Okay, I can help expand on the concepts mentioned in the "os.docx" draft. Here's a more detailed explanation of each section:

**1. Introduction to Operating Systems**

* **1.1 Definition and Functions:**
  + An Operating System (OS) acts as the fundamental software layer managing all hardware and software components of a computer. Think of it as the government of a computer; it sets the rules and manages resources so that applications (like word processors or web browsers) can run effectively and users can interact with the machine without needing to know the intricate hardware details.
  + **Core Functions Expanded:**
    - **Resource Management:** The OS is responsible for allocating and managing the computer's resources. This includes deciding which program gets to use the CPU and for how long, assigning blocks of memory to running applications, handling requests to read from or write to devices like disks or printers, and organizing data storage.
    - **Process Management:** A process is essentially a program that is currently running. The OS oversees the entire lifecycle of a process, from its creation (starting an application), scheduling its time on the CPU, managing its state (running, waiting, etc.), and finally, its termination (closing the application).
    - **Memory Management:** The OS keeps track of which parts of the main memory (RAM) are currently being used and by which processes. It allocates memory space to new processes and deallocates it when processes finish, ensuring that processes don't interfere with each other's memory space.
    - **File Management:** This involves organizing data into files and directories on storage devices (like hard drives or SSDs). The OS provides functions to create, delete, open, close, read, and write files, and controls access permissions.
    - **I/O Management:** The OS manages communication between the CPU/memory and peripheral devices (Input/Output devices like keyboards, mice, monitors, disks). It handles the complexities of different hardware devices, providing a simpler interface for applications.
    - **Protection and Security:** The OS implements mechanisms to protect the system and user data from unauthorized access or interference. This includes user authentication, access control lists for files, and preventing processes from accessing memory or resources allocated to other processes.
* **1.2 Evolution of Operating Systems:**
  + **Serial Processing (1940s-1950s):** Early computers like ENIAC had no OS. Programmers interacted directly with the hardware, running one program (job) from start to finish before the next could begin. This was incredibly inefficient.
  + **Batch Systems (1950s-1960s):** To improve efficiency, similar jobs were grouped (batched) together and run sequentially by an operator. The OS's role was minimal, primarily transferring control automatically from one job to the next.
  + **Multiprogramming (1960s):** This was a major leap. The OS kept multiple jobs in memory simultaneously. When one job had to wait (e.g., for I/O), the OS switched the CPU to another ready job, significantly increasing CPU utilization.
  + **Time-Sharing Systems (1970s):** An extension of multiprogramming, time-sharing (or multitasking) allowed multiple users to interact with the system concurrently via terminals. The OS rapidly switches the CPU between user jobs, giving each user the impression they have dedicated access (e.g., Unix).
  + **Real-Time Systems:** These systems are critical for applications where timing is crucial. They must process data and respond within strict time limits (e.g., industrial control, flight control systems).
  + **Distributed Systems (1980s+):** Connect multiple independent computers over a network, allowing them to share resources and cooperate. The OS makes this network of machines appear as a single cohesive system (e.g., modern cloud platforms).
  + **Modern Systems:** Today's OSs typically combine many features: supporting multiple users and tasks simultaneously, operating across distributed networks, and often incorporating virtualization technologies.
* **1.3 OS Architecture Models:**
  + **Monolithic:** The entire OS (including scheduling, file system, networking, device drivers) runs as a single large program in kernel space. (+) Fast communication between components. (-) Difficult to modify or debug; a bug in one part can crash the whole system.
  + **Layered:** The OS is organized into layers, each built upon the services of the layer below it. (+) Modular design, easier debugging. (-) Can be slower due to overhead passing requests between layers.
  + **Microkernel:** Only essential OS functions (like basic process/memory management, communication) reside in the kernel. Other services (file systems, drivers) run as user-space processes. (+) More reliable and secure, easier to extend. (-) Performance overhead due to frequent user/kernel mode switches for service requests.
  + **Exokernel:** Provides minimal hardware abstractions, allowing application-level libraries maximum control over hardware resources. (+) Highly flexible and potentially high performance. (-) Very complex to program for.
  + **Virtual Machine:** An OS (the host) allows other operating systems (guests) to run concurrently on the same hardware, each believing it has its own machine. (+) Strong isolation, ability to run different OS types. (-) High resource consumption (CPU, memory).
* **1.4 System Calls and APIs:**
  + **System Call:** The mechanism applications use to request a service from the OS kernel. It's like a function call, but it transitions the processor from user mode to kernel mode to perform privileged operations (e.g., reading a file, creating a process). Common types manage processes, files, devices, system information, and communication.
  + **API (Application Programming Interface):** A set of functions, protocols, and tools for building software. OS APIs (like POSIX for Unix-like systems, Win32 for Windows) provide a more convenient, higher-level interface for programmers than raw system calls, often wrapping one or more system calls within a single API function.
* **1.5 OS Structures:**
  + **Kernel:** The core, privileged part of the OS that directly interacts with hardware and provides fundamental services.
  + **Shell:** The command-line interpreter or graphical interface that allows users to interact with the kernel and run programs.
  + **Utilities:** Additional programs and tools provided with the OS for system management and user tasks (e.g., file managers, text editors, system monitors).

