Detailed Notes by Chapter

The textbook is organized into ten major parts, consisting of 21 chapters and 4 appendices.

Part One: Overview (Chapters 1–2)

Goal/Theme: Explain what operating systems are, what they do, and how they are designed and constructed, covering traditional PC/server and mobile operating systems. The presentation is motivational and explanatory, avoiding internal algorithm details.

Chapter 1: Introduction

Important Topics & Concepts:

• **Operating System Role:** Software that manages computer hardware, provides a basis for application programs, and acts as an intermediary between the user and the hardware.

• **Computer System Organization:** Comprises hardware, the operating system, application programs, and the user.

• **Interrupts:** A key mechanism for hardware to interact with the operating system; hardware devices alert the CPU via interrupts that require attention. The interrupt routine is located using an interrupt vector stored in low memory.

• **Storage Hierarchy:** Memory is organized according to speed and cost, with registers being the fastest and most costly, followed by cache, main memory, and secondary storage.

• **Volatile vs. Nonvolatile Storage (NVS):** Volatile storage (memory) loses content when power is removed; NVS (like HDDs and NVM) retains contents permanently. NVS includes mechanical (HDDs, magnetic tape) and electrical (NVM, SSD) types.

• **Multiprocessor Systems:** Most common are **Symmetric Multiprocessing (SMP)** systems, where each processor performs all tasks. Newer architectures include **Non-Uniform Memory Access (NUMA)**, where CPUs have local memory accessed via a fast local bus.

• **System Utilization: Multiprogramming** allows several jobs in memory to maximize CPU utilization. **Multitasking** (time sharing) extends multiprogramming with rapid CPU switching for fast response time.

• **Protection (Dual-Mode):** Hardware provides two modes—**user mode** and **kernel mode**—to prevent user programs from interfering with system operations. Privileged instructions can only be executed in kernel mode.
• **Resource Management:** OS is responsible for **Process Management** (creating/deleting processes, scheduling, synchronization, communication), **Memory Management** (tracking memory use, allocation/freeing), **Storage Management** (file systems, disk scheduling, partitioning), and **I/O System Management** (buffering, caching, spooling, device drivers).
• **Virtualization:** Abstracting hardware into different execution environments.
• **Kernel Data Structures:** Fundamental structures prevalent in OS design, including lists, stacks, queues, trees, and maps.
• **Updates (10th Ed.):** Includes updated coverage of multicore systems, new coverage of NUMA systems and Hadoop clusters, and new motivation for OS study.

Chapter 2: Operating-System Structures

Important Topics & Concepts:
• **Operating-System Services:** OS provides an environment for program execution and services to programs and users.
• **User and Operating-System Interface:** Approaches include Command-Line Interface (command interpreter) and Graphical User Interface (GUI).
• **System Calls:** Mechanism for user programs to request OS services. Categories include process control, file management, device management, information maintenance, communications, and protection.
• **Process Control System Calls:** Include methods for process creation (fork(), CreateProcess()) and management, interprocess communication (e.g., shared memory create()), and synchronization (acquire lock(), release lock()).
• **System Services (Utilities):** Provide a convenient environment for program development and execution; some are interfaces to system calls, others are more complex.
• **Design Principles:** Key principle is the separation of **mechanism** (how to do something) from **policy** (what will be done).

• **Operating-System Structures:**

  ○ **Monolithic Structure:** Places all kernel functionality into a single, static binary file running in one address space.

  ○ **Loadable Kernel Modules (LKMs):** Allows a dynamic, modular kernel (like Linux) while retaining monolithic performance benefits, used primarily for device drivers and file systems.

  ○ **Hybrid Systems:** Combine structures (e.g., Windows, macOS, iOS, Android).

• **Operating-System Debugging:** Finding and fixing errors, including performance tuning (removing bottlenecks). Tools use counters (ps, top, vmstat) or tracing (strace, BCC toolset).

• **Updates (10th Ed.):** Revised discussion of design/implementation, updated coverage of Android/iOS, revised system boot (focusing on GRUB), new sections on linkers/loaders, and BCC debugging toolset.

Part Two: Process Management (Chapters 3–5)

Goal/Theme: Describes the process concept, concurrent execution, process scheduling, interprocess communication, threads, and multi-core systems.

Chapter 3: Processes

Important Topics & Concepts:

• **Process Concept:** A program in execution; an active entity (whereas a program is a passive entity). A system consists of concurrently executing processes (OS code and user code).

