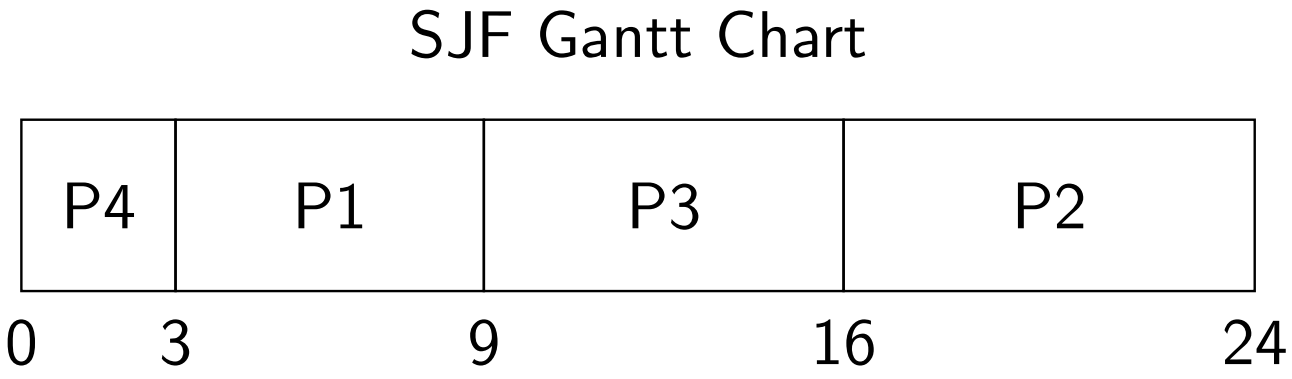
- Consider three processes: P1 with burst time 24, P2 with burst time 3, P3 with burst time 3.
- If they arrive in order P1, P2, P3, the Gantt chart shows P1 running first.
- Waiting times: P1 = 0, P2 = 24, P3 = 27.
- Average waiting time: (0 + 24 + 27) / 3 = 17 milliseconds.
- This diagram visualizes the non-preemptive sequential execution.

Shortest-Job-First (SJF) Scheduling I

Associates with each process the length of its next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst.

- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- SJF algorithm is provably optimal in that it gives the minimum average waiting time for a given set of processes.
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request.
- Often used in long-term scheduling where users specify a time limit when they submit jobs.
- Can be either preemptive (Shortest Remaining Time First) or non-preemptive.

SJF Gantt Chart Diagram I



- Consider processes with burst times: P1=6, P2=8, P3=7, P4=3.
- The order of execution based on shortest burst is P4, P1, P3, P2.
- Waiting times: P4 = 0, P1 = 3, P3 = 9, P2 = 16.
- Average waiting time: (0 + 3 + 9 + 16) / 4 = 7 milliseconds.
- Compared to FCFS, SJF significantly reduces the average waiting time.

Priority Scheduling (Non-Preemptive) I

A priority is associated with each process, and the CPU is allocated to the process with the highest priority (usually denoted by the smallest integer).

- SJF is a special case of general priority scheduling where priority is the inverse of the predicted next CPU burst.
- In a non-preemptive priority algorithm, once a process gets the CPU, it runs to completion of its burst.
- Major problem: Indefinite blocking (starvation), where low-priority processes may wait indefinitely.
- Solution to starvation: Aging, a technique of gradually increasing the priority of processes that wait in the system for a long time.
- Priorities can be defined internally (e.g., memory limits) or externally (e.g., importance to the user).

Round Robin (RR) Scheduling I

Designed especially for time-sharing systems, similar to FCFS but preemption is added to switch between processes.

- A small unit of time, called a time quantum or time slice, is defined (typically 10 to 100 ms).
- The ready queue is treated as a circular queue. The scheduler goes around allocating the CPU to each process for a time interval of up to one time quantum.
- If a process has a burst less than the quantum, it releases the CPU voluntarily.
- If the burst is longer than the quantum, a timer goes off, causes an interrupt, and the process is preempted and put at the tail of the ready queue.
- The performance of the RR algorithm depends heavily on the size of the time quantum.

Title Slide I

Course: Fundamentals Operating System

- Unit: Concepts of Process
- Topic: Process Scheduling: Scheduling Criteria
- Topic: Scheduling Algorithms: First Come First Serve, Shortest Job First, Round Robin, Non-Preemptive Priority based Scheduling

Introduction to Process Scheduling I

Process scheduling is the core of multiprogrammed operating systems, designed to maximize CPU utilization by switching the CPU among processes.

- Objective: Have some process running at all times to maximize CPU utilization.
- The CPU scheduler selects a process from the ready queue and allocates the CPU to it.
- A ready queue contains all processes residing in main memory that are ready and waiting to execute.
- Dispatcher: The module that gives control of the CPU to the process selected by the short-term scheduler.
- Dispatch latency is the time it takes for the dispatcher to stop one process and start another running.

Scheduling Criteria I

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.

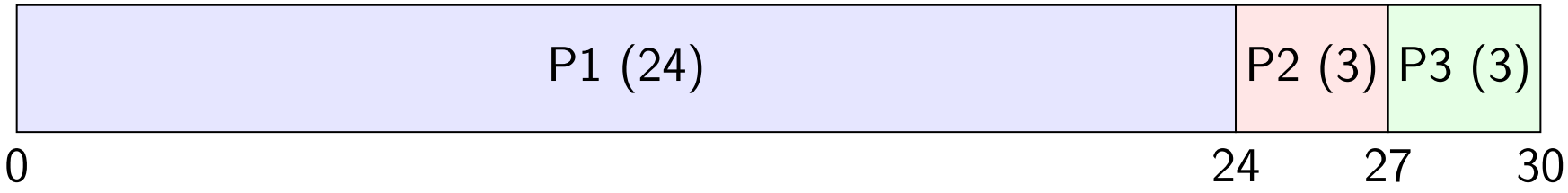
- CPU Utilization: Keep the CPU as busy as possible (ranging from 0% to 100%, conceptually 40% to 90% in real systems).
- Throughput: The number of processes that are completed per time unit.
- Turnaround Time: The interval from the time of submission of a process to the time of completion.
- Waiting Time: The sum of the periods spent waiting in the ready queue.
- Response Time: The time from the submission of a request until the first response is produced.

First-Come, First-Served (FCFS) Scheduling I

The simplest CPU-scheduling algorithm where the process that requests the CPU first is allocated the CPU first.

- Implementation is straightforward and managed using a FIFO (First-In, First-Out) queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- Non-preemptive algorithm: Once the CPU has been allocated to a process, it keeps the CPU until it releases it.
- Drawback: Average waiting time is often quite long and highly sensitive to the order of process arrival.
- Convoy Effect: Short processes waiting behind a long process, leading to lower CPU and device utilization.

FCFS Gantt Chart Example I



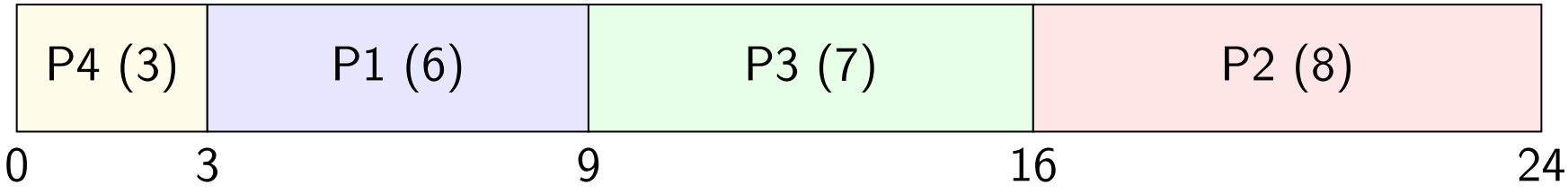
- Assume processes arrive at time 0 in the order: P1(burst=24), P2(burst=3), P3(burst=3).
- Waiting times: P1=0, P2=24, P3=27.
- Average waiting time: $(0 + 24 + 27) / 3 = 17$.
- If they arrived in order P2, P3, P1, the average waiting time would be $(0 + 3 + 6) / 3 = 3$.
- This clearly demonstrates the convoy effect where process execution order drastically changes waiting times.

Shortest-Job-First (SJF) Scheduling I

This algorithm associates with each process the length of the process's next CPU burst. The CPU is assigned to the process with the smallest next CPU burst.

- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- SJF is provably optimal in that it gives the minimum average waiting time for a given set of processes.
- Difficulty: Knowing the length of the next CPU request in short-term scheduling.
- Prediction: The next CPU burst is generally predicted as an exponential average of measured lengths of previous CPU bursts.

SJF Gantt Chart Example I



- Processes arrive at time 0: P1(6), P2(8), P3(7), P4(3).
- Execution order is based on shortest burst: P4, P1, P3, P2.
- Waiting times: P4=0, P1=3, P3=9, P2=16.
- Average waiting time: $(0 + 3 + 9 + 16) / 4 = 7$.
- By moving shorter processes before longer ones, SJF decreases waiting time for short processes more than it increases waiting time for long processes.

Round-Robin (RR) Scheduling I

The Round-Robin algorithm is similar to FCFS, but preemption is added to switch between processes. It is designed especially for time-sharing systems.

- A small unit of time, called a time quantum or time slice, is defined (typically 10 to 100 milliseconds).
- The ready queue is treated as a circular queue. The scheduler goes around the ready queue, allocating the CPU to each process for up to 1 time quantum.
- If a process's CPU burst exceeds 1 time quantum, the timer will go off, causing an interrupt to the OS, and a context switch will be executed.
- The interrupted process is put at the tail of the ready queue.
- Performance depends heavily on the size of the time quantum (should be large with respect to context switch time).

Non-Preemptive Priority Scheduling I

A priority is associated with each process, and the CPU is allocated to the process with the highest priority.

- Equal-priority processes are scheduled in FCFS order.
- SJF is a special case of priority scheduling where priority is the inverse of the predicted next CPU burst.
- Priorities can be defined internally (e.g., time limits, memory requirements, open files) or externally (e.g., importance of process, user paying for it).
- Non-preemptive: A newly arrived high-priority process waits at the head of the ready queue until the currently running process finishes its CPU burst.
- Starvation (Indefinite Blocking): Low-priority processes may wait indefinitely. Solution: Aging (gradually increasing the priority of waiting processes).

Comparison of Scheduling Algorithms I

The selection of a scheduling algorithm fundamentally affects system performance, fairness, and responsiveness.

- FCFS: Simple, non-preemptive, suffers from convoy effect, suitable for batch systems.
- SJF: Optimal average waiting time, cannot be implemented exactly at the CPU scheduling level, requires burst time prediction.
- Round-Robin: Excellent response time for interactive systems, high context switching overhead if the time quantum is too small.
- Priority Scheduling: Allows prioritization of critical tasks, but suffers from starvation which necessitates aging.
- Modern OSs use multilevel feedback queue scheduling which combines the advantages of these basic algorithms.

Title Slide I

Course: Fundamentals Operating System

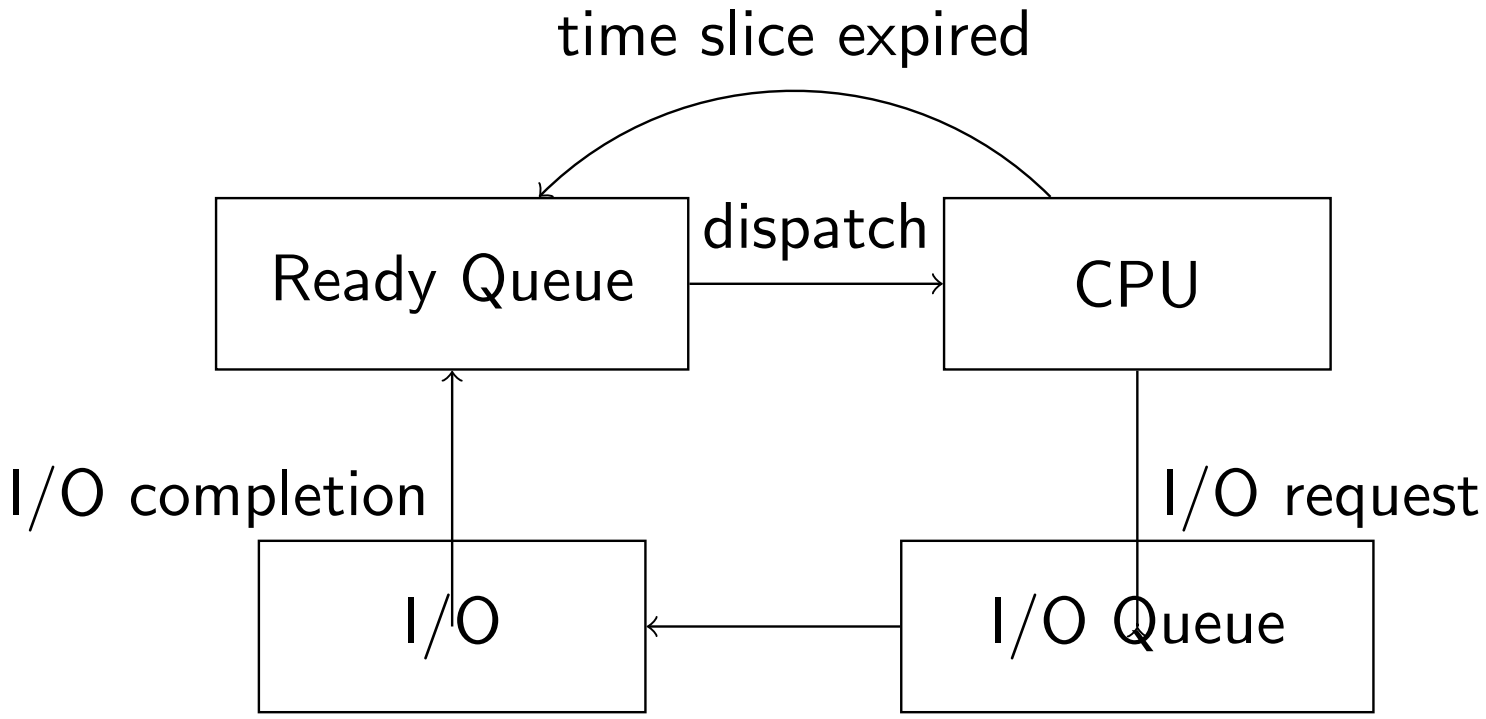
- Unit: Concepts of Process

Introduction to Process Scheduling I

Process scheduling is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

- Multiprogramming aims to maximize CPU utilization.
- Time sharing aims to switch the CPU among processes so frequently that users can interact with each program while it is running.
- The process scheduler selects an available process for program execution on the CPU.

Scheduling Queues I



- Job Queue: Set of all processes in the system.
- Ready Queue: Set of all processes residing in main memory, ready and waiting to execute.
- Device Queues: Set of processes waiting for an I/O device.

Scheduling Criteria (Performance Metrics) I

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. We need criteria to compare these algorithms.

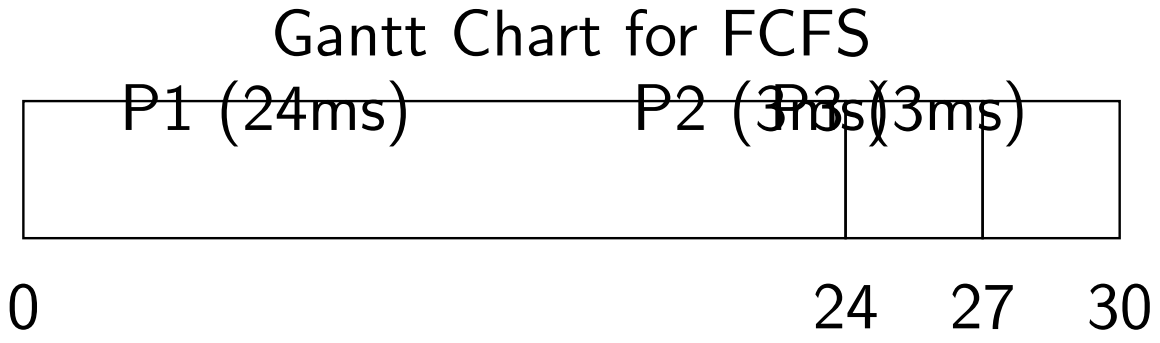
- CPU utilization: Keep the CPU as busy as possible (0-100%).
- Throughput: Number of processes that complete their execution per time unit.
- Turnaround time: Amount of time to execute a particular process (submission to completion).
- Waiting time: Amount of time a process has been waiting in the ready queue.
- Response time: Amount of time it takes from when a request was submitted until the first response is produced.

Optimization Goals for Scheduling I

A good scheduling algorithm should optimize the following performance metrics to ensure efficient system operation and fair resource allocation.

- Maximize CPU utilization and Throughput.
- Minimize Turnaround time.
- Minimize Waiting time.
- Minimize Response time.
- In most cases, we optimize the average measure. However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average.

First-Come, First-Served (FCFS) Scheduling I



- Simplest CPU-scheduling algorithm.
- The process that requests the CPU first is allocated the CPU first.
- Implementation is easily managed with a FIFO queue.
- Non-preemptive algorithm.
- Suffers from the Convoy Effect: short process behind long process.

Shortest-Job-First (SJF) Scheduling I

Associates with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

- SJF is optimal: gives minimum average waiting time for a given set of processes.
- The difficulty is knowing the length of the next CPU request.
- Can be preemptive (Shortest-Remaining-Time-First) or non-preemptive.
- Preemptive SJF will preempt the currently executing process if a new process arrives with a CPU burst length less than what is remaining of the current process.
- May cause starvation for long processes.

Predicting Next CPU Burst in SJF I

Since we cannot know the exact length of the next CPU burst, we must predict it. We expect that the next burst will be similar in length to the previous ones.

- The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts.
- Formula: $\{}tau_{n+1} = \{}alpha t_n + (1 - \{}alpha) \{}tau_n$.
- t_n = actual length of n^{th} CPU burst.
- $\{}tau_n$ = past predicted value for the n^{th} CPU burst.
- $\{}alpha$ controls the relative weight of recent and past history (typically 0.5).

Round Robin (RR) Scheduling I

Designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes.

- Each process gets a small unit of CPU time (time quantum or time slice), usually 10-100 milliseconds.
- After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once.
- Performance depends heavily on the size of the time quantum.
- If q is too large, RR approaches FCFS. If q is too small, context switch overhead becomes too high.

Non-Preemptive Priority Scheduling I

A priority number (integer) is associated with each process. The CPU is allocated to the process with the highest priority (usually smallest integer $\{}equiv$ highest priority).

- SJF is a special case of general priority scheduling where priority is the inverse of predicted next CPU burst time.
- In non-preemptive priority scheduling, once the CPU is given to the process, it cannot be preempted until it completes its CPU burst.
- Problem: Starvation (indefinite blocking) - low priority processes may never execute.
- Solution: Aging - as time progresses increase the priority of the process.

Title Slide I

Course: Fundamentals Operating System

- Unit: Concepts of Process
- Lecture 13: Schedulers, Scheduling Queues, and Context Switching

Introduction to Process Scheduling I

The objective of multiprogramming is to have some process running at all times to maximize CPU utilization. The objective of time-sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

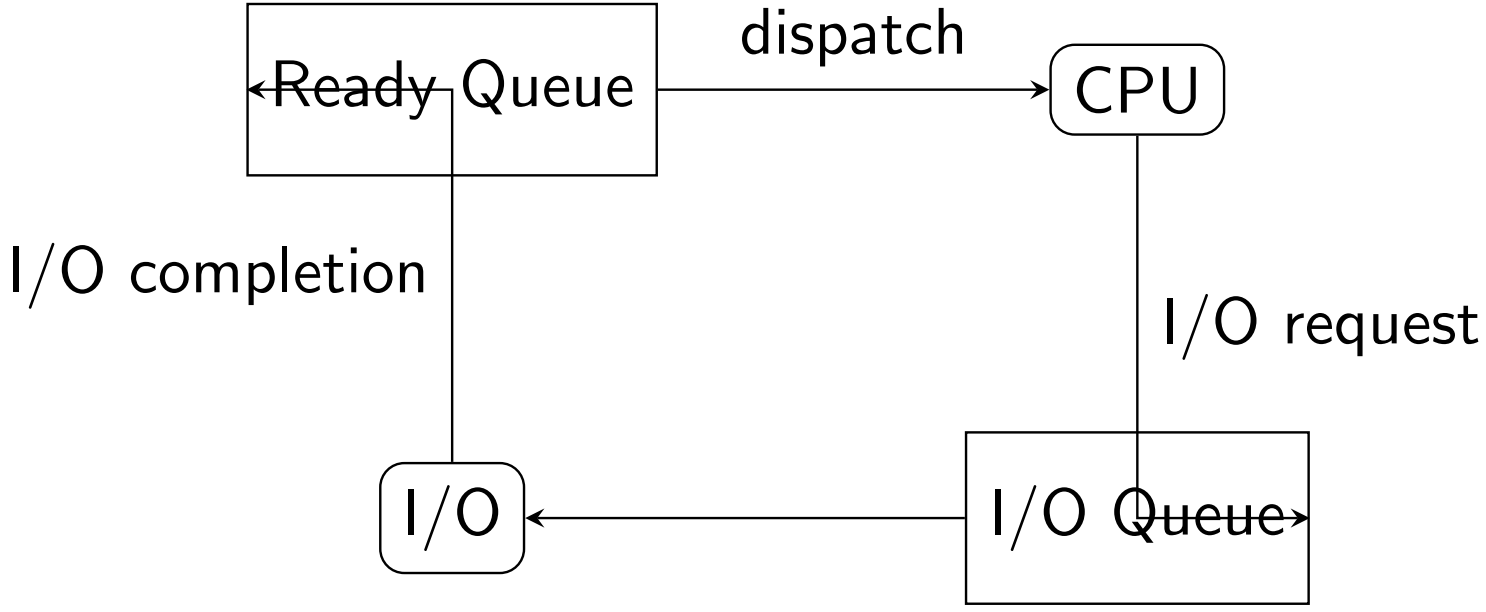
- A process migrates among the various scheduling queues throughout its lifetime.
- The OS must select, for scheduling purposes, processes from these queues in some fashion.
- The selection process is carried out by an appropriate scheduling algorithm and scheduler component.

Scheduling Queues I

As processes enter the system, they are put into various queues to manage their execution and resource usage effectively.

- Job Queue: As processes enter the system, they are put into a job queue, which consists of all processes in the system.
- Ready Queue: Processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue.
- Device Queues: The list of processes waiting for a particular I/O device is called a device queue. Each device has its own queue.

Diagram: Process Queuing Representation I



- A new process is initially put in the Ready queue.
- It waits there until it is selected for execution, or dispatched.
- Once the process is allocated the CPU and is executing, it could issue an I/O request and then be placed in an I/O queue.

Long-Term Scheduler (Job Scheduler) I

The long-term scheduler, or job scheduler, selects processes from the spool (mass-storage device) and loads them into memory for execution.

- It executes much less frequently than other schedulers; minutes may separate the creation of one new process and the next.
- It controls the degree of multiprogramming (the number of processes in memory).
- It is critical that it makes a careful selection between I/O-bound and CPU-bound processes to maintain a good process mix.

Short-Term Scheduler (CPU Scheduler) I

The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them.

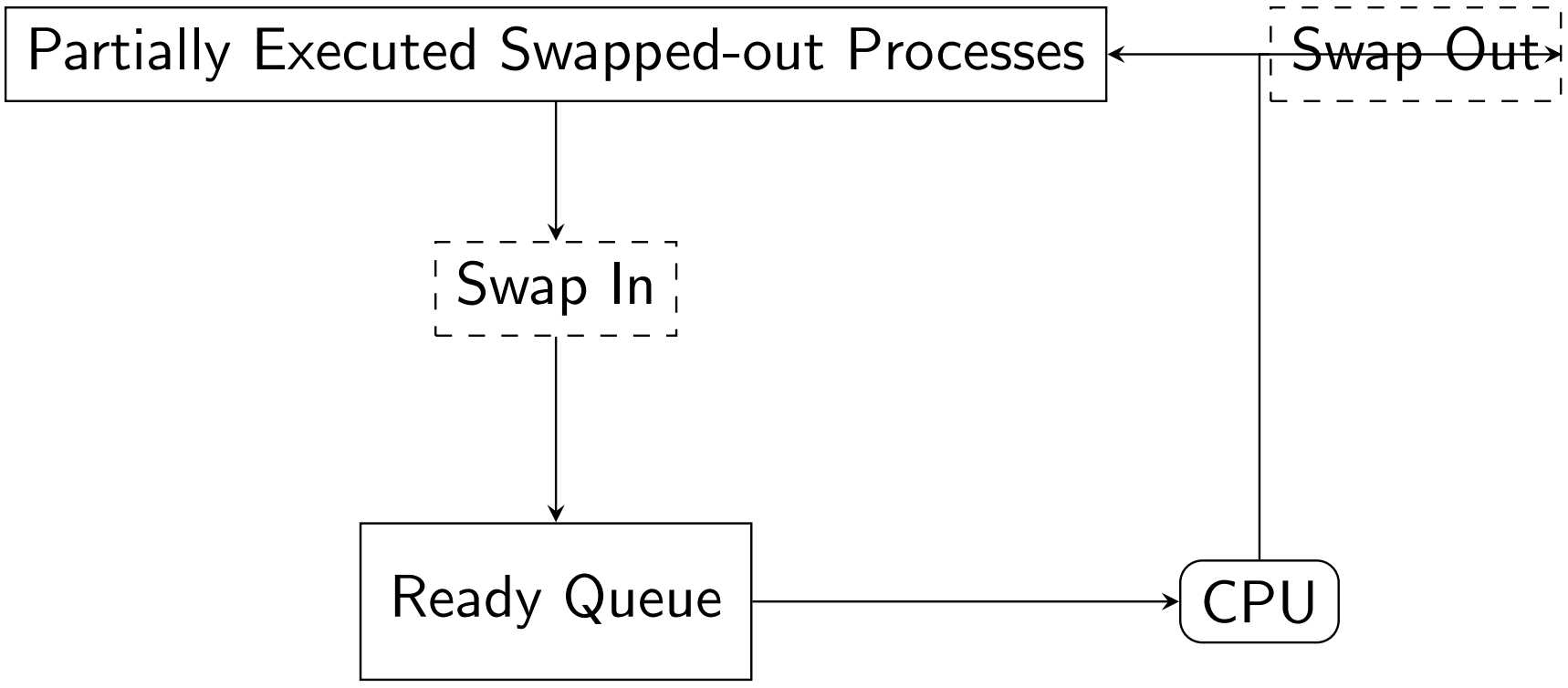
- It executes very frequently (e.g., once every 10 to 100 milliseconds) as processes block for I/O or run out of time quanta.
- Because of the short time between executions, the short-term scheduler must be very fast to minimize latency.
- If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 9% of the CPU is being wasted simply for scheduling.

Medium-Term Scheduler I

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling called the medium-term scheduler.

- Its core function is swapping: removing a process from memory (and from active contention for the CPU) and thus decreasing the degree of multiprogramming.
- Later, the process can be reintroduced into memory, and its execution can be continued where it left off.
- Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory.

Diagram: Medium-Term Scheduler and Swapping I



- Medium-term scheduling is practically implemented as part of the swapping function.
- It temporarily removes processes from main memory and places them on secondary storage (disk).
- It helps manage the degree of multiprogramming and frees up physical memory space for more critical tasks.

Context Switch I

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch.

- The context of a process is represented in the Process Control Block (PCB).
- It includes the value of the CPU registers, the process state, and memory-management information.
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

Overhead of Context Switch I

Context switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions.

- Context-switch times are highly dependent on hardware support.
- Some hardware systems provide multiple sets of registers per CPU, allowing a context switch to simply change a pointer to the current register set.
- The more complex the OS and the more state that must be saved (e.g., advanced memory management), the more work the context switch requires.

Title Slide I

Course: Fundamentals Operating System

- Unit: Concepts of Process
- Lecture 14: Overview of Schedulers
- Topics: Long-term, Short-term, Medium-term Schedulers, Scheduling Queues, Context Switch

Introduction to Process Scheduling I

Process scheduling is a core function of a multiprogramming operating system, enabling multiple processes to share CPU time efficiently.

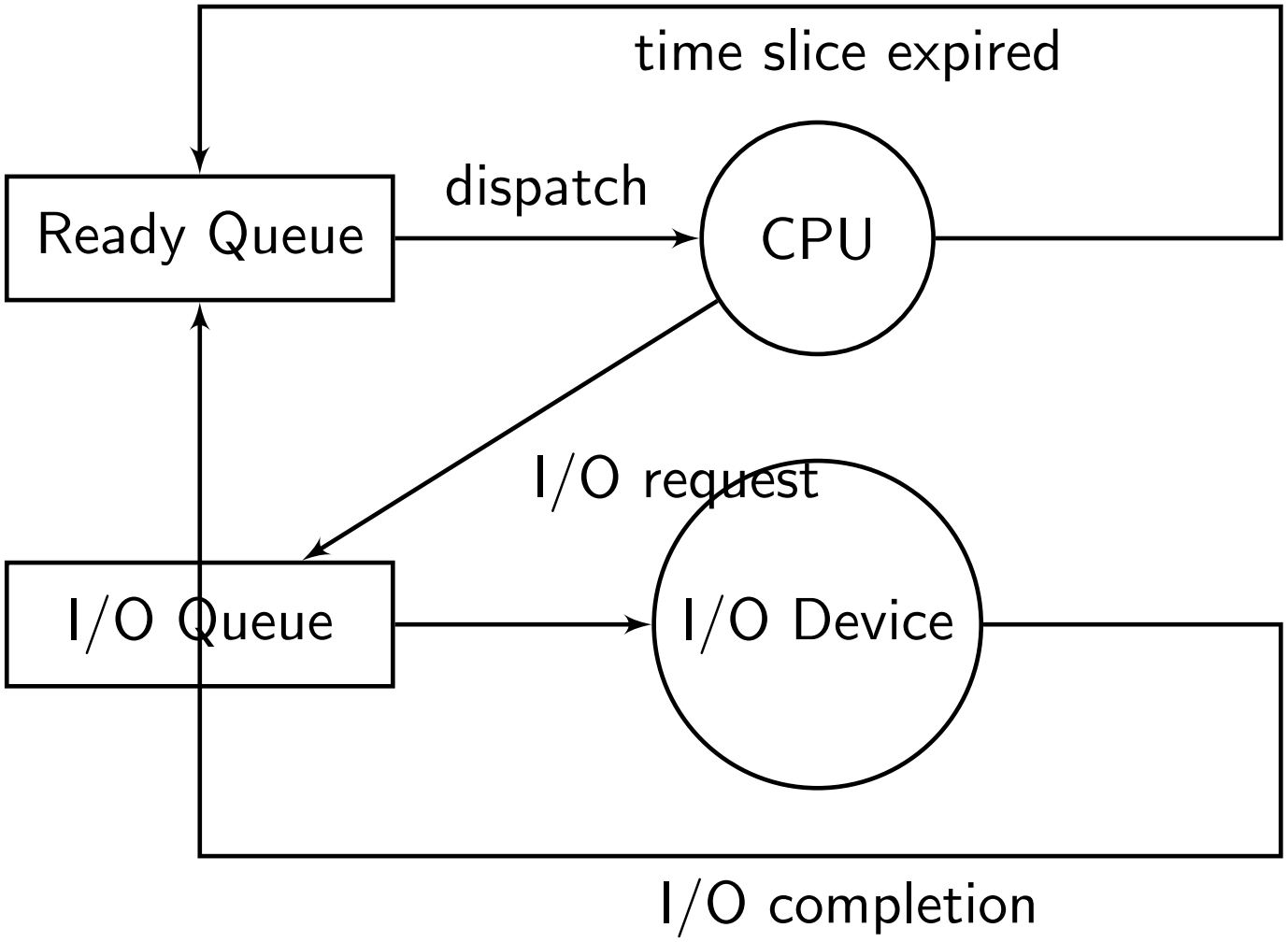
- The primary objective of multiprogramming is to have some process running at all times to maximize CPU utilization.
- Time-sharing systems aim to switch the CPU among processes so rapidly that users can interact with each program while it runs.
- The process scheduler is the OS component responsible for selecting an available process from memory for CPU execution.

Scheduling Queues I

As processes enter the system, they are organized into various scheduling queues based on their current execution state.

- Job Queue: Contains all processes in the system, usually residing on mass storage waiting to be brought into main memory.
- Ready Queue: A list of processes residing in main memory that are ready and waiting to execute on the CPU.
- Device Queue (I/O Queue): A list of processes waiting for a specific I/O device to become available.
- Processes dynamically migrate between these queues as they alternate between CPU bursts and I/O wait times.

Diagram: Scheduling Queues I



- The ready queue acts as the holding area for processes waiting to be dispatched to the CPU.

Diagram: Scheduling Queues II

- A process running on the CPU can be forcefully interrupted due to a time slice expiration (in preemptive scheduling).
- If a process requests an I/O operation, it is moved to the respective I/O queue until the I/O device completes the request.

Long-Term Scheduler (Job Scheduler) I

The long-term scheduler, or job scheduler, dictates which processes are admitted into the system for processing.

- It selects processes from the mass-storage pool and loads them into memory for execution, thereby controlling the degree of multiprogramming.
- It executes infrequently (minutes may separate process creation) because it only acts when a process leaves the system or is newly created.
- Its primary goal is to ensure a balanced mix of CPU-bound and I/O-bound processes to keep system resources optimally utilized.
- If the degree of multiprogramming is too high, system performance degrades; if too low, resources remain idle.

Short-Term Scheduler (CPU Scheduler) I

The short-term scheduler, or CPU scheduler, determines which ready process is allocated the CPU next.

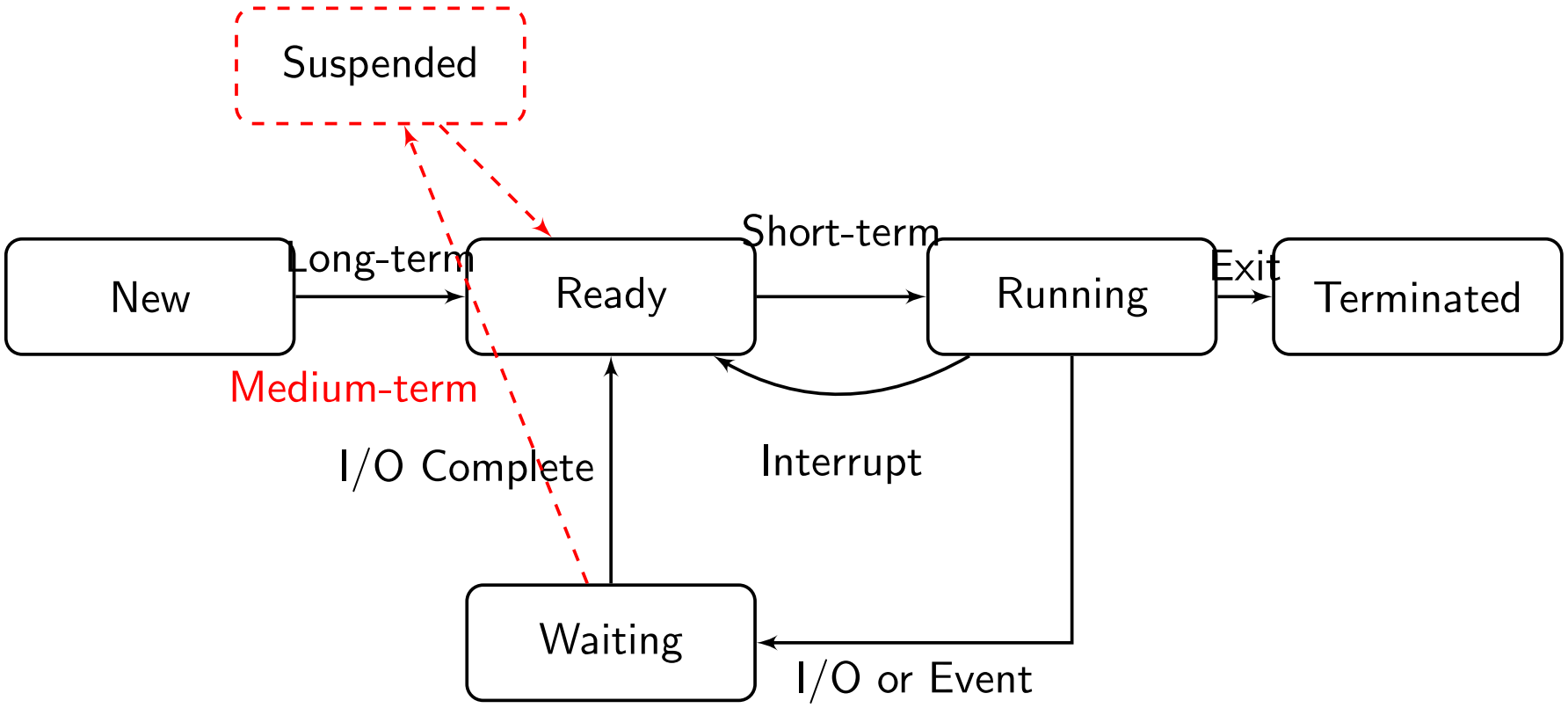
- It constantly selects a process from the ready queue and allocates the CPU to it.
- It executes highly frequently (e.g., every 10 to 100 milliseconds), so it must be exceptionally fast to minimize overhead.
- It is triggered by events such as clock interrupts, I/O completions, and operating system calls.
- A slow short-term scheduler results in excessive context switching overhead, wasting valuable CPU cycles.

Medium-Term Scheduler I

The medium-term scheduler operates in time-sharing systems to adjust the degree of multiprogramming dynamically.

- It temporarily removes processes from main memory (swapping out) and places them on secondary storage to free up memory.
- Later, the process is brought back into memory (swapped in), and its execution resumes from where it left off.
- This is highly useful for mitigating 'thrashing', a condition where the system spends more time swapping than executing.
- It primarily acts on suspended or waiting processes to alleviate memory pressure.

Diagram: Types of Schedulers and Transitions I



- Long-term scheduling manages the transition from the 'New' state to the 'Ready' state.
- Short-term scheduling is responsible for the rapid dispatch cycle between 'Ready' and 'Running'.

Diagram: Types of Schedulers and Transitions II

- Medium-term scheduling manages the transition to and from 'Suspended' states to optimize memory utilization.
- The interaction of these schedulers determines overall process throughput and turnaround time.

Context Switch I

A context switch is the mechanism by which the operating system stops executing one process and begins executing another.

- It involves saving the current state of the old process and loading the saved state of the new process.
- Process state is maintained in a data structure called the Process Control Block (PCB), which contains CPU registers, program counter, and memory limits.
- Context switching time is pure overhead because the system performs no useful computation while switching.
- Hardware support, such as multiple sets of CPU registers, can significantly reduce the time required to perform a context switch.

Summary I

Scheduling mechanisms are vital for balancing system resources and providing a responsive environment.

- Queues (Job, Ready, Device) provide the structural foundation for tracking process states and resource requests.
- The Long-Term Scheduler dictates system workload and multiprogramming levels.
- The Short-Term Scheduler is the fast-acting dispatcher determining immediate CPU allocation.
- The Medium-Term Scheduler controls memory limits via swapping processes out of active contention.
- Context Switches form the mechanical basis of multiprogramming, heavily dependent on efficient OS and hardware design.

Detailed Concept 1 I

Detailed explanation of Overview of Schedulers: Tyes of Schedulers: Long term Schedulers, Short ....

- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 2 I

Detailed explanation of Overview of Schedulers: Tyes of Schedulers: Long term Schedulers, Short ....

- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 3 I

Detailed explanation of Overview of Schedulers: Tyes of Schedulers: Long term Schedulers, Short ....