**2. Process Management**

* **2.1 Process Concept:** A process is more than just program code; it's an active instance of a program running, including its current state, memory space (code, data, stack, heap), and allocated system resources.
* **2.2 Process States:** As a process executes, it transitions between states:
  + **New:** The process is being created by the OS.
  + **Ready:** The process has all resources needed to run but is waiting for the CPU scheduler to assign it processor time.
  + **Running:** The process's instructions are currently being executed by the CPU.
  + **Waiting/Blocked:** The process is paused, unable to continue until some event occurs (e.g., waiting for data from a disk read, waiting for user input).
  + **Terminated:** The process has finished execution or has been explicitly killed.
* **2.3 Process Control Block (PCB):** A data structure maintained by the OS for every single process. It acts as the process's "passport," containing all essential information the OS needs to manage it: its unique ID (PID), current state, the location of the next instruction to execute (Program Counter), CPU register values (saved when switched out), scheduling priority, memory allocation details, open files, I/O device status, etc..
* **2.4 Process Scheduling Queues:** The OS uses queues to manage processes:
  + **Job Queue:** Contains all processes in the system, regardless of their current state.
  + **Ready Queue:** Holds all processes residing in main memory that are ready and waiting to execute.
  + **Device Queues:** For each I/O device, there's a queue of processes waiting for that specific device.
* **2.5 Process Creation:**
  + Processes can create other processes, forming a hierarchy. The creator is the **Parent Process**, and the created process is the **Child Process**.
  + **Mechanisms (Common in Unix-like systems):**
    - fork(): Creates a new child process that is nearly an exact duplicate of the parent, including memory space and open files.
    - exec(): Replaces the memory space of the current process with a new program. Often used by a child process after fork() to run a different program.
    - wait(): A parent process can use this to pause until one of its child processes terminates.
* **2.6 Process Termination:**
  + **Normal:** The process executes its last statement and asks the OS to delete it (e.g., via exit()).
  + **Abnormal:** The process is terminated due to an error (e.g., division by zero, invalid memory access) or by another process (e.g., user kills the process, parent terminates a child via abort()).
* **2.7 Inter-Process Communication (IPC):** Mechanisms allowing different processes to communicate and synchronize their actions:
  + **Shared Memory:** Multiple processes are given access to the same region of memory. (+) Very fast communication (data doesn't need copying). (-) Processes must implement their own synchronization (e.g., using semaphores) to avoid conflicts.
  + **Message Passing:** Processes communicate by sending and receiving messages via the OS. (+) Safer as the OS manages the interaction, easier synchronization. (-) Slower due to kernel involvement and data copying. Can be **Direct** (sender/receiver name each other explicitly) or **Indirect** (messages sent to/received from shared mailboxes or ports).
  + **Pipes:** Simple, typically unidirectional communication channels often used between related processes (e.g., parent and child).
  + **Sockets:** Provide endpoints for communication, commonly used for network communication between processes on different machines, but also usable locally.
  + **Remote Procedure Calls (RPC):** Allows a process to call a procedure (function) in another process, potentially on a different machine, as if it were a local call. Hides the complexities of message passing.
* **2.8 Client-Server Communication:** Common IPC patterns, especially in networked/distributed systems:
  + **Sockets:** The fundamental building block for network communication, providing low-level send/receive operations.
  + **RPC:** A higher-level abstraction built on sockets or other mechanisms, making distributed programming look more like local programming.
  + **Java RMI (Remote Method Invocation):** An object-oriented version of RPC specific to Java, allowing objects on one Java Virtual Machine (JVM) to invoke methods on objects in another JVM.