• **Process Memory Layout:** Divided into four sections: text (executable code), data (global variables), heap (dynamically allocated memory), and stack (temporary data storage for function calls).

• **Process State:** A process changes state as it executes (e.g., running, waiting, terminated).

• **Process Control Block (PCB):** Data structure acting as a repository for all data needed to start or restart a process (CPU registers, scheduling info, memory management info, I/O status, accounting).

• **Context Switch:** Mechanism to save the current process state (context) and restore a new process state when the CPU changes tasks, usually triggered by an interrupt. Context switch time is highly dependent on hardware support.

- **Interprocess Communication (IPC):** Required for cooperating processes to exchange data.
  - **Shared Memory:** Processes establish a region of memory they both map into their address spaces for direct read/write access (e.g., POSIX Shared Memory).
  - **Message Passing:** OS provides facilities for processes to communicate by exchanging messages (e.g., Mach, Windows ALPC).
  - **Pipes:** Conduits for two processes to communicate. Ordinary pipes are unidirectional and require a parent–child relationship; Named pipes (FIFOs) are more general and allow unrelated processes to communicate.
  - **Client–Server Communication:** Sockets (communication over a network) and Remote Procedure Calls (RPCs, abstracting function calls across machines; used by Android's binder framework).
- **Updates (10th Ed.):** Simplifies discussion of scheduling to only CPU scheduling issues; new coverage includes memory layout of C programs, Android process hierarchy, Mach message passing, Android RPCs, and systemd replacement for init process.

Chapter 4: Threads & Concurrency

Important Topics & Concepts:

- **Thread Concept:** A basic unit of CPU utilization; threads within the same process share code, data, and resources, enabling multiple tasks simultaneously, especially beneficial on multicore systems.
- **Benefits of Multithreading:** Responsiveness, resource sharing, economy, and scalability.
- **Concurrency vs. Parallelism:** Concurrency means multiple threads are making progress (possible on single CPU); Parallelism means multiple threads are making progress simultaneously (requires multicore system).
- **Multicore Challenges:** Dividing/balancing work, data splitting, managing data dependencies (synchronization needed), testing, and debugging.
- **Multithreading Models (Mapping User to Kernel Threads):** Many-to-one, One-to-one (used by Windows and Linux), and Many-to-many.
- **Thread Libraries:** Provide APIs for managing threads (Pthreads for POSIX/UNIX/Linux/macOS, Windows API, Java threading).

• **Implicit Threading:** Strategy where languages or API frameworks manage thread creation/management based on identified parallel tasks (e.g., thread pools, fork-join framework in Java, Grand Central Dispatch).

• **Threading Issues:** Semantics of fork() and exec() calls change; supporting thread cancellation (asynchronous/deferred); and thread-local storage.

• **OS Examples:**

  ○ **Windows Threads:** Uses the one-to-one model.

  ○ **Linux Threads:** Uses the term **task**; uses the clone() system call, passing flags to determine resource sharing (allowing tasks to behave more like processes or threads).

• **Updates (10th Ed.):** Increased coverage of APIs for concurrent/parallel programming, revised Java threads section (includes futures), updated Grand Central Dispatch (includes Swift), new sections on fork-join parallelism and Intel thread building blocks.

Chapter 5: CPU Scheduling

Important Topics & Concepts:

• **CPU Scheduling:** Task of selecting a waiting process from the ready queue and allocating the CPU to it; done by the dispatcher.

• **Scheduling Criteria:** Used to evaluate algorithms: CPU utilization, throughput, turnaround time, waiting time, and response time.

• **Scheduling Algorithms:**

  ○ **FCFS (First-Come, First-Served):** Simple, but can cause short processes to wait for long ones.

  ○ **SJF (Shortest-Job-First):** Optimal (shortest average waiting time), but difficult to predict next CPU burst length.

  ○ **RR (Round-Robin):** Allocates CPU for a fixed time quantum; preempted if quantum expires.

  ○ **Priority Scheduling:** Allocates CPU to highest priority process; ties resolved by FCFS or RR.

  ○ **Multilevel Queue:** Processes partitioned into queues arranged by priority; scheduler executes the highest priority queue.

