### **Mod 1: OS Structure and Shell**

1. **OS Structure:**
   * **Monolithic:** Single large kernel that handles everything (e.g., Linux).
   * **Microkernel:** Minimal kernel that only provides essential services (e.g., Minix).
   * **Layered Structure:** OS is divided into layers, each providing a specific function, where each layer interacts only with the layers directly above or below it.
   * Hybrid
2. **Hard Time vs. Real-Time:**
   * **Hard Real-Time:** Critical systems where missing a deadline is unacceptable (e.g., pacemaker).
   * **Soft Real-Time:** Missing a deadline is undesirable but not catastrophic (e.g., multimedia systems).
3. **Multiprogramming vs. Time Sharing:**
   * **Multiprogramming:** Multiple programs are loaded into memory, but only one executes at a time. It aims to maximize CPU utilization.
   * **Time Sharing:** A form of multiprogramming where CPU time is shared among users or processes using time slices.
4. **Shell and Shell Scripts:**
   * **Shell:** A command-line interface that interprets user commands (e.g., Bash).
   * **Shell Script:** A series of shell commands executed as a script (e.g., automation of tasks).
5. **System Calls:**
   * Interface between user programs and the kernel. System calls allow user programs to request services from the OS (e.g., open(), read(), write(), fork()).
6. **Device Drivers:**
   * Software that allows the OS to interact with hardware devices, providing an interface for I/O operations.
7. **Commands:**
   * **Linkers/Loaders:** Linkers combine object files into executable programs, while loaders load the executable into memory for execution.

### **Mod 2: Process Management**

1. **PCB (Process Control Block):**
   * Data structure that stores information about a process (e.g., PID, state, program counter, CPU registers).
2. **Process States:**
   * **New:** Process is being created.
   * **Ready:** Process is ready to execute, waiting for CPU.
   * **Running:** Process is executing.
   * **Blocked:** Process is waiting for some event (I/O, signal).
   * **Terminated:** Process has finished execution.
3. **ULT vs. KLT:**
   * **ULT (User-Level Thread):** Managed entirely by user-level libraries, not visible to the OS.
   * **KLT (Kernel-Level Thread):** Managed by the OS kernel and scheduled by it.
4. **Schedulers and Queues:**
   * **Long-term Scheduler:** Decides which processes are admitted to the system.
   * **Short-term Scheduler:** Decides which process gets CPU time.
   * **Medium-term Scheduler:** Handles swapping of processes between RAM and disk.
   * **Ready Queue:** Processes waiting to be executed.
   * **Wait Queue:** Processes waiting for an event (e.g., I/O completion).
5. **Cascading Termination:**
   * When a process terminates, it may cause other dependent processes to terminate.
6. **Zombie Process:**
   * A process that has finished execution but still has an entry in the process table because its parent has not read its exit status.
7. **Preemptive vs. Non-Preemptive Scheduling:**
   * **Preemptive:** OS can suspend a running process to allocate CPU to another (e.g., Round Robin, Shortest Job First).
   * **Non-Preemptive:** Process runs until it voluntarily gives up CPU.
8. **MLT vs. MLFQ:**
   * **MLT (Multilevel Feedback Queue):** A dynamic queue structure where processes can move between queues based on their behavior.
   * **MLFQ (Multilevel Queue):** A static queue structure where processes are permanently assigned to a queue based on priorities.

### **Mod 3: Synchronization and Deadlock**

1. **Hardware vs. Software Approach:**
   * **Hardware:** Uses hardware mechanisms (e.g., atomic instructions, interrupts).
   * **Software:** Uses algorithms and data structures (e.g., semaphores, monitors) to manage synchronization.
2. **Producer-Consumer Problem:**
   * A classic synchronization problem where a producer generates data and a consumer processes it. The challenge is managing shared buffers.
3. **Bounded vs. Unbounded Buffer:**
   * **Bounded Buffer:** Fixed-size buffer, limited number of items can be stored.
   * **Unbounded Buffer:** Infinite or practically infinite size buffer.
4. **Deadlock:**
   * A situation where a set of processes are blocked because each process is holding a resource and waiting for another.
   * **Prevention:** Ensures at least one of the Coffman conditions for deadlock (mutual exclusion, hold and wait, no preemption, circular wait) is violated.
   * **Avoidance:** Uses algorithms (e.g., Banker's Algorithm) to check resource allocation to avoid deadlock.
   * **Recovery:** Involves killing processes or rolling back to a safe state.
5. **Semaphore:**
   * A synchronization tool used to manage access to shared resources. Types:
     + **Binary Semaphore (Mutex):** Takes values 0 or 1.
     + **Counting Semaphore:** Can take multiple integer values.
   * **Dining Philosopher's Solution:** Use semaphores or mutexes to avoid deadlock and ensure mutual exclusion.
6. **Monitor:**
   * A high-level synchronization construct that combines mutexes and condition variables, providing a solution to the dining philosopher's problem and other synchronization challenges.
7. **Mutual Exclusion, Livelock, Starvation:**
   * **Mutual Exclusion:** Ensures that only one process accesses critical resources at a time.
   * **Livelock:** Processes are continuously changing states in response to each other but never making progress.
   * **Starvation:** A process is indefinitely delayed from executing due to other processes getting priority.