**3. CPU Scheduling**

* **3.1 Basic Concepts:**
  + **CPU Scheduler:** The OS component that selects which process from the ready queue should be allocated the CPU next.
  + **Dispatcher:** The module that gives control of the CPU to the process selected by the scheduler. This involves switching context, switching to user mode, and jumping to the proper location in the user program to restart it.
  + **Scheduling Criteria:** Goals used to evaluate scheduling algorithms:
    - **CPU Utilization:** Keep the CPU as busy as possible.
    - **Throughput:** Maximize the number of processes completed per unit time.
    - **Turnaround Time:** Minimize the total time a process spends in the system (waiting + running).
    - **Waiting Time:** Minimize the time a process spends waiting in the ready queue.
    - **Response Time:** Minimize the time from a request submission until the first response is produced (important for interactive systems).
* **3.2 Scheduling Algorithms:** Different strategies for choosing the next process:
  + **First-Come, First-Served (FCFS):** Simple; the process that requests the CPU first gets it first (like a queue). (-) Non-preemptive, can lead to the "convoy effect" where short processes get stuck behind a long one. *Example: A(24ms), B(3ms), C(3ms). Order A,B,C. B waits 24ms, C waits 27ms.*
  + **Shortest Job First (SJF):** Selects the process with the shortest estimated next CPU burst. Provably optimal for minimizing average waiting time. Can be **preemptive** (Shortest Remaining Time First - SRTF) or **non-preemptive**. (-) Difficult to predict the next burst length accurately. *Example: A(24), B(3), C(3). Order B,C,A (non-preemptive). B waits 0ms, C waits 3ms.*
  + **Priority Scheduling:** Each process has a priority; the CPU is allocated to the process with the highest priority. Can be **preemptive** or **non-preemptive**. (+) Flexible. (-) Can lead to **starvation** where low-priority processes never run. (Aging can be used to prevent starvation).
  + **Round Robin (RR):** Designed for time-sharing systems. Each process gets a small unit of CPU time (time quantum/slice). If not finished, it's preempted and moved to the end of the ready queue. (+) Good response time for interactive tasks. (-) Performance depends heavily on quantum size; too large approaches FCFS, too small leads to high context-switching overhead.
  + **Multilevel Queue:** The ready queue is partitioned into several separate queues (e.g., foreground/interactive, background/batch). Each queue has its own scheduling algorithm (e.g., RR for foreground, FCFS for background). Scheduling must be done between queues (e.g., fixed priority, time slice). (+) Customizable. (-) Inflexible, processes don't move queues.
  + **Multilevel Feedback Queue:** Allows processes to move between queues based on their CPU burst characteristics. If a process uses too much CPU time, it can be moved to a lower-priority queue. If it waits too long, it can be moved up. (+) Adaptive, prevents starvation better than simple priority. (-) Most complex to implement and tune.
* **3.3 Thread Scheduling:** If an OS supports kernel-level threads, it schedules these threads, not entire processes:
  + **User-Level Threads:** Managed by a thread library in user space; the kernel is unaware of them. Fast creation/switching. (-) If one thread blocks on a system call, the entire process blocks.
  + **Kernel-Level Threads:** Managed directly by the OS. Slower creation/switching. (+) Blocking one thread doesn't block others in the process.
  + **Mapping Models:** Relate user threads to kernel threads (Many-to-One, One-to-One, Many-to-Many).
* **3.4 Real-Time Scheduling:** For systems with timing constraints:
  + **Hard Real-Time:** Tasks *must* complete by their deadline; failure is catastrophic (e.g., automotive airbag deployment).
  + **Soft Real-Time:** Missing a deadline is undesirable but not fatal; the system degrades (e.g., skipping frames in video playback).
  + **Algorithms:**
    - **Rate Monotonic:** Static priority based on task frequency (shorter period = higher priority).
    - **Earliest Deadline First (EDF):** Dynamic priority based on deadlines (closest deadline = highest priority).