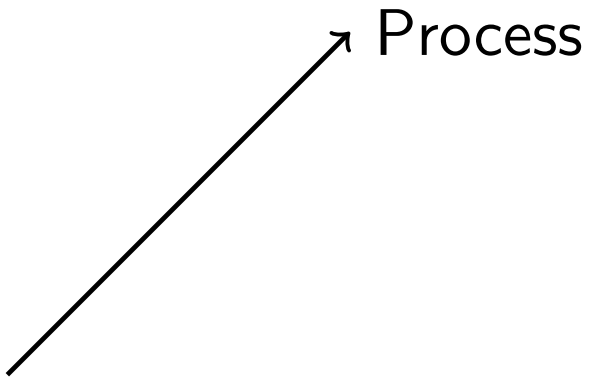
- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 4 I

Detailed explanation of Overview of Schedulers: Tyes of Schedulers: Long term Schedulers, Short ....

- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 5 I



- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 6 I

Detailed explanation of Overview of Schedulers: Tyes of Schedulers: Long term Schedulers, Short ....

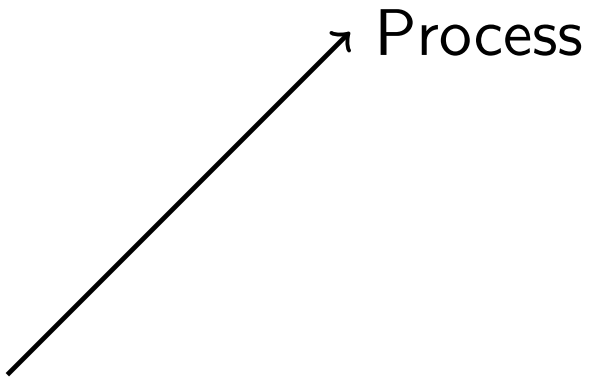
- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 7 I

Detailed explanation of Overview of Schedulers: Tyes of Schedulers: Long term Schedulers, Short ....

- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 8 I



- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 9 I

Detailed explanation of Overview of Schedulers: Tyes of Schedulers: Long term Schedulers, Short ....

- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 10 I

Detailed explanation of Overview of Schedulers: Tyes of Schedulers: Long term Schedulers, Short ....

- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Title Slide I

Course: Fundamentals Operating System

- Unit: Concepts of Process

Introduction to Process Synchronization I

In a multiprogramming environment, multiple processes execute concurrently and often need to share resources. Process Synchronization is the task of coordinating the execution of processes in a way that no two processes can access the same shared data and resources simultaneously, to avoid inconsistencies.

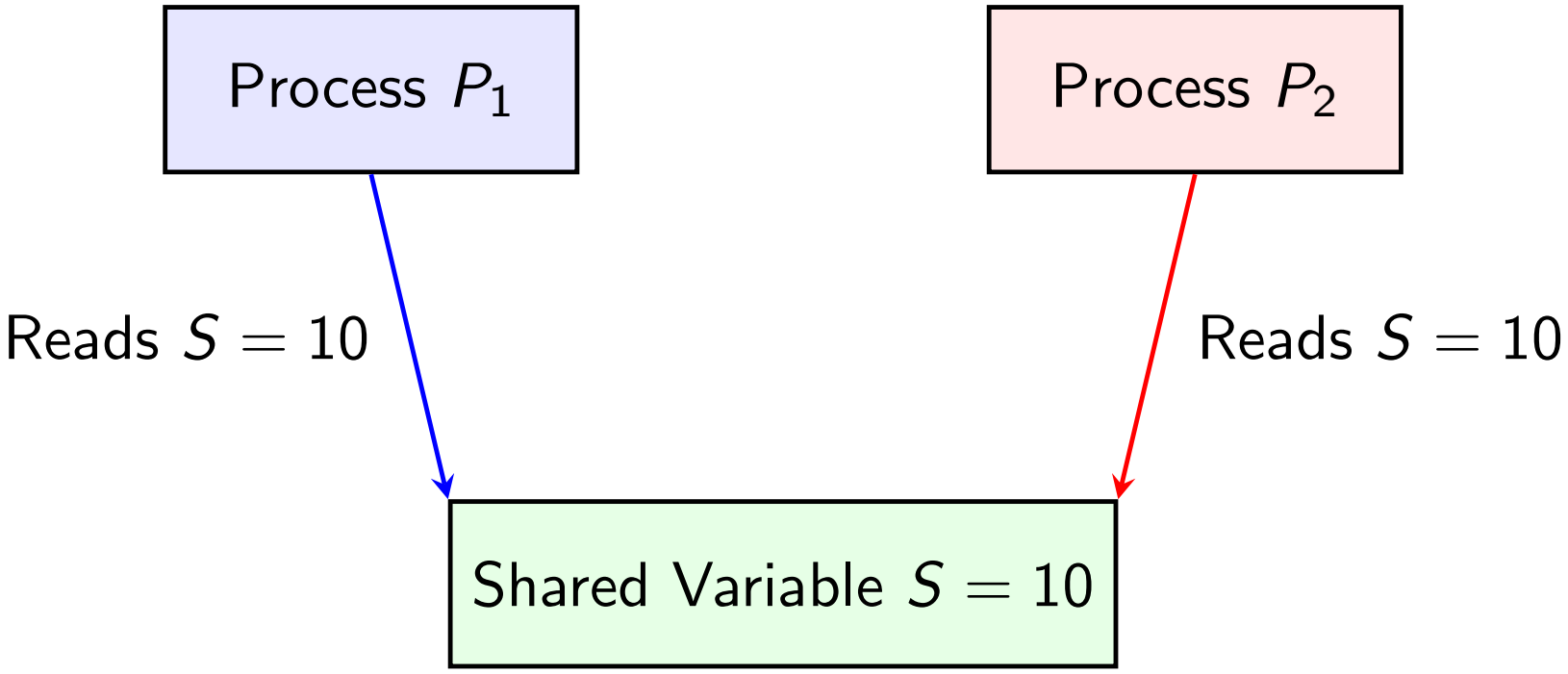
- Processes can execute concurrently or in parallel.
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Independent processes do not affect each other, but cooperating processes share state and need synchronization.

The Race Condition I

A Race Condition occurs when multiple processes or threads read and write shared data concurrently, and the final outcome depends on the exact timing of their execution (the 'race' to finish).

- The outcome of the execution depends on the particular order in which the access takes place.
- Most commonly occurs when multiple processes perform Read-Modify-Write operations on shared variables.
- To prevent race conditions, concurrent processes must be synchronized.
- Only one process should be manipulating the shared variable at any given time.

Race Condition Example I



If both read $S = 10$ simultaneously and increment it by 1, they both write back 11, instead of the correct 12.

This is a Race Condition.

- Two processes P1 and P2 attempt to increment a shared variable S.
- Both read the value 10 into their local registers.

Race Condition Example II

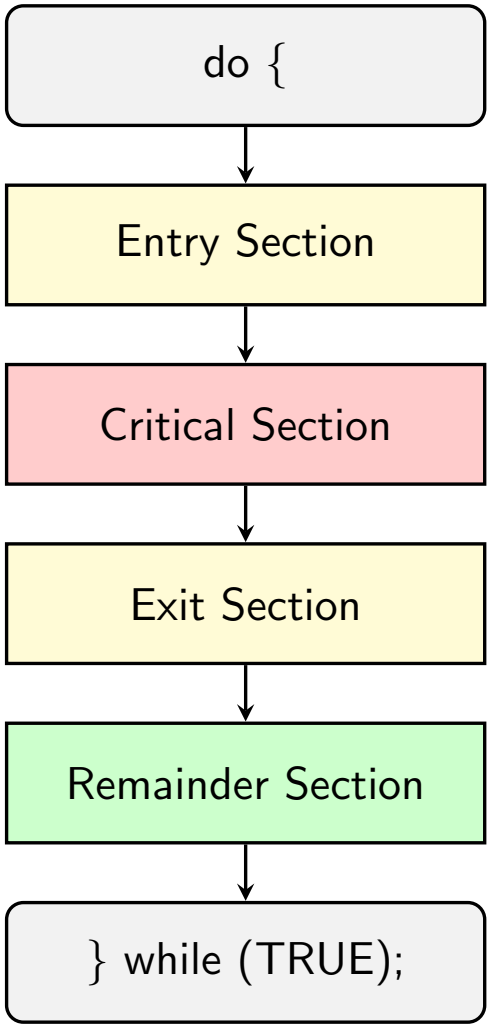
- Both increment their local registers to 11.
- Both write 11 back to memory, losing one increment operation.

The Critical Section Problem I

The Critical Section is a segment of code where a process accesses and modifies shared resources (e.g., variables, files, databases). The critical section problem is to design a protocol that the processes can use to cooperate and ensure that no two processes are in their critical sections at the same time.

- Each process must request permission to enter its critical section.
- The section of code implementing this request is called the Entry Section.
- The critical section may be followed by an Exit Section.
- The remaining code is the Remainder Section.

General Structure of a Process I



- Entry Section: Requests entry, usually implements locking.
- Critical Section: Performs updates on shared memory.

General Structure of a Process II

- Exit Section: Releases locks, allowing others to enter.
- Remainder Section: Non-critical operations on local data.

Requirements for a Solution I

Any valid solution to the Critical Section problem must satisfy three primary requirements: Mutual Exclusion, Progress, and Bounded Waiting. These requirements ensure that processes execute without interfering with one another and without deadlocking or starving.

- Mutual Exclusion: Only one process at a time can be inside the critical section.
- Progress: If no process is in the critical section and some processes wish to enter, the selection cannot be postponed indefinitely.
- Bounded Waiting: There must be a limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter.

Mutual Exclusion Deep Dive I

Mutual Exclusion is the most vital requirement. It guarantees that if process P_i is executing in its critical section, then no other processes can be executing in their critical sections. This isolates the read-modify-write cycle into an atomic unit.

- Prevents race conditions by serializing access to shared variables.
- Can be implemented via hardware mechanisms like disabling interrupts (only on single processors).
- Can be implemented using hardware instructions like TestAndSet or CompareAndSwap.
- Software solutions include Mutex Locks, Semaphores, and Monitors.

Progress and Bounded Waiting I

While Mutual Exclusion prevents safety violations (bad things happening), Progress and Bounded Waiting guarantee liveness (good things eventually happen). They ensure the system does not freeze and processes are fairly treated.

- Progress prevents deadlock: A process in its remainder section cannot participate in deciding who enters next.
- Progress ensures the decision of who enters next is made in finite time.
- Bounded Waiting prevents starvation: A process cannot wait forever while others keep entering the critical section.
- Strict alternation algorithms may satisfy mutual exclusion and bounded waiting, but fail progress.

Summary and Next Steps I

In summary, concurrent processes sharing memory require strict synchronization to avoid race conditions. The critical section problem models this by dividing process execution into specific regions governed by entry and exit protocols.

- Race conditions cause unpredictable data corruption.
- Critical sections must be protected by Mutual Exclusion, Progress, and Bounded Waiting.
- Next Lecture: Peterson's Solution and Hardware Synchronization primitives.
- Next Lecture: Introduction to Mutex Locks and Semaphores.

Title Slide I

Course: Fundamentals Operating System

- Unit: Concepts of Process
- Lecture 17: Process Synchronization

Introduction to Process Synchronization I

Process synchronization refers to the coordination of executing processes to ensure orderly access to shared resources and prevent data inconsistency.

- In a multiprogramming environment, multiple processes can execute concurrently.
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Synchronization is crucial for shared memory, files, and inter-process communication mechanisms.

Cooperating vs. Independent Processes I

Processes in an operating system can be either independent or cooperating. Synchronization is primarily concerned with cooperating processes.

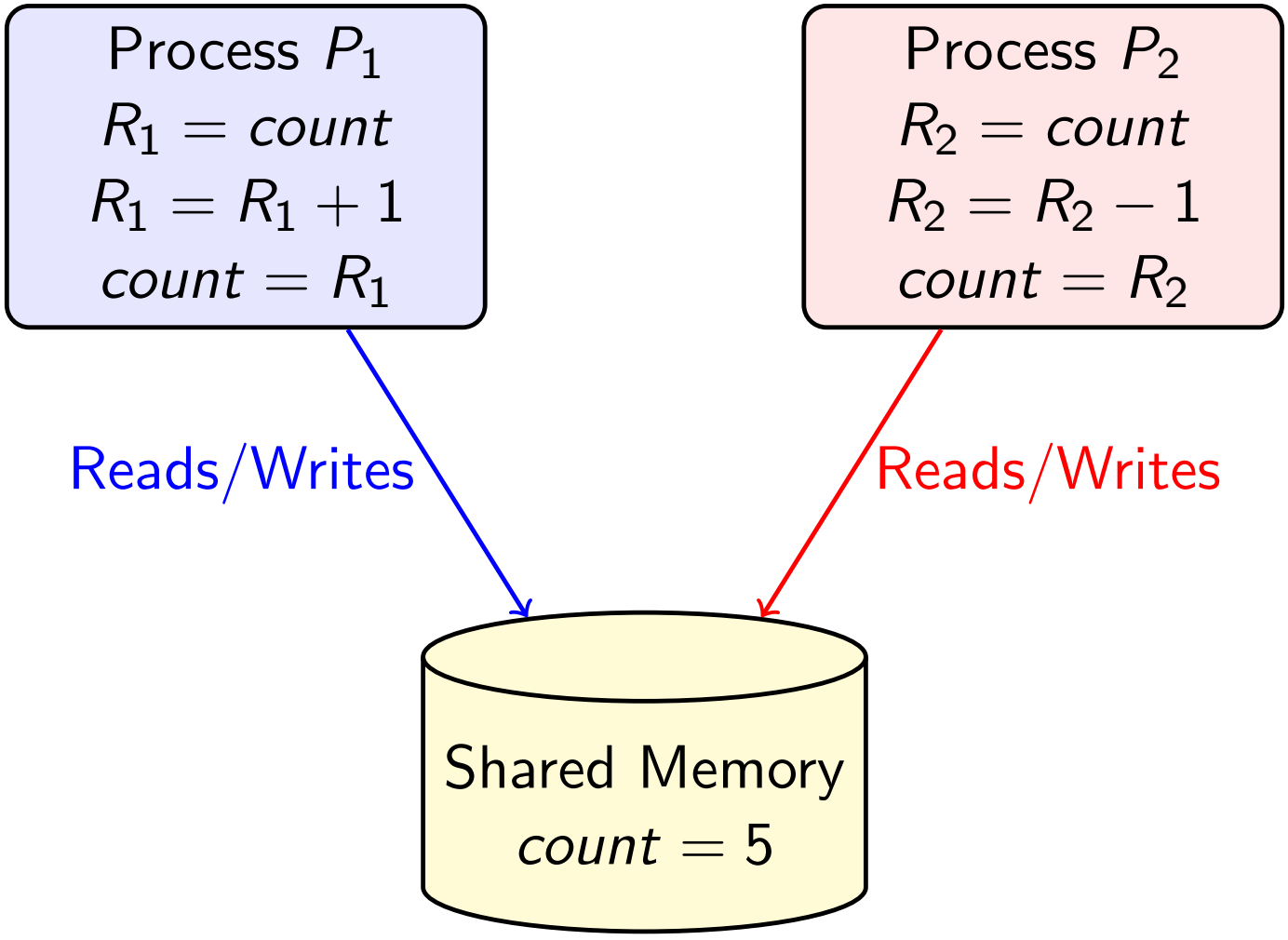
- Independent Process: Cannot affect or be affected by the execution of another process.
- Cooperating Process: Can affect or be affected by other executing processes.
- Reasons for cooperation: Information sharing, computation speedup, modularity, and convenience.
- Cooperating processes share logical address space or data through messages/shared memory.

The Race Condition I

A race condition is a situation where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place.

- Occurs when multiple threads or processes read and write a shared data item.
- The final result depends on the relative timing of their execution (who finishes last).
- To guard against race conditions, concurrent processes must be synchronized.
- A classic example is the producer-consumer problem sharing a 'counter' variable.

Race Condition Example Diagram I



- Process P1 increments the count, while P2 decrements it.
- Both read count=5, P1 calculates 6, P2 calculates 4.

Race Condition Example Diagram II

- If P1 writes first (count=6), then P2 writes (count=4), the result is 4 (incorrect).
- The interleaving of low-level machine instructions causes the race condition.

The Critical Section Problem I

A critical section is a segment of code in which a process may be changing common variables, updating a table, writing a file, etc.

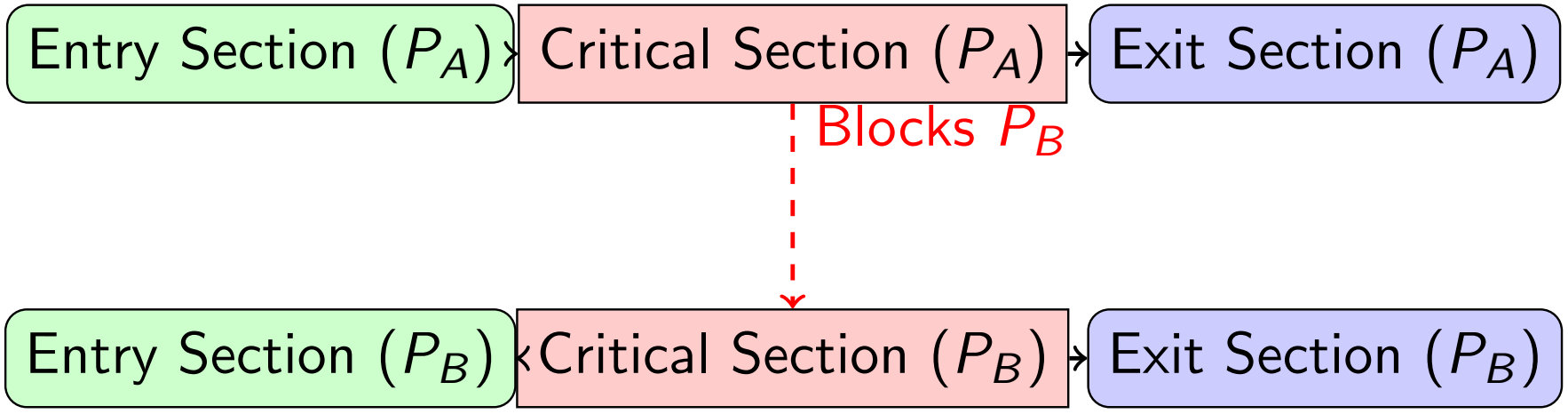
- The important feature is that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- The execution of critical sections by the processes is mutually exclusive in time.
- Each process must request permission to enter its critical section. The section of code implementing this request is the entry section.
- The critical section may be followed by an exit section, and then the remainder section.

Requirements for Critical Section Solution I

A valid solution to the critical-section problem must satisfy three essential requirements: Mutual Exclusion, Progress, and Bounded Waiting.

- Mutual Exclusion: If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
- Progress: If no process is in its critical section and some processes wish to enter, the selection of the processes that will enter next cannot be postponed indefinitely.
- 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 a request to enter its critical section and before that request is granted.
- The solution must not depend on relative execution speeds of the processes.

Mutual Exclusion Concept Diagram I



- Mutual exclusion guarantees only one process holds the lock at any given time.
- Process A enters the critical section and locks it.
- Process B attempts to enter but is blocked in its entry section.
- Process B must wait until Process A executes the exit section and releases the lock.

Entry and Exit Sections Implementation I

The synchronization between processes is generally achieved using software algorithms (like Peterson's Solution) or hardware instructions (like TestAndSet or CompareAndSwap).

- Entry Section: Usually implemented as a 'while' loop that checks and acquires a lock. Sometimes called 'acquire()'.
- Critical Section: The actual code manipulating shared data, which must be executed quickly.
- Exit Section: Releases the lock, allowing other waiting processes to enter. Sometimes called 'release()'.
- Remainder Section: The non-critical part of the code that does not access shared resources.

Summary & Conclusion I

Process synchronization is a fundamental requirement for modern operating systems to support concurrency and parallelism safely.

- Race conditions threaten data integrity when shared resources are accessed concurrently.
- The critical section is the code segment where shared variables are manipulated.
- Mutual exclusion is the primary requirement to prevent race conditions.
- A robust synchronization solution must also ensure progress (no deadlock) and bounded waiting (no starvation).

Detailed Concept 1 I

Detailed explanation of Process Synchronization: Critical Section, Mutual Exclusion, Race Condition.

- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 2 I

Detailed explanation of Process Synchronization: Critical Section, Mutual Exclusion, Race Condition.

- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 3 I

Detailed explanation of Process Synchronization: Critical Section, Mutual Exclusion, Race Condition.

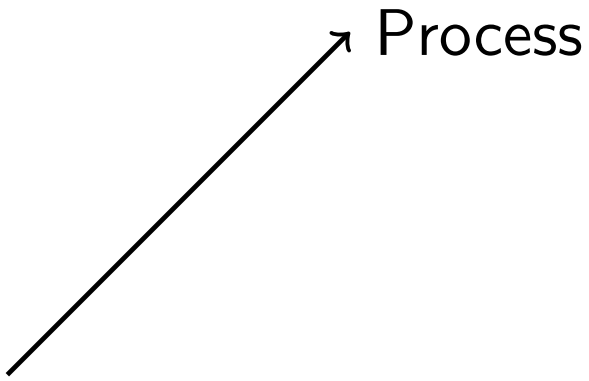
- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 4 I

Detailed explanation of Process Synchronization: Critical Section, Mutual Exclusion, Race Condition.

- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 5 I



- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 6 I

Detailed explanation of Process Synchronization: Critical Section, Mutual Exclusion, Race Condition.

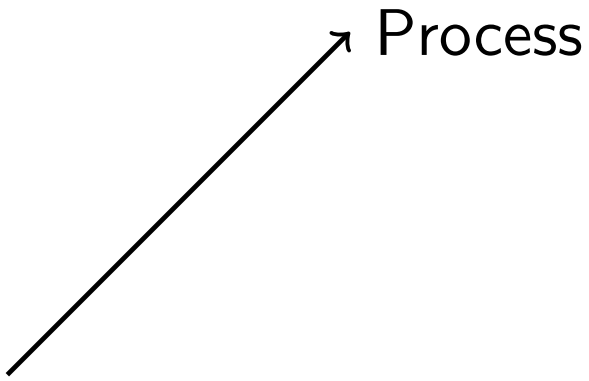
- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 7 I

Detailed explanation of Process Synchronization: Critical Section, Mutual Exclusion, Race Condition.

- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 8 I



- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 9 I

Detailed explanation of Process Synchronization: Critical Section, Mutual Exclusion, Race Condition.

- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Detailed Concept 10 I

Detailed explanation of Process Synchronization: Critical Section, Mutual Exclusion, Race Condition.

- Fundamental principle 1
- Engineering application 2
- Important consideration 3

Title Slide I

Course: Fundamentals Operating System

- Unit: Concepts of Process

Definition of Deadlock I

A deadlock is a situation where a set of processes are permanently blocked because each process is holding a resource and waiting for another resource acquired by some other process.

- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources; if they are not available, the process enters a waiting state.
- Deadlock occurs when waiting processes can never change state because the resources they requested are held by other waiting processes.
- Real-world analogy: Traffic gridlock at an intersection where four cars arrive at the same time and each waits for the other to move.

Necessary Conditions for Deadlock I

For a deadlock to arise, four conditions must hold simultaneously in a system (known as Coffman conditions).

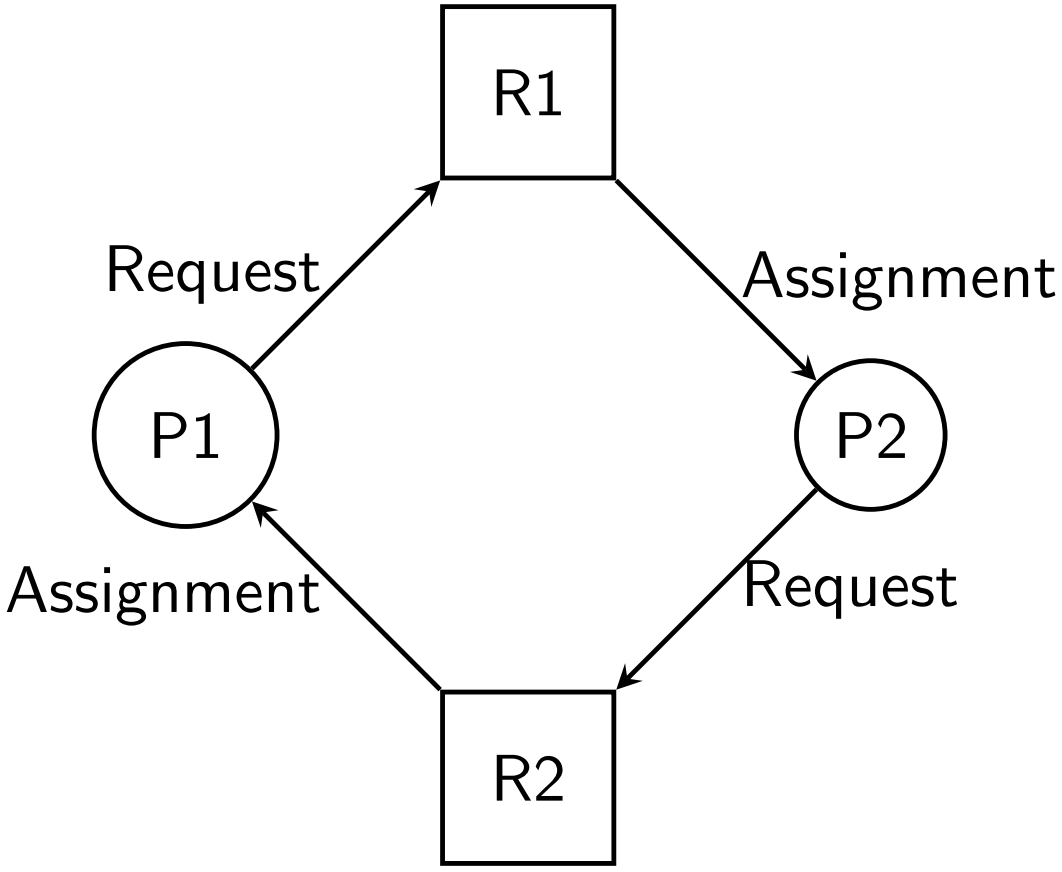
- Mutual Exclusion: At least one resource must be held in a non-sharable mode; only one process can use the resource at a time.
- Hold and Wait: A process must be holding at least one resource and waiting to acquire additional resources that are currently held by other processes.
- No Preemption: Resources cannot be preempted; a resource can be released only voluntarily by the process holding it after its task is completed.
- Circular Wait: A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for P2, and Pn is waiting for P0.

Resource Allocation Graph (RAG) I

A directed graph used to model resource allocation and identify deadlocks, consisting of a set of vertices V and edges E.

- Vertices are partitioned into two types: P = {P1, ..., Pn} (processes) and R = {R1, ..., Rm} (resource types).
- Request Edge (P -¿ R): A directed edge from process Pi to resource type Rj indicates Pi has requested an instance of Rj and is waiting.
- Assignment Edge (R -¿ P): A directed edge from resource type Rj to process Pi indicates an instance of Rj has been allocated to Pi.
- If a RAG contains no cycles, then no deadlock exists. If it contains a cycle, a deadlock may exist (certainly exists if each resource has only one instance).

Diagram: Resource Allocation Graph Cycle I



- The diagram illustrates a cyclic wait: Process P1 holds an instance of R2 and requests R1.
- Process P2 holds an instance of R1 and requests R2.
- Because each resource type has exactly one instance, this closed cycle directly implies a state of deadlock.

Dealing with Deadlock: General Strategies I

Operating systems employ different strategies to handle the potential of deadlocks based on the system design and workload.

- Ignore the problem completely, under the assumption that deadlocks are extremely rare (The Ostrich Algorithm).
- Use a protocol to proactively ensure the system will never enter a deadlock state (Deadlock Prevention or Avoidance).
- Allow the system to enter a deadlock state, detect its occurrence, and execute a routine to recover from it (Detection and Recovery).

Ignorance of Deadlock (Ostrich Algorithm) I

The simplest approach is to pretend deadlocks never happen, prioritizing system performance over absolute correctness.

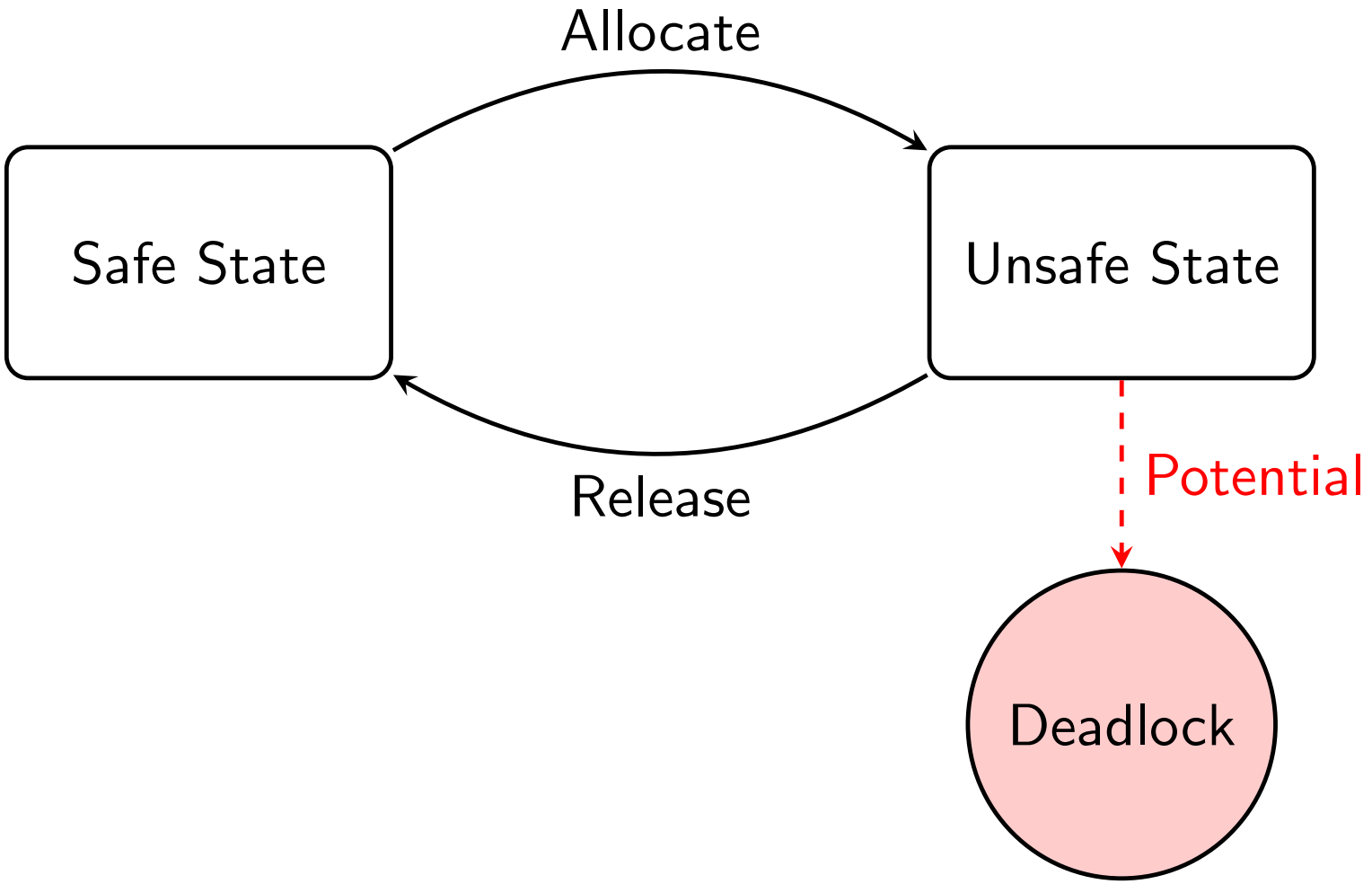
- Named after the ostrich sticking its head in the sand to ignore a problem.
- If deadlocks happen very rarely (e.g., once a year), it is cheaper to ignore them than to incur the constant computational overhead of prevention algorithms.
- When a deadlock does occur, the system or affected programs freeze, usually requiring manual user intervention (e.g., force-killing a process or rebooting).
- Most general-purpose operating systems, including UNIX, Linux, and Windows, fundamentally rely on this approach for resource management.

Deadlock Detection and Recovery I

If a system does not prevent deadlocks, it must provide algorithms to examine the system state and recover when a deadlock is detected.

- Detection algorithms periodically check for cycles in a wait-for graph (for single instances) or run a Banker's-like matrix algorithm (for multiple instances).
- Recovery via Process Termination: The OS can abort all deadlocked processes simultaneously or abort them one at a time until the deadlock cycle is broken.
- Recovery via Resource Preemption: Successively preempt resources from specific processes and give them to others until the cycle is resolved.
- Challenges include selecting a victim (minimizing cost), rollback (returning a process to a safe state), and avoiding starvation (ensuring the same process isn't always the victim).

Dynamic Deadlock Avoidance I



- Avoidance requires the OS to have a priori information about the maximum number of resources of each type that a process will request.

Dynamic Deadlock Avoidance II

- The system dynamically decides whether granting a resource request will leave the system in a 'Safe State'.
- A Safe State exists if there is a sequence of all processes such that each process can complete its execution given current available resources.
- An Unsafe State is not necessarily a deadlock, but it implies a deadlock might occur. The Banker's Algorithm prevents entering unsafe states.

Deadlock Prevention I

Prevention mechanisms proactively constrain resource requests to ensure that at least one of the four necessary conditions cannot occur.

- Mutual Exclusion: Make resources sharable where possible (e.g., read-only files), though some resources (like printers) are inherently non-sharable.
- Hold and Wait: Require a process to request and be allocated all its resources before execution begins. This can lead to low resource utilization and starvation.
- No Preemption: If a process holding resources requests another that is unavailable, all currently held resources are preempted and released implicitly.
- Circular Wait: Impose a total linear ordering on all resource types and mandate that each process requests resources strictly in an increasing order of enumeration.

Title Slide I

Course: Fundamentals Operating System

- Unit: Concepts of Process

What is a Deadlock? I

A formal definition of deadlock in an operating system context.

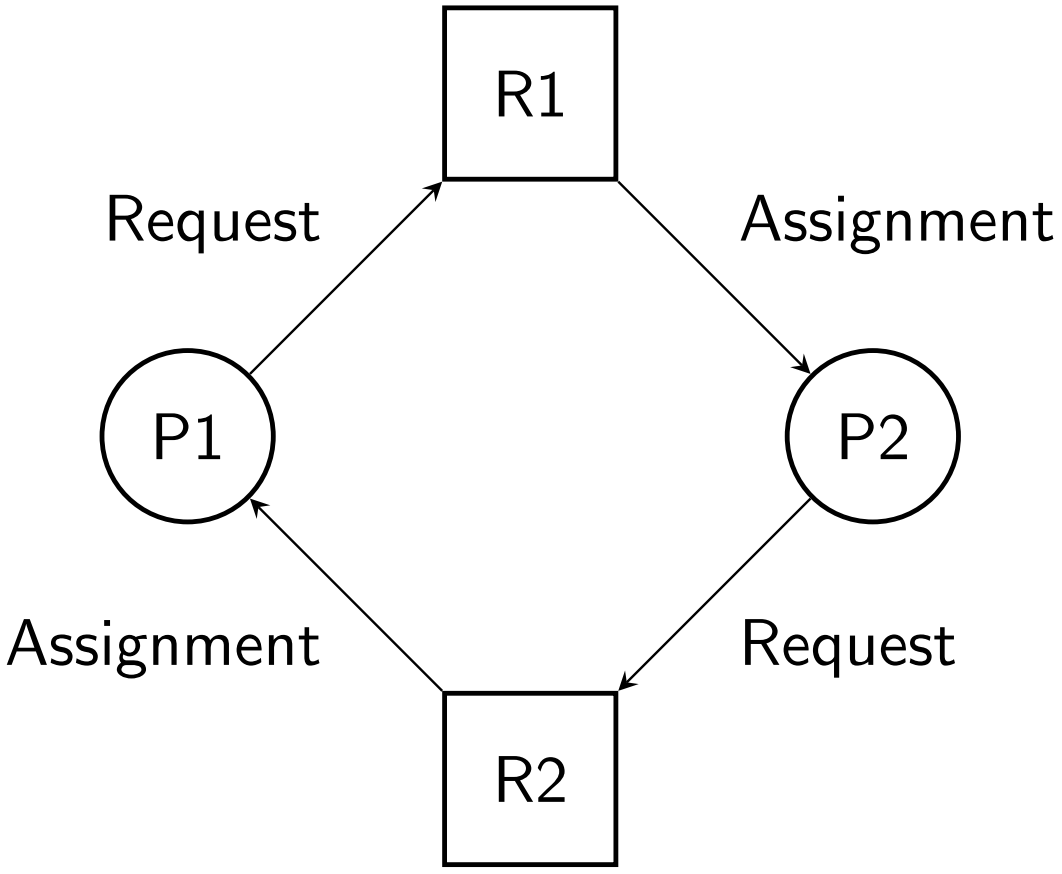
- A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set.
- Typically, the event is the release of a currently held resource.
- The system grinds to a halt, as none of the blocked processes can execute, release resources, or be awakened.
- Example: Process A holds Printer and requests Scanner. Process B holds Scanner and requests Printer.

System Model for Resource Allocation I

Understanding how resources are utilized by processes.

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- Resource types (R1, R2, ..., Rm) include CPU cycles, memory space, files, and I/O devices (like printers).
- Each process utilizes a resource in a sequence: Request, Use, and Release.
- If a request cannot be granted immediately, the requesting process must wait until it can acquire the resource.

Resource Allocation Graph (RAG) I



- A directed graph used to represent the state of the system and precisely describe deadlocks.
- Vertices are partitioned into two types: P = {P1, P2, ..., Pn} (processes) and R = {R1, R2, ..., Rm} (resource types).

Resource Allocation Graph (RAG) II

- A Request Edge (Pi -¿ Rj) signifies that process Pi has requested an instance of resource type Rj and is waiting.
- An Assignment Edge (Rj -¿ Pi) signifies that an instance of resource type Rj has been allocated to process Pi.
- If the graph contains no cycles, then no process in the system is deadlocked.

The Four Necessary Conditions for Deadlock I

A deadlock situation can arise if and only if the following four conditions hold simultaneously in a system:

- 1. Mutual Exclusion: At least one resource must be held in a non-sharable mode; if another process requests it, it must wait.
- 2. Hold and Wait: A process must be holding at least one resource and waiting to acquire additional resources held by other processes.
- 3. No Preemption: Resources cannot be preempted; a resource can be released only voluntarily by the process holding it.
- 4. Circular Wait: A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for P1, P1 for P2, and Pn for P0.

Dealing with Deadlock: Strategies I

Operating systems employ different methods to handle deadlocks depending on the frequency and cost of prevention.

- Ignorance (The Ostrich Algorithm): Pretend deadlocks never occur in the system. Used by most modern OSes (UNIX, Windows).
- Prevention: Ensure that the system will never enter a deadlock state by structurally negating one of the four necessary conditions.
- Avoidance: Require the OS to be given additional information in advance concerning which resources a process will request.
- Detection and Recovery: Allow the system to enter a deadlock state, detect it, and then recover from it.

Ignorance of Deadlock (Ostrich Algorithm) I

The simplest and most widely used approach to handling deadlocks in commodity operating systems.

- The OS completely ignores the problem, effectively 'sticking its head in the sand' like an ostrich.
- Rationale: Deadlocks occur rarely in general-purpose systems (e.g., once a year).
- The overhead (performance cost) of preventing, avoiding, or detecting deadlocks is far greater than simply rebooting the system when a deadlock occurs.
- Handling it is left up to the user (e.g., killing a frozen application or restarting the computer).

Deadlock Prevention I

Methods for preventing deadlocks by ensuring that at least one of the necessary conditions cannot hold.

