## Session 1: Operating System Fundamentals (2T)

This session introduces the foundational concepts of Operating Systems, their purpose, historical evolution, and key components.

- **Introduction to Operating System**
    - **Concept:** An Operating System (OS) is software that manages computer hardware and software resources and provides common services for computer programs. It acts as an intermediary between the user and the computer hardware.
    - **Purpose:**
        - **Resource Management:** Manages CPU, memory, I/O devices, and other hardware resources efficiently.
        - **Process Management:** Controls the execution of programs (processes).
        - **Memory Management:** Allocates and deallocates memory space for programs.
        - **File Management:** Organizes and manages files and directories.
        - **User Interface:** Provides a way for users to interact with the computer (GUI, CLI).
        - **Security & Protection:** Protects system resources and user data.
        - **Error Detection:** Handles and reports errors.
        - **System Services:** Provides services like program execution, I/O operations, communications, etc.
- **History of Operating System in Brief**
    - **Early Days (1940s-1950s):** No OS. Programmers interacted directly with hardware. Batch processing emerged, where jobs were grouped and run sequentially.
    - **Mainframe Era (1960s):** Introduction of simple batch systems, then spooling, and finally timesharing systems (e.g., CTSS, Multics, Unix). Timesharing allowed multiple users to share a single mainframe by rapidly switching between their tasks.
    - **Personal Computer Era (1970s-1980s):** Development of microprocessors led to personal computers. Early OS examples include CP/M, MS-DOS, Apple DOS/Mac OS. These were single-user, single-tasking systems initially.
    - **Graphical User Interface (GUI) Era (1980s-1990s):** Apple Macintosh (1984) popularized GUIs. Windows (starting with 1.0 in 1985) brought GUIs to the PC market. Linux emerged in 1991.
    - **Networked & Distributed OS (1990s-2000s):** Focus on networking, distributed systems, client-server models.
    - **Mobile & Cloud OS (2000s-Present):** Emergence of mobile OS (iOS, Android) and OS specifically designed for cloud environments and virtualization.
- **System Components/Services**
    - **Kernel:** The core of the OS. It controls everything in the system, manages low-level operations (process scheduling, memory allocation, I/O). It runs in **kernel mode** (privileged mode).
    - **Shell/User Interface:** Provides interaction with the user. Can be a Command Line Interface (CLI) or Graphical User Interface (GUI).
    - **File System:** Manages and organizes files and directories on storage devices.
    - **Process Management:** Deals with creation, scheduling, termination, and synchronization of processes.

    - **Memory Management:** Handles the allocation and deallocation of main memory.
    - **I/O System:** Manages input/output operations and device drivers.
    - **Networking:** Handles network communication protocols.
    - **Security & Protection:** Mechanisms to control access to resources and protect data.
    - **Error Handling:** Detects and handles various errors.

- **Interrupts & System Calls**

    - **Interrupts:**
        - **Definition:** An event that alters the normal execution flow of a program. It's a signal to the CPU that an event has occurred that requires immediate attention.
        - **Purpose:** Allows the OS to respond to external events (e.g., I/O completion, keyboard input, timer expiry) or internal events (e.g., division by zero, invalid memory access).
        - **Types:**
            - **Hardware Interrupts:** Generated by hardware devices (e.g., keyboard pressing a key, disk controller finishing a data transfer).
            - **Software Interrupts (Traps/Exceptions):** Generated by software. Can be caused by an error (e.g., trying to access protected memory) or by a specific instruction (a **system call**).
        - **Mechanism:** When an interrupt occurs, the CPU suspends its current task, saves its state, jumps to an **Interrupt Service Routine (ISR)**, executes the ISR, restores the saved state, and resumes the interrupted task.
    - **System Calls:**
        - **Definition:** The programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.
        - **Purpose:** Provide an interface between a process and the operating system. User programs cannot directly access hardware or protected memory; they must request the OS to perform such operations on their behalf via system calls.
        - **Mechanism:** When a program needs an OS service (e.g., `read()` from a file, `write()` to the screen, `fork()` to create a new process), it makes a system call. This generates a software interrupt (trap), which transfers control from user mode to kernel mode, allowing the OS to execute the privileged operation. After the service is completed, control returns to the user program in user mode.
        - **Examples:** `open()`, `read()`, `write()`, `close()`, `fork()`, `exec()`, `exit()`.

- **Introduction to Process Management**

    - **Process:** A program in execution. It's an active entity, unlike a program (which is a passive entity).
    - **Components of a Process:**
        - **Text Section:** The program code.
        - **Data Section:** Global variables.
        - **Heap:** Dynamically allocated memory during runtime.
        - **Stack:** Holds temporary data (function parameters, return addresses, local variables).
        - **Process Control Block (PCB):** A data structure maintained by the OS for each process, containing process state, program counter, CPU registers, CPU scheduling information, memory management info, accounting info, I/O status info.

- **Process States and Life Cycle**

- A process transitions between various states during its lifetime:
  - **New:** The process is being created.
  - **Ready:** The process is waiting to be assigned to a processor. It's in main memory, ready to run.
  - **Running:** Instructions are being executed on the CPU. At any given time, only one process per CPU core can be in this state.
  - **Waiting (Blocked):** The process is waiting for some event to occur (e.g., I/O completion, reception of a signal, availability of a resource). It cannot execute until the event occurs.
  - **Terminated:** The process has finished execution. Resources are deallocated.
- **Transitions:**
  - New -> Ready (admitted)
  - Ready -> Running (dispatch/schedule)
  - Running -> Ready (interrupt/time slice expired)
  - Running -> Waiting (I/O request/event wait)
  - Waiting -> Ready (I/O complete/event occurs)
  - Running -> Terminated (exit)

- **Multithreading**
  - **Thread:** A lightweight unit of execution within a process. A process can have multiple threads, each executing a part of the program's code concurrently.
  - **Difference from Process:**
    - **Processes:** Independent, have separate address spaces, resource-heavy to create/switch.
    - **Threads:** Share the same address space, code, data, and OS resources (open files, signals) of their parent process. Each thread has its own program counter, register set, and stack. Lighter-weight to create/switch.
  - **Benefits:**
    - **Responsiveness:** A multi-threaded application can remain responsive to user input even if a part of it is blocked or performing a long operation.
    - **Resource Sharing:** Threads within the same process share memory and resources, simplifying communication.
    - **Economy:** Cheaper to create and context switch between threads than processes.
    - **Scalability:** On multi-core processors, multiple threads can run in parallel, leading to faster execution for CPU-bound tasks.
  - **Types:**
    - **User-Level Threads (ULTs):** Managed by a thread library in user space. The kernel is unaware of their existence. Fast to create/switch, but if one ULT blocks, the entire process blocks.
    - **Kernel-Level Threads (KLTs):** Managed directly by the OS kernel. Slower to create/switch, but if one KLT blocks, other threads in the same process can still run. Most modern OS (Linux, Windows) support KLTs.
    - **Hybrid (Many-to-Many):** A flexible model where multiple user-level threads are mapped to a smaller or equal number of kernel-level threads.

## Session 2: Process Scheduling & Synchronization (2T)

This session delves into how the OS manages CPU time among processes and how it ensures proper coordination when multiple processes or threads access shared resources.