• **Thread Scheduling: Process-Contention Scope (PCS)** scheduling occurs among threads of the same process; **System-Contention Scope (SCS)** scheduling occurs among all threads in the system (used by Windows and Linux).
• **Multi-Processor Scheduling (SMP):** Each processor is self-scheduling. Strategies include **Load Balancing** (equalizing load across cores) and maximizing **Processor Affinity** (keeping threads on the same CPU to retain cache contents).
• **Real-Time CPU Scheduling:**
  ○ **Soft Real-Time:** Gives preference to real-time tasks.
  ○ **Hard Real-Time:** Provides timing guarantees; task must be serviced by its deadline. Algorithms include **Rate-Monotonic** (static priority based on shorter period) and **Earliest-Deadline-First (EDF)** (priority based on earlier deadline).
• **Operating System Examples:**
  ○ **Linux CFS (Completely Fair Scheduler):** Assigns a proportion of CPU time based on *virtual runtime* (vruntime); supports NUMA-aware load balancing.
  ○ **Windows 10:** Uses a preemptive, 32-level priority scheme.
  ○ **Solaris:** Uses six scheduling classes mapped to a global priority; CPU-intensive threads get lower priority/longer time quanta, I/O-bound get higher priority/shorter time quanta.
• **Updates (10th Ed.):** Revised multilevel queue and multicore scheduling; integrated coverage of NUMA-aware scheduling and Linux CFS modifications; new coverage of heterogeneous multiprocessing and Windows 10 scheduling.

Part Three: Process Synchronization (Chapters 6–8)

Goal/Theme: Methods for process synchronization and deadlock handling.

Chapter 6: Synchronization Tools

Important Topics & Concepts:

• **Critical Section Problem:** Protocol design for processes to cooperatively share data, avoiding race conditions.
• **Requirements for Solutions:** (1) **Mutual Exclusion** (only one process in critical section at a time), (2) **Progress** (cooperative determination of next entrant), and (3) **Bounded Waiting** (limit on wait time before entry).

• **Kernel Types:** Nonpreemptive kernels inherently prevent race conditions on kernel structures as only one process is active in kernel mode.
• **Peterson's Solution:** A software-based solution for two processes that satisfies all three critical section requirements.
• **Hardware Support:** Includes memory barriers, the compare-and-swap (CAS) instruction, and atomic variables.
• **Mutex Locks:** Binary lock (available or not); process must acquire before critical section and release afterward to enforce mutual exclusion.
• **Semaphores:** Synchronization tool with an integer value, capable of solving various synchronization problems beyond just mutual exclusion.
• **Monitors:** High-level abstract data type for synchronization; uses **condition variables** to allow processes to wait for specific conditions.
• **Liveness:** Refers to hazards where a process waits indefinitely, violating progress/bounded waiting (e.g., deadlock, livelock).
• **Priority Inversion:** Scheduling challenge where a high-priority process waits for a low-priority process to release a resource/lock.
• **Updates (10th Ed.):** Significant new coverage on architectural issues (instruction reordering, delayed writes), introduction of lock-free algorithms using CAS, and details on memory models/barriers.

Chapter 7: Synchronization Examples

Important Topics & Concepts:

• **Classic Synchronization Problems:** Examples used to test new synchronization schemes:
  ○ **Bounded-Buffer Problem:** Producers produce items, consumers consume them, sharing a fixed-size buffer.
  ○ **Readers–Writers Problem:** Readers can access shared data concurrently, but writers require exclusive access.
  ○ **Dining-Philosophers Problem:** Illustrates the challenge of allocating multiple resources (chopsticks) without deadlock.

• **Synchronization within the Kernel:** Linux uses atomic variables, spinlocks, and mutex locks. On single-core systems, spinlocks are replaced by enabling/disabling kernel preemption.
• **POSIX Synchronization:** Provides mutex locks, and both **Named Semaphores** (easily shared by unrelated processes) and **Unnamed Semaphores** (placed in shared memory).
• **Java Synchronization:** Built-in **Monitors** (language level) and API support for reentrant locks, semaphores, and condition variables.
• **Alternative Approaches:**
○ **Transactional Memory:** Memory read-write operations that execute atomically; the system handles atomicity rather than the developer using explicit locks (avoids deadlock).
○ **Functional Languages (e.g., Erlang, Scala):** Disallow mutable state, making them immune to race conditions and critical section issues.
• **Updates (10th Ed.):** Focuses on classical problems and API support for solving them; includes new coverage of POSIX semaphores and condition variables, and a new section on Java synchronization.