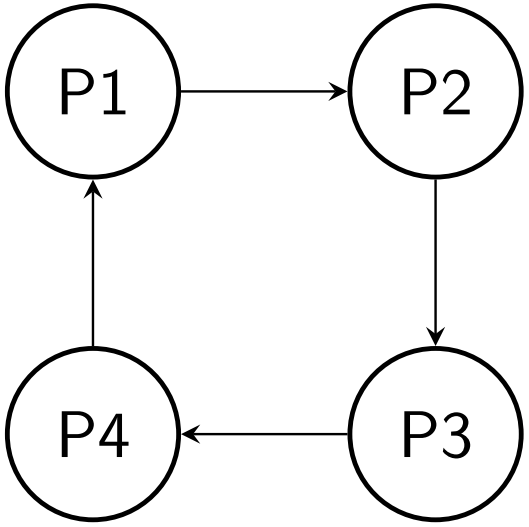
- Mutual Exclusion: Difficult to deny for intrinsically non-sharable resources (like a mutex lock), but read-only files can bypass this.
- Hold and Wait: Require a process to request and be allocated all its resources before it begins execution (low resource utilization).
- No Preemption: If a process holding some resources requests another resource that cannot be immediately allocated, all currently held resources are preempted.
- Circular Wait: Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Dynamic Deadlock Avoidance (Banker's Algorithm) I

Ensures that the system never enters an unsafe state, thus avoiding deadlocks dynamically.

- The OS simulates the allocation of predetermined maximum possible amounts of all resources, and then makes an 's-state' check.
- Safe State: The system can allocate resources to each process in some order and still avoid a deadlock.
- Unsafe State: A state where a deadlock could potentially occur; avoidance algorithms ensure the system never enters an unsafe state.
- Banker's Algorithm uses available, max, allocation, and need matrices to determine if granting a request leaves the system in a safe state.

Deadlock Detection and Recovery I



- If a system does not employ a prevention or avoidance algorithm, a deadlock may occur, necessitating detection and recovery.
- Detection: In single-instance resource environments, OS uses a Wait-for Graph (shown above). A cycle indicates a deadlock.
- Recovery by Process Termination: Abort all deadlocked processes, or abort one process at a time until the deadlock cycle is eliminated.

Deadlock Detection and Recovery II

- Recovery by Resource Preemption: Successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

Title Slide I

Course: Fundamentals Operating System

- Unit: Memory Management
- Lecture 21: Logical and physical address, Swapping

Address Binding I

Address binding is the process of mapping a logical address to a physical address.

- Programs on disk, ready to execute, form an input queue.
- Addresses in source programs are generally symbolic (like variable names).
- Binding of instructions and data to memory addresses can happen at compile time, load time, or execution time.
- Execution time binding requires hardware support for address maps (e.g., base and limit registers).

Logical Address I

An address generated by the CPU is commonly referred to as a logical address.

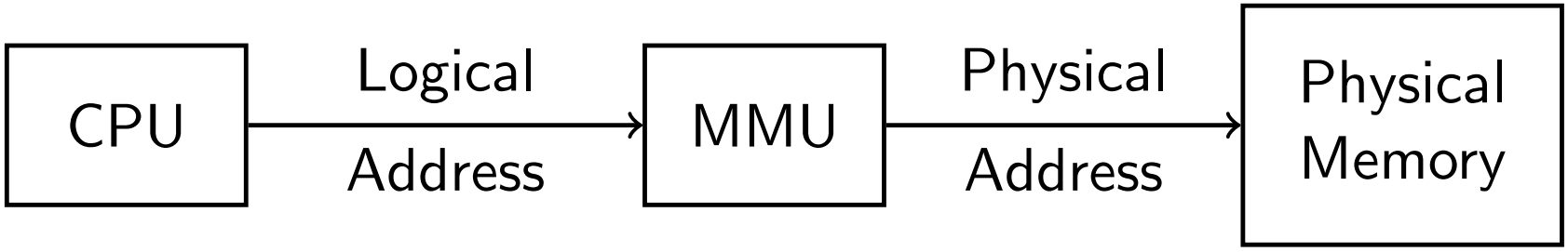
- Also widely known as a virtual address.
- The set of all logical addresses generated by a program is known as its logical address space.
- Logical addresses do not refer to actual physical memory locations but are used as a reference point.
- The user program deals only with logical addresses; it never sees the real physical addresses.

Physical Address I

A physical address is an address seen by the memory unit.

- It refers to an actual physical location in the RAM (Main Memory) where data is stored.
- The set of all physical addresses corresponding to logical addresses is the physical address space.
- Compile-time and load-time address-binding schemes result in identical logical and physical addresses.
- Execution-time address-binding schemes result in differing logical and physical addresses.

Hardware Address Translation I



- The CPU generates a logical address during program execution.
- The MMU (Memory-Management Unit) intercepts this logical address in hardware.
- The MMU quickly translates the logical address into a physical address.
- The translated physical address is sent across the memory bus to fetch or store data.

Memory-Management Unit (MMU) I

The run-time mapping from virtual to physical addresses is done by a hardware device called the MMU.

- Many methods and architectures exist to implement this address mapping.
- A simple scheme uses a relocation register (sometimes called a base register).
- 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 process never accesses physical memory directly; the MMU handles the indirection transparently.

What is Swapping? I

Swapping is a memory management technique where a process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

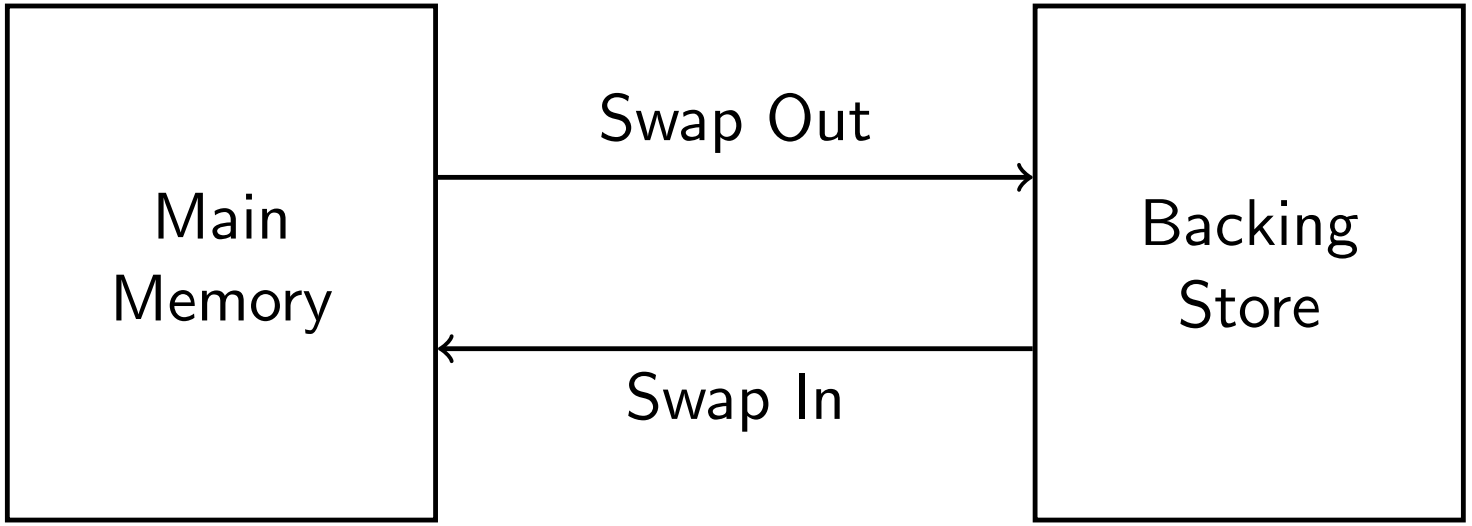
- Total physical memory space demanded by all processes can exceed the actual installed physical memory.
- Swapping increases the effective degree of multiprogramming in the system.
- It makes it possible to run large programs that wouldn't fit into the available physical RAM all at once.
- The backing store is typically a fast disk large enough to accommodate copies of all memory images for all users.

The Swapping Mechanism I

The OS system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run.

- When the CPU scheduler decides to execute a process, it invokes the dispatcher.
- The dispatcher checks to see whether the next scheduled process is currently residing in main memory.
- If not, and there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.
- It then reloads CPU registers and transfers control to the selected process to resume execution.

Swapping Process Visualization I



- Swap Out: Evicting a process from Main Memory and saving its exact state to the Backing Store to free up RAM.
- Swap In: Loading a process from the Backing Store back into Main Memory so the CPU can resume its execution.
- The backing store must provide direct, high-speed access to these memory images to minimize latency.

Performance Considerations I

Swapping requires a significant amount of time, primarily dictated by the speed of the disk transfer.

- The major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.
- Standard monolithic swapping is rarely used in modern OS due to unacceptably high context-switch times.
- Modified versions of swapping are found on many systems (e.g., swapping is only enabled when free memory falls below a critical threshold).
- Modern systems typically use paging in conjunction with swapping for much more fine-grained and efficient memory management.

Logical and physical address, Swapping I

Course: Fundamentals Operating System

- Unit: Memory Management
- Lecture 22
- Focus: Hardware Address Translation and Memory Allocation

Background: Memory Management I

Before processes can be executed, their instructions and data must be loaded into main memory.

- The CPU fetches instructions from memory according to the value of the program counter.
- Main memory and processor registers are the only general-purpose storage the CPU can access directly.
- A typical execution cycle involves fetching the instruction, decoding it, fetching operands, and storing the result in memory.
- Memory management involves protecting the operating system from access by user processes and protecting processes from one another.

Logical vs. Physical Address Space I

An address generated by the CPU is fundamentally different from the address seen by the physical memory unit.

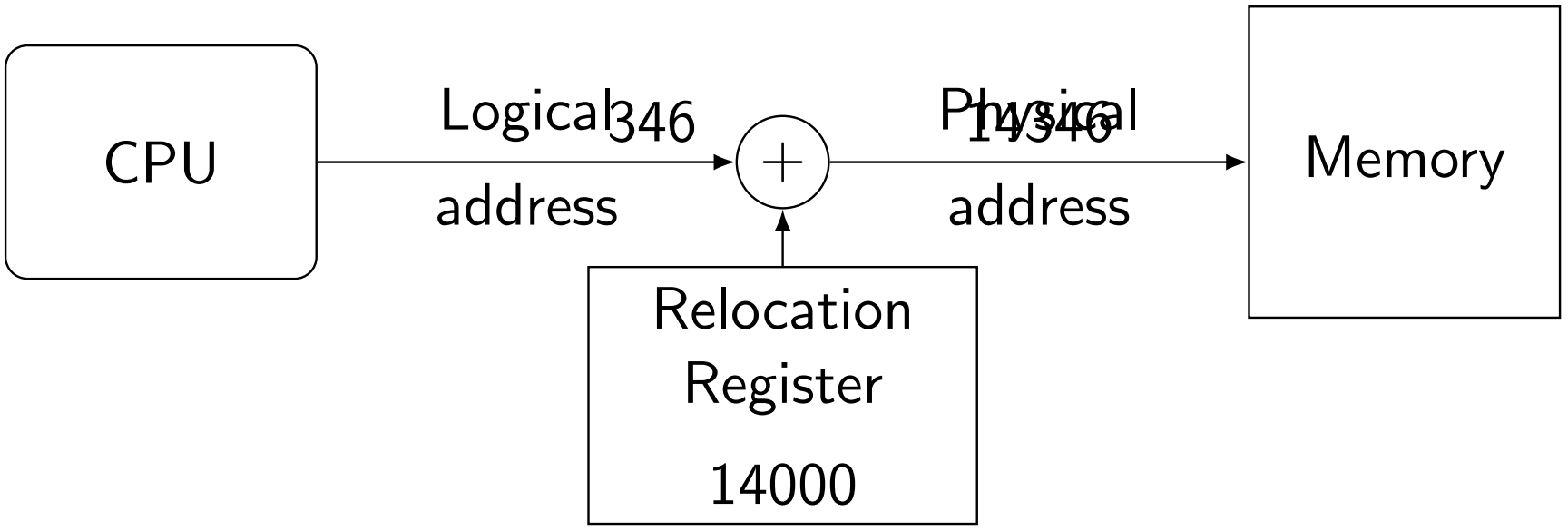
- Logical Address (Virtual Address): An address generated by the CPU during program execution. The user program deals entirely with logical addresses.
- Physical Address: An address seen by the memory unit—that is, the address loaded into the memory-address register.
- Logical Address Space: The set of all logical addresses generated by a program.
- Physical Address Space: The set of all physical addresses corresponding to these logical addresses.
- Compile-time and load-time address-binding methods result in identical logical and physical addresses. Execution-time binding results in differing addresses.

The Memory-Management Unit (MMU) I

The run-time mapping from virtual to physical addresses is handled by a hardware device called the Memory-Management Unit (MMU).

- The user program never sees the real physical addresses; it operates strictly within the logical address space.
- The MMU dynamically translates the logical addresses into physical addresses at the time of execution.
- A simple MMU scheme utilizes a base register (often called a relocation register).
- The value in the relocation register is added to every address generated by a user process when it is sent to memory.

Dynamic Relocation using a Relocation Register I



- The CPU generates a logical address (e.g., 346).
- The relocation register contains the base address of the process in physical memory (e.g., 14000).
- The MMU hardware adds the relocation register value to the logical address.
- The resulting physical address (14346) is then sent to the memory unit.

Dynamic Relocation using a Relocation Register II

- This transparent mechanism allows the OS to place the process anywhere in physical memory.

Hardware Address Protection I

To prevent a process from accessing memory it does not own, the system employs limit and relocation registers.

- The relocation register holds the smallest legal physical memory address for the process.
- The limit register specifies the size of the range of logical addresses.
- Each logical address generated is checked against the limit register: if it is smaller, it is valid; otherwise, a trap to the OS occurs.
- If valid, the address is added to the relocation register to form the physical memory address.
- This hardware check ensures strict isolation and protects the memory space of each process from unauthorized access.

Introduction to Swapping I

A process must be in memory to be executed, but it can be temporarily swapped out of memory to a backing store and later brought back in.

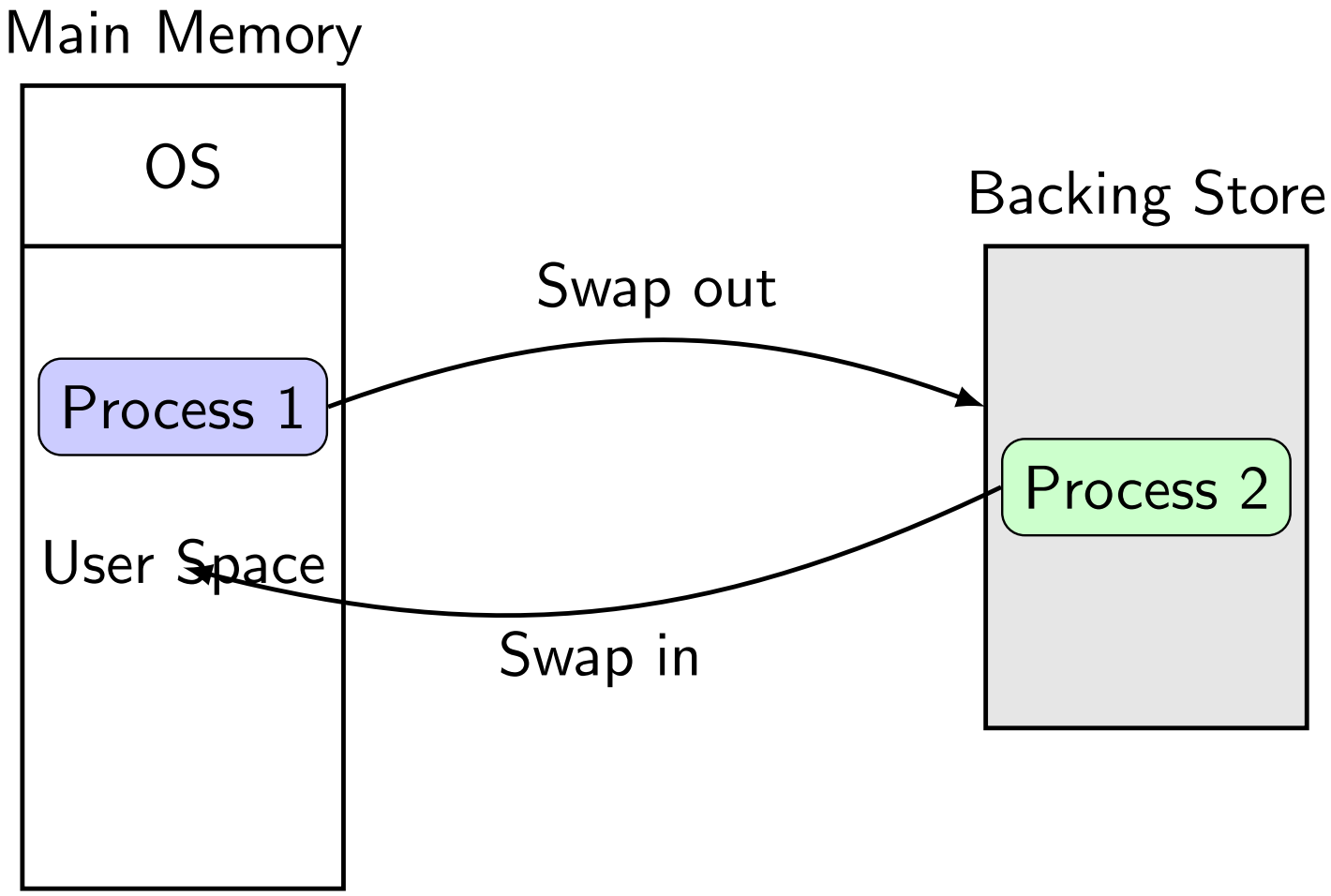
- Swapping makes it possible for the total physical address space of all active processes to exceed the real physical memory of the system.
- It increases the degree of multiprogramming by allowing more processes to run than can fit in RAM simultaneously.
- The backing store is typically a fast secondary storage disk large enough to accommodate copies of all memory images.
- The OS maintains a ready queue of all processes whose memory images are either on the backing store or in main memory and are ready to execute.

The Standard Swapping Mechanism I

Standard swapping involves moving entire processes between main memory and the backing store.

- Roll out, roll in: A swapping variant used in priority-based scheduling. A lower-priority process is rolled out so a higher-priority process can be rolled in.
- When swapped out, a process's exact execution state and memory footprint are preserved on the backing store.
- Whether a swapped-out process returns to the exact same physical address depends on the address binding method.
- With load-time binding, it must return to the same location. With execution-time binding (using MMU), it can be swapped back into any available physical space.

Swapping of Two Processes I



- The Operating System resides in low memory, and user processes reside in high memory.

Swapping of Two Processes II

- When memory is full and Process 2 needs to run, Process 1 is 'swapped out' to the backing store.
- Process 2, which was previously stored on the disk, is then 'swapped in' to the available main memory space.
- The dispatcher manages this procedure, interacting closely with memory management and disk I/O scheduling.

Performance Impact of Swapping I

The context-switch time in a swapping system can be extremely high compared to regular CPU process switching without swapping.

- The major component of swap time is transfer time, which is directly proportional to the amount of memory swapped.
- For example, swapping a 100MB process to a disk with a transfer rate of 50MB/s takes 2 seconds just for writing.
- Since standard swapping involves swapping one process out and another in, the total context switch time is essentially doubled.
- To optimize, operating systems must accurately track dynamic memory usage to swap only necessary portions.
- Modern systems generally disable standard entire-process swapping, instead relying on paging to swap specific pages (virtual memory).

Title Slide I

Course: Fundamentals Operating System

- Unit: Memory Management
- Lecture 23: Memory Allocation
- Instructor: Computer Science Professor

Introduction to Memory Allocation I

Memory allocation is the process of assigning blocks of main memory to various running processes.

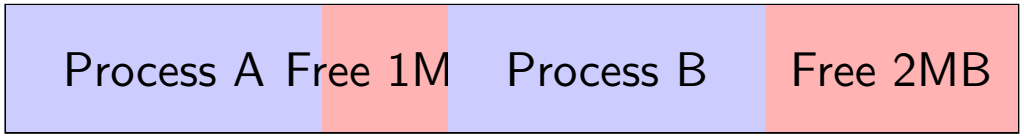
- Main memory must accommodate both the operating system (usually in low memory with interrupt vectors) and user processes (in high memory).
- Memory is typically divided into two partitions: one for the resident OS and one for user processes.
- Static Allocation: Performed at compile time, size is fixed and variables are assigned addresses in the data segment.
- Dynamic Allocation: Performed at run time, allows processes to request memory as needed from the heap or for the stack.

Contiguous Memory Allocation I

In contiguous memory allocation, each process is contained in a single contiguous section of memory.

- Relocation registers protect user processes from each other and from changing operating system code and data.
- Base register contains the value of the smallest physical address.
- Limit register contains the range of logical addresses - each logical address must be less than the limit register.
- The MMU (Memory Management Unit) maps logical addresses dynamically by adding the value in the relocation register.

Memory Fragmentation I



Total free memory is 3MB, but a 3MB process cannot be allocated contiguous space.

- External Fragmentation: Total memory space exists to satisfy a request, but it is not contiguous.
- Internal Fragmentation: Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- Compaction is a solution to external fragmentation: shuffle memory contents to place all free memory together in one large block.
- Compaction is only possible if relocation is dynamic and done at execution time.

Dynamic Storage-Allocation Problem I

Strategies on how to satisfy a request of size n from a list of free holes in contiguous allocation.

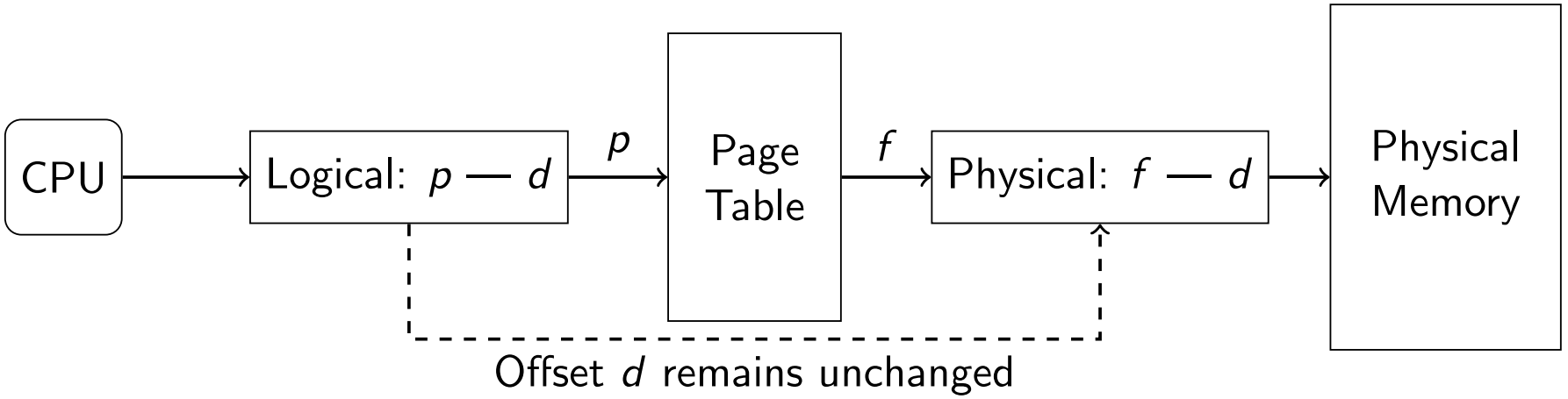
- First-Fit: Allocate the first hole that is big enough. Fast but can leave small, unusable fragments.
- Best-Fit: Allocate the smallest hole that is big enough. Requires searching the entire list unless ordered by size. Produces the smallest leftover hole.
- Worst-Fit: Allocate the largest hole. Also requires searching the entire list. Produces the largest leftover hole, which might be useful for another process.
- Simulations show that First-Fit and Best-Fit generally perform better than Worst-Fit in terms of speed and storage utilization.

Non-Contiguous Allocation: Paging I

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory.

- Divides physical memory into fixed-sized blocks called frames (size is a power of 2, typically 4KB).
- Divides logical memory into blocks of the same size called pages.
- To run a program of size n pages, need to find n free frames and load the program.
- Sets up a page table to translate logical to physical addresses.
- Solves the problem of external fragmentation entirely, but internal fragmentation can still occur in the last page of a process.

Paging Hardware Architecture I



- Logical address generated by CPU is divided into two parts: Page number (p) and Page offset (d).
- Page number (p) is used as an index into a page table which contains the base address (f) of each page in physical memory.
- Page offset (d) is combined with base address (f) to define the physical memory address that is sent to the memory unit.
- Translation happens entirely in hardware via the MMU, making it extremely fast.

Translation Look-aside Buffer (TLB) I

A TLB is a special, small, fast-lookup hardware cache associative memory used to reduce memory access time.

- Each page table access requires a memory access. Without a TLB, paging requires two memory accesses (one for page table, one for data).
- The TLB stores recent translations (page number -¿ frame number).
- TLB Hit: If page number is in TLB, frame number is retrieved immediately. Time is highly optimal.
- TLB Miss: If page number is not in TLB, memory reference to page table is made, and the translation is loaded into the TLB for future references.
- Effective Access Time (EAT) heavily depends on the TLB hit ratio.

Segmentation I

Segmentation is a memory-management scheme that supports the user view of memory, rather than a system-centric flat view.

- A program is a collection of segments (e.g., main program, procedures, arrays, stack, symbol table).
- Logical address consists of a two tuple: ¡segment-number, offset¿.
- Segment table maps two-dimensional physical addresses; each entry has a segment base and a segment limit.
- Unlike paging where the OS divides memory blindly, segmentation divides it logically, facilitating easier protection and code sharing.
- Because segments vary in size, segmentation can suffer from external fragmentation, unlike paging.

Summary & Future Topics I

Memory allocation techniques have evolved to provide efficient, protected, and flexible memory environments.

- Contiguous allocation is simple but suffers from external fragmentation.
- Paging eliminates external fragmentation by using fixed-size pages and frames, relying on a page table and TLB for translation.
- Segmentation groups logical components together, aiding sharing and protection but reintroducing external fragmentation.
- Modern operating systems typically use a combination: Paged Segmentation (segments are divided into pages).
- Next Lecture: Virtual Memory and Demand Paging (Lecture 24).

Title Slide I

Course: Fundamentals Operating System

- Unit: Memory Management
- Lecture 24: Memory Allocation

Introduction to Memory Allocation I

Memory allocation is the process of assigning blocks of memory to distinct computer programs on request.

- Main memory must accommodate both the operating system and user processes.
- Memory is usually divided into two partitions: one for the resident OS and one for user processes.
- The OS is typically placed in either low memory or high memory depending on the location of the interrupt vector.
- Hardware support via base and limit registers protects user processes from each other and protects the OS code.

Contiguous Memory Allocation I

In contiguous memory allocation, each process is contained in a single contiguous section of memory.

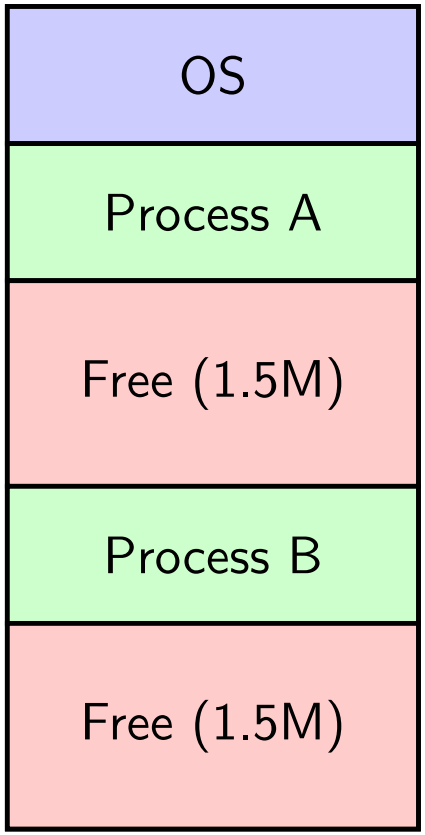
- Relocation registers are used to protect user processes from each other.
- Memory is divided into multiple partitions of fixed or variable sizes.
- Variable-partition sizes allow for more efficient memory utilization.
- The OS maintains information about allocated partitions and free partitions (holes).

Dynamic Storage-Allocation Problem I

How to satisfy a request of size 'n' from a list of free holes?

- First-Fit: Allocate the first hole that is big enough. Fast but can leave small fragmented holes.
- Best-Fit: Allocate the smallest hole that is big enough. Produces the smallest leftover hole, but requires searching the entire list.
- Worst-Fit: Allocate the largest hole. Produces the largest leftover hole, which may be more useful.
- Simulations have shown that First-Fit and Best-Fit are generally better than Worst-Fit in terms of speed and storage utilization.

Fragmentation Concepts I



External Fragmentation:
Total free = 3M, but
cannot satisfy a 2M request.

- External Fragmentation exists when total memory space is sufficient to satisfy a request, but it is not contiguous.
- Internal Fragmentation occurs when allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Fragmentation Concepts II

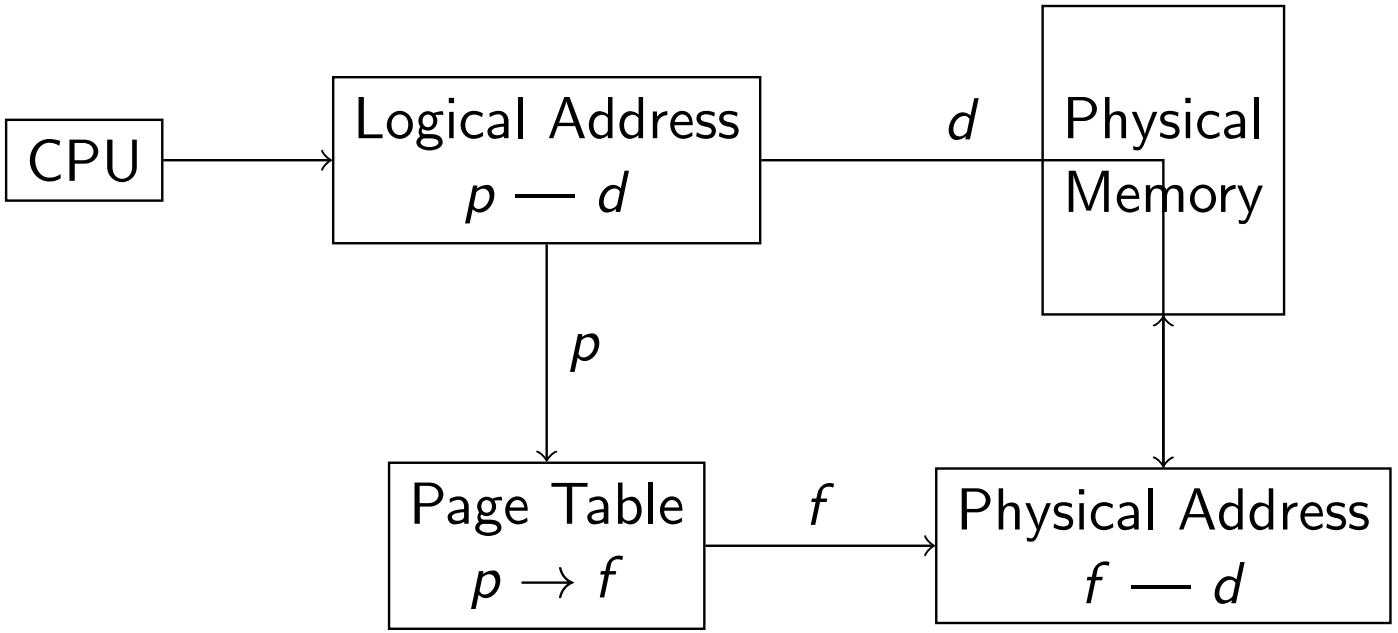
- Compaction can be used to reduce external fragmentation by shuffling memory contents to place all free memory together in one large block.
- Compaction is only possible if relocation is dynamic and done at execution time.

Non-Contiguous Memory Allocation: Paging I

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory.

- Physical memory is divided into fixed-sized blocks called frames.
- Logical memory is divided into blocks of the same size called pages.
- To run a program of size n pages, we need to find n free frames and load the program.
- The OS sets up a page table to translate logical addresses to physical addresses.
- Paging avoids external fragmentation but can still suffer from internal fragmentation.

Paging Hardware and Address Translation I



- The logical address generated by the CPU is divided into two parts: a page number (p) and a page offset (d).
- The page number (p) is used as an index into a page table.
- The page table contains the base address of each page in physical memory (frame number, f).
- The physical address is formed by combining the frame number (f) and the page offset (d).

Translation Look-aside Buffer (TLB) I

A TLB is a special, small, fast-lookup hardware cache used to reduce the time taken to access a user memory location.

- Storing the page table in main memory results in two memory accesses: one for the page table and one for the data.
- The TLB stores a small subset of page table entries.
- If a page number is found in the TLB (TLB hit), the frame number is retrieved quickly without accessing main memory.
- If a page number is not found in the TLB (TLB miss), the page table in main memory must be accessed.
- Effective Access Time (EAT) depends heavily on the TLB hit ratio.

Segmentation I

Segmentation is a memory-management scheme that supports the user view of memory.

- A program is a collection of segments. A segment is a logical unit such as: main program, procedure, function, local variables, etc.
- Logical address consists of a two tuple: ¡segment-number, offset¿.
- A segment table maps two-dimensional user-defined addresses into one-dimensional physical addresses.
- Each entry in the segment table has a segment base and a segment limit.
- Segmentation eliminates internal fragmentation but can suffer from external fragmentation.

Summary and Conclusion I

Modern operating systems use complex memory allocation mechanisms, often combining paging and segmentation.

- Memory management is critical for multiprogramming and system performance.
- Contiguous allocation suffers from significant fragmentation issues.
- Paging solves external fragmentation and simplifies memory allocation by using fixed-size blocks.
- TLBs are essential for making paging systems performant.
- Segmentation aligns memory management with how programmers view logical units.

Title Slide I

Course: Fundamentals Operating System

- Unit: Memory Management
- Lecture 25: Contiguous Memory Allocation
- Topics: Fixed & Dynamic Partitioning, Fragmentation, Relocation, Allocation Techniques

Introduction to Contiguous Memory Allocation I

In contiguous memory allocation, each process is contained in a single contiguous section of memory. Memory is generally divided into two partitions: one for the resident operating system and one for user processes.

- Main memory must accommodate both the OS and various user processes efficiently.
- Contiguous allocation is one of the early methods where each process is loaded into a single unbroken memory block.
- Requires hardware support for memory mapping and protection to prevent processes from accessing each other's memory.
- Implemented using either fixed (static) partitioning or dynamic (variable) partitioning techniques.

Multiprogramming with Fixed Partitioning I

Memory is divided into several fixed-size partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is strictly bound by the number of partitions.

- Partitions are created at system generation time and their sizes do not change dynamically.
- When a partition is free, a process is selected from the input queue and loaded into it.
- Simple to implement and requires minimal operating system overhead.
- Suffers from severe memory waste if the process size is significantly smaller than the partition size.

Internal Fragmentation I

Internal fragmentation occurs when the memory allocated to a process is slightly larger than the requested memory. The size difference is memory internal to a partition but not being used.

- Predominantly a severe issue in fixed partitioning schemes.
- Happens because memory blocks are allocated in fixed sizes regardless of the exact request size.
- The wasted space is trapped inside the allocated block and cannot be utilized by any other process.
- Example: A 2MB process loaded into a 4MB fixed partition wastes 2MB of memory internally.

Diagram: Fixed Partitioning and Internal Fragmentation I

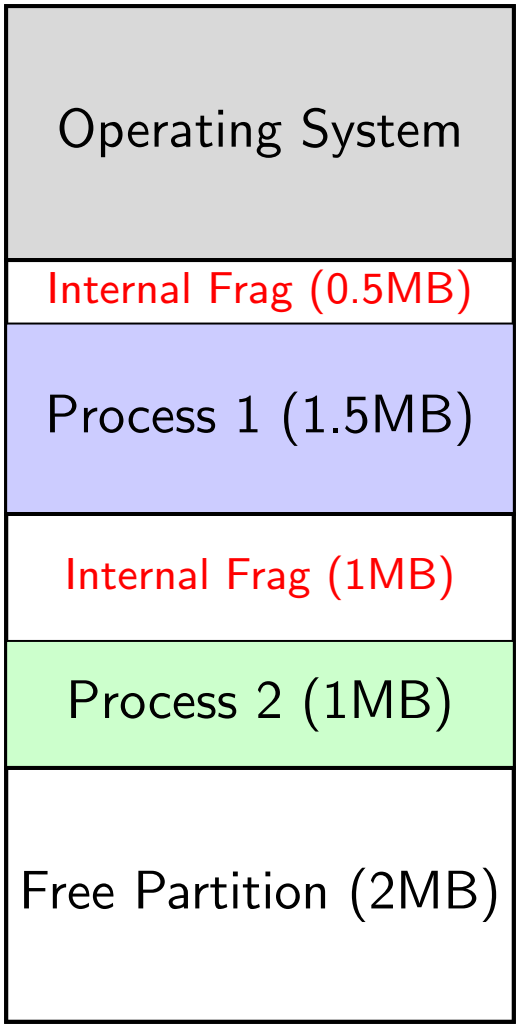


Diagram: Fixed Partitioning and Internal Fragmentation II

- Visual representation of memory divided into fixed 2MB partitions.
- Shows how allocating processes smaller than 2MB leaves unusable gaps within partitions.
- The operating system occupies the top reserved section of memory.

Multiprogramming with Dynamic Partitioning I

In dynamic partitioning, partitions are created dynamically based on the exact size required by the processes. Memory is not divided a priori.

- When a process arrives, it is allocated memory exactly equal to its size from an available contiguous hole.
- Eliminates internal fragmentation entirely since there is no wasted space inside a partition.
- The OS maintains a table indicating which parts of memory are available (holes) and which are occupied.
- Over time, as processes enter and leave, memory becomes fragmented into many small non-contiguous holes.

External Fragmentation and Compaction I

External fragmentation exists when total free memory space is sufficient to satisfy a request, but it is not contiguous, meaning the storage is fragmented into a large number of small holes.

- Primarily affects dynamic partitioning systems.
- Can lead to severe memory utilization problems where a large process cannot be loaded despite enough total free memory.
- Compaction is a solution: shuffling memory contents to place all free memory together in one large contiguous block.
- Compaction is only possible if relocation is dynamic and done at execution time. It is an extremely expensive operation.

Dynamic Storage-Allocation Techniques I

When a process arrives and needs memory, the OS must search for a free hole large enough to accommodate it using specific allocation algorithms.