- **Basic Concepts (CPU Scheduling)**
    - **CPU-I/O Burst Cycle:** Process execution consists of a cycle of CPU execution (CPU burst) and I/O wait (I/O burst). Processes typically alternate between these two states.
    - **CPU Scheduler (Short-Term Scheduler):** Selects from among the processes in the ready queue and allocates the CPU to one of them.
    - **Dispatcher:** The module that gives control of the CPU to the process selected by the short-term scheduler. It involves:
        - Switching context (saving state of old process, loading state of new process).
        - Switching to user mode.
        - Jumping to the proper location in the user program to restart execution.
    - **Dispatch Latency:** The time taken by the dispatcher to stop one process and start another.
- **Scheduling Criteria**
    - **CPU Utilization:** Keep the CPU as busy as possible (range 0-100%).
    - **Throughput:** Number of processes completed per unit time.
    - **Turnaround Time:** Total time from submission of a process to its completion (sum of waiting, ready, execution, and I/O times).
    - **Waiting Time:** Total time a process spends waiting in the ready queue.
    - **Response Time:** Time from submission of a request until the first response is produced (for interactive systems).
    - **Fairness:** Ensure each process gets a fair share of the CPU.
- **Scheduling Algorithms**
    - **First-Come, First-Served (FCFS):** Processes are executed in the order they arrive. Simple, but can lead to long waiting times (convoy effect). Non-preemptive.
    - **Shortest-Job-First (SJF):** The process with the shortest CPU burst time is executed next. Optimal for minimum average waiting time. Can be preemptive (Shortest-Remaining-Time-First, SRTF) or non-preemptive. Requires knowing future burst times, which is difficult.
    - **Priority Scheduling:** Each process is assigned a priority, and the CPU is allocated to the process with the highest priority. Can lead to starvation (low-priority processes might never run). Can be preemptive or non-preemptive. Aging (gradually increasing priority of long-waiting processes) can prevent starvation.
    - **Round Robin (RR):** Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. If the process doesn't finish within the quantum, it's preempted and added to the end of the ready queue. Good for timesharing, provides good response time. Preemptive.
    - **Multilevel Queue Scheduling:** Processes are permanently assigned to different queues based on characteristics (e.g., foreground interactive processes, background batch processes). Each queue has its own scheduling algorithm.
    - **Multilevel Feedback Queue Scheduling:** Allows processes to move between queues based on their CPU burst behavior. Used to separate processes with different CPU burst characteristics. Highly configurable and commonly used in general-purpose OS.
- **Linux Scheduling Policies**
    - Linux uses a sophisticated scheduler, primarily the **Completely Fair Scheduler (CFS)** for normal processes since kernel 2.6.

- **Normal vs. Real-Time Scheduling:**
  - **Normal Scheduling (`SCHED_OTHER`):** This is the default policy. CFS aims to provide fair CPU time to all processes, giving an illusion that each process runs on its own dedicated CPU. It doesn't use fixed time slices but rather virtual runtimes to determine fairness.
  - **Real-Time Scheduling (`SCHED_FIFO`, `SCHED_RR`):** These policies are for time-critical applications that require strict adherence to deadlines. They are priority-based and preemptive.
    - `SCHED_FIFO` (First-In, First-Out): Real-time processes run until they block or explicitly yield the CPU. Higher-priority `SCHED_FIFO` processes always preempt lower-priority ones.
    - `SCHED_RR` (Round Robin): Similar to `SCHED_FIFO` but with a time quantum. If a process exceeds its quantum, it's moved to the end of its priority queue.
- **Priorities:**
  - **Nice Values:** For `SCHED_OTHER` processes. A numerical value from -20 (highest priority) to 19 (lowest priority). A lower nice value means a process gets a larger share of the CPU.
    - To view nice value: `ps -l` (NI column)
    - To start with a specific nice value: `nice -n 10 command`
    - To change nice value of a running process: `renice +5 -p PID`
  - **Real-time Priorities:** For `SCHED_FIFO` and `SCHED_RR` processes. Range from 1 (lowest RT priority) to 99 (highest RT priority). These processes run before any `SCHED_OTHER` process and require elevated privileges to set.

- **Critical–section Problem**
  - **Concept:** A situation where multiple processes or threads concurrently try to access and modify shared data.
  - **Critical Section:** A segment of code where shared resources are accessed. Only one process/thread should be allowed to execute its critical section at any given time to prevent data inconsistency.
  - **Problem:** Designing a protocol to ensure that when one process is executing in its critical section, no other process can be executing in its critical section.
  - **Requirements for a Solution:**
    1. **Mutual Exclusion:** If process P1 is executing in its critical section, then no other processes can be executing in their critical sections.
    2. **Progress:** If no process is executing in its critical section and some processes want to enter their critical sections, then only those processes that are not executing in their remainder section can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
    3. **Bounded Waiting:** There must be a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- **Critical Region**
  - This term is often used interchangeably with "critical section," referring to the *specific code segment* that accesses shared resources. More formally, in some programming languages or OS contexts, a "critical region" might refer to a language construct or block of code that implicitly

ensures mutual exclusion for shared data accessed within it, usually through compiler-managed locks or atomic operations.

- **Semaphores & Mutex**
  - **Synchronization Tools:** Used to solve the critical-section problem and other synchronization issues.
  - **Semaphore:**
    - **Concept:** An integer variable (or an abstract data type) that, apart from initialization, can only be accessed through two standard atomic operations: `wait()` (also known as `P` or `down`) and `signal()` (also known as `V` or `up`).
    - **Types:**
      - **Counting Semaphore:** Its value can range over an unrestricted domain. Used to control access to a resource that has multiple instances.
      - **Binary Semaphore:** Its value can only be 0 or 1. Used to implement mutual exclusion (like a mutex lock).
    - `wait()` **Operation:** Decrements the semaphore value. If the value becomes negative, the process blocks until it becomes non-negative.
    - `signal()` **Operation:** Increments the semaphore value. If there are processes blocked on this semaphore, one of them is unblocked.
  - **Mutex (Mutual Exclusion Lock):**
    - **Concept:** A simplified version of a binary semaphore that is specifically designed for mutual exclusion. A mutex provides a lock that only one thread can hold at a time.
    - **Key Difference from Binary Semaphore:** A mutex typically has "ownership." Only the thread that acquired (locked) the mutex can release (unlock) it. A binary semaphore, however, can be signaled by a different process than the one that waited on it.
    - **Operations:** `acquire()` (lock) and `release()` (unlock).
- **Producer-Consumer Problem**
  - **Concept:** A classic synchronization problem where a "producer" process generates data and places it into a shared buffer, and a "consumer" process retrieves data from the buffer.
  - **Challenges:**
    - Ensuring the producer doesn't add data to a full buffer.
    - Ensuring the consumer doesn't try to remove data from an empty buffer.
    - Ensuring mutual exclusion when accessing the buffer itself.
  - **Solution using Semaphores:**
    - A mutex semaphore (`mutex`) for mutual exclusion to the buffer.
    - A counting semaphore (`empty`) initialized to buffer size, indicating empty slots.
    - A counting semaphore (`full`) initialized to 0, indicating filled slots.
    - **Producer:** `wait(empty); wait(mutex); // add item; signal(mutex); signal(full);`
    - **Consumer:** `wait(full); wait(mutex); // remove item; signal(mutex); signal(empty);`
- **Monitors**

- **Concept:** A high-level synchronization construct that simplifies concurrent programming by providing mutual exclusion implicitly. It's a collection of procedures, variables, and data structures grouped together, with a built-in mechanism to ensure that only one process can be active within the monitor at any given time.
- **Key Features:**
  - **Mutual Exclusion:** Only one process can execute any procedure within the monitor at a time.
  - **Condition Variables:** Used for thread synchronization within the monitor. Threads can `wait()` on a condition variable (suspending themselves) and `signal()` a condition variable (waking up a waiting thread).
- **Advantages:** Easier to use and less error-prone than raw semaphores because mutual exclusion is handled automatically by the language/system.