Chapter 8: Deadlocks

Important Topics & Concepts:

• **Deadlock Definition:** A situation where every process in a set is waiting for an event (usually a resource) that can only be caused by another waiting process in the set. Synchronization tools (mutex locks, semaphores) are common sources of deadlock.
• **Four Necessary Conditions for Deadlock:** Deadlock is only possible if all four conditions hold simultaneously.

1. **Mutual Exclusion:** At least one resource is nonsharable.
2. **Hold and Wait:** A process holds resources while waiting for more.
3. **No Preemption:** Resources cannot be forcibly taken from a process.
4. **Circular Wait:** A chain of processes exists, where each is waiting for a resource held by the next process in the chain.

• **Modeling:** Deadlocks can be precisely described using a **Resource-Allocation Graph**; a cycle in the graph indicates a possible deadlock.
• **Methods for Handling Deadlocks:** Prevention, Avoidance, or Detection/ Recovery.
• **Deadlock Prevention:** Ensures that at least one of the four necessary conditions cannot hold (eliminating circular wait is often the most practical technique).
• **Deadlock Avoidance:** Requires processes declare maximum resource needs; the **Banker's Algorithm** dynamically examines the resource-allocation state to ensure the system remains in a **safe state** (where deadlock is impossible) before granting resources.
• **Deadlock Detection:** Algorithm that evaluates resource/process status on a running system to find deadlocked sets.
• **Recovery from Deadlock:** Involves aborting one or more processes in the cycle, or preempting resources (requiring victim selection and process **rollback** to a safe state).
• **Updates (10th Ed.):** Minor updates, including new sections on livelock and deadlock as a liveness hazard, and coverage of Linux lockdep and BCC deadlock detector tools.

Part Four: Memory Management (Chapters 9–10)

Goal/Theme: Managing main memory during execution to improve CPU utilization and speed.

Chapter 9: Main Memory

Important Topics & Concepts:

• **Address Translation:** OS converts a **Logical Address** (CPU generated) into a **Physical Address** (seen by memory unit) via a **Memory Management Unit (MMU)**.
• **Protection:** Hardware mechanisms like base and limit registers protect the OS from user processes and processes from each other.
• **Contiguous Memory Allocation:** Each process is allocated a single, contiguous block of memory. Main problem is **external fragmentation**.

• **Paging:** Most common memory management technique. Physical memory is divided into fixed-size **frames**; logical memory into equal-sized **pages**.

  ○ Paging avoids external fragmentation but introduces **internal fragmentation** (unused space within the final page of a process).

  ○ **Translation Look-aside Buffer (TLB):** A hardware cache for page-table entries, improving address translation speed.

  ○ **Protection:** Implemented using protection bits (read/write/execute) and a **valid–invalid bit** in the page table entries.

  ○ **Shared Pages:** Used for sharing common, reentrant code (like system libraries) by having multiple page tables map to the same physical frames.

• **Structure of the Page Table:** Methods to handle large logical address spaces: **Hierarchical Paging** (paging the page table itself, e.g., two-level or four-level for 64-bit architectures), **Hashed Page Tables**, and **Inverted Page Tables**.

• **Swapping:** Moving pages to/from secondary storage (backing store).

• **Updates (10th Ed.):** Added coverage of ARMv8 64-bit architecture; swapping coverage focuses on pages rather than processes; segmentation coverage was eliminated.

Chapter 10: Virtual Memory

Important Topics & Concepts:

• **Virtual Memory:** Technique allowing execution of processes not entirely in memory. Benefits include programs larger than physical memory, efficient process creation, and shared memory/files.

• **Demand Paging:** Pages are loaded into memory only when they are referenced (demanded).

• **Page Fault:** Occurs when a process accesses a page marked invalid (not in memory). Requires servicing the interrupt, reading the page from the backing store into a free frame, updating tables, and restarting the instruction. Must maintain a low page-fault rate for performance.

  ○ **Major/Hard Page Faults:** Page not in memory.

  ○ **Minor/Soft Page Faults:** Page is legal but needs to be mapped (e.g., shared library loaded by another process).