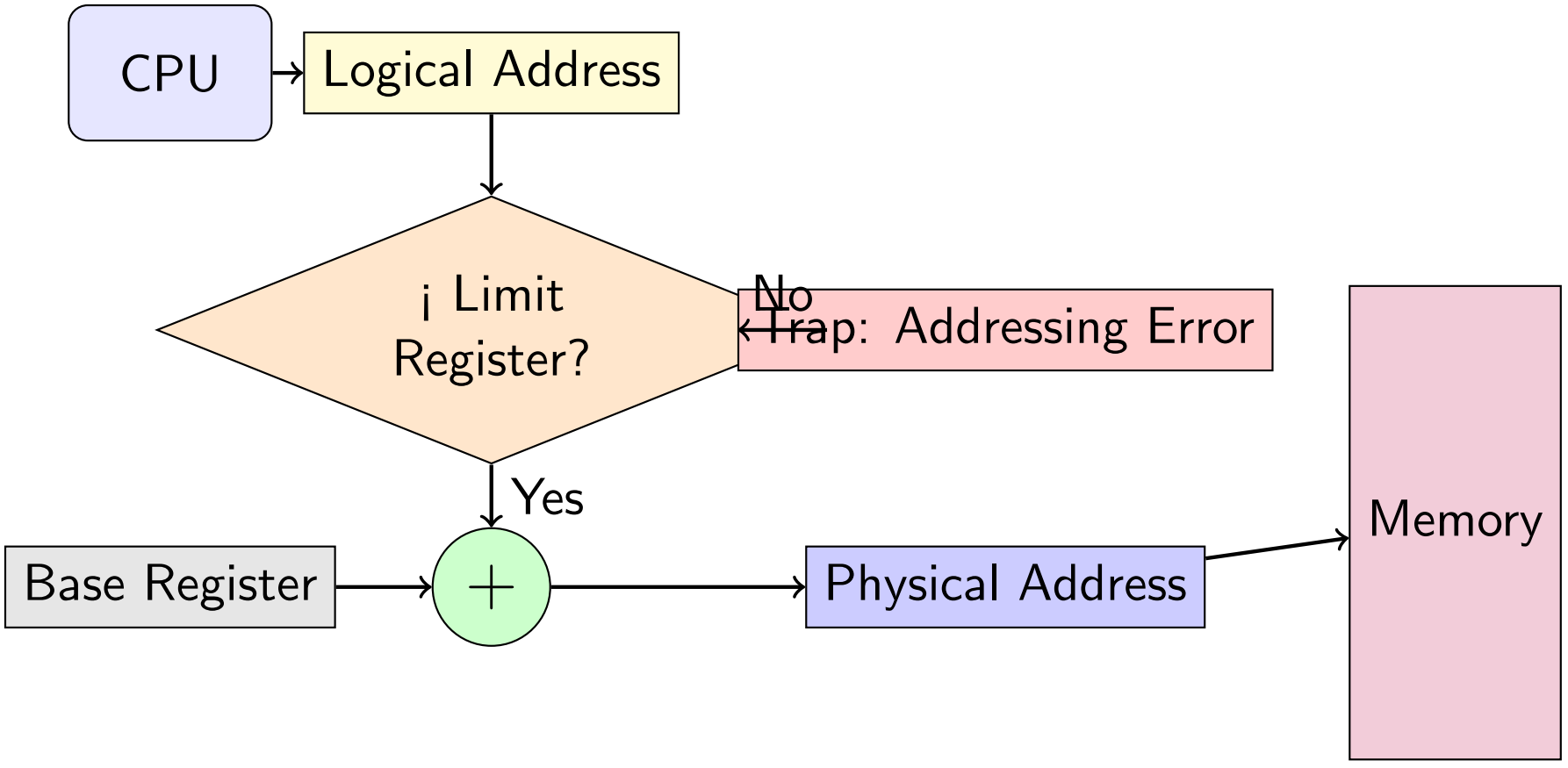
- First-Fit: Allocates the first hole that is big enough. Fast searching, but may leave small useless holes at the beginning of memory.
- Best-Fit: Allocates the smallest hole that is big enough. Requires searching the entire list unless sorted by size. Produces the smallest leftover hole.
- Worst-Fit: Allocates the largest available hole. Produces the largest leftover hole, which may be more useful than the tiny holes left by Best-Fit.
- Simulations show First-fit and Best-fit generally perform better than Worst-fit in terms of time and storage utilization.

Memory Relocation and Protection Mechanism I

Contiguous allocation requires robust hardware mechanisms to relocate process addresses at runtime and protect memory spaces of different processes and the OS from malicious or accidental access.

- Base Register (Relocation Register): Contains the value of the smallest physical address for a process. Added to every logical address.
- Limit Register: Contains the range of logical addresses (the size of the process). Each logical address must be strictly less than this limit.
- The MMU (Memory Management Unit) maps logical addresses dynamically by adding the base register value.
- If a process generates an address outside the limit, a trap (fatal addressing error) is generated to the OS, ensuring memory protection.

Diagram: Hardware Support for Relocation and Protection I



- Logical address generated by the CPU is first checked against the Limit Register.

Diagram: Hardware Support for Relocation and Protection II

- If the address is valid (less than the limit), it is added to the Base Register to form the final physical address.
- This hardware sequence occurs for every single memory reference, enforcing protection and enabling transparent runtime relocation.

Title Slide I

Course: Fundamentals Operating System

- Unit: Memory Management
- Lecture 26: Contiguous Memory Allocation and Partitioning

Contiguous Memory Allocation Overview I

Main memory must accommodate both the operating system and various user processes. In contiguous memory allocation, each process is contained in a single contiguous section of memory.

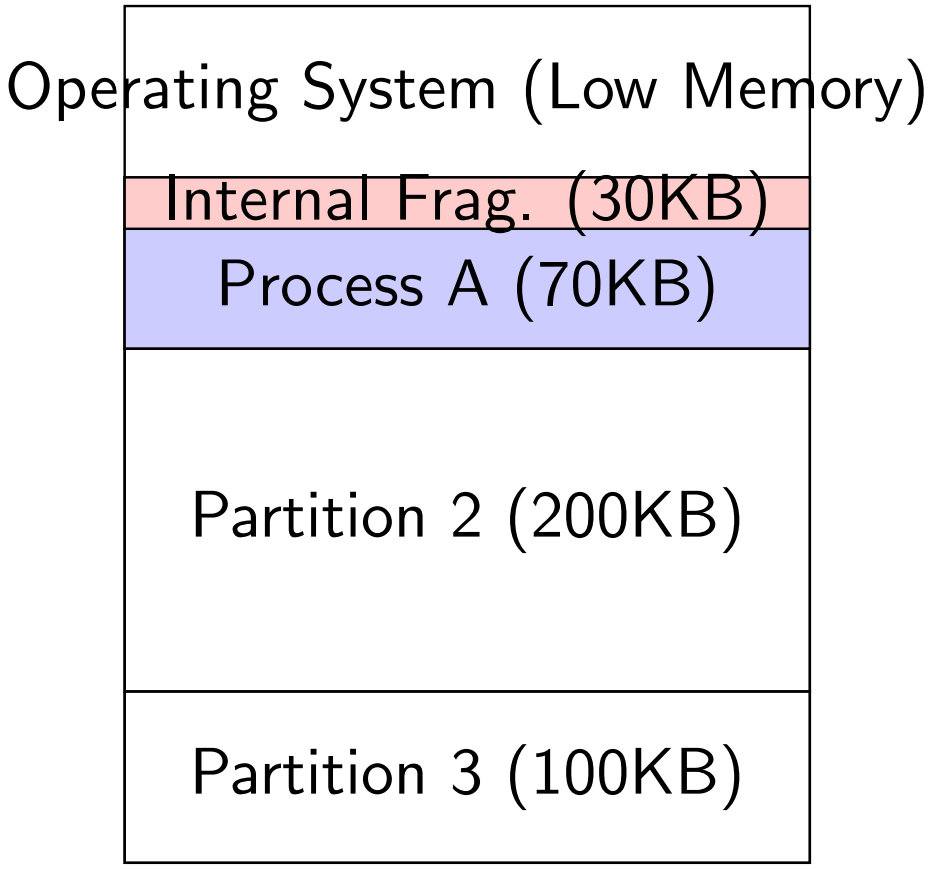
- Memory is usually divided into two partitions: one for the resident operating system, and one for user processes.
- Operating system can be placed in either low memory or high memory. (Typically low memory with interrupt vector).
- Contiguous allocation requires a protection mechanism to ensure user processes cannot access OS memory or other user processes' memory.

Multiprogramming with Fixed Partitioning (MFT) I

In MFT (Multiprogramming with a Fixed number of Tasks), memory is divided into a number of static partitions at system generation time.

- Each partition can contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions.
- When a partition is free, a process is selected from the input queue and loaded into the free partition.
- When the process terminates, the partition becomes available for another process.
- This method suffers from internal fragmentation if the loaded process is smaller than the partition size.

Internal Fragmentation Illustration I



- Internal fragmentation occurs when allocated memory is slightly larger than the requested memory.
- The size difference is memory internal to a partition, but it is not being used by the process.

Internal Fragmentation Illustration II

- The unused 30KB in Partition 1 cannot be allocated to another process because the partition boundary is fixed.

Multiprogramming with Dynamic Partitioning (MVT) I

In MVT (Multiprogramming with a Variable number of Tasks), the OS keeps a table indicating which parts of memory are available and which are occupied.

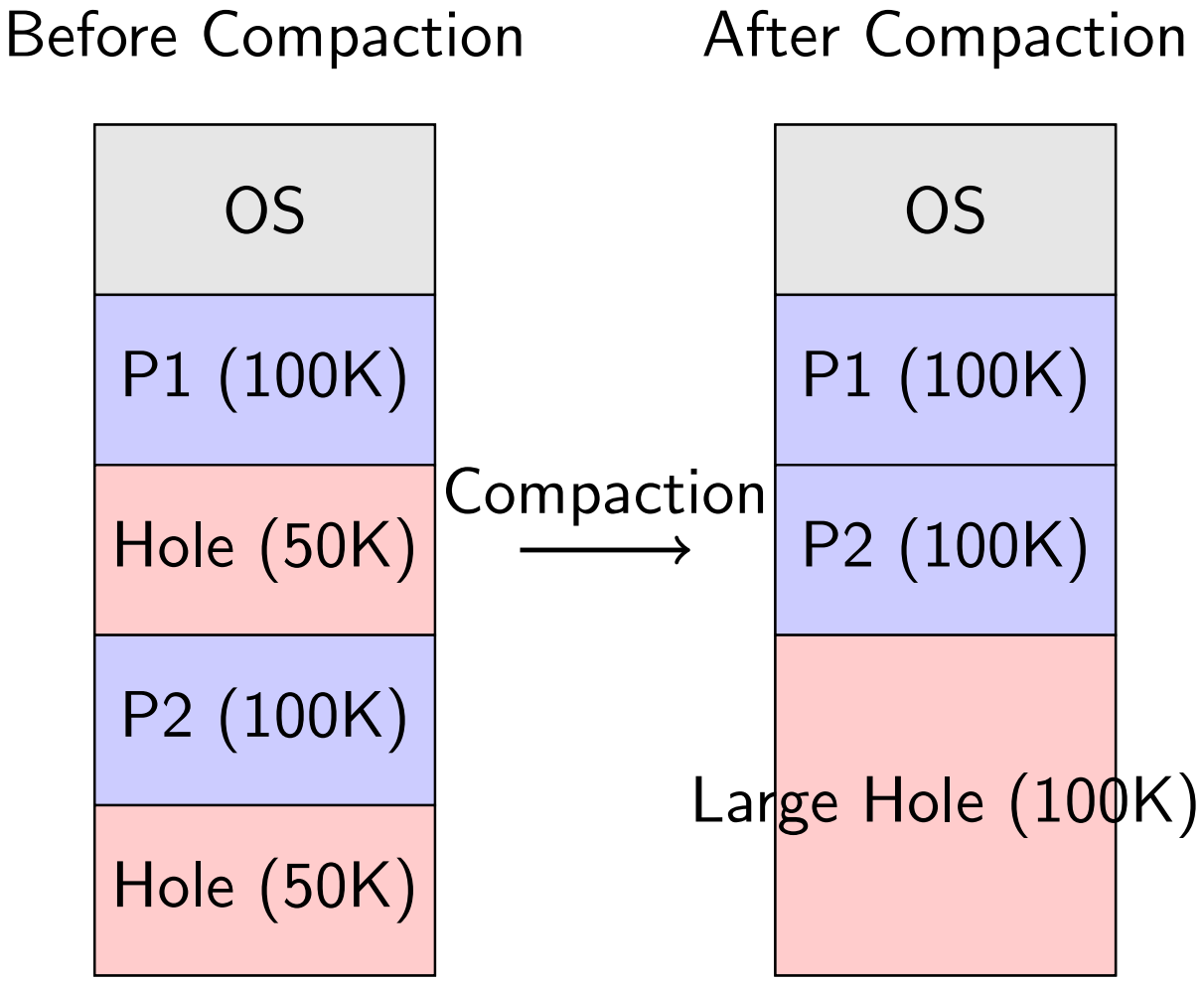
- Initially, all memory is available for user processes and is considered one large block of available memory, a hole.
- When a process arrives and needs memory, the OS searches for a hole large enough for this process.
- If a hole is found, it is allocated to the process, and the rest is left as a smaller hole.
- Eliminates internal fragmentation, as processes are allocated exactly as much memory as they request.

External Fragmentation I

As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous.

- Dynamic partitioning algorithms suffer from external fragmentation.
- Worst case: A block of free (or wasted) memory between every two processes.
- Statistical analysis of first-fit reveals the 50-percent rule: given N allocated blocks, another 0.5 N blocks will be lost to fragmentation.
- This means one-third of memory may be unusable!

Memory Compaction Process I



- Compaction is a solution to external fragmentation.
- Goal is to shuffle the memory contents so as to place all free memory together in one large block.

Memory Compaction Process II

- Compaction is possible ONLY if relocation is dynamic, and is done at execution time.
- Requires changing base registers to reflect the new starting address of shifted processes.

Memory Relocation and Protection Mechanism I

To prevent a process from accessing memory it does not own, the system utilizes hardware support through base (relocation) and limit registers.

- Relocation (Base) Register: Contains the value of the smallest physical address of the process.
- Limit Register: Contains the range of logical addresses (size of the process).
- The MMU dynamically maps logical address to physical address by adding the value in the relocation register.
- If a logical address is greater than or equal to the limit register, a trap to the operating system occurs (addressing error).

Allocation Techniques: First-Fit and Best-Fit I

When a process arrives requiring memory of size n, the OS must choose a hole from the set of available holes. These are the dynamic storage allocation problem solutions.

- First-Fit: Allocate the first hole that is big enough. Searching can start either at the beginning or where the previous search ended. Generally fast.
- Best-Fit: Allocate the smallest hole that is big enough. Must search the entire list unless it is ordered by size.
- Best-fit produces the smallest leftover hole, but requires more search time than first-fit and creates tiny, useless leftover fragments.

Allocation Techniques: Worst-Fit I

An alternative dynamic storage allocation strategy that takes an opposite approach to best-fit.

- Worst-Fit: Allocate the largest available hole. Again, we must search the entire list, unless it is sorted by size.
- This strategy produces the largest leftover hole, which may be more useful than the smaller leftover holes generated by a best-fit approach.
- Simulations show that both first-fit and best-fit are generally better than worst-fit in terms of decreasing time and storage utilization.

Title Slide I

Course: Fundamentals Operating System

- Unit: Memory Management
- Topic: Non-Contiguous Memory Allocation - Paging and Segmentation
- Lecture: 27

Introduction to Non-Contiguous Memory Allocation I

Non-contiguous memory allocation allows a process to be dispersed across various physical memory locations, overcoming the limitations of contiguous allocation schemes.

- Eliminates the problem of external fragmentation by allowing processes to use any available free memory chunks, irrespective of their location.
- Avoids the costly overhead of memory compaction required in dynamic contiguous allocation.
- Requires active hardware support for dynamic address translation during execution, adding to the system architecture's complexity.
- The two most prominent non-contiguous memory management techniques used in modern operating systems are Paging and Segmentation.

Overview of Paging I

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory by dividing both logical and physical memory into fixed-sized blocks.

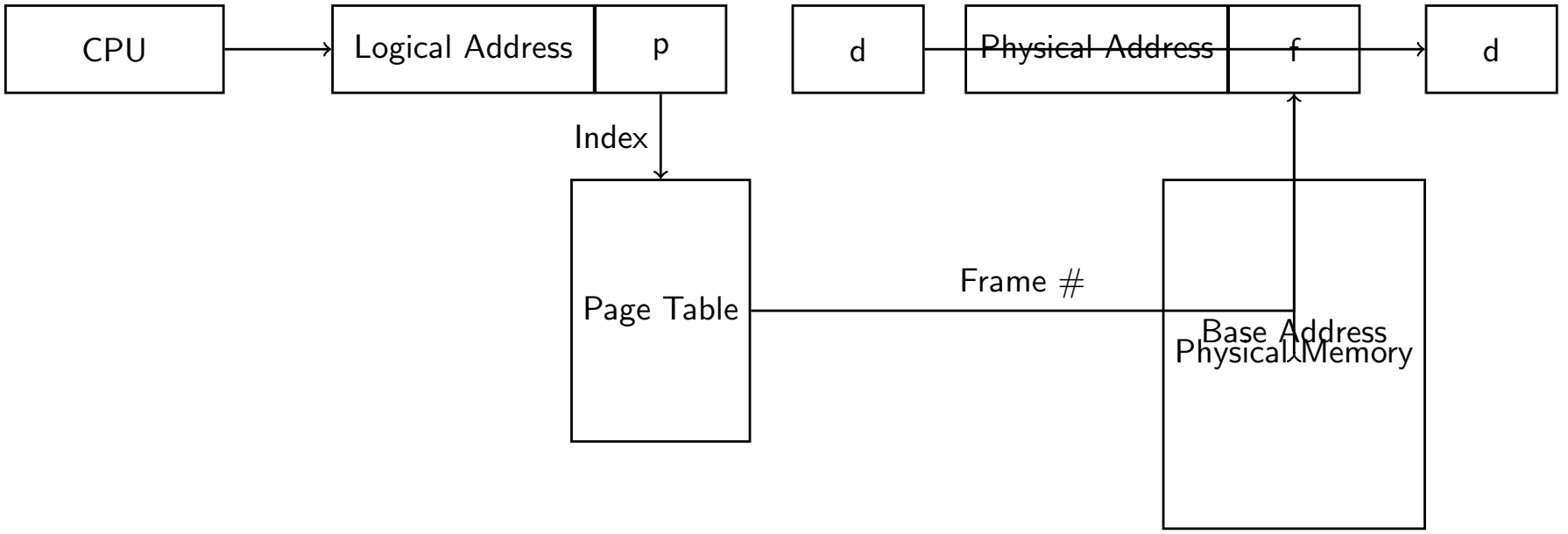
- Physical memory is broken down into fixed-sized blocks known as 'frames' (typically ranging from 512 bytes to 16 MB, depending on the architecture).
- Logical memory is divided into blocks of the same size called 'pages'.
- When a process is to be executed, its pages are loaded into any available memory frames from their backing store (disk).
- Since any page can fit into any frame, paging completely eliminates external fragmentation.
- However, paging can suffer from internal fragmentation on the very last page if a process's memory requirement is not an exact multiple of the page size.

Paging: Address Translation Mechanism I

The fundamental concept behind paging relies on a hardware-supported address translation mechanism that maps logical pages to physical frames.

- Every logical address generated by the CPU is divided into two distinct components: a page number (p) and a page offset (d).
- The page number (p) serves as a direct index into a system data structure called the 'page table'.
- The page table maps each page number to a base physical address, effectively providing the frame number (f) where the page resides in memory.
- The hardware combines the frame number (f) with the original page offset (d) to construct the final physical address sent to the memory unit.
- Physical Address computation: (Frame Number * Page Size) + Offset.

Paging: Address Translation Diagram I



- The CPU generates a Logical Address (p, d).
- The page number 'p' acts as an index into the Page Table.
- The Page Table yields the corresponding Frame number 'f'.
- The physical address is constructed by directly combining the frame number 'f' and the original offset 'd'.

Paging Hardware and the TLB I

Storing the page table in main memory slows down memory access. A Translation Look-aside Buffer (TLB) is used to accelerate the translation process.

- The Page-Table Base Register (PTBR) points to the page table stored in main memory. Therefore, every data/instruction access requires two memory reads (one for the table, one for data).
- To solve this 2x slowdown, an associative, high-speed hardware cache memory called the TLB is used to store recent page-table entries.
- When a logical address is generated, the TLB is searched for page 'p' concurrently. If found (TLB hit), the frame number is immediately extracted.

Paging Hardware and the TLB II

- If the page number is not found (TLB miss), a normal memory lookup into the page table is performed, and the retrieved entry is then cached in the TLB.
- Some TLB entries are wired down (cannot be removed), heavily utilized by critical OS kernel code.

Overview of Segmentation I

Segmentation is a memory-management scheme that directly supports the user's/programmer's logical view of memory, rather than partitioning memory strictly by fixed hardware sizes.

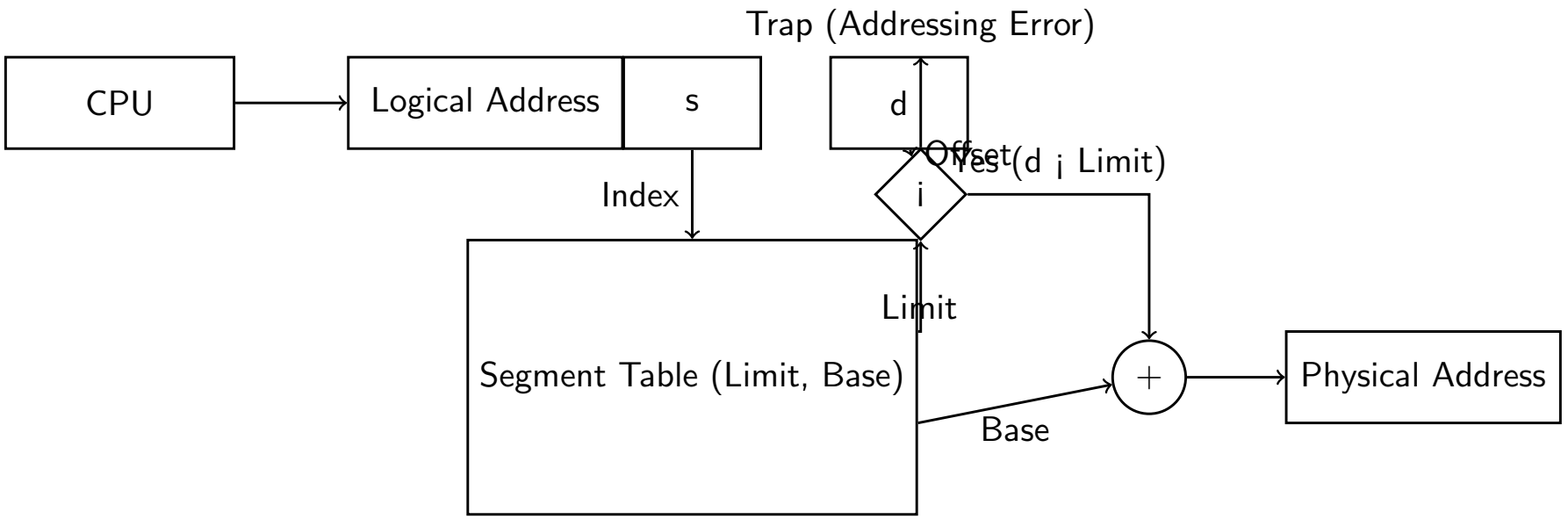
- A program is viewed as a collection of logically related segments (e.g., main program, global variables, stack, objects, arrays, symbol table).
- Unlike pages, segments are variable in size, reflecting the actual size of the logical structure they represent.
- A logical address in this scheme is a tuple: ¡segment-number, offset¿.
- Because segments vary in size, segmentation completely eliminates internal fragmentation.
- However, allocating contiguous variable-sized memory blocks for segments introduces external fragmentation as processes are swapped in and out.

Segmentation: Address Translation Mechanism I

Mapping two-dimensional logical addresses to one-dimensional physical addresses is accomplished via a segment table.

- Each entry in the segment table has a segment base and a segment limit.
- The 'segment base' stores the starting physical address where the segment resides in main memory.
- The 'segment limit' specifies the exact length of the segment.
- During translation, the segment number 's' indexes into the segment table to find the limit and base.
- The MMU hardware compares the offset 'd' against the segment limit. If d ¡ limit, the address is valid; otherwise, it triggers a trap (addressing error/segmentation fault).

Segmentation: Address Translation Diagram I



- The CPU produces a logical address containing Segment number (s) and Offset (d).
- Segment number 's' is used to look up the Base and Limit in the Segment Table.
- The hardware verifies that the requested offset is strictly less than the segment limit.
- If valid, the physical address is calculated mathematically as Base + Offset.

Paging vs. Segmentation I

While both prevent contiguous allocation constraints, they differ structurally and conceptually.

- Block Size: Paging uses fixed-size blocks (pages/frames); Segmentation uses variable-size blocks based on logical program units.
- Fragmentation: Paging suffers from internal fragmentation; Segmentation suffers from external fragmentation.
- User View: Paging is invisible to the user/programmer (handled entirely by hardware and OS); Segmentation is visible and aligns with the user's structural view.
- Security/Protection: Segmentation naturally allows independent protection levels per segment (e.g., read-only for code segments); Paging requires protection bits per page, grouping logically disjoint data.

Paging vs. Segmentation II

- Hybrid Systems: Modern OS architectures (like MULTICS or Intel x86) combine both into 'Paged Segmentation'—segments are formed and then divided into pages to eliminate external fragmentation.

Title Slide I

Course: Fundamentals Operating System

- Unit: Memory Management
- Topic 1: Overview of Paging and Address Translation
- Topic 2: Overview of Segmentation and Address Translation

Motivation: Contiguous vs. Non-Contiguous Allocation I

Contiguous memory allocation requires a single continuous block of physical memory for a process, which leads to external fragmentation.

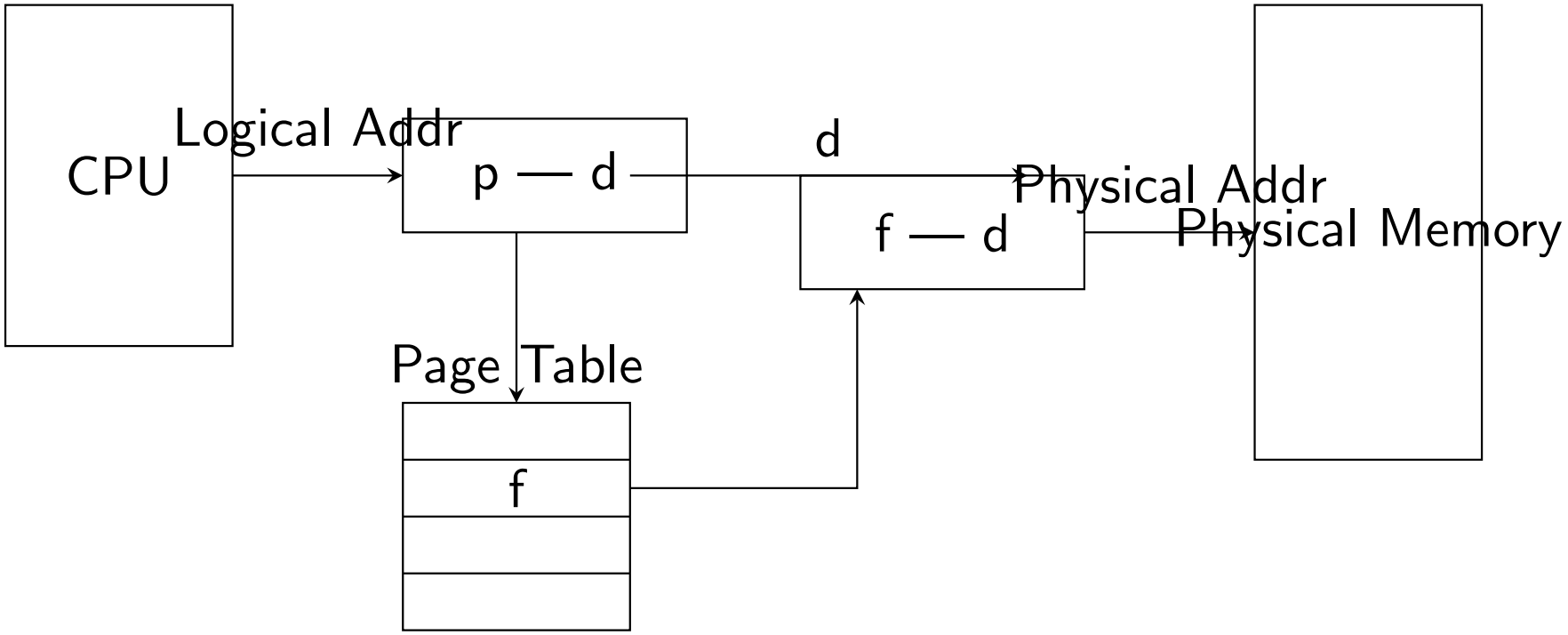
- External Fragmentation: Free memory spaces are scattered and too small to hold a new process, even if total free space is sufficient.
- Compaction can solve external fragmentation, but it involves immense overhead in moving running programs.
- Non-contiguous allocation divides a process into smaller chunks (pages or segments) that can be placed in any available physical memory locations.
- Benefits: Eliminates external fragmentation and simplifies memory allocation algorithms.

Overview of Paging I

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory.

- Physical memory is broken down into fixed-sized blocks called Frames.
- Logical memory (the process) is broken down into blocks of the same size called Pages.
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.
- The page size is determined by hardware architecture, typically a power of 2 ranging from 4 KB to 1 GB.
- The operating system maintains a Page Table to keep track of which page maps to which frame.

Paging Address Translation Architecture I



- Every logical address generated by the CPU is divided into two parts: a Page Number (p) and a Page Offset (d).
- The Page Number (p) is used as an index into the process's page table.
- The Page Table contains the base address (Frame Number, f) of each page in physical memory.

Paging Address Translation Architecture II

- The physical address is formed by appending the Page Offset (d) to the Frame Number (f).

Address Translation Details in Paging I

The Memory Management Unit (MMU) handles the dynamic translation of logical to physical addresses.

- If the logical address space is 2^m bytes, and the page size is 2^n bytes.
- The high-order m-n bits of a logical address designate the page number (p).
- The low-order n bits designate the page offset (d).
- No arithmetic operations (like addition) are required to calculate the physical address; bits are simply concatenated.
- The size of the page table is proportional to the size of the logical address space (number of pages).

Characteristics & Trade-offs of Paging I

While paging solves external fragmentation, it introduces its own set of structural trade-offs.

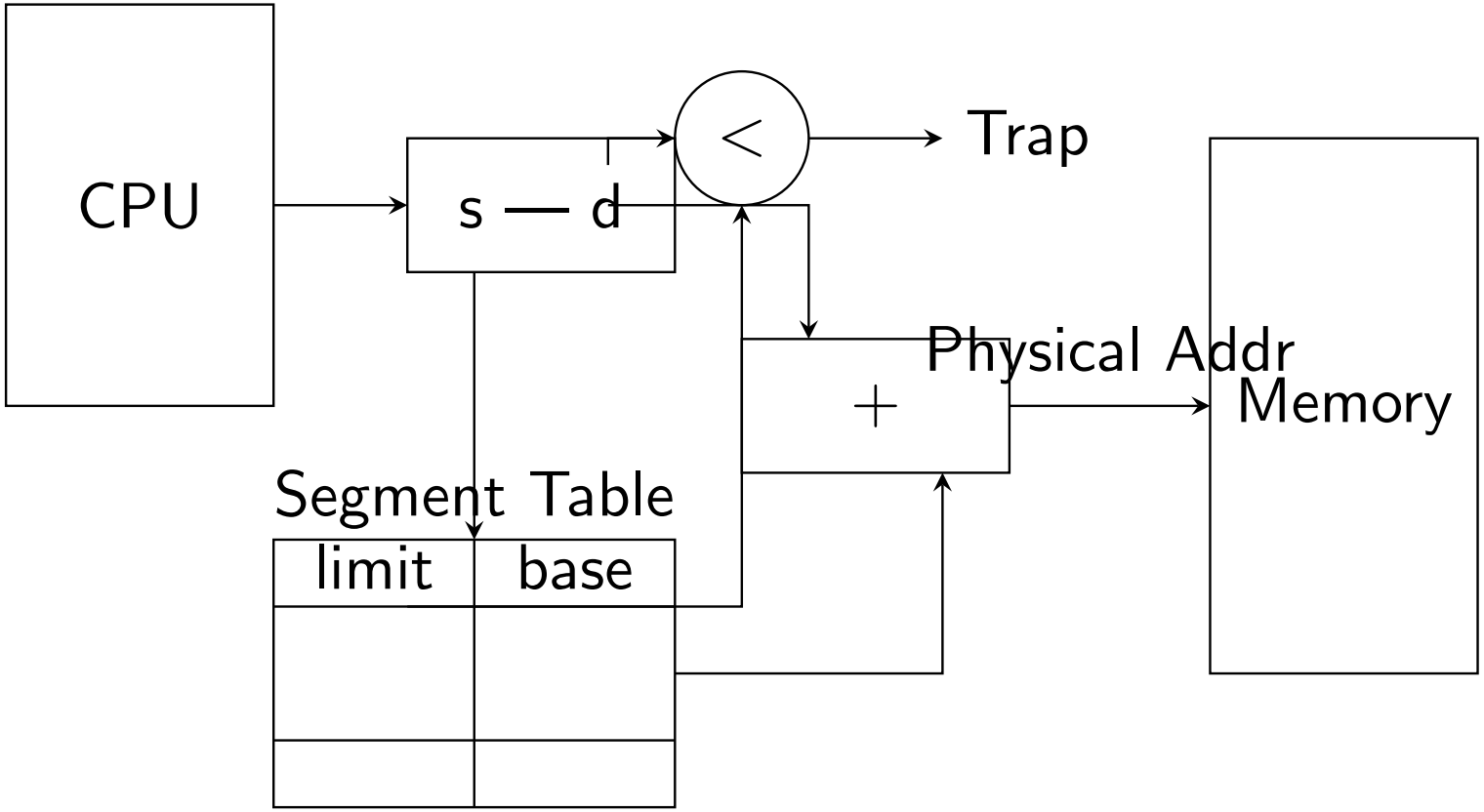
- Internal Fragmentation: The last page of a process may not be completely full. The remaining unused space in that frame is wasted.
- Memory Access Time penalty: Address translation requires an extra memory access to read the page table, effectively doubling access time.
- Translation Lookaside Buffer (TLB): A fast associative hardware cache used to store recently used page table entries, accelerating translation and mitigating the access penalty.
- Memory Protection: Easily achieved by associating protection bits (Read, Write, Execute) with each frame in the page table.

Overview of Segmentation I

Segmentation is a memory management scheme that aligns with the programmer's view of memory.

- A program is viewed as a collection of logical segments (e.g., main program, functions, arrays, stack, symbol table).
- Unlike pages which are fixed in size, segments are of variable sizes, strictly corresponding to logical program units.
- A logical address in this model consists of a two-tuple: ¡segment-number, offset¿.
- Segmentation naturally facilitates the sharing of code or data among multiple processes (e.g., shared dynamic libraries) at the semantic segment level.

Segmentation Address Translation Architecture I



- The CPU generates a logical address containing Segment Number (s) and Offset (d).
- The Segment Table acts as a mapping table. Each entry has a 'base' and a 'limit'.

Segmentation Address Translation Architecture II

- The 'base' contains the starting physical address where the segment resides in memory.
- The 'limit' specifies the length of the variable-sized segment.
- Hardware rigorously checks if d ¡ limit. If not, a trap (segmentation fault) is generated to prevent illegal memory access.
- Physical address is calculated dynamically by adding the base address and offset.

Characteristics & Details of Segmentation I

Segmentation provides strong logical isolation but reintroduces the issue of external fragmentation.

- Because segments vary in size depending on program structures, dynamically allocating memory for them leaves holes, leading to external fragmentation.
- Standard memory allocation strategies like First-fit, Best-fit, or Worst-fit must be employed to find contiguous chunks of physical memory for segments.
- Compaction can be used to merge free space and mitigate external fragmentation, but it imposes high computational overhead.
- Protection and sharing are intuitive and straightforward because a segment represents an isolated, semantically meaningful unit.

Comparative Analysis: Paging vs. Segmentation I

Paging and Segmentation offer distinct paradigms for memory management, and modern Operating Systems often hybridize them.

- Division: Paging divides memory into fixed-size physical blocks; Segmentation divides it into variable-size logical semantic blocks.
- Fragmentation: Paging is prone to internal fragmentation within frames; Segmentation is strictly prone to external fragmentation.
- Visibility: Paging is completely transparent to the user/programmer; Segmentation is visible to the programmer mapping logical constructs.
- Address Generation: Paging relies on hardware-level bit concatenation; Segmentation requires physical arithmetic addition.

Comparative Analysis: Paging vs. Segmentation II

- Hybrid Systems: Modern architectures (e.g., x86) combine both approaches using Paged Segmentation to leverage the logical benefits of segments without external fragmentation.

Title Slide I

Course: Fundamentals Operating System

- Unit: Memory Management
- Lecture 29: Page Replacement Algorithms - FIFO and LRU
- Focus: Managing physical memory effectively when page faults occur

Introduction to Page Replacement I

Page replacement is a critical component of virtual memory management that occurs when a page fault happens and there are no free frames available in main memory.

- When a process requests a page not currently in memory, a page fault is generated by the hardware.
- The OS must bring the requested page from secondary storage into a free physical frame.
- If all frames are occupied, an existing page in memory must be evicted to make room.
- The goal of a page replacement algorithm is to select a 'victim page' that minimizes future page faults.

The Page Replacement Process I

The mechanism involves several well-defined steps to swap out the victim and swap in the new page, updating memory management data structures accordingly.

- 1. Locate the desired page on the backing store (disk).
- 2. Find a free frame. If none exists, use a page replacement algorithm to select a victim frame.
- 3. Write the victim page to the backing store (if modified/dirty) and update the page table to mark it invalid.
- 4. Read the desired page into the newly freed frame.
- 5. Update the page table and restart the user process.

First-In, First-Out (FIFO) Algorithm I

The FIFO page replacement algorithm is the simplest to implement, treating the physical memory frames as a circular queue.

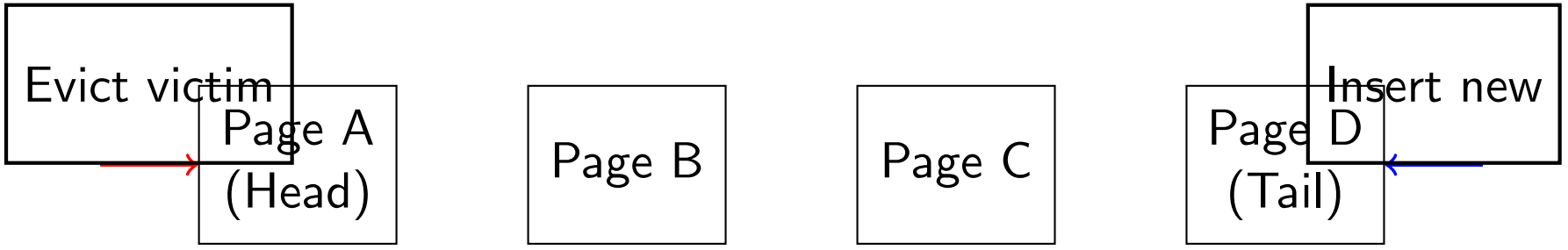
- FIFO associates with each page the time when that page was brought into memory.
- When a page must be replaced, the oldest page is chosen as the victim.
- Implementation typically uses a FIFO queue: new pages are added to the tail, and victims are selected from the head.
- Advantage: Low overhead and very simple implementation.
- Disadvantage: Its performance is often poor as it completely ignores the frequency or recency of page usage.

Belady's Anomaly I

A counter-intuitive phenomenon observed in some page replacement algorithms, particularly FIFO, where increasing the number of physical frames actually increases the number of page faults.

- Intuitively, providing a process with more memory (frames) should reduce the number of page faults.
- However, for specific reference strings (e.g., 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5), FIFO experiences more faults with 4 frames than with 3 frames.
- This occurs because FIFO does not possess the 'stack property' (stack algorithms never suffer from Belady's Anomaly).
- Algorithms like LRU and Optimal guarantee that increasing frame count never increases page faults.

FIFO Page Replacement Diagram I



- The diagram illustrates a standard FIFO queue used for page replacement tracking.
- New pages are inserted at the tail of the queue when brought into physical memory.
- When a page fault occurs and a victim is needed, the page at the head (the oldest) is indiscriminately evicted.
- This strict chronological eviction can remove a heavily used page simply because it was loaded early in the process.

Least Recently Used (LRU) Algorithm I

LRU is based on the principle of locality of reference, assuming that pages heavily used in the recent past will likely be used again in the near future.