- **Deadlocks**
  - **Concept:** A situation where two or more processes are permanently blocked because each is waiting for a resource that is held by another process in the same set.
  - **Four Necessary Conditions (Coffman Conditions) for Deadlock:**
    1. **Mutual Exclusion:** At least one resource must be held in a non-sharable mode; only one process can use the resource at any given time.
    2. **Hold and Wait:** A process holding at least one resource is waiting to acquire additional resources held by other processes.
    3. **No Preemption:** Resources cannot be forcibly taken from a process holding them; they must be released voluntarily by the process after it has completed its task.
    4. **Circular Wait:** A set of processes {P0, P1, ..., Pn} exists such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.
  - **Strategies for Handling Deadlocks:**
    - **Deadlock Prevention:** Design a system to prevent at least one of the four necessary conditions from holding.
      - Break Mutual Exclusion (not always possible).
      - Break Hold and Wait (e.g., acquire all resources at once, or release all held resources before requesting new ones).
      - Break No Preemption (e.g., allow OS to forcibly take resources).
      - Break Circular Wait (e.g., impose an ordering on resource types).
    - **Deadlock Avoidance:** Require processes to declare maximum resource needs in advance. The OS dynamically checks resource allocation state to ensure no circular wait condition can arise (e.g., **Banker's Algorithm**).
    - **Deadlock Detection:** Allow deadlocks to occur, then detect them, and recover. Requires a detection algorithm and a recovery strategy.
    - **Deadlock Recovery:**
      - **Process Termination:** Abort all deadlocked processes, or abort one process at a time until the deadlock cycle is broken.
      - **Resource Preemption:** Preempt resources from some processes and give them to others until the deadlock is broken.

## Session 3&4: Memory Management (4T+2L)

These sessions cover how operating systems manage computer memory, from basic concepts to advanced techniques like virtual memory. The lab portion will involve observing memory usage in Linux.

- **Introduction to Memory Management**
    - **Purpose:** To manage the primary memory (RAM) efficiently, protect processes from each other, allow sharing of memory, and provide an abstraction of a large, uniform memory space to programs.
    - **Memory Hierarchy:** Cache, Main Memory (RAM), Secondary Storage (Disk).
    - **Binding of Instructions and Data to Memory:**
        - **Compile Time:** Absolute code generated. Must know where process resides in memory at compile time.
        - **Load Time:** Relocatable code generated. Final address binding done at load time.
        - **Execution Time:** Most flexible. Binding postponed until run time. Required if process can be moved during execution (e.g., by swapping). Modern OS use this.
- **Logical Address vs. Physical Address**
    - **Logical Address (Virtual Address):** An address generated by the CPU. This is the address space seen by the program.
    - **Physical Address:** The actual address available on the memory unit.
    - **Memory Management Unit (MMU):** A hardware device that maps logical addresses to physical addresses at runtime.
    - **Address Space:**
        - **Logical Address Space:** The set of all logical addresses generated by a program.
        - **Physical Address Space:** The set of all physical addresses corresponding to the logical addresses.
- **Dynamic Linking**
    - **Concept:** A linking method where linking of libraries or modules is postponed until execution time. Instead of including copies of all necessary library routines into the executable file at compile time (static linking), only a small stub is included.
    - **Mechanism:** When the program runs and calls a library function, the dynamic linker/loader loads the required library into memory (if not already loaded) and resolves the call.
    - **Benefits:**
        - **Memory Savings:** Multiple programs can share the same copy of a dynamically linked library in memory.
        - **Smaller Executables:** Executable files are smaller as they don't contain full library code.
        - **Easier Updates:** Libraries can be updated independently without recompiling applications that use them.
    - **Example:** Shared libraries (.so files on Linux, .dll files on Windows).
- **Memory Management Techniques**
    - **Contiguous Allocation**
        - **Concept:** Each process is allocated a single, contiguous block of memory.
        - **Methods:**

        - **Fixed Partitioning:** Memory is divided into a fixed number of partitions, each of a fixed size. Simple, but leads to **internal fragmentation** (unused space within a partition if process is smaller than partition) and limits number of processes.
        - **Variable Partitioning (Dynamic Partitioning):** Partitions are created dynamically as processes arrive, matching their size. More flexible.
    - **Challenges:**
        - **External Fragmentation:** Enough total memory space exists, but it's scattered in small, non-contiguous holes, preventing allocation of a larger process.
        - **Compaction:** Shuffling memory contents to consolidate free space. Expensive.

- **Segmentation**

    - **Concept:** Memory is divided into logical units called **segments**, each corresponding to a logical part of the program (e.g., code, data, stack, subroutines). Each segment can be of variable size.
    - **Mechanism:** A logical address consists of a `(segment-number, offset)` pair. The OS maintains a **segment table** (Base, Limit) for each process. The MMU checks if `offset < Limit` and calculates `Physical Address = Base + Offset`.
    - **Advantages:** Supports logical view of memory, allows better sharing and protection.
    - **Disadvantages:** Still suffers from external fragmentation.

- **Paging**

    - **Concept:** Divides both physical memory (into fixed-size **frames**) and logical memory (into fixed-size **pages**) into equally sized blocks.
    - **Mechanism:** A logical address consists of a `(page-number, page-offset)` pair. The **page number** is used as an index into a **page table** (maintained by the OS) to find the corresponding **frame number** in physical memory. `Physical Address = (frame-number * frame-size) + page-offset`.
    - **Advantages:** Eliminates external fragmentation. Efficient for memory allocation.
    - **Disadvantages:** Introduces **internal fragmentation** (last page might not be fully used). Page table can be large.
    - **Translation Lookaside Buffer (TLB):** A small, fast cache used to speed up address translations by storing recent page-number to frame-number mappings.

- **Segmentation with Paging**

    - **Concept:** A hybrid approach (e.g., used in Multics, some Intel architectures) where segments are further divided into pages.
    - **Mechanism:** A logical address is first translated by a segment table (to get a segment's base address and length). This base address then points to a page table for that segment. The offset within the segment is used to find the page within that segment.
    - **Advantages:** Combines the benefits of both: logical view of segmentation with no external fragmentation of paging.

- **Virtual Memory**

    - **Concept:** A technique that allows the execution of processes that are not entirely in memory. It separates the user's logical memory from physical memory.

- **Benefits:**
  - Allows programs to be larger than physical memory.
  - Allows more programs to run concurrently (increased multiprogramming degree).
  - Fewer I/O operations needed to load/swap programs.
- **Swapping**
  - **Concept:** The process of temporarily moving a process (or parts of it) from main memory to secondary storage (swap space on disk) and then bringing it back into memory for execution.
  - **Purpose:** Allows more processes to share limited physical RAM, enabling higher multiprogramming levels.
- **Demand Paging**
  - **Concept:** A key aspect of virtual memory where pages are only loaded into physical memory when they are actually needed (i.e., referenced).
  - **Mechanism:** When a process tries to access a page that is not in memory, a **page fault** occurs.
  - **Advantages:** Less I/O, less memory consumed, faster response.
- **Page-Fault Exception Handler**
  - **Steps when a page fault occurs:**
    1. The MMU detects that the page is not in memory and generates a trap to the OS.
    2. The OS checks if the memory access was valid (i.e., the page is part of the process's valid address space but just not loaded).
    3. If invalid, terminate the process (e.g., segmentation fault).
    4. If valid, find a free frame in physical memory.
    5. If no free frame, a **page-replacement algorithm** is used to select a victim page to evict from memory. The victim page is written to disk if it has been modified (dirty bit set).
    6. The desired page is read from disk into the free frame.
    7. The page table is updated to reflect the new mapping.
    8. The CPU registers are restored, and the interrupted instruction is restarted (or the process continues).
- **Page-Replacement Algorithms**
  - When a page fault occurs and no free frames are available, the OS must choose a page in memory to evict (victim page).
  - **FIFO (First-In, First-Out):** Replaces the page that has been in memory the longest. Simple, but suffers from Belady's Anomaly (more frames can lead to more page faults).
  - **LRU (Least Recently Used):** Replaces the page that has not been used for the longest period of time. Optimal but difficult to implement precisely (requires tracking timestamps or linked lists). Often approximated.
  - **Optimal Page Replacement (OPT):** Replaces the page that will not be used for the longest period of time in the future. Ideal but impossible to implement in practice (requires knowing the future). Used as a benchmark.