• **Copy-on-Write (CoW):** Allows parent and child processes to share the same pages initially; a private copy is made only when a page is modified, speeding up process creation.
• **Page Replacement Algorithms:** Needed when no free frames are available: FIFO, Optimal (OPT), LRU, and **LRU-Approximation Algorithms** (e.g., Clock algorithm, used by most modern OSes).
• **Allocation of Frames: Global replacement** (selects victim from any process) or **Local replacement** (selects victim only from the faulting process).
• **Thrashing:** Occurs when a process lacks enough frames to hold its active pages (locality), leading to excessive paging activity.
• **Locality and Working Set:** The **Locality Model** states that programs access memory in localized patterns. The **Working Set** is the set of pages a process is actively using; managing frames to fit the working set prevents thrashing.
• **Memory Compression:** Compresses pages to save space; used on mobile systems (like iOS and Android) as an alternative to paging.
• **Kernel Memory Allocation:** Techniques like **Buddy System** and **Slab Allocation** are used to manage kernel memory conservatively, often without paging restrictions.
• **TLB Reach:** Amount of memory accessible via the TLB (entries * page size); increasing page size increases reach.
• **Updates (10th Ed.):** Includes new coverage of compressed memory, major/minor page faults, and updated memory allocation on NUMA systems and OS examples (Linux and Windows 10).

Part Five: Storage Management (Chapters 11–12)

Goal/Theme: Describes mass storage and I/O handling, including device control, interfaces, and performance.

Chapter 11: Mass-Storage Structure

Important Topics & Concepts:

• **Secondary Storage:** Provided primarily by **Hard Disk Drives (HDDs)** (mechanical) and **Nonvolatile Memory (NVM) devices** (electrical, e.g., SSDs).

• **HDD Structure:** Data organized in platters, tracks, sectors, and cylinders. Performance is dominated by seek time and rotational latency.
• **NVM Devices (SSDs):** Use flash NAND die chips; managed by a controller and a **Flash Translation Layer (FTL)**. Write performance is non-uniform due to the necessity of garbage collection and erasing blocks.
• **HDD Scheduling:** Algorithms like SCAN and C-SCAN optimize effective bandwidth and response time by minimizing head movement.
• **NVM Scheduling:** Typically uses simple FCFS, although some modify this to merge adjacent requests (e.g., Linux NOOP scheduler).
• **Error Detection and Correction:** Devices use techniques like Error Correction Codes (ECC) to detect and correct bit errors.
• **Storage Device Management:** Includes low-level formatting, partitioning, volume creation, and management of **bad blocks**.
• **Swap-Space Management:** Used for paging (moving individual pages) combined with virtual memory.
• **RAID (Redundant Array of Independent Disks):** Provides redundancy and performance improvement through striping. RAID Level 1 uses **mirroring** for high reliability. RAID Levels 5 and 6 use striping with parity/ECC redundancy.
• **Updates (10th Ed.):** New coverage of NVM devices (flash and SSDs), updated RAID coverage, and discussion of cloud/object storage.
Chapter 12: I/O Systems
Important Topics & Concepts:
• **I/O Subsystem:** Manages and controls I/O operations and devices, separating the kernel from device complexities.
• **I/O Hardware:** Interaction involves handshaking between the host and device controller, often managed via polling or interrupts. Large transfers are managed by **DMA (Direct Memory Access)** to offload the CPU.
• **I/O Interfaces:** Operating systems abstract physical devices into **block devices** (e.g., disk drives, supporting read/write/seek) and **character devices** (e.g., keyboards, supporting transfer one character at a time).
• **Kernel I/O Subsystem Services:**

  - **Buffering:** Using memory to store data during transfer, compensating for device speed mismatches and supporting copy semantics.
  - **Caching:** Using fast memory (like main memory) to hold copies of frequently accessed data.
  - **Unified Virtual Memory:** Systems that use page caching to cache both process pages and file data simultaneously.
  - **Power Management:** Critical in mobile OSes (e.g., Android uses power collapse and wakelocks).
- **Performance:** I/O is a major bottleneck; efficiency improvements focus on minimizing CPU execution time for drivers, reducing context switches, and minimizing memory bus load from data copies.
- **Updates (10th Ed.):** Updated technologies and performance numbers; expanded coverage of power management for mobile OSes and synchronous/asynchronous I/O.

Part Six: File System (Chapters 13–15)

Goal/Theme: Discusses how file systems provide mechanisms for online storage and access, covering structures, algorithms, and interfaces.