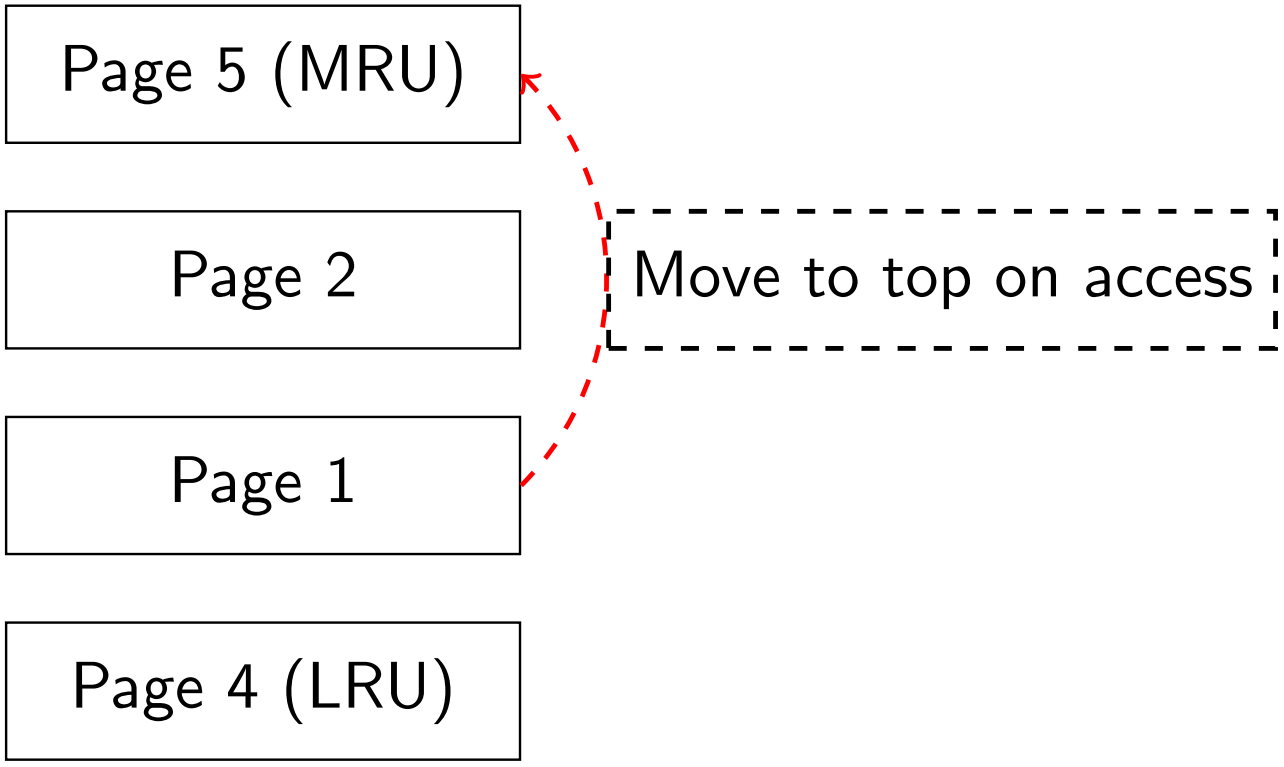
- LRU replaces the page that has not been used for the longest period of time.
- It actively associates with each page the exact time of its last use.
- When replacement is necessary, LRU chooses the page with the oldest last-use time.
- Advantage: Performs much better than FIFO and is mathematically proven to not suffer from Belady's Anomaly.
- Disadvantage: High hardware assistance is required (e.g., time-of-use counters or a stack) to track recency on every memory reference.

LRU Implementation Challenges I

True LRU is expensive to implement in software due to the constant overhead of updating reference data on every single memory access.

- Counter implementation: Add a logical clock to the CPU. Every page table entry has a time-of-use field. On access, copy the clock to this field. Search for the smallest value on replacement.
- Stack implementation: Keep a stack of page numbers. On reference, move the page to the top. The bottom page is the LRU victim. Requires modifying multiple pointers per access.
- Because of these high costs, modern operating systems often use LRU-approximation algorithms (like the Clock or Second-Chance algorithm) instead of true LRU.

LRU Stack Implementation Diagram I



- This diagram shows a stack-based implementation of the LRU algorithm using a doubly-linked list.
- The Most Recently Used (MRU) page is always kept at the top of the stack.
- The Least Recently Used (LRU) page, which is the prime victim candidate, naturally falls to the bottom of the stack.

LRU Stack Implementation Diagram II

- Whenever a page is accessed, it is immediately extracted from its current position and pushed to the top, meaning the bottom is always the LRU.

Conclusion and Comparison I

Selecting the right page replacement algorithm involves a trade-off between implementation complexity, hardware requirements, and performance (minimizing page faults).

- FIFO is simple to build but suffers from high page fault rates and Belady's Anomaly.
- LRU provides excellent performance by exploiting temporal locality, but exact tracking is often prohibitively expensive.
- The Optimal (OPT) algorithm replaces the page that will not be used for the longest time in the future; it's impossible to implement but used for theoretical benchmarking.
- Real-world systems compromise by employing LRU approximations to balance low runtime overhead with highly predictive replacement.

Title Slide I

Course: Fundamentals Operating System

- Unit: Memory Management
- Lecture 30: Page replacement algorithm - FIFO, LRU

Introduction to Page Replacement I

Page replacement is essential in demand paging when a page fault occurs and physical memory is full.

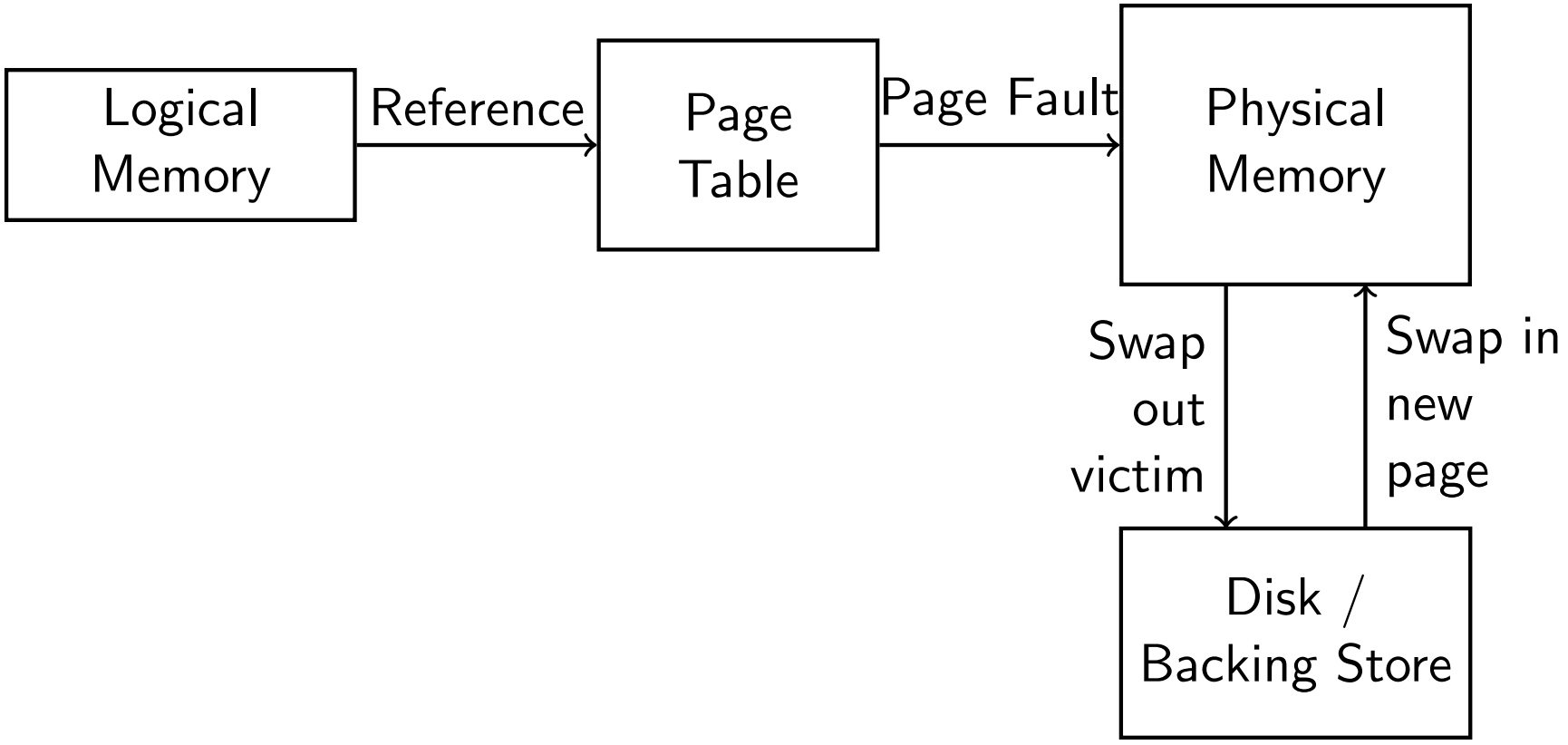
- Demand paging systems bring pages into memory only when they are referenced.
- If no physical frames are free, an existing page in memory must be selected as a victim and swapped out.
- The fundamental goal of a page replacement algorithm is to minimize the page-fault rate.
- Choosing a good victim page heavily impacts overall system performance by preventing thrashing.

The Page Replacement Process I

When a page fault happens without free frames, a standard sequence of steps is followed.

- 1. Find the location of the desired page on the disk (backing store).
- 2. Use a page-replacement algorithm to select a victim frame.
- 3. Write the victim frame to the disk and update the page and frame tables.
- 4. Read the desired page into the freed frame and update tables.
- 5. Restart the instruction that caused the page fault.

Diagram: Page Replacement Flow I



- The flow highlights the overhead of replacement when no free frames exist.
- Two disk accesses are required for replacement (swap out, swap in), effectively doubling the page-fault service time.

Diagram: Page Replacement Flow II

- A 'dirty bit' or 'modify bit' can optimize this by only writing the victim back if it has been modified.

First-In-First-Out (FIFO) Algorithm I

FIFO is the simplest page-replacement algorithm, operating strictly on arrival time.

- The OS maintains a queue of all pages currently in memory.
- When a page must be replaced, the oldest page (the head of the queue) is chosen as the victim.
- When a new page is brought into memory, it is inserted at the tail of the queue.
- Advantage: Extremely simple to understand and implement.
- Disadvantage: Performance is generally poor because it ignores how heavily a page is used; an active page might be swapped out just because it is old.

Belady's Anomaly I

FIFO is susceptible to an unexpected and counter-intuitive behavior known as Belady's Anomaly.

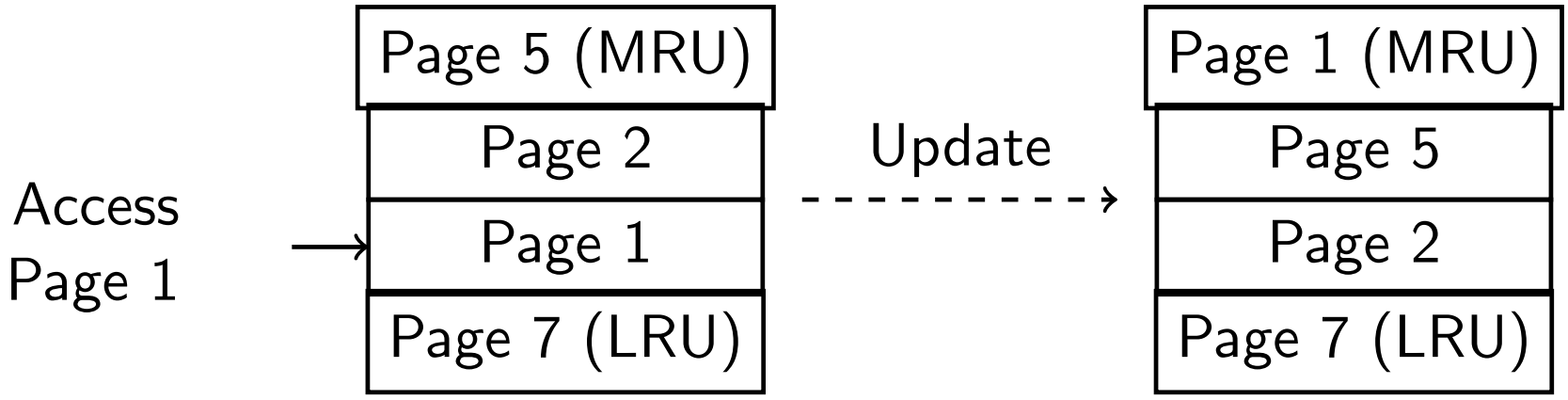
- Usually, allocating more physical frames to a process should result in fewer page faults.
- Belady's Anomaly is the phenomenon where increasing the number of page frames actually increases the number of page faults.
- This occurs because the FIFO queue merely cycles out the oldest page without regard for future reference patterns.
- Algorithms belonging to the 'stack algorithm' class (like LRU) are mathematically guaranteed never to exhibit Belady's Anomaly.

Least Recently Used (LRU) Algorithm I

LRU leverages the principle of temporal locality to make better replacement decisions.

- If a page has not been used for a long time, it is unlikely to be used in the near future.
- LRU replaces the page that has not been used for the longest period of time.
- It provides near-optimal performance and is frequently used as a benchmark.
- The main challenge is determining how to efficiently track historical page usage, as this requires hardware assistance on every memory access.

Diagram: LRU Implementation using a Stack I



- One method to implement LRU is using a stack of page numbers.
- Whenever a page is referenced, it is removed from its current position and pushed onto the top of the stack.
- The top always contains the Most Recently Used (MRU) page, while the bottom contains the victim (LRU).
- This avoids searching for a victim, but updating the stack on every reference requires expensive pointer manipulation.

LRU Implementation via Counters I

An alternative LRU implementation uses hardware counters or logical clocks.

- Every page-table entry is augmented with a time-of-use field, and a logical clock is added to the CPU.
- The CPU clock is incremented for every memory reference.
- Upon accessing a page, the current clock value is copied into the time-of-use field of its page-table entry.
- When replacement is needed, the system searches the page table for the entry with the smallest time value.
- Drawbacks include the cost of searching the page table and handling clock overflow.

Summary: FIFO vs LRU I

Both algorithms address the same problem but differ drastically in complexity and efficiency.

- FIFO is straightforward and requires minimal overhead, but blindly evicts old pages and risks Belady's Anomaly.
- LRU is highly effective at minimizing page faults by predicting future use based on past access.
- LRU requires significant hardware support (stacks or counters) making true LRU very expensive.
- In modern operating systems, hardware-supported approximations of LRU (like the Clock/Second-Chance algorithm) are used to balance cost and performance.

Title Slide I

Course: Fundamentals Operating System

- Unit: Files System

File Attributes I

A file is a named collection of related information recorded on secondary storage. Every file is characterized by its attributes.

- Name: The symbolic file name which is the only information kept in human-readable form.
- Identifier: A unique tag (usually a number) identifying the file within the file system.
- Type: Needed for systems that support different types of files.
- Location: A pointer to a device and to the location of the file on that device.
- Size: The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size.
- Protection: Access-control information determining who can read, write, execute, etc.

File Attributes II

- Time, date, and user identification: Data for creation, last modification, and last use (useful for protection, security, and usage monitoring).

File Operations I

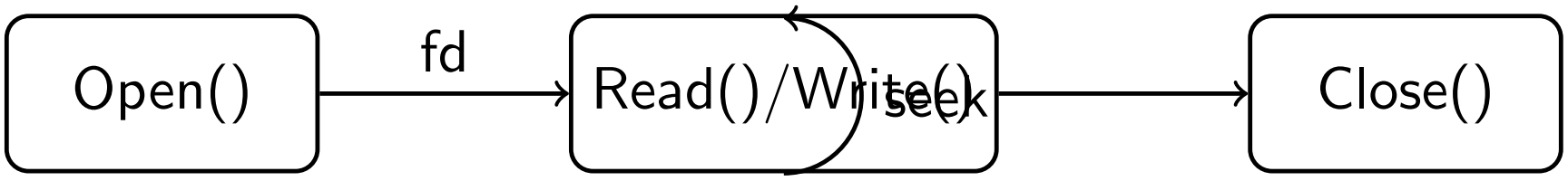
The operating system provides system calls to create, read, write, reposition, delete, and truncate files.

- Create: Allocate space in the file system and make an entry in the directory.
- Write: A system call specifying both the file name and the information to be written. The system maintains a write pointer.
- Read: A system call specifying the file name and where (in memory) the next block of the file should be put.
- Reposition (Seek): The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value.
- Delete: Search the directory for the named file, release all file space, and erase the directory entry.

File Operations II

- Truncate: Erase the contents of a file but keep its attributes and directory entry.

File Operations Workflow I



- To avoid searching the directory for every operation, most systems require an open() system call.
- The OS maintains an open-file table containing information about all open files.
- The file descriptor (fd) is used to access the file for read/write operations.

File Types I

A common technique for implementing file types is to include the type as part of the file name (extension).

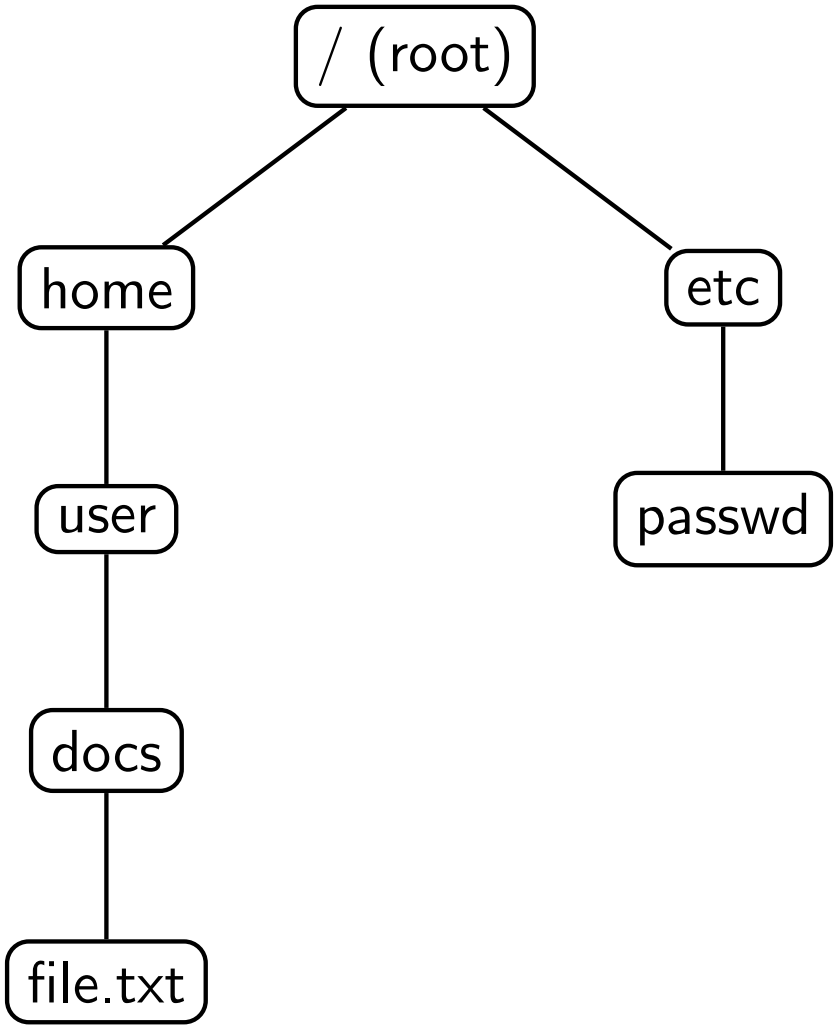
- Executable: ready-to-run machine-language program (e.g., .exe, .bin).
- Object: compiled, machine language, not linked (e.g., .o, .obj).
- Source code: source code in various languages (e.g., .c, .cpp, .java).
- Batch: commands to the command interpreter (e.g., .bat, .sh).
- Text: textual data, documents (e.g., .txt, .doc).
- Multimedia: binary files containing audio or video (e.g., .mp3, .mp4, .avi).
- Archive: related files grouped into one file, sometimes compressed (e.g., .zip, .tar).

File Paths: Absolute vs. Relative I

A path name is a sequence of directory names followed by the file name, specifying a file's location in the file system.

- Absolute Path: Begins at the root directory and follows a path down to the specified file, giving the unambiguous exact location.
- Example of Absolute Path: /home/user/docs/file.txt (in UNIX/Linux) or C:\{}Users\{}user\{}docs\{}file.txt (in Windows).
- Relative Path: Defines a path related to the current working directory.
- Example of Relative Path: If the current directory is /home/user, the relative path to the same file is docs/file.txt.
- The special dot (.) and double-dot (..) names refer to the current and parent directories, respectively.

Directory Structure Tree I



- The tree structure is the most common directory structure.
- Absolute path traverses from the root (/) downwards.

Directory Structure Tree II

- Relative paths navigate based on a starting directory other than root.

File Security and Protection I

File systems must implement protection mechanisms to control which users have access to which files.

- Protection mechanisms provide controlled access by limiting the types of file access that can be made.
- Types of access that can be controlled: Read, Write, Execute, Append, Delete, and List (for directories).
- Security is concerned with the environment the system operates in, protecting against unauthorized access, data destruction, or data alteration.
- The most common approach to the protection problem is to make access dependent on the identity of the user.

Protection Mechanisms: ACLs and Unix Permissions I

Different operating systems implement various methods to store and enforce access control information.

- Access-Control Lists (ACLs): Specifies user names and the types of access allowed for each user. Can be long and difficult to manage.
- Unix Permissions: A condensed version of an ACL classifying users into three categories: Owner, Group, and Universe (Others).
- For each category, Read (r), Write (w), and Execute (x) bits are maintained (e.g., rwxr-xr–).
- Capability Lists: Maintained per user, listing the objects (files) the user has access to and the permitted operations.
- Role-Based Access Control (RBAC): Assigns permissions to specific roles rather than individual users, simplifying administration.

Summary I

A brief overview of the key concepts regarding file systems and operations.

- Files are logical units of information storage managed by the OS, defined by various attributes (name, type, size, protection).
- Standard file operations include create, read, write, seek, and delete, facilitated via system calls like open().
- Files can be classified into different types, often designated by their file extensions.
- File paths can be absolute (starting from root) or relative (starting from the current directory).
- Protection is critical, typically implemented using access-control lists, UNIX-style permission bits, or capability lists.

Title Slide I

Course: Fundamentals Operating System

- Unit: Files System

What is a Directory? I

A directory is a symbol table that translates file names into their corresponding directory entries, which contain metadata or pointers to the files' data blocks.

- Serves as an index or symbol table for files.
- Provides a mapping from file names to file data.
- Contains attributes like name, location, size, and protection settings.
- Allows logical grouping of files by properties such as purpose or user.

Common Directory Operations I

Operating systems provide a standard set of system calls to manipulate directory structures.

- Search for a file: Finding a specific entry in the directory.
- Create a file: Adding a new directory entry.
- Delete a file: Removing an entry from the directory.
- List a directory: Enumerating all files within the directory.
- Rename a file: Changing the name attribute of an existing file.
- Traverse the file system: Accessing every directory and file systematically.

Single-Level Directory I

The simplest directory structure where all files are contained in the same directory, making it easy to support and understand.

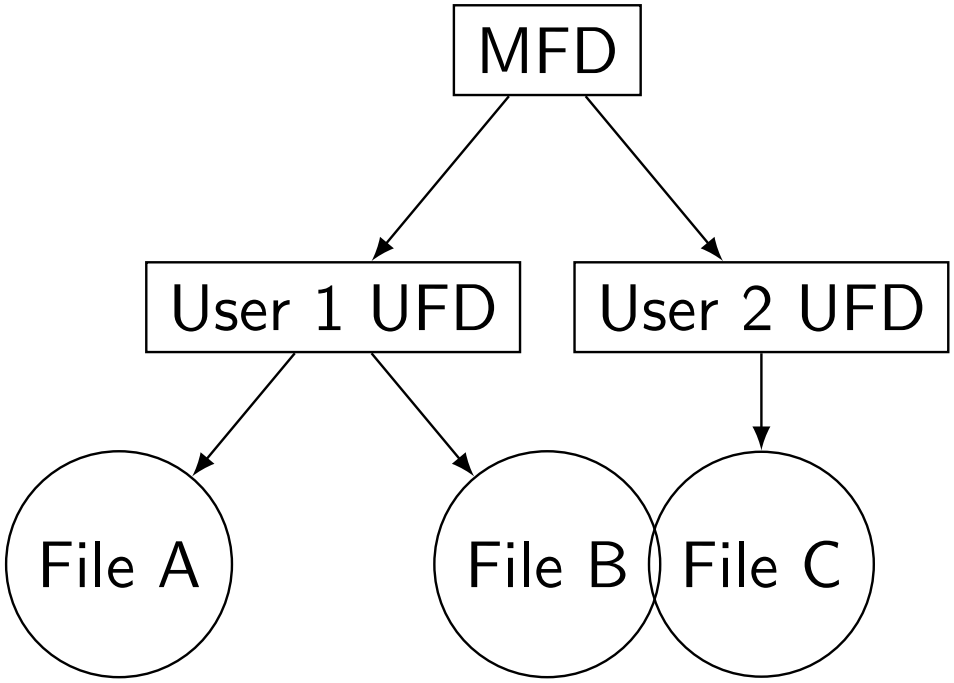
- All files reside in a single directory.
- Significant limitation: Naming collision (no two files can have the same name).
- Poor scalability: Difficult to manage as the number of files increases or in multi-user systems.
- Not suitable for modern operating systems, but used in early or simple embedded OS.

Two-Level Directory I

Introduces a Master File Directory (MFD) and User File Directories (UFD) to solve name collision across different users.

- Each user has their own User File Directory (UFD).
- Master File Directory (MFD) keeps track of all UFDs.
- Solves the name collision problem between different users.
- Path names are introduced (e.g., /user1/fileA).
- Isolates users from one another, which is good for privacy but complicates sharing.

Diagram of a Two-Level Directory I



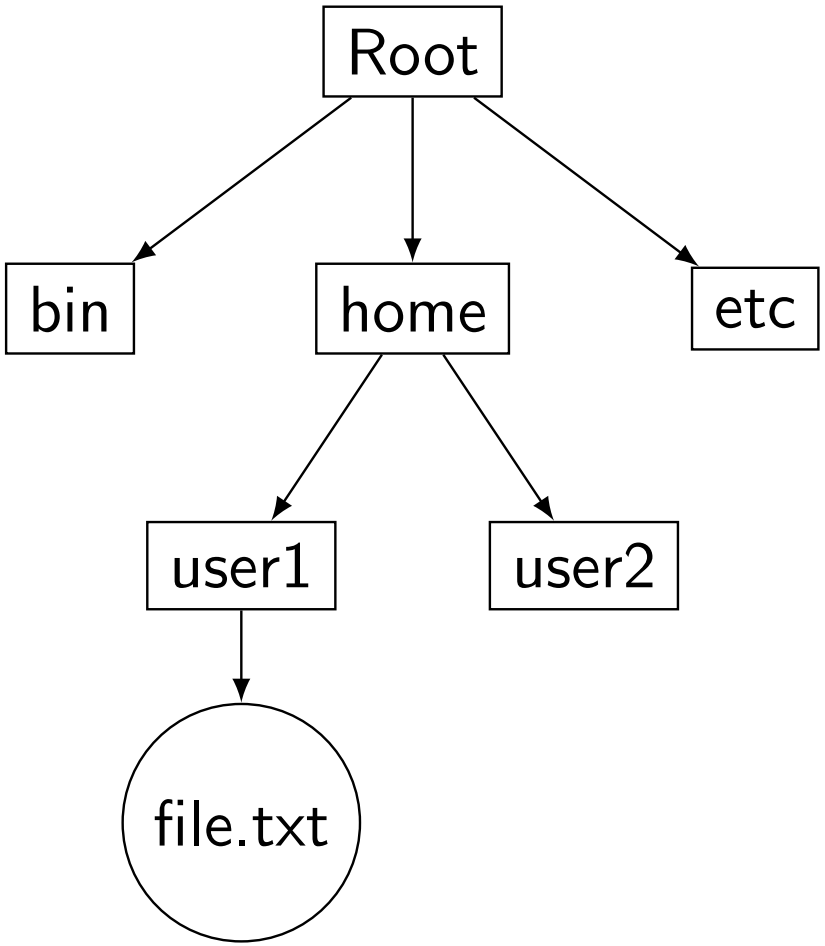
- The root represents the Master File Directory.
- Intermediate nodes represent User File Directories.
- Leaf nodes represent the actual files.
- Demonstrates isolation among users.

Tree-Structured Directory I

Extends the two-level directory to an arbitrary height, creating the familiar hierarchical file system.

- Most common directory structure used in modern operating systems (Windows, Linux, macOS).
- Directory contains both files and subdirectories.
- Uses absolute and relative pathnames for file access.
- Current Directory (working directory) concept is introduced for relative resolution.
- Efficient searching and grouping capability for users.

Diagram of a Tree-Structured Directory I



- Hierarchical structure with a single root node.
- Subdirectories can nest infinitely (up to filesystem limits).
- Nodes are either directories (internal) or files (leaves).

Diagram of a Tree-Structured Directory II

- Path resolution starts from root (absolute) or current node (relative).

Acyclic-Graph Directory I

A generalization of the tree structure that allows directories to share subdirectories and files, forming a Directed Acyclic Graph (DAG).

- Allows file sharing by linking (hard links and symbolic/soft links).
- Same file or subdirectory may be in two different directories.
- More complex to traverse; algorithms must avoid infinite loops (though cycles are not present, repeated visits can occur).
- Deletion requires reference counting (for hard links) to avoid dangling pointers.
- Used extensively in UNIX/Linux systems.

General Graph Directory I

Removes the restriction of acyclicity, allowing links to point to ancestor directories and creating potential cycles.

- Allows maximum flexibility in linking files and directories.
- Cycles can cause infinite loops in directory traversal and search operations.
- Requires complex garbage collection schemes to reclaim disk space, as reference counts may never reach zero.
- File systems often restrict the creation of hard links to directories to prevent cycles, essentially enforcing a DAG structure for hard links.

Title Slide I

Course: Fundamentals Operating System

- Unit: Files System

Introduction to Directory Structures I

A directory is a symbol table that translates file names into their corresponding directory entries. The directory itself is generally treated as a specialized file by the operating system.

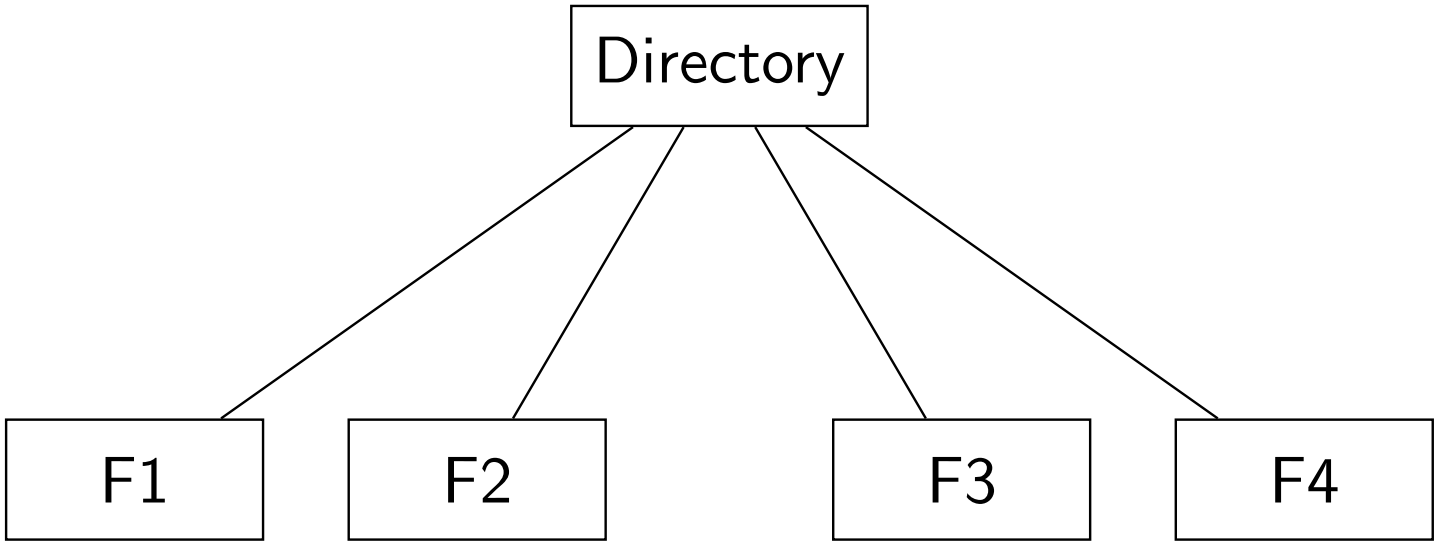
- A directory structure is required to organize files efficiently and logically on storage media.
- Key operations supported by directories: Search, Create, Delete, List, Rename, and Traverse.
- Different directory structures offer varying degrees of flexibility, naming capabilities, and implementation complexity.
- Choosing an appropriate directory structure directly impacts system performance, scalability, and user convenience.

Single-Level Directory I

The simplest directory structure design, where all files in the system are contained within the same, singular root directory.

- All system and user files coexist in one flat directory space.
- Files must have globally unique names; naming collisions occur frequently in multi-user environments.
- Very straightforward to implement and maintain, but scales exceptionally poorly as the file count increases.
- Does not support logical grouping of related files, rendering file management unwieldy for large systems.

Single-Level Directory Diagram I



- A single root node connects directly to all file nodes.
- Visually demonstrates the complete lack of hierarchical segregation.
- Illustrates why naming collisions become an immediate bottleneck when file count grows.

Two-Level Directory I

A structure that partitions the directory space by creating a separate directory for each user, resolving the primary naming collision issue.

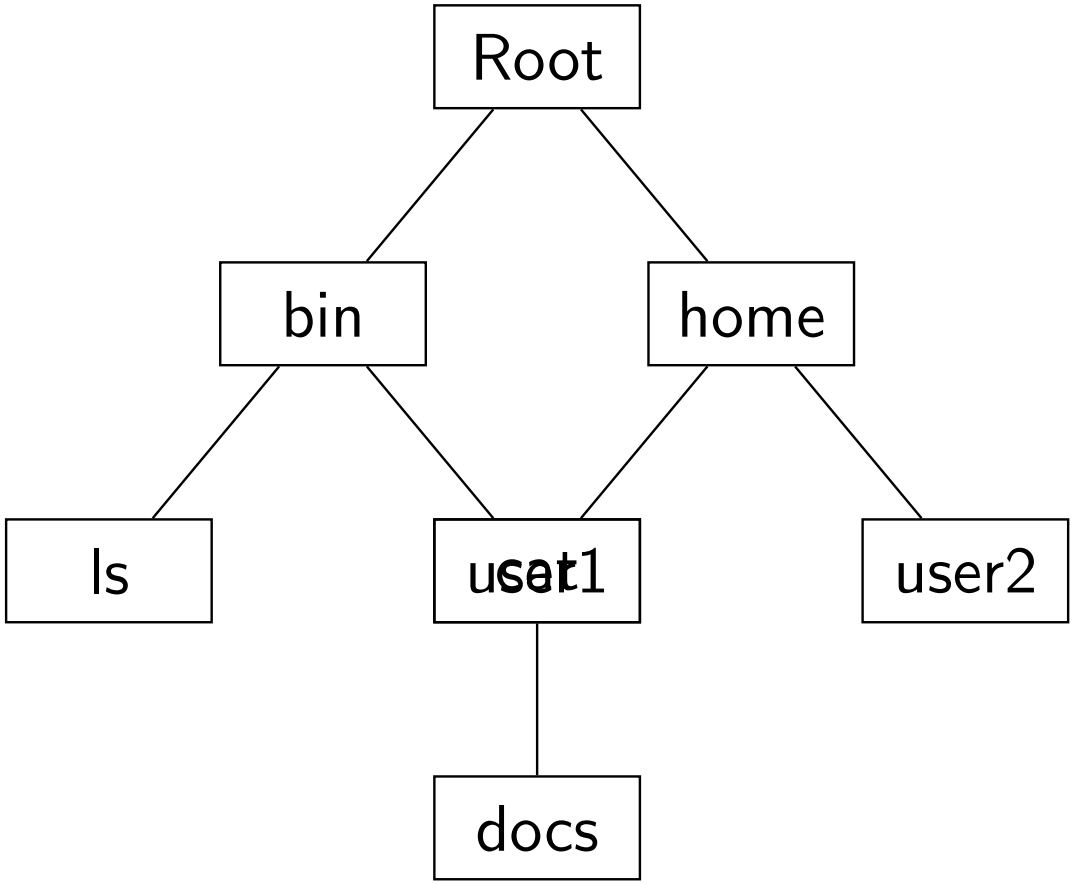
- Consists of a Master File Directory (MFD) that points to individual User File Directories (UFD).
- Different users can use the exact same file name without collision (e.g., User A and User B can both have a 'test.c').
- Provides basic security and isolation between different users' files.
- Still lacks the ability for users to internally group their own files, as each UFD remains a flat, single-level directory.

Tree-Structured Directory I

A hierarchical, tree-based structure that allows users to create arbitrary subdirectories, accommodating logical grouping and deep nesting of files.

- Generalizes the two-level directory concept to an arbitrary depth, creating a strict hierarchy.
- Utilizes absolute (from root) and relative (from current directory) pathnames for file resolution.
- Allows highly efficient file searching and logical categorization by projects or applications.
- Prohibits the sharing of files between directories; each file has exactly one parent node, forming a strict tree without cross-links.

Tree-Structured Directory Diagram I



- Depicts a strict hierarchy where every file and subdirectory (except root) has exactly one unique parent.
- Pathnames are formed by traversing from the root node down to the leaf nodes.

Tree-Structured Directory Diagram II

- Deleting a directory node mandates the cascading deletion of all its subdirectories and child files.

Acyclic-Graph Directory I

An extension of the tree structure that allows directories to share subdirectories and files, creating a Directed Acyclic Graph (DAG) rather than a strict tree.

- Enables seamless file and directory sharing between multiple users or group projects.
- Implemented using aliasing mechanisms like hard links or symbolic (soft) links.
- A single physical file may be resolved via multiple different absolute pathnames.
- Complicates file deletion: OS must employ reference counting to ensure actual file data is only removed when all links are deleted.

General Graph Directory Structure I

The most permissive directory structure that allows arbitrary linking between directories, including the creation of cycles.

- Provides maximum flexibility for interconnecting files and directories without topological restrictions.
- Cycles pose severe risks: recursive search or traversal algorithms can enter infinite loops.
- Garbage collection is highly complex; simple reference counting cannot reclaim isolated cyclical reference islands.
- Most modern OS (including UNIX-like systems) restrict users from hard-linking directories to strictly maintain an Acyclic Graph.

Summary & Performance Trade-offs I

Operating systems employ different directory architectures to balance simplicity, user isolation, organizational capability, and sharing.

- Single-level: Maximum simplicity, but completely lacks scalability and naming isolation.
- Two-level: Introduces basic user isolation, but limits internal file organization.
- Tree-structured: Solves deep organization requirements, but inherently prohibits file sharing.
- Acyclic Graph: The modern standard; balances organization and sharing, but complicates link resolution and deletion.
- General Graph: Offers ultimate flexibility, but introduces unmanageable risks with infinite loops and complex garbage collection.

Title Slide I

Course: Fundamentals Operating System

- Unit: File Systems
- Topic: Disk Space Allocation Methods
- Subtopics: Contiguous, Linked, Indexed

Introduction to Disk Space Allocation I

The primary goal of disk space allocation methods is to efficiently utilize disk space while allowing fast access to file contents.

- Secondary storage is a direct-access device, allowing data to be accessed in any order.
- Files are stored on disks in non-volatile storage, typically divided into fixed-size blocks (e.g., 4KB).
- An allocation method dictates how disk blocks are allocated for files.
- Key criteria for evaluating allocation methods include storage efficiency (minimizing fragmentation) and data access speed (sequential vs. direct access).
- The three major methods are contiguous, linked, and indexed allocation, each with distinct trade-offs.