- **LFU (Least Frequently Used):** Replaces the page with the smallest count of references.
- **MFU (Most Frequently Used):** Replaces the page with the largest count of references (less common).
- **Clock Algorithm (Second Chance):** A practical approximation of LRU. Uses a reference bit; if 0, evict; if 1, reset to 0 and move to next.

- **Thrashing**

- **Concept:** A phenomenon where a system spends most of its time swapping pages in and out of memory rather than executing useful work. It occurs when a process does not have "enough" frames allocated to it, leading to a very high page-fault rate.
- **Cause:** The sum of the sizes of active memory requirements (working sets) of all processes exceeds the available physical memory.
- **Symptoms:** Low CPU utilization, high disk activity (swapping).
- **Solutions:**
  - Provide more physical memory.
  - Reduce the degree of multiprogramming (suspend some processes).
  - Implement a proper working-set model to ensure processes get enough frames.
  - Use page-fault frequency schemes to adjust process frame allocations.

## Session 5: File Systems & Secondary Storage (2T)

This session covers how operating systems manage files and directories, and the underlying structure and management of secondary storage devices like hard drives.

- **File System Interface and Implementation**

- **Files and Directories Operations**

- **File:** A named collection of related information that is recorded on secondary storage. It's the logical storage unit.
  - **Attributes:** Name, ID, Type, Location, Size, Protection, Time, Date, User ID.
  - **Types:** Regular (text, binary), Directory, Character Special, Block Special, Symbolic Link, Socket, Pipe.
  - **Access Methods:** Sequential access (read/write next), Direct access (random access to blocks), Indexed access (using an index).
- **Directory:** A collection of files and subdirectories. Provides mapping between file names and their disk locations.
  - **Operations:** `create`, `delete`, `open`, `read`, `write`, `seek`, `truncate` (for files).
  - `mkdir` (create directory), `rmdir` (delete directory), `ls` (list contents), `cd` (change directory).
  - **Structures:**
    - **Single-Level Directory:** All files in one directory (simple, but name conflicts).
    - **Two-Level Directory:** Each user has their own directory (isolates users).
    - **Tree-Structured Directory:** Most common (hierarchical, allows grouping).
    - **Acyclic-Graph Directory:** Allows sharing of files and directories (using links).
    - **General Graph Directory:** Allows cycles (requires garbage collection for dangling pointers, more complex).

- **File System Mounting and Protection Methods**
  - **File System Mounting:** The process of making a file system (located on a specific storage device or partition) accessible through the directory tree. A "mount point" is an empty directory where the new file system is attached.
    - **Example (Linux):** `mount /dev/sdb1 /mnt/data`
  - **Protection Methods:** Regulating access to files and directories.
    - **Access Control Lists (ACLs):** List of permissions for specific users/groups. More granular control than traditional Unix permissions.
    - **Traditional Unix Permissions:** `rwx` (read, write, execute) for Owner, Group, and Others.
      - `chmod` **command:** Used to change permissions (e.g., `chmod 755 myfile.txt`, `chmod u+x,g-w myfile.txt`).
      - `chown` **command:** Change file owner.
      - `chgrp` **command:** Change file group.
- **File System Structures and Implementation**
  - **On-Disk Structures:**
    - **Boot Block:** Contains bootstrap program.
    - **Super Block:** Contains file system metadata (type, size, number of blocks/inodes, free block/inode lists).
    - **Inodes (Index Nodes):** Data structure that stores metadata about a file (owner, permissions, size, timestamps, and pointers to data blocks). Every file and directory has an inode.
    - **Data Blocks:** Where the actual file content is stored.
  - **In-Memory Structures:**
    - **Mount Table:** Information about mounted file systems.
    - **Directory-Entry Cache:** Speeds up path lookups.
    - **System-Wide Open File Table:** Contains an entry for each opened file.
    - **Per-Process Open File Table:** Contains an entry for each file a process has opened.
    - **Buffer Cache:** Caches frequently accessed disk blocks.
  - **Directory Implementation:**
    - **Linear List:** Simple list of file names and pointers to data blocks/inodes. Slow for large directories.
    - **Hash Table:** Hash filename to find pointer to its directory entry (faster lookups).
  - **Allocation Methods:** How disk blocks are allocated to files.
    - **Contiguous Allocation:** Each file occupies a contiguous set of blocks. Simple, fast sequential access. Suffers from external fragmentation.
    - **Linked Allocation:** Each block points to the next block of the file. No external fragmentation. Slow random access. Requires space for pointers.
    - **Indexed Allocation:** Uses an **index block** (or inode) which contains pointers to all data blocks of the file. Good for both sequential and random access. No external fragmentation. Max file size limited by index block size.

- **Secondary Storage Structure**
  - **Disk Structure**

- **Platters:** Circular disks coated with magnetic material.
- **Tracks:** Concentric rings on a platter.
- **Sectors:** Angular divisions of a track. The smallest unit of data transfer.
- **Cylinders:** A set of tracks that are at the same radial distance from the center on all platters.
- **Disk Controller:** Manages the disk and communicates with the computer.
- **Access Time:**
  - **Seek Time:** Time to move read/write heads to the correct cylinder.
  - **Rotational Latency:** Time for the desired sector to rotate under the head.
  - **Transfer Time:** Time to actually read/write the data.

- **Disk Scheduling and Management**

  - **Purpose:** To minimize disk access time, primarily seek time, by optimizing the order of disk I/O requests.
  - **Algorithms:**
    - **FCFS (First-Come, First-Served):** Simplest, but may lead to high seek times.
    - **SSTF (Shortest-Seek-Time-First):** Selects the request with the minimum seek time from the current head position. Can lead to starvation.
    - **SCAN (Elevator Algorithm):** The head moves in one direction, servicing all requests in its path, then reverses direction.
    - **C-SCAN (Circular SCAN):** Similar to SCAN, but when the head reaches the end, it immediately returns to the beginning of the disk without servicing requests on the return trip. Provides more uniform wait times.
    - **LOOK/C-LOOK:** Variations of SCAN/C-SCAN where the head only goes as far as the last request in each direction, then reverses, rather than going all the way to the end of the disk.
  - **Disk Management:**
    - **Partitioning:** Dividing a disk into logical sections.
    - **Formatting:** Low-level (physical sectors) and logical (file system structure).
    - **Boot Block:** Contains bootstrap loader program.
    - **Bad Block Management:** Identifying and managing sectors that become unreliable.

- **Swap-Space Management**

  - **Concept:** The portion of secondary storage (disk) that is used by the OS to temporarily hold pages of memory that have been swapped out from RAM.
  - **Purpose:** Extends the apparent size of physical memory, allowing the execution of larger programs or more programs simultaneously.
  - **Types:**
    - **Swap File:** A normal file on a file system used for swapping. Easier to create/resize.
    - **Swap Partition:** A dedicated disk partition formatted specifically for swap space. Generally faster due to raw disk access.
  - **Management:** The OS manages allocation and deallocation of swap space. Disk scheduling algorithms are often applied to swap I/O requests.

## Session 6: Linux History, Operations, and Basic Commands (2T+6L)

This session introduces the Linux operating system, its architecture, and essential command-line operations for navigating, manipulating files, and managing permissions. The lab component involves hands-on practice with these commands.