### **Mod 4: File Systems and I/O**

1. **File Structure and Architecture:**
   * File structure refers to how data is stored in files (e.g., text files, binary files). File architecture describes how files and directories are organized in the OS.
2. **File Allocation:**
   * The method used to store file data on disk (e.g., contiguous, linked list, indexed).
3. **Hashed File Organization:**
   * Uses a hash function to compute the location of the file or data on disk, providing fast access.
4. **Inode:**
   * Data structure in Unix-like systems that stores metadata about a file, such as its size, owner, permissions, and data block addresses.
5. **Disk Scheduling:**
   * Algorithms that determine the order in which disk I/O operations are processed (e.g., FCFS, SSTF, SCAN, C-SCAN,LOOK ,C-LOOK).
6. **I/O Buffering:**
   * Storing data temporarily in memory (buffer) while transferring between devices and memory to improve efficiency.

### **Mod 5: Memory Management**

1. **Swapping:**
   * The process of moving processes in and out of main memory (RAM) to the disk to manage memory efficiently.
2. **Memory Management:**
   * Techniques to handle the allocation, tracking, and deallocation of memory (e.g., paging, segmentation).
3. **Paging Structure:**
   * Memory management scheme that eliminates the need for contiguous allocation of physical memory, dividing memory into fixed-size pages.
4. **Fragmentation:**
   * **External Fragmentation:** Unused memory between allocated regions.
   * **Internal Fragmentation:** Wasted memory within allocated regions due to fixed-size allocation.
5. **Segmentation:**
   * Memory management scheme that divides memory into segments, each representing a different logical unit (e.g., code, data, stack).
6. **TLB (Translation Lookaside Buffer):**
   * A cache that stores recently used page table entries to speed up virtual to physical address translation.

**POST (Power-On Self-Test):** When a computer is turned on, it runs POST, which is a diagnostic test to check if hardware components like memory, disk drives, and the processor are working correctly before loading the operating system.

**Booting Process:** The process where a computer loads the operating system into memory after the POST. This includes loading the bootloader (like GRUB in Linux), which then loads the OS kernel, initializes drivers and system processes, and starts user applications.

**Scheduling Algorithm in Linux:** Linux primarily uses the Completely Fair Scheduler (CFS) for process scheduling, designed to provide fair CPU time to processes by balancing tasks and maintaining performance.

**Software System Sequence:** This is the order of operations or events in a software system, often represented in a sequence diagram showing how different parts of the system interact over time, like how user actions trigger responses from the system.  
  
**Translation Lookaside Buffer (TLB)** is a cache in the CPU’s memory management unit that stores recent virtual-to-physical address translations. When the CPU needs to access memory, it first checks the TLB:

* **TLB Hit**: If the address is in the TLB, it quickly retrieves the physical address.
* **TLB Miss**: If not, it consults the page table, which takes longer.

This speeds up memory access by reducing the need for repeated, slower page table lookups.