Contiguous Allocation I

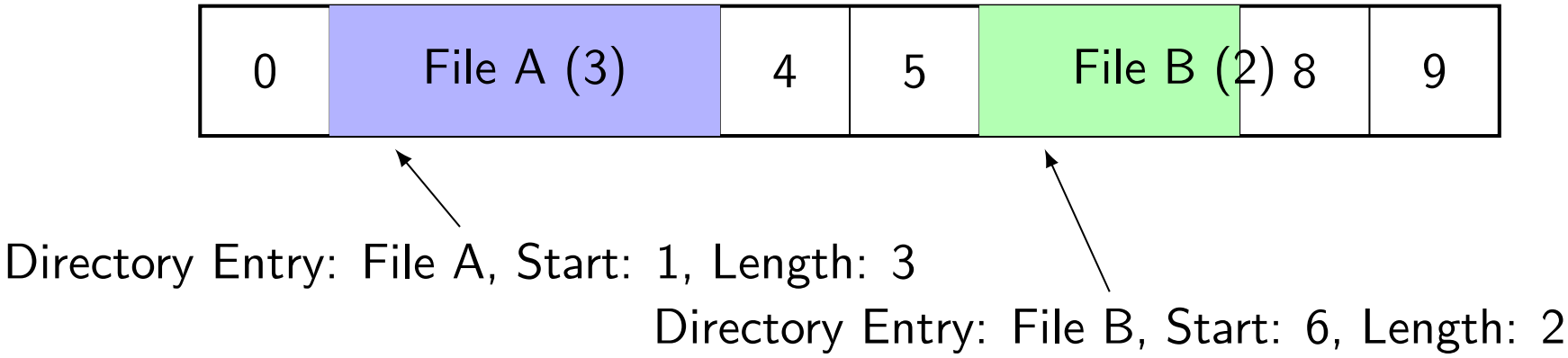
Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk.

- Disk addresses define a linear ordering on the disk, making sequential access very fast with minimal disk head movement.
- The directory entry for each file needs only the starting block address and the length (in blocks) of the area allocated to the file.
- Supports both sequential access and direct access efficiently. To access block i of a file starting at block b, we immediately access block b+i.
- Major drawback: Suffers from severe external fragmentation. As files are created and deleted, free disk space is broken into little pieces.
- Compaction can be used to resolve external fragmentation, but it is extremely time-consuming.

Contiguous Allocation II

- Another drawback is the difficulty in determining how much space is needed for a file when it is created, preventing easy file growth.

Contiguous Allocation Diagram I



- The directory contains the file name, starting block, and length of the allocation.
- Files occupy a sequential series of contiguous blocks, ensuring rapid data retrieval without disk seeks.

Linked Allocation I

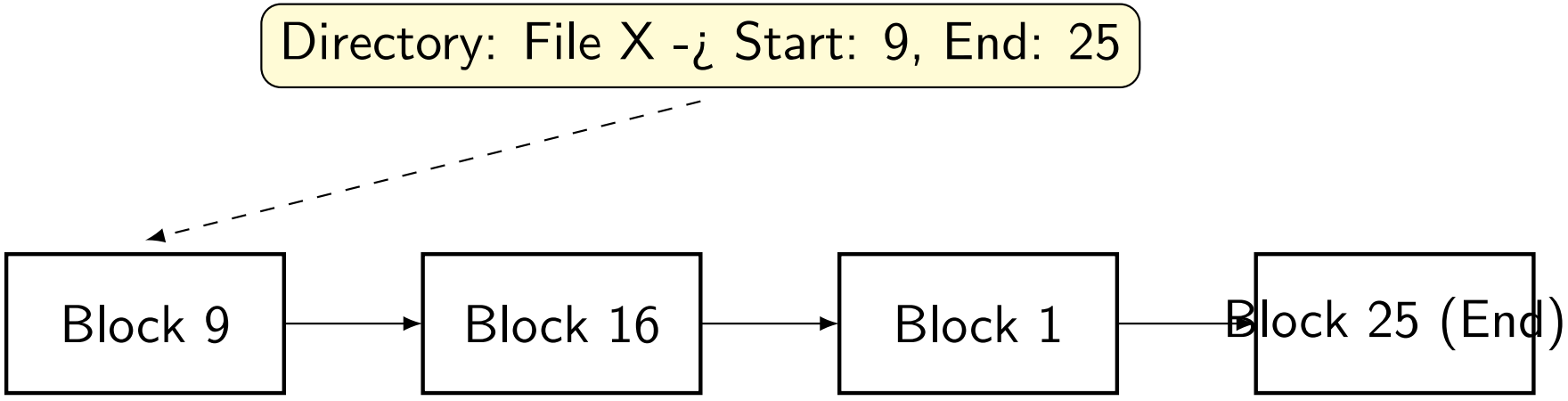
Linked allocation solves the external fragmentation and size-declaration problems of contiguous allocation.

- Each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk.
- The directory contains a pointer to the first and last blocks of the file.
- Each block contains a pointer to the next block. These pointers consume a small portion of each disk block.
- No external fragmentation occurs, as any free block can be used to satisfy a request. Files can grow easily.
- Major disadvantage: It is inefficient for direct access. Finding the i-th block requires traversing the linked list from the beginning.

Linked Allocation II

- Reliability is a concern; if a pointer is lost or damaged due to a bad disk sector, the remaining linked blocks become inaccessible.

Linked Allocation Diagram I



- The directory only needs to keep track of the first and last block addresses.
- Each individual block contains the payload data along with a pointer to the subsequent logical block in the file.

File Allocation Table (FAT) I

An important variation of linked allocation is the File Allocation Table (FAT), notably used in MS-DOS and OS/2.

- A section of disk at the beginning of each partition is set aside to contain the table.
- The 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.
- The table entry indexed by that block number contains the block number of the next block in the file.
- This chain continues until the last block, which has a special end-of-file value (EOF).
- Improves direct access significantly because the FAT can be cached in main memory, allowing fast traversal of pointers without physical disk I/O.

File Allocation Table (FAT) II

- However, if the FAT is not cached, finding a specific block can result in a significant number of expensive disk head seeks.

Indexed Allocation I

Indexed allocation brings all the file block pointers together into one central location called the index block.

- Solves the direct-access problem of pure linked allocation by providing direct access to any block of the file.
- Each file has its own index block, which is an array of disk-block addresses. The i-th entry points to the i-th data block of the file.
- The directory contains the address of the index block.
- To read the i-th block, the pointer in the i-th index block entry is used to find and read the desired block.
- Supports direct access efficiently and suffers from no external fragmentation, as any free block can be mapped into the index.
- Disadvantage: Wasted space due to index block overhead. Every file must have an index block, even very small files consisting of just one or two data blocks.

Indexed Allocation - Handling Large Files I

Handling files larger than what a single index block can map requires advanced hierarchical mechanisms.

- Linked Scheme: Link multiple index blocks together. The last word in the index block points to the next index block, rather than a data block.
- Multilevel Index: Use a first-level index block to point to a set of second-level index blocks, which in turn point to the actual file data blocks.
- Combined Scheme (UNIX INODE): Keep several direct block pointers in the inode structure itself for fast access to small files.
- The inode also contains a single indirect pointer, a double indirect pointer, and a triple indirect pointer to accommodate exceptionally large files.

Indexed Allocation - Handling Large Files II

- This elegant approach allows the UNIX file system to support files that are extremely large while keeping access to small files very fast and overhead low.

Comparison of Allocation Methods I

Different access patterns and file sizes dictate the optimal allocation method for a file system.

- Contiguous Allocation: Excellent for sequential and direct access, but suffers heavily from external fragmentation and inflexible file growth.
- Linked Allocation: Excellent for sequential access and completely avoids external fragmentation, but performs poorly for direct access and poses reliability risks.
- FAT (File Allocation Table): A linked variant that resolves the direct access problem if the FAT is small enough to be cached entirely in memory.
- Indexed Allocation: Excellent for direct access and avoids external fragmentation, but introduces space overhead due to index block storage.

Comparison of Allocation Methods II

- Modern File Systems: Often use combined or extent-based approaches (like ext4 extents or NTFS Master File Table) to balance the trade-offs of contiguous and indexed allocations.

Title Slide I

Course: Fundamentals Operating System

- Unit: Files System
- Lecture 35: Secondary Storage Structure: Physical Structure of Disk

Introduction to Secondary Storage I

Magnetic disks provide the bulk of secondary storage for modern computer systems.

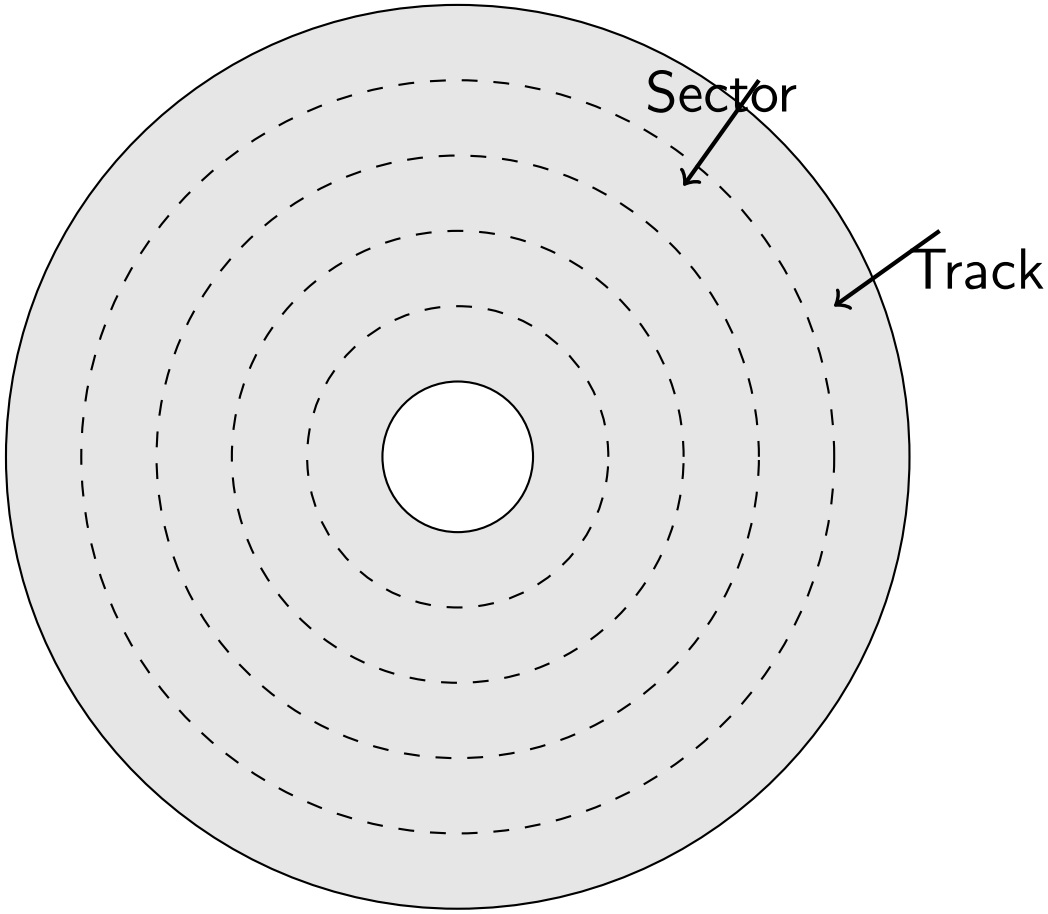
- Non-volatile memory used to store large amounts of data permanently.
- Slower than main memory (RAM) but significantly larger in capacity.
- Conceptually, disks are viewed as large one-dimensional arrays of logical blocks.
- The logical block is the smallest unit of transfer (typically 512 bytes or 4 KB).

Anatomy of a Magnetic Disk I

A magnetic disk drive consists of several mechanical and electronic components working together.

- Platters: Circular, rigid disks typically made of aluminum, glass, or ceramic, coated with magnetic material.
- Spindle: The central axis on which platters are mounted and rotated at high speeds (e.g., 5400 to 15000 RPM).
- Read/Write Heads: Coils that fly just above the surface of the platter to read or alter the magnetic polarity.
- Actuator/Disk Arm: Moves the read/write heads collectively across the platters.

Disk Geometry: Tracks and Sectors I



Tracks and Sectors

- Track: A concentric circular ring on the surface of a platter where data is recorded.

Disk Geometry: Tracks and Sectors II

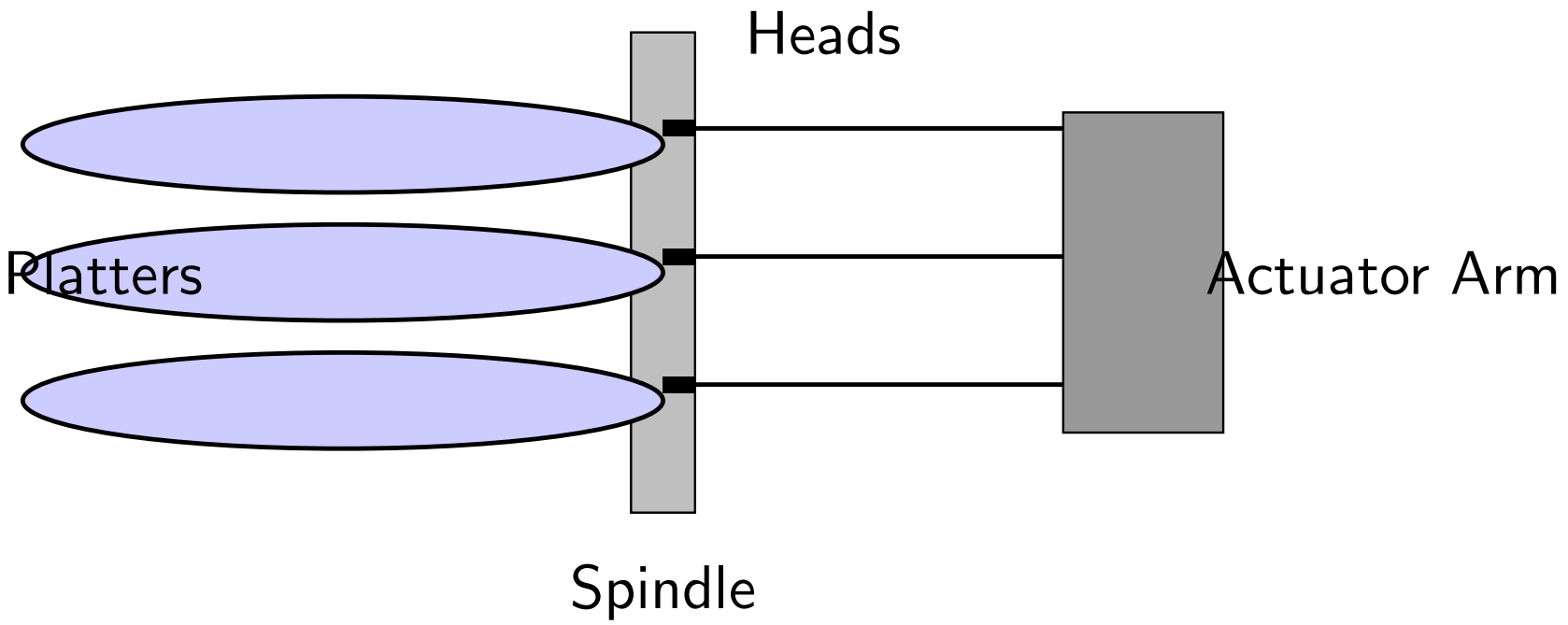
- Sector: A pie-shaped wedge dividing the tracks into smaller, addressable blocks of data (e.g., 512 bytes).
- Sector 0 is typically the first sector of the first track on the outermost cylinder.

Disk Geometry: Cylinders I

A cylinder is formed by all the tracks at a given radius across all platters in a disk drive.

- If a disk has 'n' platters, a cylinder consists of '2n' tracks (assuming both sides of each platter are used).
- Data is often written cylinder-by-cylinder to minimize the movement of the read/write heads.
- Moving heads to a different track (cylinder) is a slow mechanical operation called 'seeking'.

Disk Arm and Read/Write Heads I



- The disk arm (actuator assembly) moves all read/write heads simultaneously across the platters.
- Heads do not touch the platter; they fly on a microscopic cushion of air.
- A head crash occurs when the head makes physical contact with the platter surface, destroying data.

Disk Speed and Performance Metrics I

Disk performance is dominated by the mechanical movement of the disk arm and the rotation of the platters.

- Positioning Time (Random-Access Time): Total time taken to reach the desired data block.
- Transfer Rate: The rate at which data flows between the drive and the computer.
- Average access time is typically in the range of 5 to 15 milliseconds for magnetic disks.

Seek Time and Rotational Latency I

These are the two most significant delays in accessing data on a magnetic disk.

- Seek Time: The time necessary to move the disk arm to the desired cylinder (track). Average is about 3-9 ms.
- Rotational Latency: The time waiting for the desired sector to rotate under the read/write head.
- Average rotational latency is one-half of a full rotation (e.g., ˜4.16 ms for a 7200 RPM drive).
- Total Access Time = Seek Time + Rotational Latency + Transfer Time.

Disk Formatting: Physical and Logical I

Before a disk can store data, it must be structured and formatted physically and logically.

- Low-level formatting (Physical): Divides the disk into sectors that the disk controller can read/write. Adds headers, trailers, and Error-Correcting Codes (ECC).
- Partitioning: Dividing the disk into one or more cylinder groups (e.g., C: and D: drives), treated as individual logical disks.
- Logical formatting: The OS stores initial file-system data structures, such as FAT or inodes, onto the disk.

Summary and Key Takeaways I

Understanding the physical structure is crucial for OS disk scheduling algorithms.

- Magnetic disks are mechanically complex, relying on platters, spindles, and actuator arms.
- Data is organized hierarchically into cylinders, tracks, and sectors.
- Performance is severely limited by mechanical delays: seek time and rotational latency.
- OS disk scheduling algorithms aim to minimize seek time by optimizing the order of head movement.

Title Slide I

Course: Fundamentals Operating System

- Unit: Disk Management
- Lecture 36: Disk Scheduling Algorithm - SCAN, CSCAN

Introduction to Disk Scheduling I

Disk scheduling is the technique used by the operating system to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller.
- The primary goal is to minimize the seek time, which is the time taken for the disk arm to move the heads to the cylinder containing the desired sector.
- Secondary goals include minimizing rotational latency and ensuring fairness among requests to prevent starvation.

Recap: Limitations of FCFS and SSTF I

Before exploring SCAN and C-SCAN, it is important to understand why earlier algorithms like First Come First Serve (FCFS) and Shortest Seek Time First (SSTF) are insufficient.

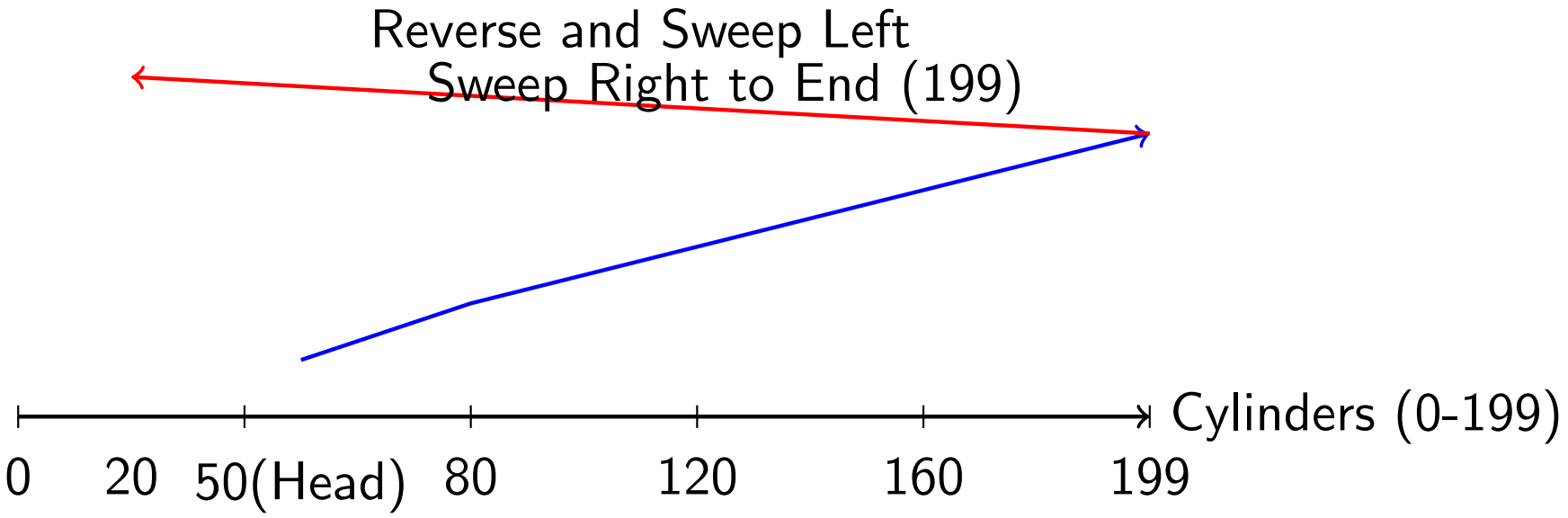
- FCFS serves requests in the exact order they arrive, leading to wild swings of the disk head and poor performance (high average seek time).
- SSTF selects the pending request closest to the current head position. While this reduces average seek time significantly compared to FCFS, it suffers from a major flaw: starvation.
- In SSTF, a continuous stream of requests localized to a specific region can cause distant requests to wait indefinitely, making it unsuitable for highly loaded systems.

SCAN Algorithm (Elevator Algorithm) I

The SCAN disk scheduling algorithm is often referred to as the elevator algorithm due to its similar operating principle to a building elevator.

- The disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder.
- Once it reaches the end of the disk, the direction of head movement is reversed, and servicing continues in the opposite direction.
- This bidirectional sweep ensures that all regions of the disk are covered, eliminating the starvation problem inherent in SSTF.
- If a request arrives just behind the head's current position, it must wait for the head to reach the end of the disk and sweep back, leading to variable wait times.

SCAN Algorithm Diagram I



- The diagram illustrates a sequence of requests. The head starts at cylinder 50 and moves towards the higher cylinder numbers.
- It services requests at 80, 120, and 160 until it hits the absolute boundary at 199.
- After reaching 199, the head reverses direction and services the remaining request at 20.
- Total head movement is calculated as (199 - 50) + (199 - 20) = 328 cylinders.

Pros and Cons of SCAN I

While SCAN effectively resolves the most glaring issues of SSTF, it introduces its own set of performance characteristics.

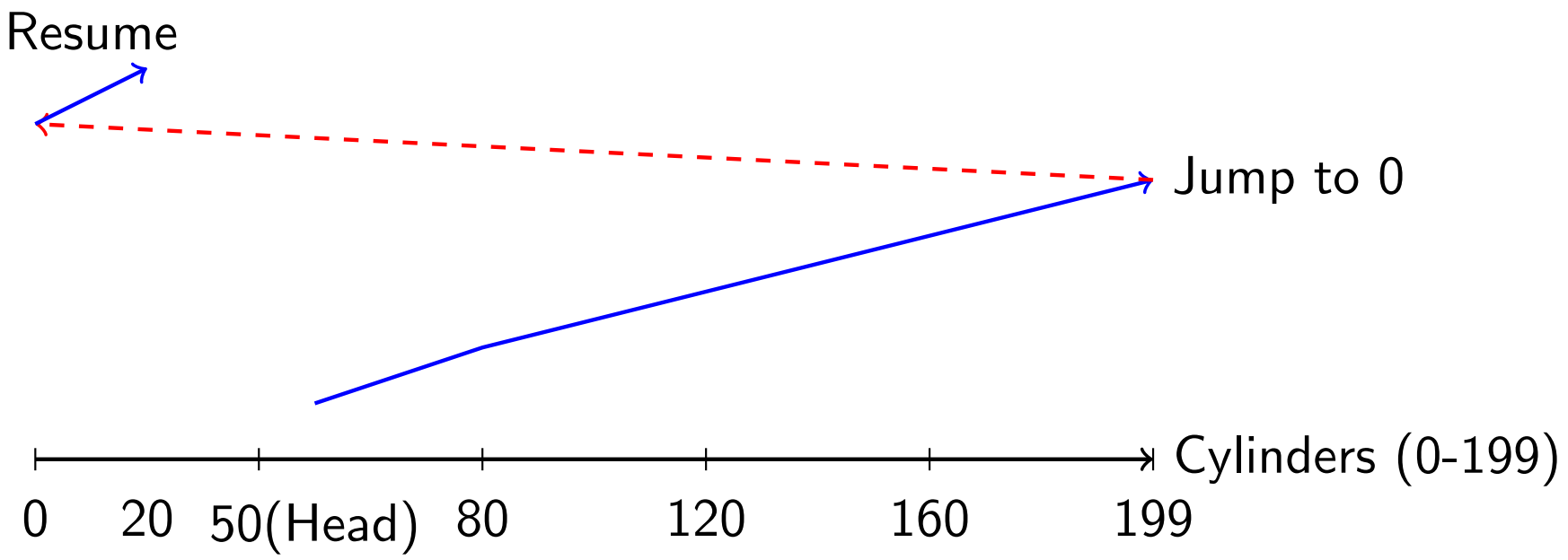
- Advantage: Simple, easy to understand, and completely eliminates the starvation problem for all disk requests.
- Advantage: Variance in response time and average response time is lower compared to SSTF in heavily loaded scenarios.
- Disadvantage: The algorithm forces the head to travel to the very end of the disk, even if there are no requests pending in that extremity.
- Disadvantage: Cylinders in the middle of the disk get serviced twice as often (once in each direction) compared to the innermost and outermost cylinders, creating an unfair waiting time distribution.

C-SCAN (Circular SCAN) Algorithm I

C-SCAN is a variant of SCAN designed to provide a more uniform wait time by treating the cylinders as a circular list that wraps around from the final cylinder to the first one.

- Like SCAN, the head moves from one end of the disk to the other, servicing requests along the way.
- However, when the head reaches the end of the disk, it immediately returns to the beginning of the disk WITHOUT servicing any requests on the return trip.
- After returning to the beginning, it resumes servicing requests moving in the original direction.
- This approach effectively treats the disk as if it were a single circular track, significantly reducing the maximum delay experienced by new requests arriving at the other end of the disk.

C-SCAN Algorithm Diagram I



- The head starts at 50, moving towards 199, servicing 80, 120, and 160.
- Upon reaching the end (199), a rapid seek occurs directly to cylinder 0 (dashed red line). No requests are serviced during this return trip.
- After resetting to cylinder 0, the head resumes its sweep in the original direction, finally servicing the request at 20.

C-SCAN Algorithm Diagram II

- Total head movement includes the full jump from 199 to 0, though seek time for a continuous sweep back may be optimized by the hardware.

Comparing SCAN and C-SCAN I

Choosing between SCAN and C-SCAN depends on the specific workload and the importance of predictable response times.

- Throughput: Both algorithms offer excellent throughput, far exceeding FCFS and avoiding the pitfalls of SSTF.
- Variance in Wait Time: C-SCAN provides a much more uniform waiting time. In SCAN, requests at the edges wait longer than requests in the middle. C-SCAN treats all cylinders equally.
- Total Head Movement: SCAN typically results in slightly less total head movement because C-SCAN includes the 'empty' return sweep across the entire disk.
- Fairness: C-SCAN is fundamentally fairer, ensuring that a cluster of requests at one end doesn't disproportionately delay requests at the other end.

Summary and Variations I

SCAN and C-SCAN form the foundation for modern disk scheduling, with practical implementations often utilizing optimized variations.

- LOOK and C-LOOK: These are practical enhancements to SCAN and C-SCAN. Instead of traversing to the absolute edges of the disk (e.g., 0 and 199), the head only moves as far as the furthest request in each direction.
- LOOK prevents the unnecessary traversal to the disk edges when no requests exist there, saving significant time.
- Operating systems often dynamically switch algorithms based on current load, using SSTF for light loads and switching to LOOK/C-LOOK under heavy I/O pressure.

Summary and Variations II

- Modern storage devices like SSDs use highly specialized internal controllers (FTL - Flash Translation Layer) where traditional mechanical scheduling like SCAN is less relevant, but the concepts remain crucial for HDD management.

Title Slide I

Course: Fundamentals Operating System

- Unit: Unix/LinuxOperatingSystem

Introduction to Unix and Linux I

Linux is a family of open-source Unix-like operating systems based on the Linux kernel, an operating system kernel first released on September 17, 1991, by Linus Torvalds.

- Unix was originally developed in 1969 at AT&T's Bell Labs by Ken Thompson, Dennis Ritchie, and others.
- Linux was created by Linus Torvalds as a free and open-source alternative to MINIX, heavily inspired by Unix design principles.
- Linux is typically packaged in a Linux distribution (e.g., Ubuntu, Fedora, Debian) which includes the kernel and supporting system software and libraries.
- The philosophy of Linux emphasizes modularity, stability, security, and the paradigm that 'everything is a file'.

Key Design Principles of Unix/Linux I

The architecture of Linux adheres strictly to the fundamental design principles established by early Unix systems to maintain simplicity and power.

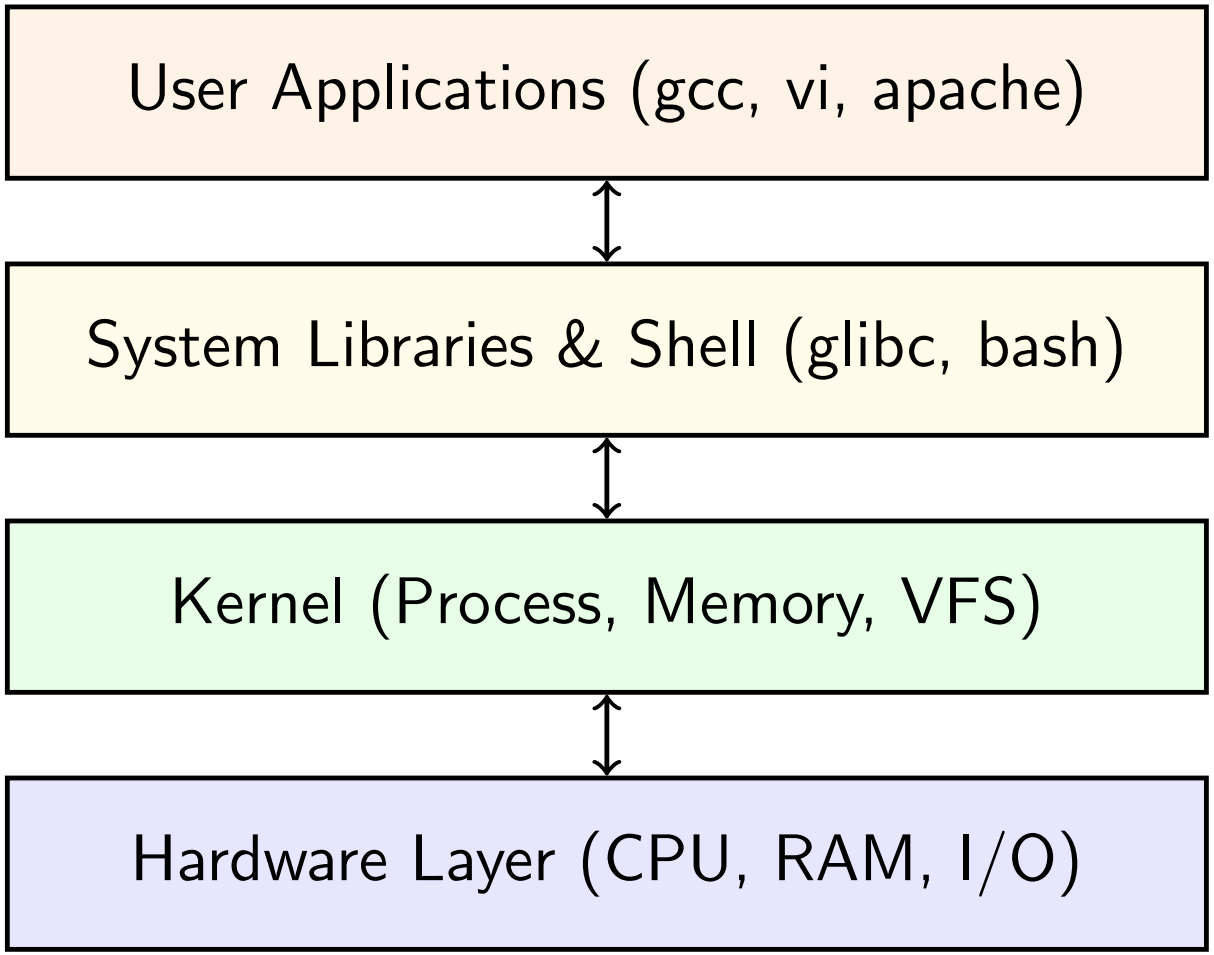
- Modularity: The system consists of small, simple programs that perform a single task well and can be piped together.
- Single Hierarchical File System: All files and directories are located under a single root directory (/), regardless of the physical storage layout.
- Everything is a File: Hardware devices, regular files, directories, sockets, and even active processes are represented as files.
- Multi-user and Multitasking: Multiple users can access system resources concurrently, and multiple processes can run preemptively.

Core Components of the OS I

A typical Linux/Unix operating system is composed of several distinct functional layers that isolate hardware interaction from user-level applications.

- Hardware: The physical components such as CPU, RAM, Disk, and I/O devices.
- Kernel: The core of the OS that operates in a privileged mode, interacting directly with hardware and managing resources.
- Shell and System Libraries: The interface between user applications and the kernel, providing a standard set of functions (e.g., glibc).
- User Applications: Programs such as text editors, compilers, and command-line utilities running in an unprivileged user space.

Basic Architecture Overview I



- The architecture is highly layered, providing strong isolation and abstraction between components.

Basic Architecture Overview II

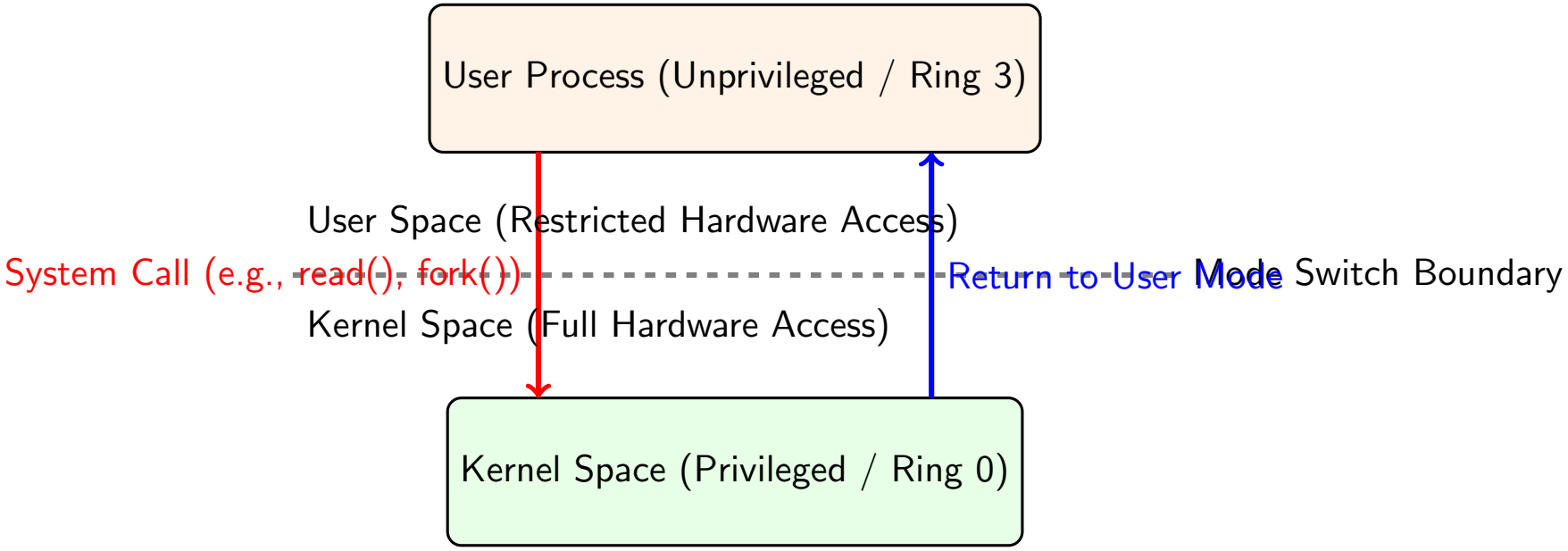
- User applications communicate with the kernel primarily through system library APIs rather than direct hardware access.
- The kernel translates standardized system calls into specific machine instructions executed by the physical hardware.

The Linux Kernel Deep Dive I

The kernel is the most fundamental part of a Linux operating system. It acts as a resource manager and a bridge between applications and hardware.

- Process Management: Handles creation, preemptive scheduling, and termination of processes, allocating CPU time.
- Memory Management: Manages physical RAM, virtual memory abstraction, paging, and swapping to disk.
- Device Drivers: Contains code to interact with specific hardware controllers, presenting a unified interface to the OS.
- File System Management: Provides the Virtual File System (VFS) layer to support multiple underlying filesystem types (ext4, XFS).
- Network Stack: Implements core network protocols (TCP/IP) for socket-based communication.

System Calls and Execution Modes I



- Modern architectures use CPU privilege rings to protect critical OS functions from rogue user processes.
- Linux uses User Mode for everyday applications and Kernel Mode for executing core OS code.
- A system call triggers a software interrupt (trap), safely transitioning CPU execution from User Mode to Kernel Mode.

The Shell and Command Line Interface I

The shell is a command-line interpreter that provides a traditional textual user interface for interacting with the operating system.

- It accepts commands typed by the user, parses them, forks new processes, and executes the requested programs.
- Common Unix/Linux shells include Bash (Bourne Again SHell), Zsh, and tcsh.
- The shell itself is not part of the kernel; it is simply a standard user-space application.
- It supports powerful features like input/output redirection (¿, ¡), pipes (—) for chaining commands, and shell scripting.

The File System Hierarchy Standard (FHS) I

Linux systems use the Filesystem Hierarchy Standard (FHS) to define a consistent directory structure and dictate where files should be placed.

- / (Root): The top-level directory containing the entire filesystem hierarchy.
- /bin and /sbin: Contain essential user command binaries and system administration executables.
- /etc: Stores host-specific system-wide configuration files.
- /dev: Contains special device nodes (character and block devices) representing hardware.
- /proc and /sys: Virtual filesystems maintained by the kernel that provide real-time access to kernel data structures and system info.

Summary and Conclusion I

Linux is a powerful, flexible, and robust operating system architecture derived directly from classic Unix philosophies.

- The strict separation of User Space and Kernel Space ensures system stability, preventing a single crashed app from halting the OS.
- The 'everything is a file' abstraction dramatically simplifies the programmatic interface for developers.
- The highly layered, modular architecture allows hardware, the kernel, and userland applications to be developed and updated independently.
- Understanding this foundational architecture is essential for systems programming and advanced operating system concepts.

Title Slide I

Course: Fundamentals Operating System

- Unit: Unix/Linux Programming

Introduction to Unix/Linux Programming I

Unix/Linux programming primarily involves interacting with the operating system through system calls and library functions to manage system resources.

- The POSIX standard defines the API for Unix-like operating systems, ensuring portability across different environments.
- C is the primary language for systems programming in Linux due to its low-level memory access and performance efficiency.
- System programming encompasses critical tasks like process management, file I/O operations, IPC, and network communication.
- A deep understanding of the execution boundary between user space and kernel space is essential for writing secure and efficient code.

System Calls vs. Library Functions I

System calls are direct requests to the kernel, whereas library functions are wrappers executing in user space that may or may not invoke system calls.