Chapter 13: File-System Interface

Important Topics & Concepts:

- **File Concept:** A logical storage unit that provides a uniform view of stored information on nonvolatile secondary storage.
- **File Operations:** Standard primitives include create, open, read, write, reposition, delete, and truncate.
- **File Locking:** Mechanisms for synchronizing access to shared files; locks can be **exclusive** (write access) or **shared** (read access). Locking may be **mandatory** (OS enforces) or **advisory** (up to programmers).
- **Access Methods:**
  - **Sequential Access:** Reads or writes are made sequentially, advancing a file pointer (tape model).
  - **Direct Access (Relative Access):** Reads or writes based on block numbers (useful for databases).

○ **Indexed Access:** Builds an index (containing pointers to data blocks) on top of direct access.

• **Directory Structure:** A symbol table translating file names into file control blocks (FCB, which contain file information and location). Common structures include Tree-Structured and Acyclic-Graph structures.

• **Protection:** Mechanisms to control improper access to files. The most common approach uses three user classifications: **Owner, Group, and Other (Universe)**, each assigned permissions (read, write, execute). **Access Control Lists (ACLs)** provide finer-grained control.

• **Memory-Mapped Files:** Treat file I/O as routine memory access by mapping the file contents into a process's virtual address space, useful for shared memory.

• **Updates (10th Ed.):** Improved coverage of directory structures and updated coverage of protection; expanded memory-mapped files section.

Chapter 14: File-System Implementation

Important Topics & Concepts:

• **File System Structure (Layered Design):** Levels progress from physical device handling (I/O control, device drivers) up to managing symbolic file names and metadata (logical file system, managing FCBs/inodes).

• **Allocation Methods (How disk space is allocated):**

○ **Contiguous Allocation:** Requires finding a single large hole of free space; suffers external fragmentation.

○ **Linked Allocation:** Each block contains a pointer to the next block; efficient space usage, but poor direct access. File Allocation Table (FAT) places pointers in a separate array.

○ **Indexed Allocation:** Uses an index block containing pointers to all file data blocks. Supports large files via **multilevel index** or a **combined scheme** (e.g., UNIX inode using direct, single, double, triple indirect blocks).

• **Free-Space Management:** Methods include bit vectors and linked lists, with optimizations like grouping and counting.

• **Performance:** Achieved via caching (buffer cache, unified virtual memory/page cache). Write performance: **Synchronous writes** (caller waits for data to reach device) vs. **Asynchronous writes** (data buffered in cache, control returns immediately).

• **Recovery:** Consistency checker programs (like fsck) detect and fix inconsistencies caused by crashes. **Journaling** (log-based recovery) writes all metadata changes to a sequential log before applying them to the file system structure to ensure consistency. Systems like WAFL/ZFS use **copy-on-write** to new blocks to avoid overwriting old data.

• **Backup/Restore:** Full backups and subsequent incremental backups are used to recover from data loss.

• **Updates (10th Ed.):** Updated with coverage of TRIM and the Apple File System (APFS); expanded discussion of journaling and performance.

Chapter 15: File-System Internals

Important Topics & Concepts:

• **File System Context:** Files are stored on storage devices organized into partitions or volumes.

• **Partitions and Mounting:** A device can have multiple partitions (raw or cooked). **Mounting** attaches a file system to a directory in the existing file hierarchy, making it available for access.

• **Virtual File Systems (VFS):** Layer of abstraction that separates file system logic from underlying implementation details, allowing OS to support multiple file system types seamlessly.

• **Remote File Systems:** Accessing files over a network (e.g., NFS).

• **Consistency Semantics:** Rules specifying how multiple users access a shared file simultaneously (when modifications are observable by others). Related to synchronization algorithms.

• **NFS (Network File System):** Uses remote procedure calls (RPC) and maintains two caches (file-attribute cache and file-blocks cache) to boost performance.

Part Seven: Security and Protection (Chapters 16–17)

Goal/Theme: Mechanisms for security (defending integrity) and protection (controlling access to resources).

Chapter 16: Security

Important Topics & Concepts:

• **Security vs. Protection:** Security measures confidence that integrity is preserved; Protection is the mechanism controlling access.

• **The Security Problem:** Threats exist at the hardware, operating system, application, and human levels.

• **Program Threats:** Writing programs to create security breaches (e.g., Trojan Horses, Viruses, Buffer Overflows, Covert Channels). Viruses can be Multipartite or Armored (obfuscated).