**4. Concurrency and Synchronization**

* **4.1 Principles of Concurrency:** Managing multiple threads or processes that execute overlapping in time, potentially interacting with each other. The goal is usually to improve performance (by doing multiple things "at once") or responsiveness. It introduces challenges like race conditions and deadlocks.
* **4.2 The Critical Section Problem:**
  + **Critical Section:** A segment of code within a process/thread where it accesses shared resources (data structures, variables, files) that must not be concurrently accessed by other processes/threads.
  + **Requirements for a Solution:**
    - **Mutual Exclusion:** If one process is executing in its critical section, no other process can be executing in its critical section for the same shared resource.
    - **Progress:** If no process is in its critical section and some processes wish to enter, the selection of the next process cannot be postponed indefinitely.
    - **Bounded Waiting:** There must be a limit on the number of times 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 (prevents starvation).
* **4.3 Synchronization Mechanisms:** Tools to control access to critical sections:
  + **Lock Variables (Busy Waiting):** Simple flag indicating if the critical section is busy. Processes continuously check the lock ("spin"). (+) Simple. (-) Wastes CPU cycles (busy waiting).
  + **Semaphores:** An integer variable accessed only through two atomic operations:
    - wait() (or P()): Decrements the semaphore value. If the value becomes negative, the process blocks until the value is non-negative.
    - signal() (or V()): Increments the semaphore value. If processes are blocked on this semaphore, one is unblocked.
    - **Binary Semaphore (Mutex):** Value is 0 or 1, used for mutual exclusion. *Example: Initialize mutex = 1. Before critical section: wait(mutex). After: signal(mutex).*
    - **Counting Semaphore:** Value can be any non-negative integer, used to control access to a resource with multiple instances.
  + **Monitors:** A higher-level language construct that encapsulates shared data and the procedures that operate on it. Only one process can be active within the monitor at a time, ensuring mutual exclusion automatically. Uses **condition variables** for processes to wait for specific conditions inside the monitor (wait, signal).
  + **Message Passing:** Processes can synchronize by sending and receiving messages. A process might block waiting for a message needed to proceed.
* **4.4 Classical Synchronization Problems:** Used to test synchronization primitives:
  + **Producer-Consumer (Bounded Buffer):** One or more producers create data items and place them in a shared buffer; one or more consumers remove items. Requires synchronization to prevent producers from adding to a full buffer and consumers from removing from an empty buffer. Solved using semaphores (for buffer counts and mutual exclusion) or monitors.
  + **Readers-Writers:** Multiple processes (readers) can read a shared resource concurrently, but only one process (writer) can access it at a time for writing, and no readers can access it while a writer is writing. Variants prioritize either readers or writers.
  + **Dining Philosophers:** Models processes competing for limited resources (philosophers needing two forks to eat). Illustrates the problem of deadlock and starvation. Solutions involve careful resource allocation strategies (e.g., picking up forks in a specific order, using an arbitrator).
* **4.5 Deadlock and Starvation:**
  + **Deadlock:** A situation where two or more processes are blocked indefinitely, each waiting for a resource held by another process in the set.
  + **Starvation:** A process is repeatedly denied access to a resource it needs, even though the resource becomes available, often due to scheduling decisions (e.g., low priority) or synchronization mechanisms.