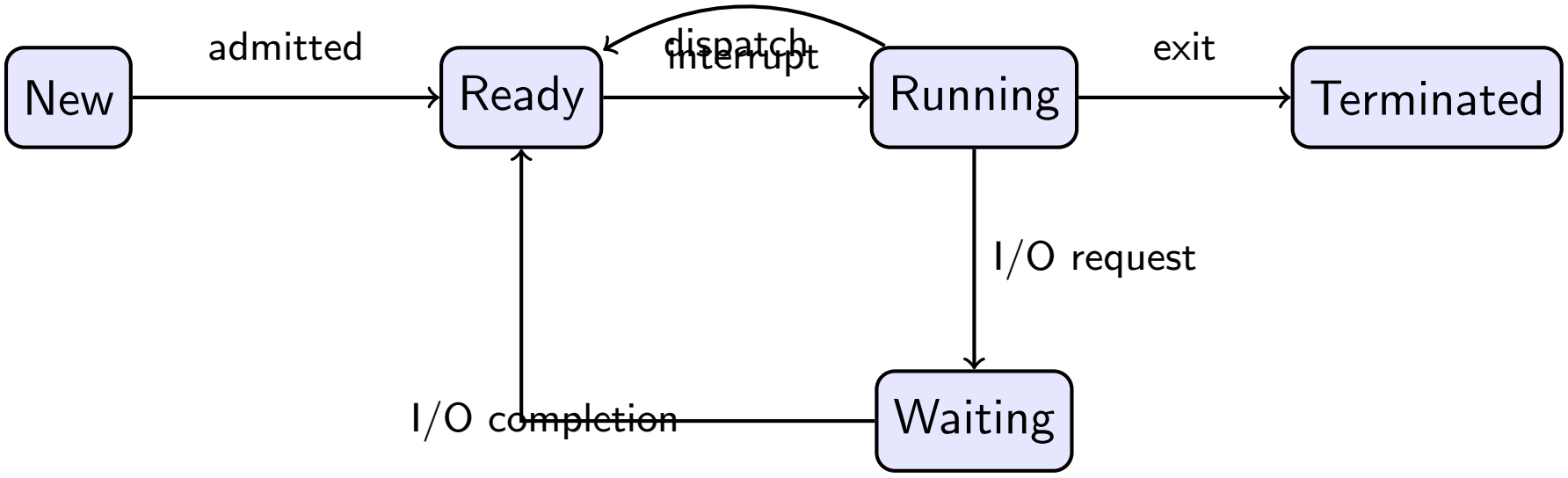
- System Calls (e.g., read(), write(), fork()) trigger a software interrupt to perform a context switch from user mode to kernel mode.
- Library Functions (e.g., printf(), fopen(), malloc()) provide buffered, easier-to-use interfaces on top of raw system calls.
- A standard library function like printf() performs internal buffering and eventually calls the write() system call to output data.
- Context switches introduced by system calls incur CPU overhead, which buffered library functions attempt to minimize.

Process Control: Fork, Exec, and Wait I

The lifecycle of a process in Unix is predominantly controlled by the fork(), exec(), and wait() family of system calls.

- fork() creates a new process (child) by duplicating the calling process (parent), returning 0 to the child and the child's PID to the parent.
- The exec() family (e.g., execl, execvp) replaces the current process memory image with a new executable program, maintaining the same PID.
- wait() and waitpid() allow a parent process to pause execution until its child terminates, harvesting the exit status.
- Zombie processes occur when a child terminates but its parent has not yet waited for it, leaving an uncollectable entry in the system process table.

Process State Transition Diagram I



- Processes transition between New, Ready, Running, Waiting (Blocked), and Terminated states.
- The kernel scheduler decides which Ready process gets CPU time, transitioning it to the Running state.
- The Waiting state is entered when a process requests I/O or explicitly calls a sleep/wait function.
- Hardware interrupts or expired time slices can preempt a Running process, returning it to the Ready queue.

Inter-Process Communication (IPC) I

Since processes have fully isolated memory spaces, the kernel provides IPC mechanisms for them to exchange data and synchronize execution.

- Pipes (unnamed) provide unidirectional byte streams exclusively between related processes (e.g., parent and child).
- FIFOs (named pipes) exist as special files in the file system and allow unrelated processes to communicate.
- Shared Memory (POSIX or System V) allows multiple processes to map the same physical memory segment, offering the fastest IPC.
- Message Queues provide discrete message passing boundaries, while Semaphores are primarily used for synchronization to prevent race conditions.

File I/O Basics: Everything is a File I

In Unix/Linux, hardware devices, pipes, network sockets, and regular text/binary files are all represented and accessed identically via file descriptors.

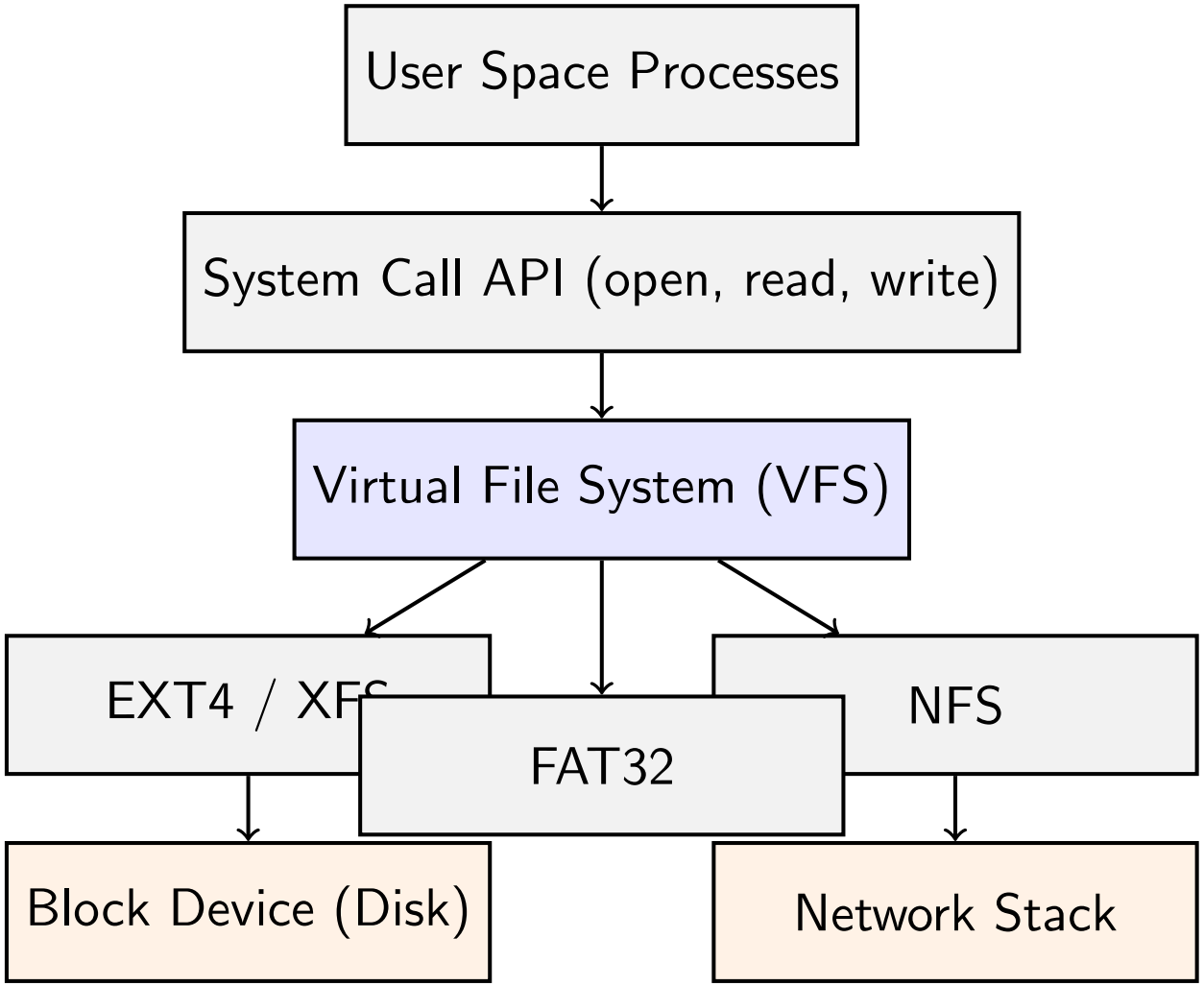
- A file descriptor is a non-negative integer serving as a process-specific handle to an open file (0: STDIN, 1: STDOUT, 2: STDERR).
- The open() system call returns a new file descriptor, while close() releases it back to the operating system.
- read() and write() operate on unbuffered sequential byte streams, making them foundational for all data transfer.
- lseek() repositions the kernel's read/write file offset pointer, allowing random access within regular files.

Signals: Asynchronous Event Notifications I

Signals are software interrupts delivered by the kernel to a process to notify it of important, asynchronous events like user inputs or memory faults.

- Common signals include SIGINT (Ctrl+C for interrupt), SIGKILL (immediate force terminate), and SIGSEGV (segmentation fault).
- Processes can register custom signal handlers using the sigaction() system call to catch and respond to signals gracefully.
- SIGKILL (signal 9) and SIGSTOP cannot be caught, blocked, or ignored, ensuring the kernel always has the ultimate authority to stop rogue processes.
- Signal handling routines must exclusively use async-signal-safe (reentrant) functions to prevent unpredictable behavior.

Virtual File System (VFS) Architecture I



- VFS provides a unified kernel abstraction layer, allowing programs to use identical system calls for drastically different underlying file systems.

Virtual File System (VFS) Architecture II

- It translates generic file operations into the specific disk or network actions required by drivers like EXT4, FAT32, or NFS.
- Inodes represent file metadata (permissions, ownership) and disk location, while dentries represent directory hierarchies.
- This highly modular object-oriented architecture allows Linux to support dozens of file systems concurrently.

Multithreading and Synchronization (Pthreads) I

POSIX Threads (Pthreads) allow concurrent execution paths within a single process, sharing the exact same virtual memory space.

- Threads are lightweight compared to full processes, as they share the heap, open files, and global data, but maintain independent call stacks.
- pthread_create() spawns a new thread of execution, while pthread_join() forces a thread to wait for another to complete.
- Mutexes (pthread_mutex_t) ensure mutually exclusive access to shared variables, completely preventing concurrent data corruption.
- Condition variables (pthread_cond_t) allow threads to sleep efficiently until a specific condition becomes true, avoiding CPU-wasting busy-waiting.

Introduction to shell and commands I

Course: Fundamentals Operating System

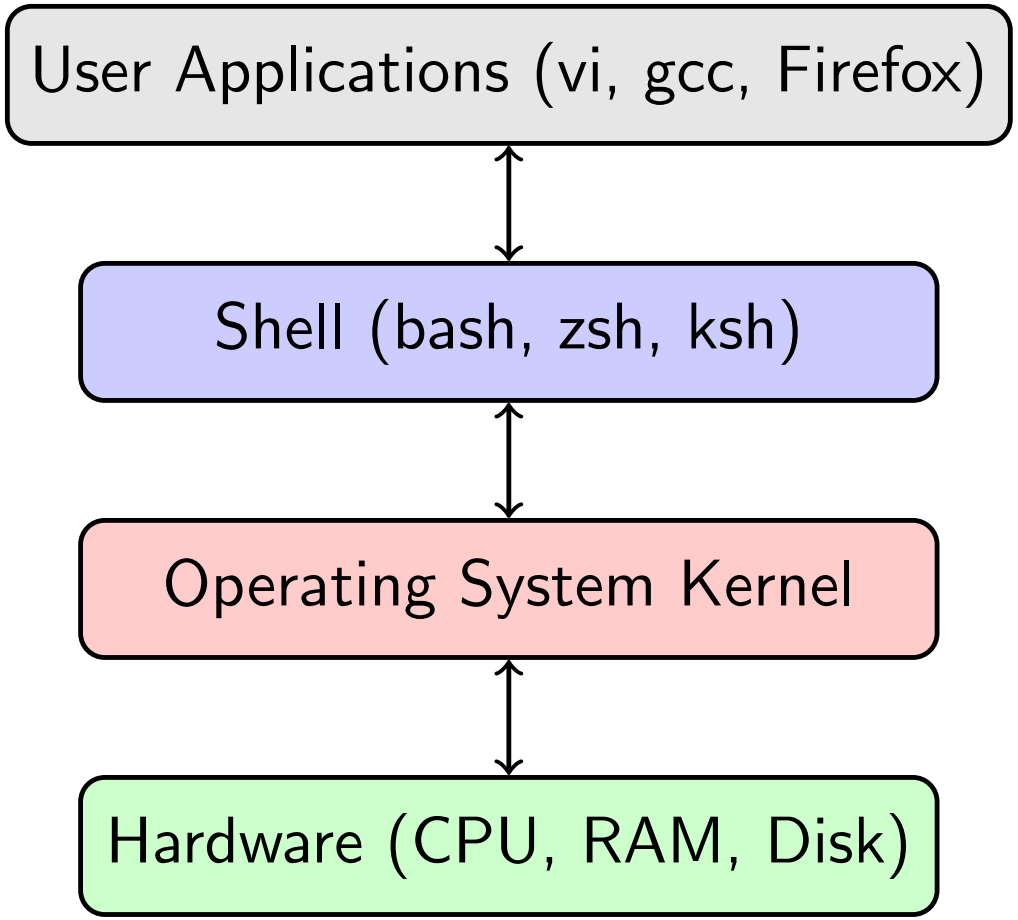
- Unit: Unix/LinuxOperatingSystem

What is a Shell? I

A shell is a command-line interpreter that provides a traditional user interface for the Unix and Linux operating systems.

- It acts as an intermediary between the user and the kernel, interpreting commands entered by the user.
- The shell reads commands from standard input or a script file and translates them into system calls for the kernel.
- It is not part of the OS kernel, but rather an ordinary user-level program that runs in user space.
- Features include input/output redirection, pipelining, variables, and a robust scripting language for automation.

Architecture of Unix/Linux I



- The hardware forms the foundation, managed directly by the kernel.
- The kernel exposes an API via system calls.

Architecture of Unix/Linux II

- The shell utilizes these system calls to interact with the kernel on behalf of user applications.
- Users interact with the shell either interactively via a terminal or through background scripts.

Common Types of Unix Shells I

Several distinct shell families have evolved over the history of Unix, each with unique features and syntax.

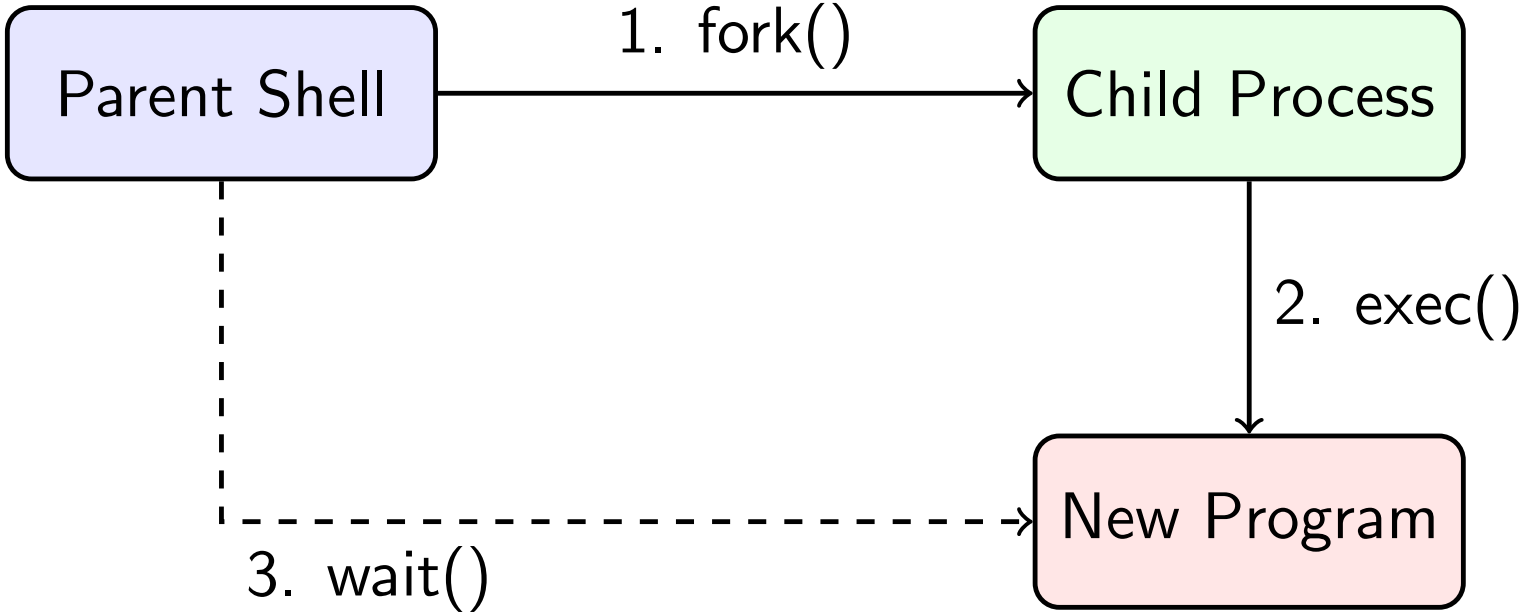
- Bourne Shell (sh): The original standard Unix shell written by Stephen Bourne at Bell Labs.
- C Shell (csh): Developed by Bill Joy, featuring syntax resembling the C programming language and introducing command history.
- Korn Shell (ksh): Developed by David Korn, it combines features of both sh and csh and is backwards-compatible with sh.
- Bourne Again Shell (bash): The default shell on most GNU/Linux systems, heavily expanding upon sh with modern interactive features.
- Z Shell (zsh): An extremely feature-rich shell with advanced auto-completion, often used as the default on modern macOS.

Standard I/O Streams and Redirection I

Every process in Unix is automatically provided with three standard data streams upon creation.

- Standard Input (stdin, file descriptor 0): The default source of input, usually the keyboard.
- Standard Output (stdout, file descriptor 1): The default destination for output, usually the terminal display.
- Standard Error (stderr, file descriptor 2): The destination for error messages, also defaulting to the terminal.
- Output Redirection (¿): Redirects stdout to a file, overwriting the file (e.g., 'echo hello ¿ file.txt').
- Append Redirection (¿¿): Redirects stdout to a file, appending to it rather than overwriting.
- Error Redirection (2¿): Redirects stderr independently (e.g., 'find / 2¿ errors.txt').

The Command Execution Lifecycle I



- 1. fork(): The shell creates a nearly identical copy of itself (the child process).
- 2. exec(): The child process replaces its memory space with the new program to be executed (e.g., /bin/ls).
- 3. wait(): The parent shell suspends its execution until the child process terminates and returns an exit status.

The Command Execution Lifecycle II

- Background execution (&) allows the parent shell to bypass the wait() step and return to the prompt immediately.

Pipes and Filters I

Pipes (—) are a fundamental Unix philosophy concept, allowing the chaining of small, single-purpose utilities.

- A pipe connects the standard output (stdout) of one process directly to the standard input (stdin) of another.
- Syntax: 'command1 — command2' ensures the data flows continuously from left to right.
- Filters are commands designed to read stdin, transform the data, and write to stdout.
- Common filters include 'grep' (searching text), 'awk' (pattern scanning), 'sed' (stream editing), and 'sort' (ordering lines).
- Example: 'ls -l — grep ".txt" — wc -l' counts the number of text files in a directory.

Shell and Environment Variables I

Variables in the shell store data such as system configuration paths, usernames, and behavioral settings.

- Local Variables: Exist only within the current shell instance and are not inherited by child processes.
- Environment Variables: Exported using the 'export' command; they are inherited by all child processes spawned by the shell.
- PATH: A critical environment variable containing a colon-separated list of directories to search for executable commands.
- HOME: Defines the current user's home directory path.
- The 'env' or 'printenv' commands display all current environment variables.

Basic Command Overview I

The shell is useless without the myriad of GNU core utilities that provide actual functionality.

- File Navigation: 'pwd' (print working directory), 'cd' (change directory), 'ls' (list directory contents).
- File Manipulation: 'cp' (copy), 'mv' (move/rename), 'rm' (remove/delete files).
- Directory Management: 'mkdir' (make directory), 'rmdir' (remove empty directory).
- File Viewing: 'cat' (concatenate and print), 'less' (paginate output), 'head' / 'tail' (view start/end of a file).
- Permissions: 'chmod' (change file mode bits), 'chown' (change file owner and group).

Conclusion and Summary I

The shell remains an indispensable tool for power users, system administrators, and developers.

- It provides a flexible, powerful environment for interacting with the operating system kernel.
- Understanding standard streams, pipes, and redirection is crucial for composing complex tasks from simple tools.
- The fork-exec model dictates how the shell manages process creation and execution.
- Next Lecture: We will introduce Shell Scripting, combining commands into executable programs with control structures.

Title Slide I

Course: Fundamentals Operating System

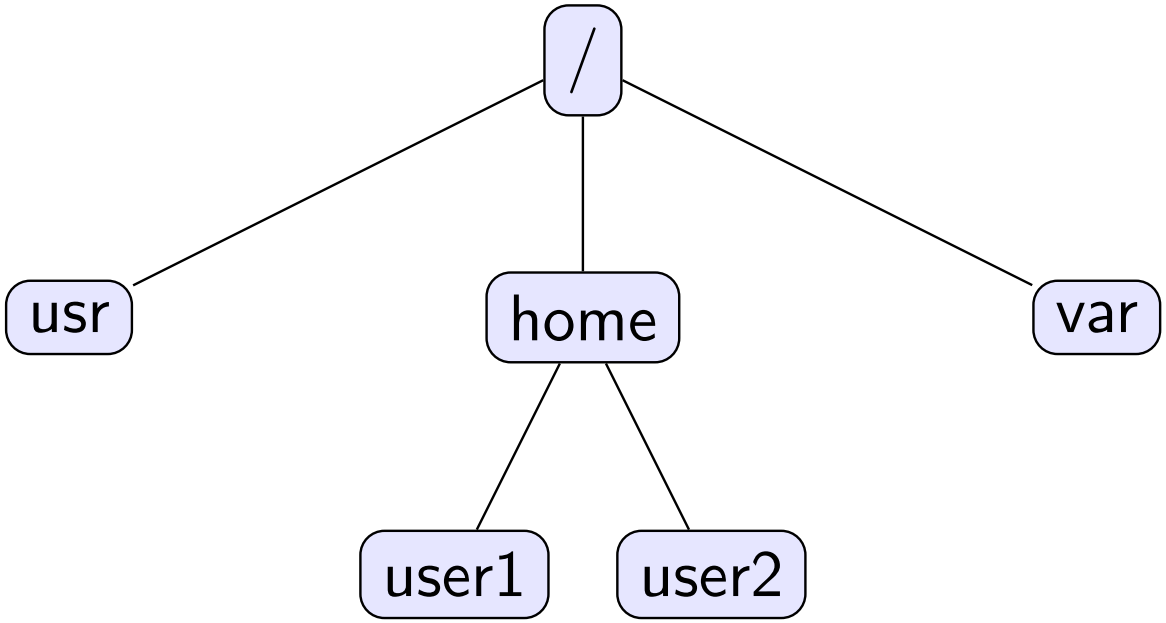
- Unit: Unix/LinuxOperatingSystem
- Lecture 40: Essential Unix/Linux File and Directory Commands

Directory Navigation and Management I

Core commands for navigating and structuring the Unix hierarchical filesystem.

- pwd (Print Working Directory): Outputs the absolute path of the current directory, utilizing the getcwd() system call.
- cd (Change Directory): Changes the current working directory. It is a shell built-in rather than an external program, as it must alter the shell process's environment (chdir() system call).
- mkdir (Make Directory): Creates new directories. Options like -p allow creation of parent directories if they do not exist.
- rmdir (Remove Directory): Deletes empty directories. Will fail if the directory contains any files or subdirectories.

Diagram: Unix Hierarchical Filesystem I



- The root directory (/) is the top-level directory in the hierarchy.
- The cd command traverses this tree using absolute (starting from /) or relative paths.
- The pwd command computes the path from root to the current node.

Listing and Viewing Files I

Commands to inspect directory contents and read file contents sequentially.

- ls (List): Displays directory contents. Uses readdir() to read directory entries and stat() for metadata (e.g., ls -l).
- Common ls flags: -l (long format detailing permissions, owner, size), -a (shows hidden files starting with '.'), -h (human-readable sizes).
- cat (Concatenate): Reads files sequentially, writing them to standard output. Useful for viewing short files or concatenating multiple files into one.
- Behind the scenes, cat opens a file (open()), reads it in blocks (read()), and writes to stdout (write(1, ...)).

File Operations: Copying and Moving I

Manipulating files within the filesystem hierarchy without altering file contents.

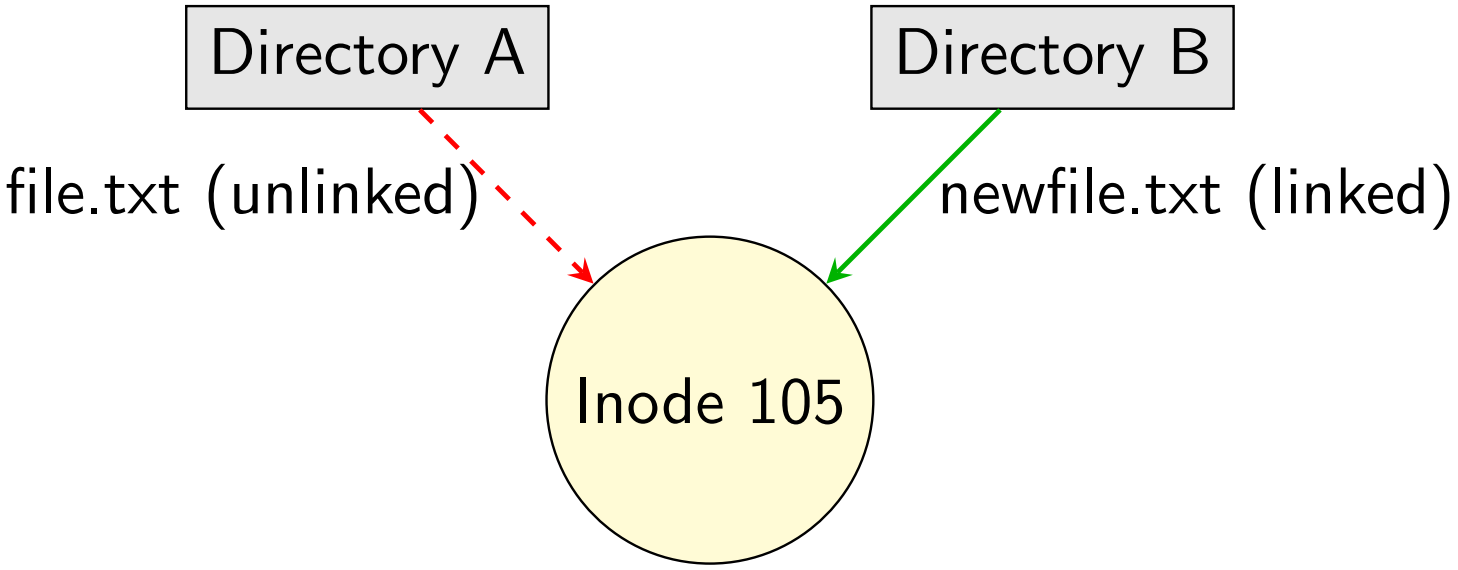
- cp (Copy): Duplicates files or directories. Requires read permission on the source and write permission on the destination directory.
- cp -r recursively copies directories, while cp -p preserves timestamps, ownership, and mode.
- mv (Move): Renames a file or moves it to a different directory. If on the same filesystem, mv merely updates the directory entries (link() and unlink() conceptually, or rename() system call), leaving data blocks untouched.
- If mv crosses filesystem boundaries, it effectively performs a cp followed by an rm.

File Deletion I

Removing file references from directories and freeing inodes/data blocks.

- rm (Remove): Unlinks a file name from its directory (unlink() system call). If the hard link count drops to zero and no process holds it open, data blocks are freed.
- rm -r recursively removes directories and their contents. Use with extreme caution.
- rm -f forces removal without prompting, ignoring nonexistent files.
- Unlike graphical trash bins, rm permanently unlinks files. Recovery tools rely on scavenging raw disk blocks before they are overwritten.

Diagram: Move (mv) Operation on the Same Filesystem I



- On the same partition, moving a file only updates directory mappings.
- The underlying inode and data blocks remain entirely unchanged.
- This makes mv a constant-time O(1) operation regardless of file size.

Text Analysis and Comparison I

Commands to analyze file properties and compare contents.

- wc (Word Count): Outputs line, word, and byte counts for a file. Counts newline (\{}n) characters for lines.
- split: Divides a file into smaller pieces based on line count (-l) or byte size (-b). Useful for transferring large files.
- cmp: Compares two files byte by byte. Exits silently if identical, or reports the byte and line number of the first difference.
- comm: Compares two sorted files line by line, outputting three columns: lines unique to file 1, unique to file 2, and common lines.
- diff: Analyzes two files and outputs the minimal set of instructions (adds, deletes, changes) needed to convert one file into the other.

Viewing File Extremities I

Commands for examining the beginning or end of files.

- head: Outputs the first 10 lines of a file by default. Adjustable via head -n ¡lines¿.
- tail: Outputs the last 10 lines of a file by default. Also adjustable via tail -n ¡lines¿.
- tail -f (Follow): A critical monitoring tool. It keeps the file open and continuously prints appended data as the file grows, heavily used for live log file monitoring (e.g., tail -f /var/log/syslog).
- Internally, tail seeks to the end of the file and reads backwards, or reads sequentially into a circular buffer.

Searching and Sorting I

Filtering and organizing textual data streams.

- grep (Global Regular Expression Print): Searches files or standard input for lines matching a pattern. Uses Deterministic Finite Automata (DFA) for fast regex matching.
- Useful grep flags: -i (case-insensitive), -v (invert match), -r (recursive search in directories).
- sort: Sorts lines of text files. Supports lexicographical or numeric (-n) sorting.
- sort can merge pre-sorted files, sort based on specific fields/columns (-k), and eliminate duplicates (-u) often combined with the uniq command.

Title Slide I

Course: Fundamentals Operating System

- Unit: Unix/LinuxOperatingSystem

Introduction to Linux Text Editors I

Text editors are essential tools in Unix/Linux operating systems for creating and modifying configuration files, writing code, and basic text processing.

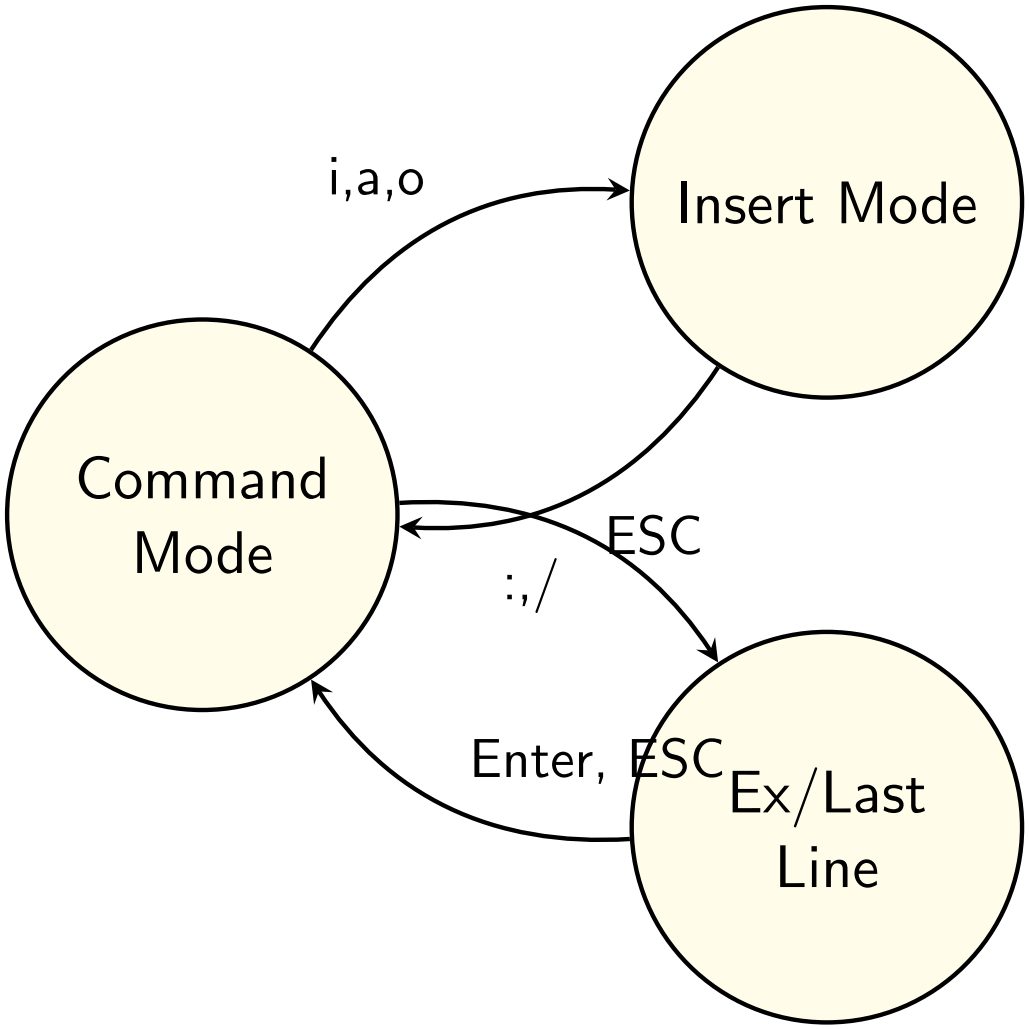
- Linux provides both CLI-based (vi, vim, nano) and GUI-based (gedit, kate) text editors.
- CLI text editors operate within the terminal and are crucial for remote server administration via SSH where GUI is unavailable.
- Choosing the right editor depends on the task context: quick edits (nano), complex text manipulation and programming (vim), or desktop environments (gedit).

The 'vi' Editor Fundamentals I

The 'vi' (visual) editor is a modal text editor introduced by Bill Joy in 1976. It is a POSIX standard and is available on virtually all Unix-like systems.

- Unlike standard text editors, 'vi' is modal, meaning the same keystrokes perform different actions depending on the current active mode.
- The three primary modes are: Command Mode (default state for navigation and manipulation), Insert Mode (for entering text), and Last Line/Ex Mode (for saving, searching, and advanced commands).
- Understanding modal editing is critical for efficient text manipulation without relying on a mouse or arrow keys, keeping fingers on the home row.

Diagram: 'vi' Editor State Transitions I



- The editor always starts in Command Mode.
- Pressing 'i' (insert), 'a' (append), or 'o' (open new line) transitions the editor to Insert Mode.

Diagram: 'vi' Editor State Transitions II

- Pressing ':' (colon) or '/' (search) in Command Mode transitions to Ex Mode.
- The ESC key is the universal way to return to Command Mode from any other state.

The 'vim' Editor (Vi IMproved) I

'vim' is a highly configurable, enhanced clone of 'vi' designed by Bram Moolenaar, adding numerous features tailored for programmers and modern users.

- Syntax Highlighting: Provides visual cues for hundreds of programming languages and configuration files.
- Multi-level Undo: Standard 'vi' only allows a single undo operation ('u'), whereas 'vim' maintains a complex undo tree.
- Visual Mode: Introduces a Visual mode that allows text selection and block operations, making copy (yank) and paste (put) far more intuitive.
- Extensibility: Supports thousands of plugins and complex configuration via the ~/.vimrc file.

The 'gedit' Editor I

'gedit' is the default text editor for the GNOME desktop environment, providing a user-friendly graphical interface while supporting advanced programming features.

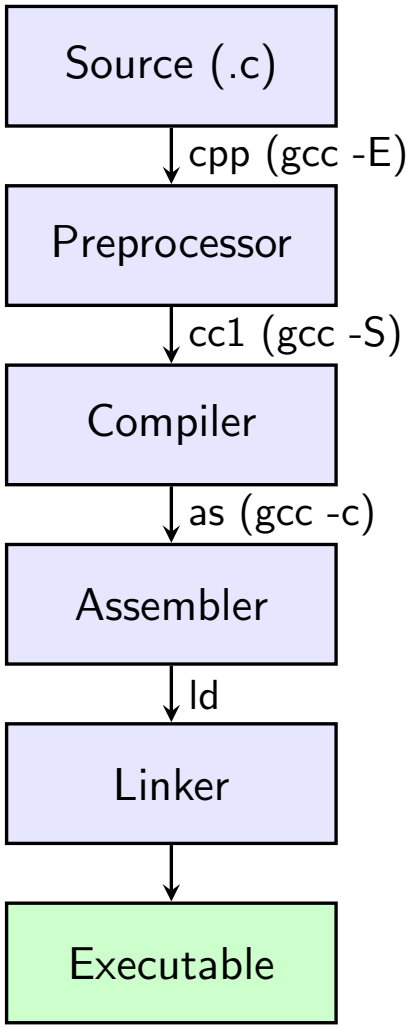
- GUI Paradigm: Uses standard mouse interactions, drop-down menus, and familiar keyboard shortcuts (Ctrl+C, Ctrl+V) favored by desktop users.
- Tabbed Interface: Allows opening and editing multiple files simultaneously within a single window instance.
- Developer Features: Includes syntax highlighting, bracket matching, line numbers, and a robust plugin architecture (e.g., Python console, spell checker).
- Integration: Seamlessly integrates with the Virtual File System (GVFS) to natively edit remote files over FTP, SSH, or HTTP.

Compiling C Programs with 'gcc' I

The GNU Compiler Collection (GCC) is the standard compiler system for Linux, essential for translating high-level C/C++ source code into executable machine binaries.

- Basic Usage: The command 'gcc program.c -o program' compiles the source file into an executable named 'program'.
- The compilation process is not monolithic; it consists of four distinct stages: Preprocessing, Compilation, Assembly, and Linking.
- Common Flags: '-Wall' enables all warning messages, '-g' includes debugging information for GDB, and '-O2' enables compiler optimizations.
- GCC essentially acts as a driver program, invoking underlying components like cpp, cc1, as, and ld sequentially.

Diagram: The GCC Compilation Pipeline I



- Preprocessor resolves #include directives and expands #define macros, outputting an expanded .i file.

Diagram: The GCC Compilation Pipeline II

- Compiler translates the expanded C code into architecture-specific assembly language instructions (.s file).
- Assembler converts the human-readable assembly instructions into machine-readable object code (.o file).
- Linker combines one or more object files along with standard C libraries (e.g., libc) to create the final executable binary.

Introduction to Linux Shell Scripting I

A shell script is a text file containing a sequence of commands for a UNIX-based operating system. It allows the automation of repetitive tasks and complex system administration workflows.

- Shebang (#!) line: The first line (e.g., #!/bin/bash) specifies the absolute path to the interpreter that should execute the script.
- Execution Permissions: Scripts must be made executable using the command 'chmod +x script.sh' before they can be run directly (via ./script.sh).
- A shell script executes commands exactly as they would be typed at the command line, but it supports full programming constructs.
- Common shells used for scripting include Bash (Bourne Again SHell, the standard on Linux), sh (Bourne Shell), and zsh.

Shell Scripting Core Constructs I

Shell scripting goes beyond sequential command execution by including essential programming paradigms such as variables, control flow, loops, and functions.

- Variables: Defined without spaces around the equals sign (VAR=value) and accessed with a dollar sign prefix ($VAR). Environment variables (like $PATH, $USER) are globally available.
- Conditionals: Utilize 'if', 'elif', 'else', and 'fi'. Test conditions evaluate expressions using brackets (e.g., [$a -eq $b] for numerical equality).
- Loops: Supports 'for', 'while', and 'until' constructs to iterate over lists of items, read lines in a file, or loop until a condition changes.

Shell Scripting Core Constructs II

- Command Substitution: Using backticks (‘cmd‘) or the modern $(cmd) syntax captures the standard output of a command and stores it in a variable for later use.

Title Slide I

Course: Fundamentals Operating System

- Unit: Unix/LinuxOperatingSystem

Introduction to Command Line Arguments I

Command line arguments allow passing parameters to a program when it is executed in the shell.