- **Linux History and Operation**
    - **The Evolution of Linux:**
        - **Unix:** Developed at Bell Labs in the late 1960s/early 1970s. Key features: modular design, command-line interface, "everything is a file," pipe concept.
        - **MINIX:** A Unix-like operating system written by Andrew S. Tanenbaum for educational purposes.
        - **Linus Torvalds:** Started developing the Linux kernel in 1991, inspired by MINIX, as a hobby project.
        - **GNU Project:** Richard Stallman initiated the GNU (GNU's Not Unix) project in 1983 to create a free and open-source Unix-like operating system. Developed many essential user-space tools (compiler, shell, utilities).
        - **Linux + GNU = GNU/Linux:** The Linux kernel combined with GNU tools forms the complete operating system commonly referred to as Linux (or GNU/Linux).
        - **Open Source & Community:** Linux development is collaborative and open, with contributions from thousands worldwide.
    - **Linux Operations as a Server:**
        - **Stability & Reliability:** Known for uptime and robust performance.
        - **Security:** Strong security model, frequent updates, active community patching.
        - **Scalability:** Can run on a wide range of hardware, from embedded systems to supercomputers.
        - **Flexibility:** Highly customizable, supports various services (web servers, databases, mail servers, file servers, virtualization).
        - **Cost-Effectiveness:** Free and open-source, reducing licensing costs.
        - **Command Line Interface (CLI):** Efficient for server administration and automation.
    - **The Architecture and Structure of Linux:**
        - **Kernel:** The core of the OS. Manages hardware, processes, memory, and I/O.
        - **Shell:** A command-line interpreter (e.g., Bash) that provides an interface to the kernel.
        - **System Utilities/Libraries:** GNU tools, standard C library (glibc), etc., provide common functionalities.
        - **Applications:** User programs.
        - **File System Hierarchy Standard (FHS):** A standardized directory structure (`/bin`, `/etc`, `/home`, `/var`, `/usr`, etc.) for organizing files and directories.

- **Basic Commands**
    - This section provides explanations and common usages for fundamental Linux commands.
    - `ls`: **List directory contents.**
        - `ls`: Lists files and directories in the current directory.
        - `ls -l`: Long format (permissions, owner, size, date).
        - `ls -a`: List all files, including hidden files (starting with `.`).
        - `ls -lh`: Long format with human-readable sizes (KB, MB).
        - `ls -d */`: List only directories.
    - `cp`: **Copy files and directories.**

    - `cp source_file destination_file`: Copy `source_file` to `destination_file`.
    - `cp -r source_directory destination_directory`: Recursively copy directory contents.
- `mv`: **Move or rename files and directories.**
    - `mv old_name new_name`: Rename `old_name` to `new_name`.
    - `mv file destination_directory`: Move `file` to `destination_directory`.
- `sort`: **Sort lines of text files.**
    - `sort file.txt`: Sorts content of `file.txt` alphabetically and prints to standard output.
    - `ls -l | sort -k5n`: Sorts `ls -l` output numerically by the 5th column (size).
- `grep`: **Search for patterns in files.**
    - `grep "pattern" file.txt`: Search for "pattern" in `file.txt`.
    - `grep -i "pattern" file.txt`: Case-insensitive search.
    - `grep -v "pattern" file.txt`: Invert match (show lines *not* containing pattern).
    - `grep -r "pattern" directory/`: Recursively search files in a directory.
    - `ps aux | grep "apache"`: Find processes related to Apache.
- `cat`: **Concatenate files and print on the standard output.**
    - `cat file.txt`: Display content of `file.txt`.
    - `cat file1.txt file2.txt > combined.txt`: Concatenate `file1.txt` and `file2.txt` into `combined.txt`.
- `head`: **Output the first part of files.**
    - `head file.txt`: Display first 10 lines of `file.txt`.
    - `head -n 5 file.txt`: Display first 5 lines.
- `tail`: **Output the last part of files.**
    - `tail file.txt`: Display last 10 lines of `file.txt`.
    - `tail -n 5 file.txt`: Display last 5 lines.
    - `tail -f /var/log/syslog`: Continuously monitor new content added to a log file.
- `man`: **Format and display the on-line manual pages.**
    - `man ls`: Display the manual page for the `ls` command.
- `locate`: **Find files by name (uses a pre-built database).**
    - `locate filename`: Quickly find files matching `filename`. Requires `updatedb` to update the database.
- `find`: **Search for files in a directory hierarchy.**
    - `find . -name "*.txt"`: Find all `.txt` files in current directory and subdirectories.
    - `find /home/user -type f -size +1G`: Find files larger than 1GB in `/home/user`.
    - `find . -mtime +7 -delete`: Find files older than 7 days and delete them.
- `diff`: **Compare two files line by line.**
    - `diff file1.txt file2.txt`: Shows differences between `file1.txt` and `file2.txt`.
- `file`: **Determine file type.**
    - `file myfile.txt`: Identifies if it's a text file, executable, image, etc.
- `rm`: **Remove files or directories.**
    - `rm file.txt`: Remove `file.txt`.
    - `rm -r directory/`: Recursively remove a directory and its contents.
    - `rm -rf directory/`: Force removal (use with extreme caution!).
- `mkdir`: **Make directories.**
    - `mkdir mydir`: Create a directory named `mydir`.
    - `mkdir -p path/to/new/directory`: Create parent directories if they don't exist.

- `rmdir`: **Remove empty directories.**
  - `rmdir empty_dir`: Removes `empty_dir` only if it's empty.
- `cd`: **Change the current working directory.**
  - `cd /home/user/documents`: Change to specific path.
  - `cd ..`: Go up one directory.
  - `cd ~`: Go to home directory.
  - `cd -`: Go to previous directory.
- `pwd`: **Print name of current working directory.**
  - `pwd`: Displays the full path of the current directory.
- `ln` and `ln -s`: **Make links between files.**
  - `ln source_file hard_link`: Create a **hard link**. Both names point to the same inode. Deleting one doesn't delete the content until all links are removed. Only works on the same filesystem.
  - `ln -s source_file symbolic_link`: Create a **symbolic (soft) link**. The link is a small file containing the path to the `source_file`. Can link across filesystems. Deleting the source breaks the link.
- `gzip` and `gunzip`: **Compress or expand files.**
  - `gzip file.txt`: Compresses `file.txt` to `file.txt.gz`.
  - `gunzip file.txt.gz`: Decompresses `file.txt.gz` to `file.txt`.
- `zip` and `unzip`: **Package and compress (archive) files; list, test and extract compressed files.**
  - `zip archive.zip file1.txt file2.txt`: Creates `archive.zip` containing specified files.
  - `zip -r archive.zip directory/`: Creates `archive.zip` with contents of `directory/`.
  - `unzip archive.zip`: Extracts contents of `archive.zip`.

- **Access Control List (ACL) and `chmod` command**
  - **ACLs:** Extended permissions providing finer-grained control than traditional Unix permissions. You can grant specific permissions to individual users or groups, beyond the owner, group, and others.
    - `getfacl file.txt`: View ACLs for a file.
    - `setfacl -m u:john:rwx file.txt`: Grant user `john` read, write, execute permissions on `file.txt`.
    - `setfacl -x u:john file.txt`: Remove ACL entry for user `john`.
  - `chmod`**: Change file permissions.**
    - **Symbolic Mode:**
      - `u` (user/owner), `g` (group), `o` (others), `a` (all).
      - `+` (add permission), `-` (remove permission), `=` (set permission exactly).
      - `r` (read), `w` (write), `x` (execute).
      - `chmod u+x,go-w myfile.sh`: Add execute for owner, remove write for group and others.
    - **Octal Mode:** Each permission set (owner, group, others) is represented by a digit (r=4, w=2, x=1).
      - `7` (rwx), `6` (rw-), `5` (r-x), `4` (r--), `0` (---).
      - `chmod 755 myfile.sh`: Owner rwx, Group r-x, Others r-x. (Common for executables/directories)
      - `chmod 644 myfile.txt`: Owner rw-, Group r--, Others r--. (Common for text files)

        - **Special Permissions:**
            - **SetUID (4000):** If set on an executable, it runs with the permissions of the file owner. `chmod u+s file` or `chmod 4755 file`.
            - **SetGID (2000):** If set on an executable, it runs with the group permissions of the file. If set on a directory, new files created in it inherit the directory's group. `chmod g+s dir` or `chmod 2775 dir`.
            - **Sticky Bit (1000):** If set on a directory, only the owner of a file (or root) can delete or rename files within that directory, even if they have write permission to the directory. `chmod +t dir` or `chmod 1777 dir`. (Common for `/tmp`).