1. **File System Structure**: This refers to how data is organized and stored on a disk. Common structures include hierarchical (tree-like) arrangements, with directories and subdirectories. Examples include FAT32, NTFS, and ext4.
2. **Paging & Segmentation**: Both are memory management schemes. Paging divides memory into fixed-sized pages, while segmentation divides memory into variable-sized segments based on logical divisions of a program.
3. **Semaphores**: Used for process synchronization, semaphores help manage resource access by signaling. They prevent race conditions in critical sections.
4. **Threads**: Lightweight processes that allow multiple sequences of execution within the same program. Threads share memory, which enhances efficiency in multitasking.
5. **Dining Philosophers Problem**: A classic synchronization problem that demonstrates challenges in avoiding deadlocks and managing shared resources.
6. **File Access Methods**: Methods include sequential (reading in order), direct (random access), and indexed (using an index to locate files quickly).
7. **Deadlock Recovery**: Methods to handle deadlocks include terminating processes, resource preemption, and process rollback.
8. **PCB (Process Control Block)**: A data structure that stores process information like state, priority, and program counter, essential for context switching.
9. **Memory Allocation - First Fit, Best Fit, Worst Fit**: Strategies to allocate memory:
   * **First Fit** allocates the first sufficient space.
   * **Best Fit** allocates the smallest space that fits.
   * **Worst Fit** allocates the largest available space.
10. **Inode**: A data structure in Unix-based systems that stores file metadata (permissions, size, owner).
11. **Race Condition**: Occurs when multiple processes or threads try to access shared resources simultaneously, leading to unpredictable outcomes.
12. **Binary Semaphore vs Mutex**: Binary semaphores are signaling mechanisms that allow two states (0 or 1). Mutexes ensure mutual exclusion, often used when only one thread can access a resource.
13. **Fragmentation**:
    * **Internal Fragmentation** occurs when fixed-size memory blocks are allocated but not fully used.
    * **External Fragmentation** happens when free memory is scattered in small blocks.
14. **Banker’s Algorithm**: A deadlock avoidance algorithm that checks resource allocation based on maximum needs and available resources.
15. **Input/Output-Bound vs. CPU-Bound Processes**: I/O-bound processes spend more time waiting for I/O, while CPU-bound processes require more CPU time.
16. **Quality Parameters of Process Scheduling**: Factors include CPU utilization, throughput, turnaround time, waiting time, and response time.
17. **Interrupt Mechanism**: An interrupt is a signal to the CPU that an event needs attention. Interrupts prioritize urgent tasks without CPU polling.
18. **Memory Hierarchy**: Consists of cache, RAM, and storage. Cache is the fastest and closest to the CPU.
19. **Round Robin - Impact of Time Quantum**: Small quantum leads to more context switching, while a large quantum can cause higher response times.
20. **Monolithic vs. Microkernel**: Monolithic kernels provide faster performance but lack modularity. Microkernels are modular and secure but slower.
21. **I/O Buffer**: Temporary storage for I/O operations, reducing CPU idle time and enabling smoother data transfer.
22. **Convoy Effect**: When short processes get delayed by a long process, reducing system throughput.
23. **Multiprogramming vs. Multitasking**: Multiprogramming allows multiple processes to reside in memory, while multitasking allows multiple tasks to run concurrently.
24. **Types of Schedulers**:
    * **Long-Term Scheduler**: Decides which processes are admitted.
    * **Short-Term Scheduler**: Chooses which process to execute next.
    * **Medium-Term Scheduler**: Temporarily removes processes from memory (swapping).
25. **Pipeline**: Technique to divide processes into stages for increased instruction throughput, often seen in CPU instruction processing.
26. **Backwaiting**: Not commonly defined, possibly related to "backing store" in virtual memory or process wait states.
27. **Monitor**: A high-level synchronization mechanism that manages shared resource access using condition variables.
28. **Resource Allocation Diagram**: Graphs show resources, processes, and allocation. Key diagrams include resource allocation graphs, RAGs with request edges, and wait-for graphs.
29. **PCB Diagram**: Visual representation of the PCB, showing its attributes like process ID, state, and program counter.
30. **OS Architecture Diagram**: Illustrates OS components and their interactions, such as kernel mode, user mode, system calls, and device drivers.
31. **Zombie & Orphan Processes**:
    * *Zombie*: Process completed but still has an entry in the process table.
    * *Orphan*: Process whose parent has terminated.
32. **Counting vs. Binary Semaphore**:
    * *Counting*: Holds any integer value, used for resource counting.
    * *Binary*: Holds 0 or 1, used for mutual exclusion.
33. **Busy Waiting**: When a process repeatedly checks for a condition instead of releasing the CPU, leading to inefficient resource use.
34. **Semaphore Final Value**: Given 11 initial resources with 9 P (wait) and 5 V (signal) operations, the final semaphore value is 11 - 9 + 5 = 7.
35. **Real-World Applications of ULT (User-Level Threads)**: Web servers, games, and applications needing high concurrency without heavy kernel involvement.
36. **Multithreading Models**:
    * *Many-to-One*, *One-to-One*, and *Many-to-Many*.,One to Many
37. **Deadlock Avoidance in Dining Philosophers**: Use resource hierarchy or allow only 4 philosophers to sit.
38. **Proberen and Verhogen**: Dutch terms for "decrement" (P) and "increment" (V) operations on semaphores.
39. **ULT vs. KLT Differences**:
    * *ULT*: User-managed, blocking affects only that thread.
    * *KLT*: Kernel-managed, can allow thread scheduling even during blocking.
40. **SCAN vs. CSCAN**:
    * *SCAN*: Moves back and forth across disk.
    * *CSCAN*: Only moves in one direction and jumps to the start.
41. **Monitors**: Synchronization structures using mutual exclusion; the enum is often used to define states.
42. **Shell Commands**: Commands like ls, cd, mkdir, rm.
43. **Interpreter vs. Compiler**:
    * *Interpreter*: Executes code line-by-line.
    * *Compiler*: Translates code to machine language before execution.
44. **Round Robin Scheduling**: Processes get equal, fixed time slots for execution.
45. **Process Image**: Includes process code, data, stack, and registers.
46. **Context Switch vs. Mode Switch**:
    * *Context Switch*: Swaps the current process with another.
    * *Mode Switch*: Changes CPU mode between user and kernel.
47. **fork() and exec()**: fork() creates a new process; exec() replaces the current process memory with a new program.
48. **Paging vs. Segmentation**:
    * *Paging*: Divides memory into fixed-size pages.
    * *Segmentation*: Divides memory into logical segments.
49. **Monitors vs. Semaphores**: Monitors offer higher-level abstraction with mutual exclusion, while semaphores provide lower-level signaling.
50. **Process Termination Scenarios**: Processes terminate normally, due to errors, or by external intervention.
51. **Memory Allocation Methods**:
    * *First Fit*: First suitable space.
    * *Next Fit*: Starts from last allocated space.
    * *Best Fit*: Smallest suitable space.
    * *Worst Fit*: Largest suitable space.
52. **PCB (Process Control Block)**: Stores process info like PID, state, and CPU registers.
53. **Shells in Unix**: Common shells are *bash*, *sh*, *csh*, and *zsh*.
54. **Banker’s Algorithm**: Used to manage resource allocation and avoid deadlock by ensuring a safe state.
55. **FAT (File Allocation Table)**: Manages file storage by linking files to storage clusters.
56. **Test and Set Instruction**: Atomic operation to check and set a value, useful for implementing locks.
57. **Creating User-Level Threads**: Created within a user process using a threading library.
58. **UNIX Commands for Process Control**: Commands like ps, kill, bg, and fg.
59. **Thread Join vs. Termination**:
    * *Join*: Waits for a thread to complete.
    * *Termination*: Ends thread execution.
60. **Semaphore Use in Common Problems**:

* Producer-Consumer, Reader-Writer, and Dining Philosophers all use semaphores for synchronization.

1. **Changes in Process Image During Execution**: Stack and registers update frequently with process state.
2. **Reader-Writer Code**: Uses semaphores to manage read-write access.
3. **Busy Waiting**: Looping to wait for a resource, consuming CPU time.
4. **Multiprogramming vs. Multitasking**: Multiprogramming has multiple programs in memory, while multitasking allows multiple tasks to run.
5. **chmod Syntax**: chmod [permissions] [file].
6. **Inode in Unix**: Stores metadata like ownership, permissions, and location of file blocks.
7. **FAT Tables**: Track file allocation and are used in simple file systems.
8. **Semaphore in Reader-Writer Problem**: Ensures mutual exclusion for readers and writers.
9. **Deadlock Conditions**: Mutual exclusion, hold and wait, no preemption, and circular wait.
10. **Semaphore Without Busy Waiting**: Using blocking instead of a loop, where a process sleeps until it can proceed.
11. **Linker and Loader**: Linker combines object files, loader loads executable code into memory.
12. **Inode Structure**: Stores file attributes like permissions, size, and disk block pointers.
13. **Process-to-Frame Mapping**: Maps logical to physical memory via page tables.
14. **Process Creation in Unix**: fork() creates a process.
15. **Cascading Termination**: When a parent terminates, all child processes are also terminated.
16. **Set & Test Instructions**: Atomic operations used for synchronization
17. **File Management (Contiguous & Chained)**: Contiguous files use sequential blocks; chained files link blocks together.
18. **Deadlock Conditions**: Ensuring non-preemption can prevent deadlocks.
19. **Thread Implementation in Java**: Uses Thread class or Runnable interface.
20. **Types of Threads**: User-level and kernel-level threads.
21. **Software Solution for Synchronization**: Includes techniques like Peterson’s Algorithm.
22. **Peterson’s Algorithm**: Ensures mutual exclusion between two processes.
23. **Linux Commands for Process & Networking**: Process commands like ps, kill, networking commands like ping, ifconfig.
24. **Livelock**: Processes constantly change state without making progress, similar to deadlock but processes can change states.