**5. Deadlocks**

* **5.1 Deadlock Characteristics:** Deadlock can only occur if these four conditions hold simultaneously:
  + **Mutual Exclusion:** At least one resource must be held in a non-sharable mode (only one process can use it at a time).
  + **Hold and Wait:** A process must be holding at least one resource and waiting to acquire additional resources currently held by other processes.
  + **No Preemption:** Resources cannot be forcibly taken away from a process holding them; they must be released voluntarily.
  + **Circular Wait:** A set of waiting processes {P0, P1, ..., Pn} must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn is waiting for a resource held by P0[cite: 45].
* **5.2 Resource Allocation Graph (RAG):** A visual way to model resource allocation:
  + Nodes: Processes (circles) and Resource Types (rectangles, dots inside represent instances).
  + Edges:
    - Request Edge: Directed edge from process P to resource R (P → R) means P requested R.
    - Assignment Edge: Directed edge from resource R to process P (R → P) means an instance of R is allocated to P.
  + **Cycle Detection:** If the graph contains **no cycles**, deadlock cannot exist. If it **contains a cycle**, deadlock *might* exist. If the cycle involves only single-instance resource types, deadlock *does* exist. *Example cycle: P1 → R1, R1 → P2, P2 → R2, R2 → P1.*
* **5.3 Deadlock Handling Strategies:**
  + **Deadlock Prevention:** Ensure that at least one of the four necessary conditions cannot hold.
    - Break Mutual Exclusion: Make resources sharable (often not possible).
    - Break Hold and Wait: Require processes to request all needed resources at once, or release all held resources before requesting new ones (can lead to low resource utilization or starvation).
    - Break No Preemption: Allow the OS to preempt resources (difficult for some resource types like printers).
    - Break Circular Wait: Impose a total ordering on all resource types and require processes to request resources in increasing order.
  + **Deadlock Avoidance:** Use information about future resource needs to ensure the system never enters an "unsafe" state (a state that *could* lead to deadlock). Requires processes to declare maximum resource needs in advance.
    - **Banker's Algorithm:** A key avoidance algorithm. The OS maintains the state of resource allocation (Available, Max need, Current Allocation, Remaining Need). Before granting a request, it checks if doing so would leave the system in a safe state (i.e., there exists a sequence of process completions even if all processes request their max resources). If safe, grant; otherwise, block the process.
  + **Deadlock Detection:** Allow the system to enter a deadlock state, detect it, and recover.
    - Detection involves periodically checking the RAG for cycles or using algorithms similar to the Banker's but just for detection.
  + **Deadlock Recovery:**
    - **Process Termination:** Abort one or more deadlocked processes (simple but loses work). Choose victims carefully (e.g., lowest priority, least progress made).
    - **Resource Preemption:** Take resources away from one or more processes and give them to others. Requires rollback/restart mechanisms for the preempted processes.

**6. Memory Management**

* **6.1 Memory Hierarchy:** Computer storage is organized in layers based on speed, cost, and volatility: Registers (fastest, smallest) → Cache → Main Memory (RAM) → Secondary Storage (SSD/HDD) → Tertiary Storage (tapes, optical) (slowest, largest). Memory management primarily deals with Main Memory.
* **6.2 Memory Management Requirements:** The OS component responsible for managing main memory must handle:
  + **Relocation:** Programs can be loaded into any available memory area; addresses must be translatable.
  + **Protection:** Prevent processes from accessing memory outside their allocated space.
  + **Sharing:** Allow processes to share specific memory regions (e.g., libraries).
  + **Logical/Physical Organization:** Bridge the programmer's view (logical address space) and the hardware reality (physical memory addresses).
* **6.3 Memory Partitioning:** How main memory is divided for processes:
  + **Fixed Partitioning:** Memory divided into fixed-size partitions at system startup. Simple, but leads to **internal fragmentation** (wasted space *within* a partition if the process is smaller than the partition).
  + **Dynamic Partitioning:** Partitions are created dynamically to fit the size needed by each process. Avoids internal fragmentation but leads to **external fragmentation** (memory becomes broken into many small, non-contiguous free blocks, none large enough for a new process, even if total free space is sufficient).
    - **Placement Algorithms** (when loading a process into a free block/hole): First-Fit (use first hole big enough), Best-Fit (use smallest hole big enough), Worst-Fit (use largest hole).
    - **Compaction:** Shuffle processes in memory to consolidate all free space into one large block. Time-consuming.

* **6.4 Address Binding:** Mapping logical addresses generated by the CPU to physical addresses in memory:
  + **Compile-Time:** If memory location is known beforehand, absolute addresses are generated. Program must be recompiled if location changes.
  + **Load-Time:** If location isn't known at compile time, relocatable code is generated. Final binding occurs when the program is loaded into memory.
  + **Execution-Time:** Binding delayed until run time. Allows processes to be moved during execution. Requires hardware support (Memory Management Unit - MMU).
* **6.5 Swapping:** A technique to run processes whose total memory needs exceed physical memory. Inactive processes are temporarily moved from main memory to a backing store (fast disk) and brought back in when needed. Forms the basis for virtual memory.