- **`chown` and `chgrp` commands**
    - `chown`: **Change file owner.**
        - `chown new_owner file.txt`: Change owner of `file.txt` to `new_owner`.
        - `chown new_owner:new_group file.txt`: Change owner and group.
        - `chown -R new_owner directory/`: Recursively change owner for a directory and its contents.
    - `chgrp`: **Change file group ownership.**
        - `chgrp new_group file.txt`: Change group of `file.txt` to `new_group`.
        - `chgrp -R new_group directory/`: Recursively change group.

- **Commands like `telnet`, `ftp`, `ssh`, and `sftp`**
    - These are network client commands.
    - `telnet`: **Remote login using TELNET protocol.** Unencrypted, insecure.
        - `telnet hostname port`: Connect to a host on a specific port. Used for basic network troubleshooting.
    - `ftp`: **File Transfer Protocol client.** Unencrypted, insecure for file transfer.
        - `ftp hostname`: Connect to an FTP server.
    - `ssh`: **Secure Shell client (remote login program).** Encrypted, secure replacement for `telnet`, `rsh`, `rcp`.
        - `ssh user@hostname`: Connect securely to a remote host.
        - `ssh -p 2222 user@hostname`: Specify a different port.
    - `sftp`: **Secure File Transfer Protocol client.** Encrypted, secure replacement for `ftp` and `scp`. Uses SSH for secure connection.
        - `sftp user@hostname`: Connect to a remote host and provides an interactive file transfer interface.

- **Basic of I/O system with `mount` and `unmount`**
    - **I/O System:** Manages communication between the computer and peripheral devices (disk drives, keyboards, printers, network interfaces).
    - **Block Devices:** Devices that store information in fixed-size blocks (e.g., hard drives, SSDs, USB drives). Data is accessed in blocks.
    - **Character Devices:** Devices that handle I/O one character at a time (e.g., keyboard, mouse, serial port).
    - **`mount`: Attach a filesystem to the file hierarchy.** Makes a storage device or network share accessible at a specific directory (mount point).

        - `mount /dev/sdb1 /mnt/usbdrive`: Mount the first partition of `sdb` to `/mnt/usbdrive`.
        - `mount -a`: Mount all filesystems listed in `/etc/fstab`.
        - `mount`: List all currently mounted filesystems.
    - `unmount` (or `umount`): **Detach mounted filesystems.**
        - `umount /mnt/usbdrive`: Unmount the filesystem mounted at `/mnt/usbdrive`.
        - `umount /dev/sdb1`: Unmount the device.
        - **Note:** You cannot unmount a filesystem if it's currently in use (e.g., if you are in that directory or a file on it is open).

- **`vi` editor**
    - A powerful, modal text editor available on virtually all Unix-like systems.
    - **Features:** Command-driven, efficient for keyboard-only usage, highly configurable, supports regular expressions.
    - **Different Modes of `vi` editor:**
        1. **Normal Mode (Command Mode):** Default mode when `vi` starts. Used for navigation, deletion, copying, pasting, and issuing commands.
            - Press `Esc` to return to Normal mode from any other mode.
        2. **Insert Mode:** Used for inserting and editing text.
            - Enter from Normal mode using:
                - `i`: insert at cursor
                - `a`: append after cursor
                - `I`: insert at beginning of line
                - `A`: append at end of line
                - `o`: open new line below current
                - `O`: open new line above current
        3. **Last Line Mode (Ex Mode/Command-line Mode):** Used for saving, quitting, searching, and advanced operations.
            - Enter from Normal mode by pressing `:`. Commands start with `:`.
            - Enter from Normal mode by pressing `/` (for search) or `?` (for backward search).
    - **Editing using `vi` editor:**
        - **Basic Navigation (Normal Mode):**
            - `h`, `j`, `k`, `l`: left, down, up, right.
            - `w`, `b`: word forward, word backward.
            - `0`, `^`: start of line, first non-blank char.
            - `$`: end of line.
            - `G`: go to end of file.
            - `gg`: go to beginning of file.
            - `nG`: go to line `n`.
        - **Basic Deletion (Normal Mode):**
            - `x`: delete character under cursor.
            - `dd`: delete current line.
            - `ndd`: delete `n` lines.
            - `dw`: delete word.
            - `d$`: delete to end of line.
            - `D`: delete to end of line.
        - **Undo/Redo (Normal Mode):**

        - `u`: undo last change.
        - `Ctrl+r`: redo.
    - **Find and replace commands (Last Line Mode):**
        - `:/pattern`: Search forward for `pattern`.
        - `?pattern`: Search backward for `pattern`.
        - `n`: next match.
        - `N`: previous match.
        - `:%s/old/new/g`: Replace all occurrences of `old` with `new` throughout the file (`%` for entire file, `g` for global on each line).
        - `:%s/old/new/gc`: Replace all occurrences with confirmation.
    - **Cut-copy-paste commands (Normal Mode):**
        - `yy`: yank (copy) current line.
        - `nyy`: yank `n` lines.
        - `yw`: yank word.
        - `p`: paste after cursor/line.
        - `P`: paste before cursor/line.
    - **The `set` command (Last Line Mode):**
        - Used to configure `vi` options.
        - `:set nu`: Show line numbers.
        - `:set nonu`: Hide line numbers.
        - `:set ic`: Ignore case in searches.
        - `:set noic`: Don't ignore case.
        - `:set autoindent`: Enable auto indentation.
        - `:set syntax on`: Enable syntax highlighting.
        - `:set paste`: Paste mode (prevents auto-indent/reformatting).
        - `:set all`: Display all options.

- **Introduction to Users and Groups**
    - **User:** An entity that can log into the system and own files/processes. Each user has a unique **User ID (UID)**. User information is primarily stored in `/etc/passwd`.
    - **Group:** A collection of users. Users belonging to a group share common access permissions to files and resources owned by that group. Each group has a unique **Group ID (GID)**. Group information is stored in `/etc/group`.
    - **Primary Group:** The group assigned to a user upon creation, and to which files created by that user (by default) belong.
    - **Supplementary Groups:** Additional groups a user can be a member of.
    - **Purpose:** Simplify access control and resource management. Instead of granting permissions to individual users, you can grant them to a group, and then add users to that group.
    - **Commands:**
        - `id`: Display user and group IDs.
        - `whoami`: Display current username.
        - `passwd`: Change user password.
        - `useradd`, `usermod`, `userdel`: Manage user accounts.
        - `groupadd`, `groupmod`, `groupdel`: Manage groups.
        - `gpasswd`: Administer `/etc/group` (add/remove users from groups).

## Session 7&8: Shell Scripting (4T+6L)

These sessions introduce the concept of shell scripting, focusing on Bash. Students will learn to write scripts for automation, including variables, conditional statements, loops, and various command-line tools. The lab component will involve extensive practical script writing.

- **Introduction to Shell**
    - **Shell:** A command-line interpreter that provides an interface for users to interact with the operating system. It reads commands typed by the user or from a script file and executes them.
    - **Role:**
        - **Interactive Command Execution:** Provides a prompt where users can type commands.
        - **Scripting Language:** Allows users to write sequences of commands (shell scripts) to automate tasks.
        - **Environment Management:** Manages environment variables, aliases, and functions.
        - **Job Control:** Manages background/foreground processes.
        - **I/O Redirection & Piping:** Redirects input/output and connects commands.
- **Different Types of Linux Shells**
    - **Bourne Shell (`sh`):** The original Unix shell. Basic functionality, still widely used as a common denominator for system scripts.
    - **Bourne-Again Shell (`bash`):** The default shell on most Linux distributions. An enhanced version of `sh` with features like command-line editing, history, aliases, and better scripting capabilities.
    - **Korn Shell (`ksh`):** Compatible with `sh`, offers more advanced features than `sh`, including command history, aliases, and array variables.
    - **C Shell (`csh`):** Syntax similar to the C programming language. Popular for interactive use but less common for scripting due to some quirks.
    - **Z Shell (`zsh`):** A powerful modern shell that combines features from `bash`, `ksh`, and `tcsh`, offering extensive customization, better auto-completion, and plugin support.
- **Bourne Again Shell (BASH)**
    - **Features:**
        - **Command Line Editing:** Edit commands on the line (like a word processor).
        - **History:** Recall and reuse previous commands (`history` command, `!n`, `!string`).
        - **Aliases:** Create shortcuts for commands (`alias ll='ls -lh'`).
        - **Tab Completion:** Auto-complete commands, file names, variables.
        - **Job Control:** Manage multiple processes.
        - **Shell Functions:** Define reusable blocks of code.
        - **Advanced Scripting Features:** Conditional statements, loops, arithmetic operations, arrays.
- **Shell Variables (environment and user defined)**
    - **Variables:** Named storage locations for data in the shell.
    - **Environment Variables:** Variables that are set by the shell and inherited by child processes. They store information about the shell's environment.
        - **Common Examples:**