• **Cryptography:** Key security defense for confidentiality and authentication.

  ○ **Symmetric Encryption:** Same key for encrypt/decrypt (e.g., AES).

  ○ **Asymmetric Encryption (Public-Key):** Different keys for encrypt/decrypt; uses key pairs (public and private).

  ○ **TLS (Transport Layer Security):** Uses public-key cryptography to establish secure symmetric session keys for network communication.

• **User Authentication:** Proving identity (e.g., passwords, multi-factor authentication). Passwords should be stored securely using one-way transformation functions (hashing) combined with a random "salt".

• **Implementing Security Defenses:** Adopting **Defense in Depth** (multiple layers of protection). Includes security policies, vulnerability assessment (penetration tests), intrusion prevention/detection, auditing, and encryption of storage devices.

• **Updates (10th Ed.):** Revised/updated terms for current threats (ransomware, remote access tools), emphasized principle of least privilege, updated encryption technologies, revised code-injection coverage, and new summary of security defenses.

Chapter 17: Protection

Important Topics & Concepts:

• **Protection Goals:** Providing mechanisms to control the access of processes and users to resources. Mechanisms determine *how* something is done; Policies determine *what* is done.

• **Principles of Protection:**

    ○ **Principle of Least Privilege:** Giving programs/users only the minimum privileges needed.
    ○ **Need-to-Know Principle:** A process can access only those objects currently required for its task.
    ○ **Compartmentalization:** Protecting individual system components with specific permissions.

• **Protection Rings/Modes:** Hierarchical structures where Ring 0 is the highest privilege (kernel mode) and Ring 3 is the lowest (user mode).
• **Domain of Protection:** Set of resources a process can access. Systems like UNIX use user ID/group ID to define domains.
• **Access Matrix:** Abstract model describing access rights (operations) of domains over objects. Rights include read, write, execute, and potentially **copy** or **owner** rights.
• **Implementing the Access Matrix:** Can be implemented via **Access Lists** (column lists per object) or **Capability Lists** (row lists per domain).
• **Capability-Based Systems:** Capabilities are protected pointers/tokens used to access objects.
• **Linux Capabilities:** Allows administrators to selectively grant fine-grained root powers to tasks, rather than giving all or nothing access.
• **Updates (10th Ed.):** Major changes include updated discussion of protection rings (referencing Bell–LaPadula and ARM TrustZones), expanded coverage of the need-to-know principle and Mandatory Access Control (MAC), and added subsections on Linux capabilities, sandboxing, and code signing.

Part Eight: Advanced Topics (Chapters 18–19)

Goal/Theme: Discusses virtual machines, networks, and distributed systems.

Chapter 18: Virtual Machines

Important Topics & Concepts:

• **Virtualization:** Abstracting physical hardware into several execution environments, creating the illusion that each environment runs on private hardware.

• **Benefits:** System consolidation (running multiple OSes on one physical server), rapid deployment via templating, and efficient management.
• **Building Blocks:** A **Virtual CPU (VCPU)** represents the guest CPU state. Virtualization relies on techniques like **trap-and-emulate** and **binary translation**.
• **Hardware Support:** Modern CPUs (Intel VT, AMD V) provide features to accelerate virtualization, particularly memory management using **Nested Page Tables (NPTs)** in hardware (EPT/RVI).
• **Types of VMs/Hypervisors:** Type 1 (Bare Metal, runs directly on hardware) and Type 2 (Hosted, runs atop a host OS).
• **OS Components:** Virtual Machine Manager (VMM) handles scheduling (e.g., co-scheduling multiple VCPUs), I/O (often using paravirtualized drivers), and memory management.
• **Memory Management Optimization:** VMMs often overcommit memory and use techniques like **ballooning** (forcing guest to relinquish memory) and **page sharing** (deduplicating identical pages).
• **Application Containment:** Virtualization techniques provided by the OS, such as containers, zones, docker, and Kubernetes.
• **Updates (10th Ed.):** Added details on hardware assistance technologies; expanded coverage of application containment; new section discussing ongoing virtualization research (e.g., unikernels, separation hypervisors).