**Deadlock Conditions**:

1. **Mutual Exclusion**: Only one process can use a resource at a time.
2. **Hold and Wait**: Processes holding resources can request additional ones.
3. **No Preemption**: Resources cannot be forcibly taken from processes.
4. **Circular Wait**: A cycle exists where each process holds a resource the next process needs.

**Deadlock Recovery**:

1. **Process Termination**: End one or more processes to break the deadlock.
2. **Resource Preemption**: Take resources from some processes and allocate them to others.
3. **Rollback**: Revert processes to an earlier safe state.
4. **Wait-for Graph**: Detect cycles in resource allocation and act to resolve deadlock.

**Dining Philosophers :   
Monitor:**  
monitor DiningPhilosophers {

enum {THINKING, HUNGRY, EATING} state[5]; // State of each philosopher

condition self[5]; // One condition variable per philosopher

// Initialization: all philosophers start thinking

DiningPhilosophers() {

for each i from 0 to 4:

state[i] = THINKING;

}

// Pickup method: try to acquire forks

procedure pickup(int i) {

state[i] = HUNGRY;

test(i); // Check if we can start eating

if (state[i] != EATING)

self[i].wait(); // Wait if forks are not available

}

// Putdown method: release forks

procedure putdown(int i) {

state[i] = THINKING; // Done eating, start thinking

test((i + 4) % 5); // Check left neighbor

test((i + 1) % 5); // Check right neighbor

}

// Test if a philosopher can eat

procedure test(int i) {

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

state[i] = EATING;

self[i].signal(); // Signal the philosopher to start eating

}

}

}

**Semaphore:**  
semaphore fork[5] = {1, 1, 1, 1, 1}; // each fork is a semaphore

philosopher(i):

wait(fork[i]); // pick up left fork

wait(fork[(i+1) % 5]); // pick up right fork

// Eat

signal(fork[i]); // put down left fork

signal(fork[(i+1) % 5]); // put down right fork

**Producer -Consumer Problem:**  
semaphore empty = N; // Counts empty slots (buffer size N)

semaphore full = 0; // Counts full slots

semaphore mutex = 1; // Mutual exclusion for buffer access

Producer:

wait(empty); // Wait if buffer is full

wait(mutex);

// Add item to buffer

signal(mutex);

signal(full); // Increment full slots count

Consumer:

wait(full); // Wait if buffer is empty

wait(mutex);

// Remove item from buffer

signal(mutex);

signal(empty); // Increment empty slots count

**Reader-writer problem:**  
semaphore rw\_mutex = 1; // controls access for writers

semaphore mutex = 1; // protects access to read\_count

int read\_count = 0;

Reader:

wait(mutex);

read\_count++;

if (read\_count == 1)

wait(rw\_mutex); // first reader locks the resource

signal(mutex);

// Read

wait(mutex);

read\_count--;

if (read\_count == 0)

signal(rw\_mutex); // last reader unlocks the resource

signal(mutex);

Writer:

wait(rw\_mutex);

// Write

signal(rw\_mutex);

**Petersons Problem:**int turn; // Indicates whose turn it is

bool flag[2]; // flag[i] = true if process i wants to enter CS

process0:

flag[0] = true;

turn = 1;

while (flag[1] && turn == 1); // wait if other process is in CS

// Critical Section

flag[0] = false; // exit CS

process1:

flag[1] = true;

turn = 0;

while (flag[0] && turn == 0); // wait if other process is in CS

// Critical Section

flag[1] = false; // exit CS

### **Virtual Memory:**

Virtual memory allows programs to use more memory than physically available by using disk space as "virtual" RAM. It provides isolation between processes and simplifies memory management.

### **Demand Paging:**

Demand paging is a memory management scheme where pages are only loaded into physical memory when they are needed (i.e., on-demand). When a page is accessed for the first time, a page fault occurs, and the page is brought into memory.

### **Page Replacement:**

When physical memory is full, the operating system must choose a page to evict and make room for a new one. Common page replacement algorithms include:

* **FIFO** (First-In, First-Out)
* **LRU** (Least Recently Used)
* **Optimal** (replace the page that won’t be used for the longest time)

### **Page Allocation:**

Page allocation refers to the process of allocating physical memory pages to virtual memory pages. It can be done dynamically or statically, depending on the system's memory management strategy.

### **Thrashing:**

Thrashing occurs when the system spends most of its time swapping pages in and out of memory, causing excessive disk I/O and poor system performance. It happens when there’s not enough physical memory for all active processes.