        - `PATH`: List of directories where the shell looks for executable commands.
        - `HOME`: User's home directory.
        - `USER`: Current user's username.
        - `SHELL`: Path to the user's default shell.
        - `PS1`: Primary prompt string.
        - `HISTSIZE`: Number of commands stored in history.
      - **Viewing:** `echo $VAR_NAME`, `printenv`, `env`.
      - **Setting:** `export VAR_NAME=value` (makes it available to child processes).
    - **User-Defined Variables:** Variables created by the user within a shell session or script.
      - **Setting:** `VAR_NAME=value` (no spaces around `=`).
      - **Accessing:** `$VAR_NAME` or `${VAR_NAME}`.
      - **Local to current shell/script unless `export`ed.**

- **Shell Files (`.bashrc`, `.profile`, `.bash_profile`, `.bash_logout`)**
    - These are configuration files read by Bash upon startup, depending on whether it's a login shell or a non-login interactive shell.
    - `.bash_profile`: Read by **login shells** (e.g., when you log in at a console or via SSH). Typically sources `.bashrc`.
    - `.bashrc`: Read by **non-login interactive shells** (e.g., when you open a new terminal window after logging in). Used for interactive shell settings like aliases, functions, prompt customization.
    - `.profile`: A more general login shell script that is often sourced by `.bash_profile` if it exists. Used for environment variables that apply to all Bourne-compatible shells.
    - `.bash_logout`: Executed when a **login shell** exits. Used for cleanup tasks (e.g., clearing the screen, removing temporary files).

- **Positional Parameters**
    - Special variables in shell scripts that hold the command-line arguments passed to the script.
    - `$0`: Name of the script itself.
    - `$1`, `$2`, `$3`, ...: Individual arguments (first, second, third, etc.).
    - `$#`: Number of arguments passed to the script.
    - `$@`: All arguments as separate words. (`"$@"` is best for iterating over arguments with spaces).
    - `$*`: All arguments as a single string. (`"$*"` is best for printing all arguments as one string).
    - `$?`: Exit status of the last executed command (0 for success, non-zero for failure).
    - `$$`: Process ID (PID) of the current shell.
    - `$!`: PID of the last background command.

- **Wild cards (`*` and `?`)**
    - Used for filename pattern matching (globbing).
    - `*`: Matches any sequence of zero or more characters.
      - `ls *.txt`: Lists all files ending with `.txt`.
      - `rm data*`: Removes all files starting with `data`.
    - `?`: Matches any single character.
      - `ls file?.txt`: Matches `file1.txt`, `fileA.txt`, etc., but not `file10.txt`.
    - `[]`: Matches any one of the characters enclosed within the brackets.
      - `ls [abc]*.txt`: Matches files starting with 'a', 'b', or 'c' and ending with `.txt`.

        - `ls [0-9].log`: Matches `0.log`, `1.log`, etc.
    - `{}`: Brace expansion (not a wildcard, but for generating strings).
        - `mkdir {dir1,dir2,dir3}`: Creates `dir1`, `dir2`, `dir3`.
        - `cp file.{txt,bak}`: Copies `file.txt` to `file.bak`.

- **Command Line Arguments**
    - Values passed to a script when it's executed. These are accessed via positional parameters.
    - **Example Script (`args.sh`):**

```
#!/bin/bash
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "Number of arguments: $#"
echo "All arguments as separate words:"
for arg in "$@"; do
    echo "  $arg"
done
echo "All arguments as a single string: $*"
```

    - **Execution:**

```
./args.sh hello world "how are you"
```

        - Output:

```
Script name: ./args.sh
First argument: hello
Second argument: world
Number of arguments: 3
All arguments as separate words:
  hello
  world
  how are you
All arguments as a single string: hello world how are you
```

- **Arithmetic in Shell Scripts**
    - Shell performs string comparisons by default. For numerical calculations, specific constructs are needed.
    - `expr` **command:** For simple integer arithmetic.

```
num=5
result=$(expr $num + 3) # Note spaces around operators
```

```
echo $result # Output: 8
```

- `(( ))` **(Arithmetic Expansion):** Preferred for integer arithmetic in Bash. More powerful.

```
num=5
(( result = num + 3 ))
echo $result # Output: 8

# Can also be used for conditional tests
if (( num > 0 )); then
    echo "Positive"
fi
```

- `$(( ))` **(Arithmetic Expansion):** Returns the result of an arithmetic expression.

```
a=10
b=3
c=$(( a * b ))
echo $c # Output: 30
d=$(( a / b ))
echo $d # Output: 3 (integer division)
```

- `bc` **(arbitrary-precision calculator):** For floating-point arithmetic.

```
echo "scale=2; 10 / 3" | bc # Output: 3.33
```

- `read` **and** `echo` **commands in shell scripts**
  - `echo`: **Display lines of text.** Used for outputting messages or variable values.
    - `echo "Hello, World!"`
    - `echo "The value of VAR is: $VAR"`
    - `echo -n "Enter your name: "` (no newline)
    - `echo -e "Line1\nLine2"` (interpret escape sequences like `\n` for newline)
  - `read`: **Read a line from standard input.** Used to get user input.
    - `read name`: Reads input into the variable `name`.
    - `read -p "Enter your age: " age`: Displays a prompt before reading input.
    - `read -s password`: Reads input silently (for passwords).
- **The** `tput` **command**
  - `tput`**:** A utility that allows shell scripts to interact with the terminal's capabilities, such as positioning the cursor, changing text colors, or clearing the screen, in a portable way (using terminfo database).
  - **Common Usages:**

    - `tput clear`: Clear the screen.
    - `tput bold`: Set text to bold.
    - `tput sgr0`: Reset all attributes (bold, color, etc.).
    - `tput setaf N`: Set foreground color (N is a number, e.g., 1 for red, 2 for green).
    - `tput setab N`: Set background color.
    - `tput cup row col`: Position cursor at `row`, `col`.
  - **Example:**

```
echo -n "$(tput setaf 1)This is red text.$(tput sgr0)"
echo -n "$(tput cup 5 10)Cursor at 5,10."
```

- **Taking decisions:**
  - `if-then-fi`

```
#!/bin/bash
if [ -f "myfile.txt" ]; then
    echo "myfile.txt exists."
fi
```

  - `if-then-else-fi`

```
#!/bin/bash
read -p "Enter a number: " num
if (( num > 0 )); then
    echo "$num is positive."
else
    echo "$num is not positive (zero or negative)."
fi
```

  - **The `test` command (file tests, string tests)**
    - The `[ ]` syntax is actually an alias for the `test` command. `[[ ]]` is a Bash-specific extension, more powerful.
    - **File Tests:**
      - `-e FILE`: True if FILE exists.
      - `-f FILE`: True if FILE exists and is a regular file.
      - `-d FILE`: True if FILE exists and is a directory.
      - `-s FILE`: True if FILE exists and has a size greater than zero.
      - `-r FILE`: True if FILE exists and is readable.
      - `-w FILE`: True if FILE exists and is writable.
      - `-x FILE`: True if FILE exists and is executable.
    - **String Tests:**
      - `STRING1 = STRING2`: True if the strings are equal.
      - `STRING1 != STRING2`: True if the strings are not equal.