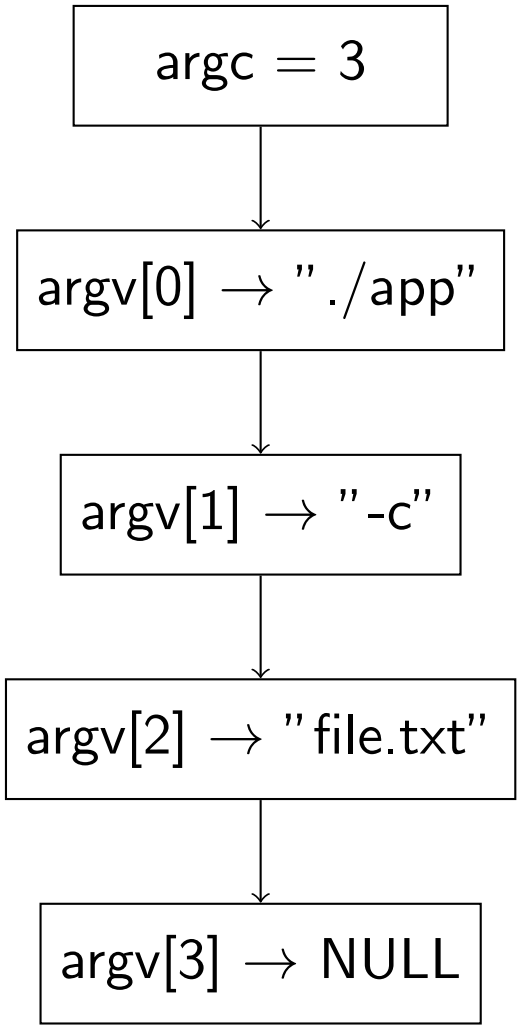
- In C and C++, the main function receives these arguments via 'int argc' and 'char *argv[]'.
- 'argc' (argument count) is an integer representing the total number of command-line arguments passed, including the program's name.
- 'argv' (argument vector) is an array of character pointers, where each pointer points to a null-terminated string representing an argument.
- The OS shell parses the input line, splits it by whitespace, and constructs the argv array before transferring control to the program's entry point.

The Structure of argv Array I

The argv array is dynamically set up by the operating system's executable loader (e.g., execve system call in Unix).

- argv[0] holds the name of the executable file as it was invoked (e.g., './program' or '/usr/bin/ls').
- argv[1] to argv[argc-1] contain the actual arguments provided by the user.
- argv[argc] is guaranteed by the C standard and POSIX to be a NULL pointer, acting as a sentinel for array traversal.
- The strings pointed to by argv are typically stored in the stack area of the process's memory space, above the environment variables.

Memory Layout of Command Line Arguments I



- This diagram illustrates the pointer relationships in the argv array when a user executes: ./app -c file.txt
- The integer argc holds the value 3.

Memory Layout of Command Line Arguments II

- The pointers in the argv array map to distinct null-terminated strings placed in the process stack by the OS.

Accessing and Processing Arguments I

Processing arguments sequentially requires iterating through the argv array from index 1 to argc - 1.

- A typical for-loop looks like: for (int i = 1; i ¡ argc; i++) { printf("%s\{}n", argv[i]); }
- Because all arguments are passed as strings, numeric arguments must be converted using functions like atoi(), strtol(), or strtod().
- Using strtol() is preferred over atoi() as it provides better error checking for invalid numerical inputs and over/underflows.
- Since the arguments are strings allocated by the OS, modifying them directly (though allowed by C) is generally discouraged.

Use Cases in Unix/Linux Utilities I

Command line arguments are the backbone of Unix philosophy, allowing small, modular tools to be composed together.

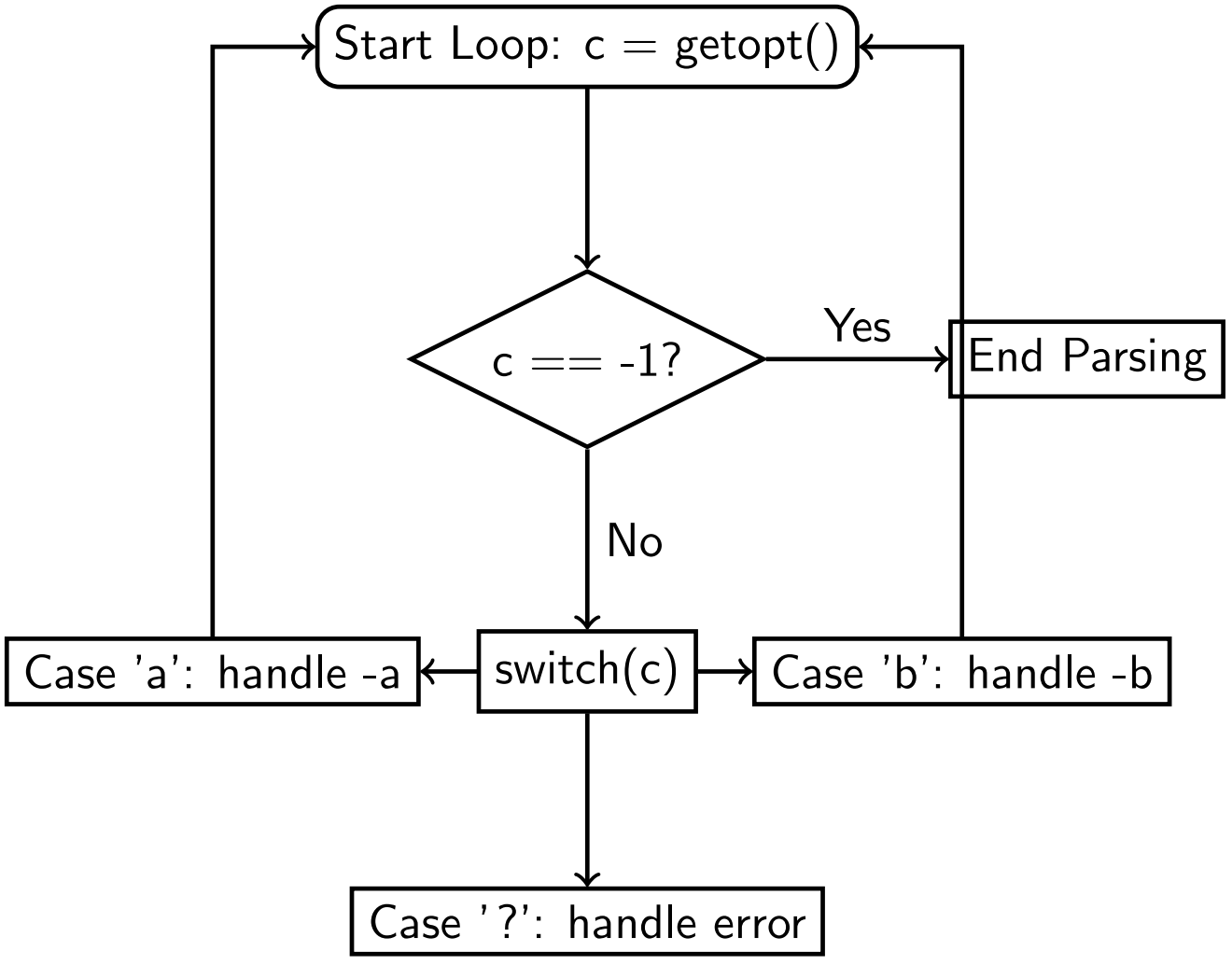
- Flags/Switches: Arguments starting with '-' or '–' modify the behavior of the program (e.g., 'ls -l' for long format, 'grep -i' for case-insensitive).
- Positional Arguments: Arguments interpreted based on their position, often used for specifying input/output files (e.g., 'cp source.txt dest.txt').
- Environment variables vs Arguments: Arguments are explicitly passed per execution, while environment variables are inherited from the parent shell process.
- The execve() system call explicitly takes an argv array as its second parameter to launch a new program with specified arguments.

Parsing Arguments with getopt() I

The POSIX getopt() function standardizes the parsing of command-line options and their arguments.

- getopt() takes argc, argv, and an 'optstring' specifying valid short options (e.g., "a:b" means 'a' requires an argument, 'b' does not).
- It sets external variables like 'optarg' (pointing to the option's argument), 'optind' (index of the next argument to process), and 'optopt' (the option character that caused an error).
- It allows for combined flags (e.g., 'tar -xzf') and handles out-of-order options transparently.
- For long options (e.g., '–help'), GNU provides the getopt_long() function.

The getopt() Parsing Flow I



- The diagram shows the standard while-loop construct used to parse options with getopt().
- The function returns -1 when all options have been parsed.

The getopt() Parsing Flow II

- A switch statement is used to handle each valid option character.
- Unrecognized options or missing arguments return '?' and are handled in the default switch case.

Command Line Arguments vs Standard Input I

It is crucial to distinguish between arguments passed during invocation and data read during execution.

- Command line arguments are passed exactly once at startup via the exec() family of system calls.
- Standard input (stdin) is a continuous stream of data read during the program's lifecycle via file descriptor 0.
- Arguments are limited in size (defined by ARG_MAX in POSIX, typically megabytes), whereas stdin can stream terabytes of data.
- Many utilities (like 'cat' or 'grep') take filenames as arguments, but if no filename is provided, they fall back to reading from stdin.

Best Practices and Error Handling I

Robust programs must validate command-line arguments rigorously to prevent unexpected behavior or security vulnerabilities.

- Always check 'argc' before accessing 'argv[n]' to prevent segmentation faults from dereferencing NULL pointers.
- Provide a usage function (e.g., 'void print_usage()') that outputs help instructions if arguments are missing or malformed.
- Validate the bounds and types of inputs strictly, especially when arguments are used to allocate memory or access files.
- In security-critical applications (like SetUID binaries), untrusted command-line arguments can lead to buffer overflows or injection attacks if not properly sanitized.

Title Slide I

Course: Fundamentals Operating System

- Unit: Unix/LinuxOperatingSystem
- Lecture 43: Arithmetic Operators, Logical Operators, Relational Operators
- Instructor: CS Professor

Introduction to Operators in Unix/Linux Shell I

Operators in Unix-like operating systems (such as Bash, sh) are essential for evaluating expressions, comparing values, and making decisions within shell scripts. Unlike traditional programming languages, shell scripts treat variables mostly as strings, making specialized operators necessary for numerical and string context evaluations.

- Shell scripts rely on operators to perform test conditions (using ‘test‘ or ‘[‘ and ‘[[‘ constructs).
- Operators are categorized primarily into Arithmetic, Relational, and Logical types.
- Understanding context is crucial: shell environments distinguish between integer arithmetic evaluation and string manipulation.

Introduction to Operators in Unix/Linux Shell II

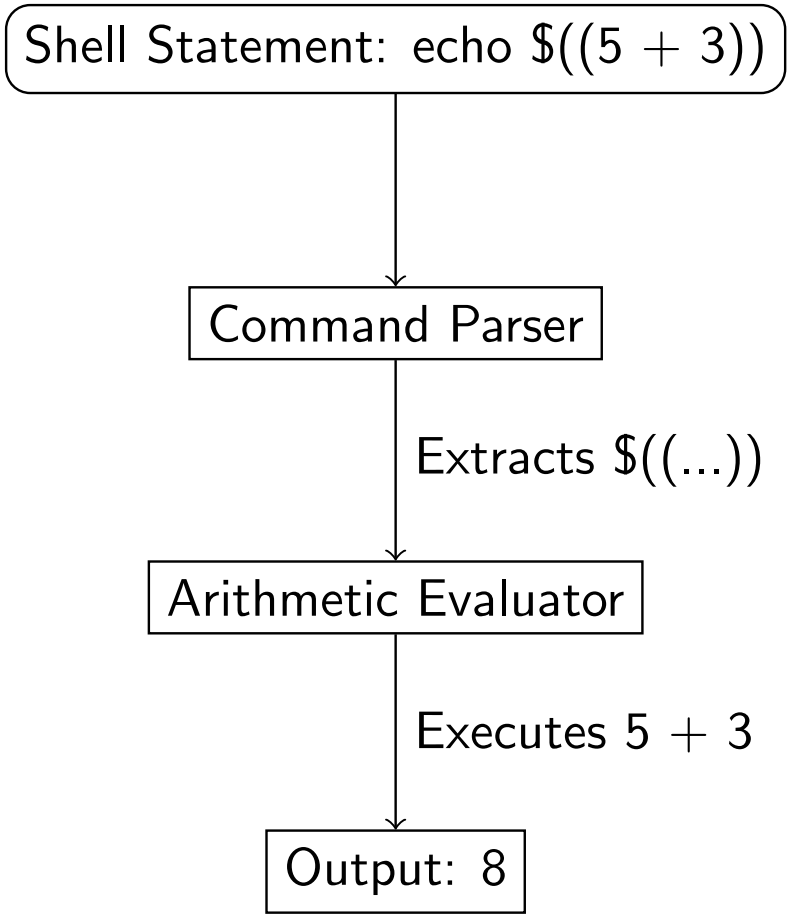
- Historically, external utilities like ‘expr‘ or ‘bc‘ were required, but modern shells provide built-in evaluation mechanisms like ‘$((...))‘.

Arithmetic Operators in Shell Scripting I

Arithmetic operators are used to perform mathematical calculations. In Bash, native integer arithmetic is supported using the double-parentheses construct ‘$((expression))‘ or the ‘let‘ command. Supported operators include addition (+), subtraction (-), multiplication (*), division (/), and modulo (%).

- Addition (+): Adds two operands (e.g., ‘$((a + b))‘).
- Subtraction (-): Subtracts second operand from the first.
- Multiplication (*): Multiplies operands.
- Division (/): Integer division (e.g., ‘$((10 / 3))‘ yields 3).
- Modulo (%): Returns remainder (e.g., ‘$((10 % 3))‘ yields 1).
- Exponentiation (**): Raises first operand to the power of the second.

Arithmetic Operator Processing in Bash I



- The shell parses the command and identifies the arithmetic expansion syntax ‘$((...))‘.
- It extracts the inner expression and passes it to the arithmetic evaluator.

Arithmetic Operator Processing in Bash II

- The evaluator performs integer math operations.
- The result is substituted back into the command line before execution.

Relational Operators (Numeric Comparisons) I

Relational operators compare two values to determine the relationship between them. In POSIX-compliant shells, numeric comparisons use specific flags inside the ‘test‘ or ‘[‘ command, rather than traditional algebraic symbols, to avoid conflicting with shell I/O redirection operators.

- ‘-eq‘: Checks if the value of two operands is equal (e.g., ‘[$a -eq $b]‘).
- ‘-ne‘: Checks if values are not equal.
- ‘-gt‘: Checks if left operand is strictly greater than the right operand.
- ‘-lt‘: Checks if left operand is strictly less than the right.
- ‘-ge‘: Greater than or equal to.
- ‘-le‘: Less than or equal to.

Relational Operators (String Comparisons) I

While numeric comparisons use hyphenated flags, string comparisons in Unix shells use algebraic symbols. It is critical to quote string variables to prevent syntax errors if a variable is empty or contains spaces.

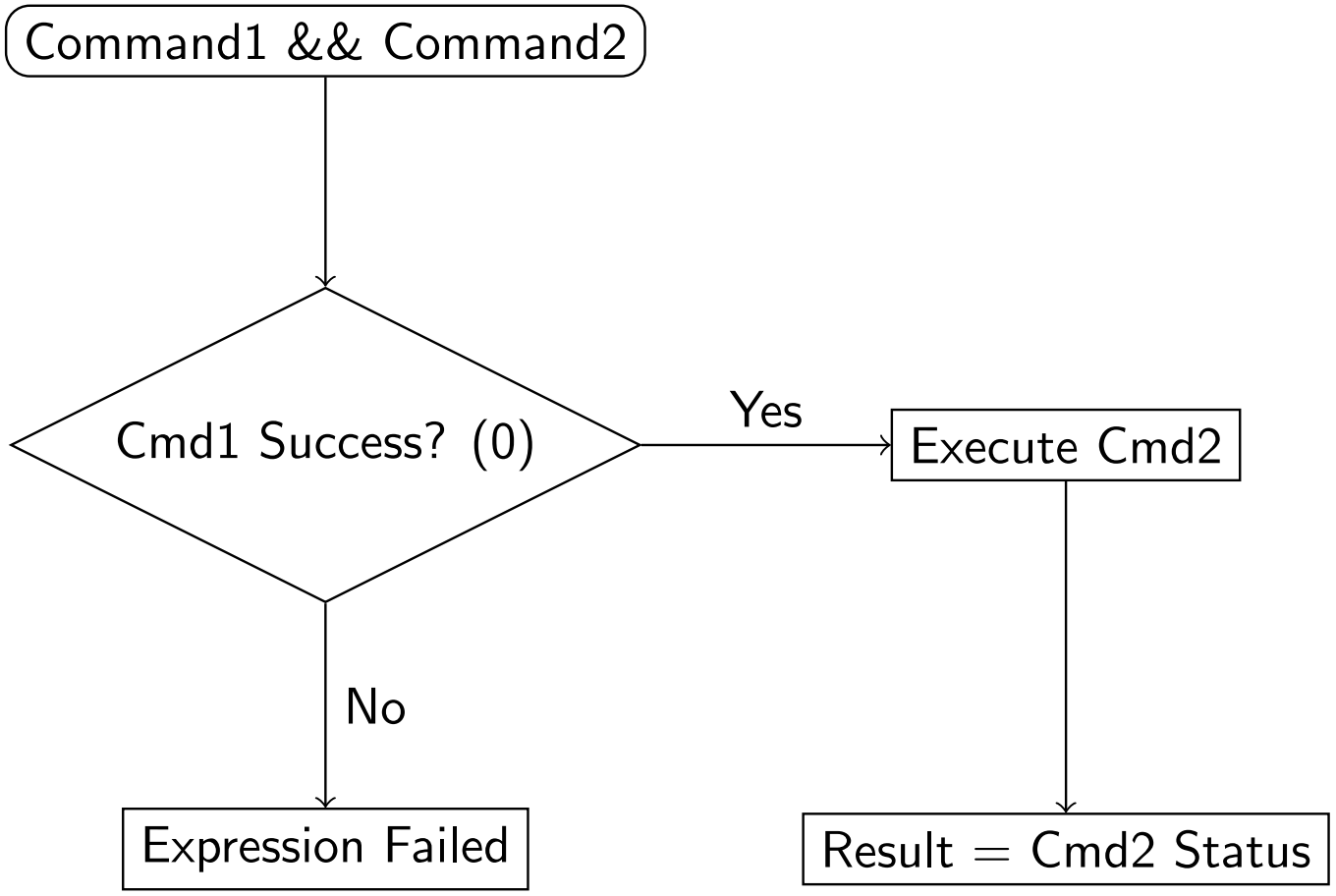
- '=' or '==': Checks if two strings are identical (e.g., '["$str1" = "$str2"]').
- '!=': Checks if two strings are not identical.
- '¡': Checks if left string sorts before right string in ASCII alphabetical order. Must be escaped in '[' as '\{}¡'.
- '¿': Checks if left string sorts after right string.
- '-z': Checks if string length is zero (empty string).
- '-n': Checks if string length is non-zero.

Logical Operators in Shell Scripts I

Logical operators are used to combine multiple conditional expressions. They evaluate to a boolean truth value (exit status 0 for true, non-zero for false). They are essential for complex decision-making in ‘if‘, ‘while‘, and ‘until‘ statements.

- Logical AND (‘-a‘ in ‘test‘, ‘&&‘ as a command list operator): True if both conditions are true.
- Logical OR (‘-o‘ in ‘test‘, ‘——‘ as a command list operator): True if at least one condition is true.
- Logical NOT (‘¡): Inverts the truth value of a condition.
- The ‘&&‘ and ‘——‘ operators implement short-circuit evaluation, executing the right operand only if necessary based on the left operand’s exit status.

Short-circuit Evaluation in Logical Operators I



- In ‘A && B‘, command ‘B‘ executes only if command ‘A‘ succeeds (exit code 0).
- In ‘A —— B‘, command ‘B‘ executes only if command ‘A‘ fails (exit code ¿ 0).

Short-circuit Evaluation in Logical Operators II

- This behavior is leveraged in shell idioms like ‘mkdir temp —— exit 1‘.
- Short-circuiting optimizes execution by bypassing unnecessary command evaluations.

Advanced Use Cases: Combining Operators I

Operators are rarely used in isolation. They are combined within conditional control structures to govern the flow of shell execution. Modern Bash provides the ‘[[...]]‘ keyword which offers safer and more powerful evaluation than the classic ‘[‘ command.

- ‘[[...]]‘ supports unescaped ‘¡‘ and ‘¿‘ for string comparison and ‘&&‘ / ‘——‘ for logical combinations internally.
- Pattern matching: ‘[[”$string” == *.txt]]‘ uses operators for globbing.
- Regular expressions: Bash 3.0+ supports ‘=~‘ for regex matching (e.g., ‘[[”$ip” =~ ^[0-9]+$]]‘).
- Combining arithmetic and logic: ‘if ((a ¿ 5 && b ¡ 10)); then ... fi‘.

Best Practices and Common Pitfalls I

Mastering shell operators requires attention to syntax strictness. A missing space or unquoted variable can lead to script failure or unexpected behavior.

- Always quote variables in ‘[‘ tests to prevent word splitting errors (e.g., ‘["$var" = "val"]‘).
- Require spaces around operators in ‘[‘ and ‘[[‘ (e.g., ‘[1 -eq 1]‘, not ‘[1-eq1]‘).
- Use ‘$((...))‘ for arithmetic instead of the deprecated ‘expr‘ utility for performance and readability.
- Prefer ‘[[...]]‘ over ‘[‘ for string and logical operations in Bash scripts due to enhanced safety and features.

Title Slide I

Course: Fundamentals Operating System

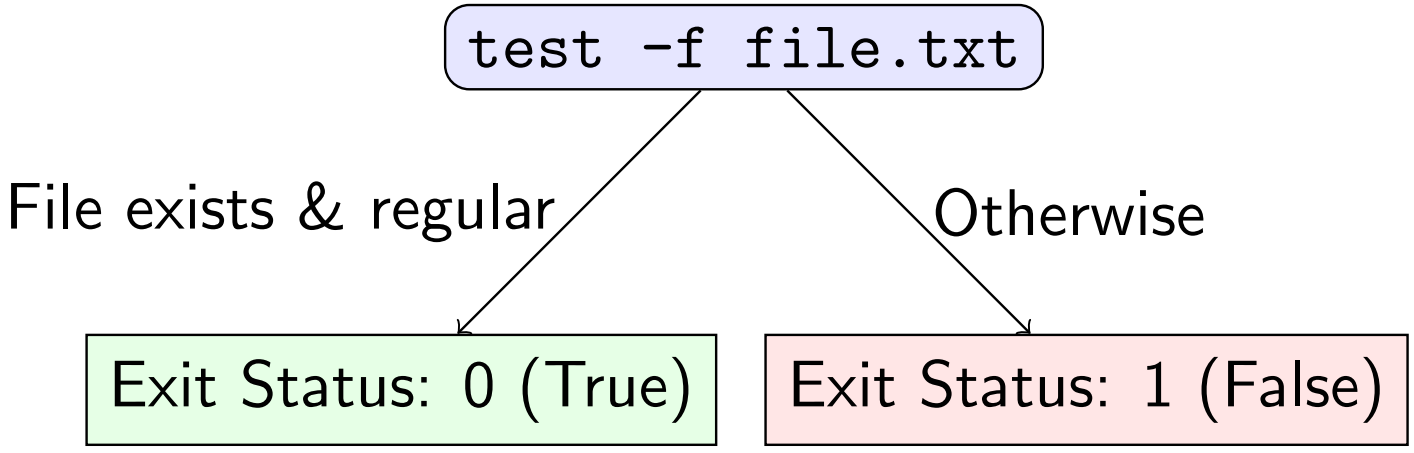
- Unit: Unix/LinuxOperatingSystem

Introduction to the test command I

The test command evaluates conditional expressions in Unix/Linux operating systems, serving as the foundation for shell script decision-making.

- It evaluates expressions to check file types, compare numeric values, and assess string properties.
- Crucial for control flow mechanisms like if-statements and while-loops in shell scripts.
- Returns an exit status of 0 to indicate true/success and 1 (or ¿0) to indicate false/failure.
- Available as a standalone binary (usually /usr/bin/test) and natively as a shell built-in for performance.

Syntax and Exit Status Flow I



- The exit status is accessed using the special shell variable $?.
- A zero (0) exit status logically equates to 'true' in shell environments.
- Any non-zero exit status (typically 1) equates to 'false' or an error state.
- The test command operates silently by default, redirecting results solely through the exit code.

File Test Operators I

Operators designed to query the filesystem and verify file metadata (e.g., existence, permissions, type).

- '-e file': Evaluates to true if the specified file or directory exists.
- '-f file': Evaluates to true if the path exists and points to a regular file (not a device or directory).
- '-d file': Evaluates to true if the path exists and points to a directory.
- '-r', '-w', '-x': Evaluate to true if the file exists and read, write, or execute permissions are respectively granted for the current user.

String Comparison Operators I

Expressions used to determine string equivalence and string length within shell scripts.

- '-z string': Evaluates to true if the length of the string is strictly zero.
- '-n string': Evaluates to true if the length of the string is non-zero.
- 'string1 = string2': Evaluates to true if both strings contain exactly the same sequence of characters.
- 'string1 != string2': Evaluates to true if the strings are not equivalent.
- Best Practice: Always double-quote variable references (e.g., "$VAR") to prevent errors if the variable is empty or contains whitespace.

Integer Comparison Operators I

Arithmetic comparison operators strictly designed for evaluating algebraic integers.

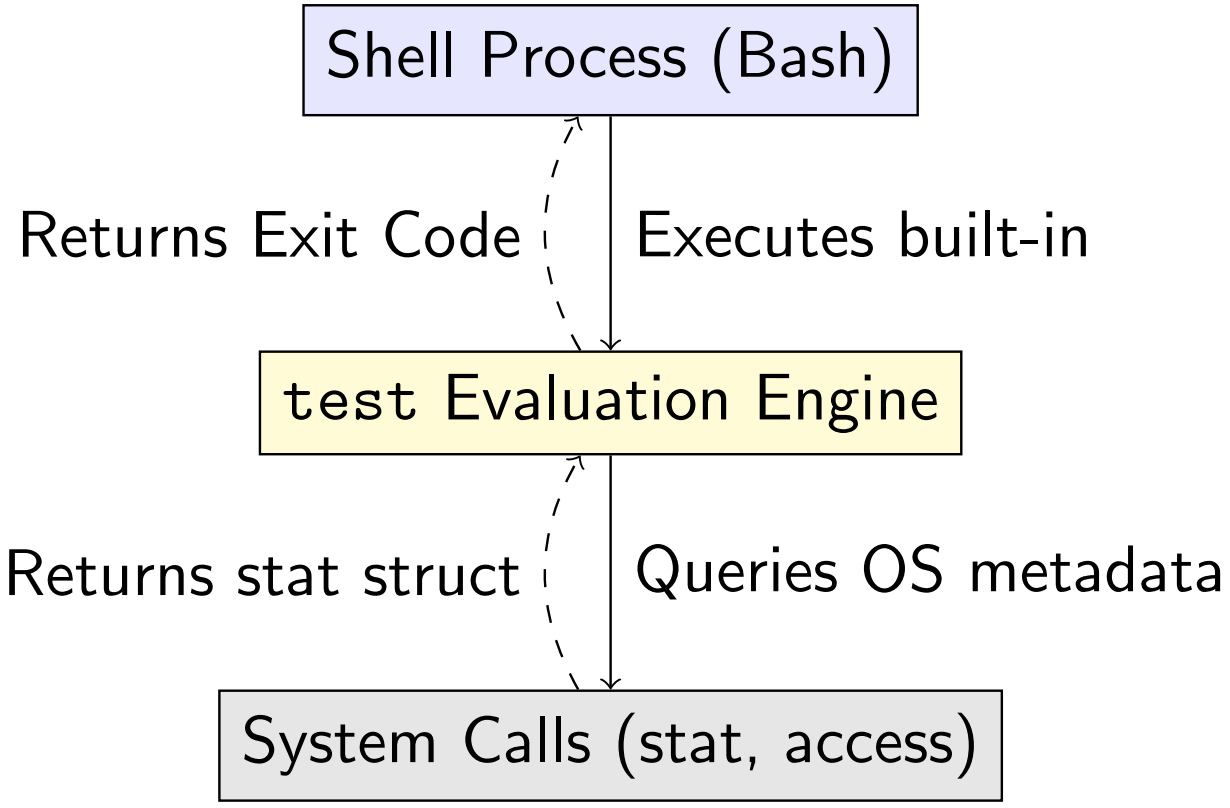
- '-eq' (equal to) and '-ne' (not equal to) check exact numeric equality.
- '-gt' (greater than) and '-ge' (greater than or equal to).
- '-lt' (less than) and '-le' (less than or equal to).
- Unlike strings, integer comparisons do not use standard symbols like '¡' or '¿', which the shell interprets as I/O redirection operators.
- Example usage: 'test 10 -gt 5' safely compares numerical magnitudes.

Logical Operators I

Constructs for combining multiple boolean expressions into a single compound condition.

- '! expr': Logical NOT. Reverses the boolean output of the following expression.
- 'expr1 -a expr2': Logical AND. True only if both expr1 and expr2 evaluate to true.
- 'expr1 -o expr2': Logical OR. True if at least one of the expressions evaluates to true.
- Warning: The '-a' and '-o' operators are considered obsolete in the POSIX standard due to ambiguity; native shell chaining ('&&', '——') is preferred.

Internal Workings and System Calls I



- File tests like '-f' and '-d' internally translate to the stat() or lstat() system calls in Linux/Unix.
- Permission checks like '-r' or '-x' map closely to the access() system call.

Internal Workings and System Calls II

- Shell built-ins execute these routines without forking a new process, improving script execution speed.
- String and integer comparisons are entirely handled in user space by the shell's expression parser.

test vs [vs [[I

Understanding the syntactic differences and evolution of condition testing tools in Unix.

- 'test' is the original command, taking arguments directly.
- '[' is historically a symlink to 'test' (or an equivalent built-in) that requires a matching ']' as the final argument.
- '[[' is a modernized shell keyword (in bash/zsh/ksh) offering enhanced features like regular expression matching ('=~').
- Within '[[...]]', variables do not undergo word splitting or path expansion, making it inherently safer and syntax-friendly.

Summary and Best Practices I

Crucial takeaways for robust shell script development using the test command.

- Use standard '[' or 'test' for maximum portability across different POSIX environments (e.g., plain sh).
- Prefer '[[...]]' when developing specifically for modern shells (bash, zsh) for safety and advanced capabilities.
- Always double-quote variables in standard '[' expressions to prevent runtime syntax errors.
- Rely on shell conditional operators ('&&' and '——') outside the test command instead of internal '-a' and '-o'.

Title Slide I

Course: Fundamentals Operating System

- Unit: Unix/Linux Operating System
- Lecture 45: Branching & Looping Constructs
- Focus: Shell Scripting Control Flow

Introduction to Shell Control Structures I

Control structures in shell scripting allow for decision-making and repetitive task execution. They form the backbone of any non-trivial automation script in Unix/Linux environments.

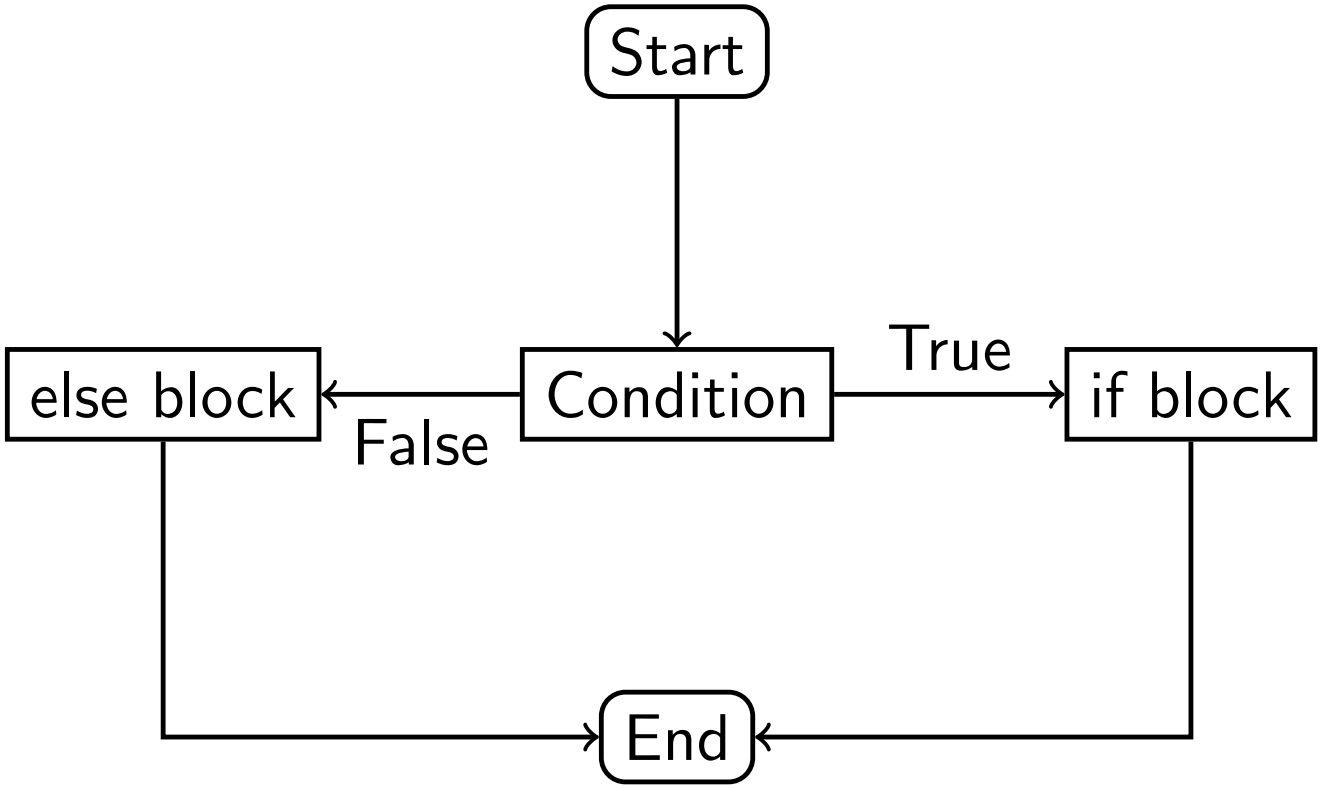
- Branching alters the sequential execution flow based on specific conditions.
- Looping permits the repeated execution of a block of code until a condition is met.
- Shell scripts primarily use exit statuses (0 for success, non-zero for failure) to evaluate conditions via the ‘test‘ command or ‘[‘/‘[[‘ brackets.
- Modern bash scripting prefers ‘[[ ]]‘ for conditional expressions due to advanced pattern matching and safety over the POSIX ‘[ ]‘.

The 'if' Statement I

The ‘if‘ statement evaluates a command's exit status and executes a block of code only if the status is exactly zero, signifying success.

- Syntax: ‘if condition; then ... fi‘
- The ‘then‘ keyword can be placed on the same line if preceded by a semicolon.
- Conditions are often evaluated using the ‘test‘ command, e.g., ‘if test -f file.txt; then‘.
- Common test operators include ‘-eq‘ (equal), ‘-ne‘ (not equal), ‘-lt‘ (less than), ‘-d‘ (directory exists), and ‘-z‘ (string is empty).

Execution Flow: if...else Statement I



- The if-else statement provides a two-way decision path.
- If the condition evaluates to true (0 exit status), the 'if' block executes.
- If the condition is false (non-zero exit status), the 'else' block executes.

Execution Flow: if...else Statement II

- Ensures exactly one of the two code blocks will be run, preventing fall-through behavior.

Multi-way Branching: if...elif...else I

The ‘elif‘ (else if) clause allows checking multiple conditions sequentially. It prevents the deep nesting of multiple ‘if...else‘ statements, maintaining script readability.

- Syntax: ‘if cond1; then ... elif cond2; then ... else ... fi‘
- Execution stops at the first condition that evaluates to true, bypassing all subsequent blocks.
- The ‘else‘ block serves as a catch-all if none of the ‘if‘ or ‘elif‘ conditions are met.
- This construct is crucial for simple menu-driven scripts where a user might input one of several options.

The 'case' Statement I

The ‘case‘ statement is a more elegant and readable alternative to multiple ‘if...elif‘ statements, especially when comparing a single variable against various string patterns.

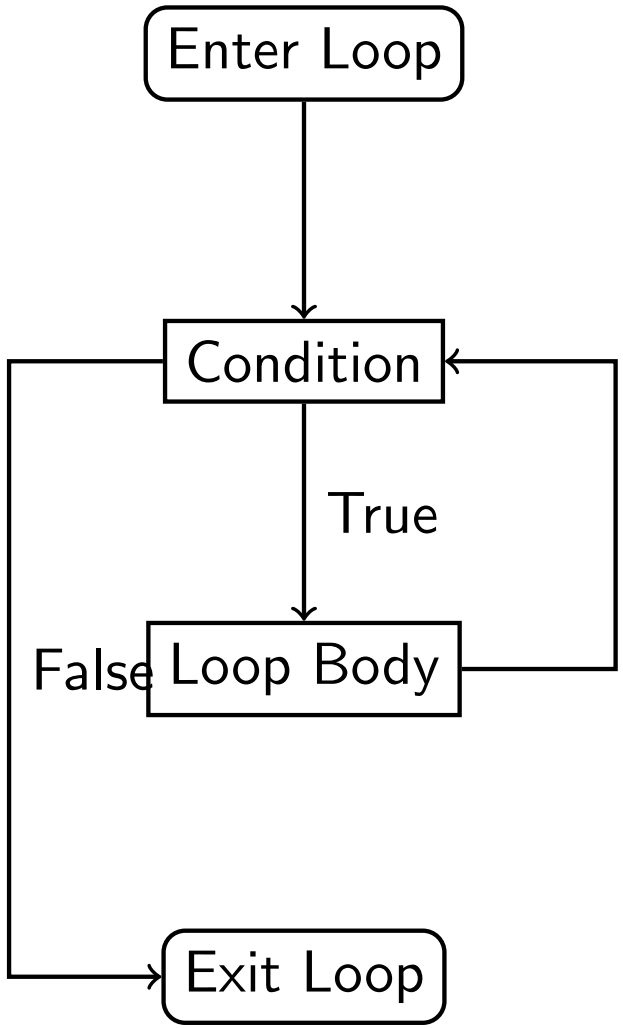
- Syntax: ‘case $var in pattern1) commands ;; pattern2) commands ;; esac‘
- Patterns can include globbing characters like ‘*‘ (any string), ‘¿ (any single character), and ‘[...]‘ (character classes).
- The double semicolon ‘;;‘ is required to terminate each pattern block, acting similarly to a 'break' statement in C or Java.
- A common idiom is using ‘*)‘ at the end as a default case to handle unexpected or fallback input.

Introduction to Looping Constructs I

Loops in Unix shells enable the automation of repetitive tasks, such as iterating over files in a directory or monitoring a system process continuously.

- Loops execute a sequence of commands multiple times based on a condition or a predefined list of items.
- Bash provides three primary looping constructs: ‘while‘, ‘until‘, and ‘for‘.
- ‘while‘ loops execute as long as a specified condition is true, whereas ‘until‘ loops execute until a condition becomes true.
- ‘for‘ loops iterate over a predefined list of words, arrays, or the output of a command substitution.

Execution Flow: while Loop I



- The ‘while‘ loop checks the condition before executing the loop body (a pre-test loop).

Execution Flow: while Loop II

- If the condition is initially false, the loop body might not execute at all.
- Care must be taken to ensure the condition eventually becomes false; otherwise, an infinite loop occurs.
- Often used with the ‘read‘ command to process files line-by-line: ‘while read line; do ... done ¡ file.txt‘.

The 'for' Loop I

The ‘for‘ loop is ideal for iterating through a known set of items. Modern bash supports both the traditional shell ‘for‘ loop and a C-style ‘for‘ loop for arithmetic operations.

- Traditional syntax: ‘for var in list; do ... done‘
- The list can be space-separated strings, brace expansions (e.g., ‘{1..5}‘), or globbed filenames (e.g., ‘*.sh‘).
- C-style arithmetic syntax: ‘for ((i=0; i¡10; i++)); do ... done‘
- Loop variables maintain their last assigned value even after the loop terminates, a behavior inherited from its environment design.

Loop Control: break and continue I

Shell scripts offer granular control over loop execution using ‘break‘ and ‘continue‘ statements, allowing for premature termination or selective skipping of loop iterations.

- The ‘break‘ command exits the current loop immediately, transferring execution control to the command following the ‘done‘ statement.
- The command ‘break n‘ can be used to break out of ‘n‘ nested loops simultaneously.
- The ‘continue‘ command skips the remaining commands in the current iteration and jumps directly to the next evaluation of the loop condition.
- Using these statements judiciously can simplify complex nested conditional logic within deeply layered loops.