**7. Virtual Memory**

* **7.1 Basic Concept:** Allows the execution of processes that may not be completely resident in physical main memory. It separates the logical memory view (what the process sees) from the physical memory (what's actually in RAM). This enables running larger programs and increases multiprogramming by keeping only necessary parts of processes in memory. Often implemented using **demand paging**.
* **7.2 Demand Paging:**
  + **Pages and Frames:** Logical memory is divided into fixed-size blocks called **pages**. Physical memory is divided into fixed-size blocks called **frames** (usually the same size as pages).
  + **Page Table:** An OS-managed data structure for each process that maps its logical pages to physical frames in memory. Entries contain the frame number and status bits (e.g., valid/invalid, present, modified).
  + **Page Fault:** An event that occurs when a process tries to access a page whose corresponding frame is not currently in physical memory (marked invalid/not present in the page table). The OS traps this fault, finds the required page on the backing store (disk), loads it into a free frame in physical memory, updates the page table, and then resumes the process.
* **7.3 Page Replacement Algorithms:** When a page fault occurs and there are no free frames, the OS must choose an existing frame to "page out" (write to disk if modified) to make room for the required page. Algorithms decide which page to replace:
  + **FIFO (First-In, First-Out):** Replaces the page that has been in memory the longest. Simple but often performs poorly (might replace a heavily used page).
  + **Optimal (OPT/MIN):** Replaces the page that will not be used for the longest period in the future. Unimplementable (requires future knowledge) but serves as a benchmark.
  + **LRU (Least Recently Used):** Replaces the page that has not been accessed for the longest time. Assumes past behavior predicts future behavior. Good performance but complex/costly to implement perfectly (needs hardware support like counters or stack).
  + **Clock (Second Chance):** An approximation of LRU using a reference bit. Uses a circular queue. When looking for a victim, if the reference bit is 1 (recently used), set it to 0 and move on; if 0, replace it. Provides a balance between FIFO's simplicity and LRU's performance.
* **7.4 Thrashing:** A situation where the system spends excessive time paging (swapping pages in and out) instead of executing processes. Occurs when processes don't have enough frames allocated to hold their working set (the set of pages actively used). Leads to frequent page faults, low CPU utilization, and potentially system collapse. Can be addressed by reducing the degree of multiprogramming or allocating more frames to processes.
* **7.5 Memory-Mapped Files:** Allows a file on disk to be mapped directly into a process's virtual address space. Accessing bytes in memory corresponds to reading/writing bytes in the file, handled efficiently via the paging system.
* **7.6 Segmentation:** Memory management scheme where memory is divided into logical units called segments (e.g., code segment, data segment, stack segment), each with a different size. Addresses specify segment name/number and offset within the segment. Can be used alongside paging (segmented paging).

**8. File Management Systems**

* **8.1 File Concepts:**
  + **File:** A logical storage unit, a named collection of related information recorded on secondary storage.
  + **Attributes:** Metadata associated with a file: Name, Type (e.g., .txt, .exe), Size, Location (pointer to device location), Protection (permissions), Timestamps (creation, modification, access).
  + **Operations:** Common actions performed on files: Create, Write, Read, Reposition (seek), Delete, Truncate, Open, Close.
* **8.2 Access Methods:** How data within a file is accessed:
  + **Sequential Access:** Information is processed in order, one record after another (like a tape). Most common method.
  + **Direct Access (Relative Access):** File viewed as numbered blocks/records; can read/write records rapidly in any order. Essential for databases.
  + **Indexed Access:** An index (like an index in a book) contains pointers to various blocks, allowing direct access to records based on a key value. Built on top of direct access.
* **8.3 Directory Structure:** How files are organized logically:
  + **Single-Level:** All files in one directory. Simple but naming conflicts arise quickly.
  + **Two-Level:** Each user has their own directory (Master File Directory pointing to User File Directories). Solves naming conflicts between users.
  + **Hierarchical (Tree-Structured):** Standard structure with a root directory and nested subdirectories. Allows complex organization.
  + **Acyclic-Graph:** Allows directories/files to be shared by having multiple paths (links) to them, but without cycles.
  + **General Graph:** Allows cycles, potentially complicating traversal and deletion.
* **8.4 File System Implementation:** How files and directories are stored on disk:
  + **Allocation Methods** (how disk blocks are allocated to files):
    - **Contiguous Allocation:** Each file occupies a contiguous set of blocks on disk. (+) Simple, good performance (minimal head movement). (-) Suffers from external fragmentation, difficult to grow files.
    - **Linked Allocation:** Each file is a linked list of disk blocks; blocks may be scattered. Each block contains a pointer to the next. (+) No external fragmentation, easy file growth. (-) Only sequential access is efficient, pointers take space, reliability issue if a pointer is lost. (FAT - File Allocation Table is a variation).
    - **Indexed Allocation:** Brings all pointers for a file together into one block called an index block. Each file has its own index block containing addresses of data blocks. (+) Supports direct access, no external fragmentation. (-) Wasted space for index blocks (especially for small files), size limits based on index block size (can be overcome with multilevel indexing).
  + **Free Space Management:** Tracking available disk blocks:
    - **Bit Vector:** A sequence of bits, one per block (0=free, 1=allocated). Simple but can be large.
    - **Linked List:** Free blocks are linked together. Simple but slow to find contiguous space.
    - **Grouping:** Store addresses of N free blocks in one free block; the last points to another block of free block addresses.
    - **Counting:** Keep track of contiguous runs of free blocks (address of first block + count).
* **8.5 Recovery:** Protecting file system integrity:
  + **Consistency Checking:** Utilities (like fsck in Unix, chkdsk in Windows) scan metadata for inconsistencies and attempt repairs after a crash.
  + **Backups:** Periodically copying file system data to offline storage.
  + **Log-Structured File Systems (LFS):** Buffer all changes (metadata and data) in memory and write sequentially to a log on disk. Improves write performance and simplifies recovery after a crash.