Chapter 19: Networks and Distributed Systems

Important Topics & Concepts:

• **Distributed Systems:** Systems where resources (clients, servers, storage) are dispersed across sites. Advantages include resource sharing, robustness, speedup, and communication.
• **Network Structure: LANs** (Local Area Networks, high speed, small distance) and **WANs** (Wide Area Networks, large distance, e.g., the Internet).
• **Communication Protocols (TCP/IP):** The foundation of the Internet. Includes IP (routing packets), UDP (unreliable, connectionless), and TCP (reliable, connection-oriented byte stream).

• **Naming:** DNS (Domain Name System) translates host names to network addresses.

• **Design Issues in Distributed Systems:**

  ○ **Robustness:** Ability to withstand failures (link, site, message loss); detection often involves a **heartbeat** procedure.

  ○ **Transparency:** System should appear to users like a conventional centralized system (hiding location/multiplicity of resources).

  ○ **Scalability:** Ability to handle increased load and expand easily.

• **Distributed File Systems (DFS):** File services across a network. Models include Client–Server (e.g., NFS) and Cluster-based (e.g., Hadoop, Google file system).

• **Remote File Access:** Achieved via **Remote Service** or **Caching** (copying data to client for faster access). Caching requires solving the **cache-consistency problem** (ensuring cached copy matches master copy).

• **Updates (10th Ed.):** Substantially updated to combine network and distributed systems coverage; added emphasis on TCP/IP and cloud storage; new coverage on distributed file systems (MapReduce, Hadoop).

Part Nine: Case Studies (Chapters 20–21)

Goal/Theme: Detailed examinations of Linux and Windows 10.

Chapter 20: The Linux System

Important Topics & Concepts:

• **History/Goals:** Started in 1991, derived from UNIX; major design goals include speed, efficiency, and adherence to POSIX standards.

• **Kernel Modules:** Supports dynamically Loadable Kernel Modules (LKMs) for device drivers and file systems, providing a modular approach with monolithic performance.

• **Process Model (Task):** Linux does not distinguish between processes and threads, referring to both as a **task**. The clone() system call controls resource sharing for creating tasks.

• **Scheduling:** Uses scheduling classes; the default scheduler is the **Completely Fair Scheduler (CFS)**, which uses virtual runtime (vruntime) to ensure a fair proportion of CPU time.

• **Memory Management:** Uses demand paging, copy-on-write, and page sharing. Kernel memory uses specialized techniques like **slab allocation**. Paging policy is a modified clock (LRU-approximation/LFU) algorithm.

• **File Systems (ext3/ext4):** Hierarchical directory tree; performs I/O through a page cache unified with the virtual memory system. Supports **Journaling** to ensure metadata consistency after crashes.

• **Security:** Follows UNIX model: Authentication (via salted one-way hash passwords) and Access Control.

Chapter 21: Windows 10

Important Topics & Concepts:

• **Design Principles:** Goals include security, reliability, compatibility, high performance, extensibility, portability, and energy efficiency.

• **Architecture:** Hybrid structure featuring a **Hyper-V Hypervisor** and an Executive layer providing core OS functions.

• **System Components (Executive):**

○ **Dispatcher:** Handles thread scheduling, context switching, and timer management.

○ **Object Manager:** Manages all manipulable entities (objects) using handles and reference counts.

○ **Memory Manager (MM):** Implements virtual memory with **demand paging and clustering**. Uses **section objects** for shared memory and memory-mapped files.

○ **Copy-on-Write (CoW):** Activated for shared data pages; a private copy is made if a process modifies the page.

○ **Physical Memory States:** Pages are tracked in seven states (free, zeroed, modified, standby, bad, transition, valid).

○ **Working-Set Management:** Assigns minimum (50 pages) and maximum (345 pages) limits to processes.

• **File System:** Uses NTFS (mentioned in recovery discussion, Chapter 14) and supports features like Terminal Services.

• **Security:** Uses Access Control Lists (ACLs) and integrity labels.

Part Ten: Appendices (A–D)

Goal/Theme: Covers influential older operating systems and detailed case studies of influential historical systems.

• **Chapter A: Influentia Operating Systems:** Discusses old influential operating systems that are no longer in use, such as Atlas, MULTICS, and IBM OS/360.

• **Chapter B: Windows 7:** Covers the history, design principles, system components, file system, and networking of Windows 7.

• **Chapter C: BSD UNIX:** Provides detailed coverage of UNIX history, design principles, process management, memory management, and file system.

• **Chapter D: Mach:** Covers the Mach operating system (which serves as the kernel foundation for macOS and iOS).