**9. Distributed Operating Systems**

* **9.1 Distributed System Concepts:** A collection of independent computers connected by a network that appears to users as a single, coherent system. Key motivations include resource sharing (hardware, software, data), computation speedup (parallelism), reliability (fault tolerance), and communication.
* **9.2 Types of Distributed Systems:** Architectures vary:
  + **Client-Server:** Clients request services from dedicated servers.
  + **Peer-to-Peer (P2P):** All nodes act as both clients and servers, sharing resources directly.
  + **Three-Tier:** Extends client-server with a middle layer (e.g., business logic).
  + **Middleware-Based:** Use a software layer (middleware) to hide heterogeneity and provide common services.
* **9.3 Communication:** Essential for distributed systems:
  + **RPC (Remote Procedure Call):** Allows calling functions on remote machines.
  + **Message Passing:** Explicit send/receive operations.
  + **Sockets:** Low-level endpoints for network communication.
* **9.4 Synchronization:** Coordinating actions across multiple machines:
  + **Clock Synchronization:** Keeping physical clocks consistent is difficult. **Logical Clocks** (Lamport timestamps, Vector clocks) provide an ordering of events without relying on physical time.
  + **Leader Election:** Algorithms (e.g., Bully, Ring) to select a unique coordinator process among nodes.
* **9.5 Distributed Mutual Exclusion:** Ensuring only one process across the entire distributed system can access a shared resource at a time. Algorithms include:
  + **Centralized:** A single coordinator grants permission.
  + **Distributed:** Processes coordinate among themselves using message passing.
  + **Token-Passing:** A unique token is passed around; only the holder can enter the critical section.
  + **Quorum-Based:** Processes request permission from a subset (quorum) of nodes.
* **9.6 Distributed File Systems (DFS):** File systems spanning multiple machines:
  + **Examples:** NFS (Network File System), AFS (Andrew File System).
  + **Features:**
    - **Transparency:** Hide the location of files from users (Location Transparency) and the fact that files might be replicated (Replication Transparency).
    - **Replication:** Storing copies of files on multiple servers for fault tolerance and performance. *Diagram Concept: A client machine interacts with a local DFS module, which contacts remote servers to access files, possibly retrieving from replicated copies.*
* **9.7 Fault Tolerance:** Designing systems to continue operating despite failures:
  + **Types of Failures:** Crash (node stops), Omission (message lost), Timing (response too early/late), Byzantine (node behaves arbitrarily/maliciously).
  + **Recovery Techniques:** Forward (find a new state to proceed from), Backward (rollback to a previous consistent state).
  + **Replication:** Key technique for fault tolerance. **Active Replication:** All replicas process requests. **Passive Replication:** One primary processes requests, backups take over if primary fails.