        - `-z STRING`: True if the string is empty (zero length).
        - `-n STRING`: True if the string is not empty.
    - **Numeric Tests (use `(( ))` or `-eq`, `-ne`, `-gt`, `-ge`, `-lt`, `-le` with `[ ]`):**
        - `[ "$num1" -eq "$num2" ]`: True if num1 equals num2 (for integers).
- **Nested `if-else`**

```
#!/bin/bash
read -p "Enter a score: " score
if (( score >= 90 )); then
    echo "Grade: A"
elif (( score >= 80 )); then # 'elif' is 'else if'
    echo "Grade: B"
elif (( score >= 70 )); then
    echo "Grade: C"
else
    echo "Grade: D or F"
fi
```

- **The `case` control structure**
    - Provides a multi-way branch based on pattern matching.

```
#!/bin/bash
read -p "Enter a fruit: " fruit
case "$fruit" in
    "apple" | "pear") # Matches apple or pear
        echo "It's a common fruit."
        ;;
    "banana")
        echo "It's yellow."
        ;;
    *) # Default case (matches anything else)
        echo "Unknown fruit."
        ;;
esac
```

- **The loop control structures**
    - **The `while` loop**
        - Executes a block of code as long as a condition is true.

```
#!/bin/bash
count=1
while [ $count -le 5 ]; do
    echo "Count: $count"
    (( count++ ))
done
```

- **The `until` loop**
  - Executes a block of code as long as a condition is false.

```
#!/bin/bash
count=1
until [ $count -gt 5 ]; do
    echo "Count: $count"
    (( count++ ))
done
```

- **The `for` loop structures**
  - **List-based `for` loop (common):** Iterates over a list of items.

```
#!/bin/bash
for file in *.txt; do
    echo "Processing $file"
    # Add file processing commands here
done

for i in "apple" "banana" "cherry"; do
    echo "Fruit: $i"
done
```

  - **C-style `for` loop (numerical iteration):**

```
#!/bin/bash
for (( i=1; i<=5; i++ )); do
    echo "Iteration: $i"
done
```

- **The `break` and `continue` statements**
  - `break`: Exits the current loop immediately.
  - `continue`: Skips the rest of the current iteration and proceeds to the next iteration of the loop.

```
#!/bin/bash
for i in {1..10}; do
    if (( i == 5 )); then
        break # Exit loop when i is 5
    fi
    if (( i % 2 == 0 )); then
        continue # Skip even numbers
```

```
    fi
    echo "Odd number: $i"
done
```

- **Shell meta-characters**
  - Special characters interpreted by the shell.
  - **Redirection:**
    - `>`: Redirect standard output to a file (overwrites). `command > file`
    - `>>`: Redirect standard output to a file (appends). `command >> file`
    - `<`: Redirect standard input from a file. `command < file`
    - `2>`: Redirect standard error to a file. `command 2> error.log`
    - `&>` or `>&`: Redirect both standard output and standard error. `command &> all.log`
    - `<<<`: Here string (provides string as input). `grep "pattern" <<< "some text"`
    - `<<EOF`: Here document (multi-line input until EOF delimiter).

      ```
      cat << EOF
      This is a multi-line
      text block.
      EOF
      ```

  - **Pipe (`|`):** Connects the standard output of one command to the standard input of another.
    - `ls -l | grep ".txt" | sort -r`
  - **Background (`&`):** Runs a command in the background.
    - `sleep 100 &`
  - **Command Substitution (`$()` or `` ` ``):** Replaces a command with its output.
    - `count=$(ls -l | wc -l)`
    - `current_dir=`pwd``
  - **Semicolon (`;`):** Separates commands on a single line.
    - `command1 ; command2`
  - **Logical AND (`&&`):** Executes the second command only if the first succeeds (exit status 0).
    - `mkdir new_dir && cd new_dir`
  - **Logical OR (`||`):** Executes the second command only if the first fails (non-zero exit status).
    - `grep "error" log.txt || echo "No errors found."`
- **Command line expansion**
  - The process by which the shell transforms the command line into actual commands and arguments before execution.
  - **Brace Expansion:** Generates arbitrary strings. `{string1,string2,...}`
    - `echo {a,b}{1,2}` -> `a1 a2 b1 b2`
  - **Tilde Expansion:** Replaces `~` with the home directory. `~/documents`
  - **Parameter Expansion:** Replaces variable names with their values. `$VAR`, `${VAR}`.
    - `${VAR:-default}`: Use default if VAR is unset or null.
    - `${VAR#pattern}`: Remove shortest matching prefix.
    - `${VAR%pattern}`: Remove shortest matching suffix.

- **Command Substitution:** Executes a command and replaces it with its output. `$()`, `` ` ``.
- **Arithmetic Expansion:** Evaluates an arithmetic expression. `$((expression))`
- **Word Splitting:** Splits the results of expansions into distinct words (arguments).
- **Filename Expansion (Globbing):** Matches patterns (`*`, `?`, `[]`) to filenames.

- **Directory stack manipulation**
  - **`pushd`:** Pushes the current directory onto a stack and then changes to the specified directory.
    - `pushd /var/log` (changes to `/var/log`, adds current dir to stack)
  - **`popd`:** Removes the top directory from the stack and changes to it.
    - `popd` (returns to the previously pushed directory)
  - **`dirs`:** Displays the contents of the directory stack.
    - `dirs -v`: Verbose output with numbers.

- **Job control, history and processes**
  - **Job Control:** Managing multiple tasks (jobs) in a single shell session.
    - `Ctrl+Z`: Suspend the foreground job (put it in background, stopped state).
    - `bg`: Resume a suspended job in the background. `bg %1` (job number 1).
    - `fg`: Bring a background job to the foreground. `fg %1`.
    - `jobs`: List current jobs (running, stopped, done).
  - **History:** The shell keeps a list of previously executed commands.
    - `history`: Display command history.
    - `!N`: Execute command number N from history.
    - `!string`: Execute the most recent command starting with `string`.
    - `!!`: Execute the last command.
    - `Ctrl+R`: Reverse search history.
  - **Processes:** Running instances of programs.
    - `ps`: Display information about running processes.
      - `ps aux`: Show all processes by all users, with user/CPU/memory usage.
      - `ps -ef`: Show full format listing, including parent PID.
    - `top`: Dynamic real-time view of running processes (like Task Manager).
    - `kill`: Send a signal to a process (terminate).
      - `kill PID`: Send SIGTERM (default).
      - `kill -9 PID`: Send SIGKILL (forceful, cannot be caught).
    - `pkill`: Kill processes by name or other attributes.
      - `pkill firefox`: Kill all Firefox processes.
    - `killall`: Kill processes by name.
      - `killall apache2`: Kill all Apache processes.

- **Built-ins and functions**
  - **Built-ins:** Commands that are part of the shell itself, not separate executable programs. They run faster because the shell doesn't need to search the `PATH` or fork a new process.
    - **Examples:** `cd`, `pwd`, `echo`, `read`, `export`, `alias`, `history`, `test`, `[`.
    - `type command_name`: Tells you if a command is a built-in, alias, function, or external executable.

- **Functions:** Blocks of shell code that can be named and executed like commands. Useful for organizing scripts and reusing code.
  - **Syntax:**

```
function_name() {
    # commands
    echo "Hello from function_name!"
}
# OR
function function_name {
    # commands
}
```

  - **Calling a function:** `function_name`
  - **Arguments:** Functions can take positional parameters ($1, $2, etc.) just like scripts.

```
greet() {
    echo "Hello, $1!"
}
greet "Alice"
```

  - **Return value:** The exit status of the last command in the function, or explicitly set with `return N`.

```
my_func() {
    echo "Doing something..."
    return 10
}
my_func
echo "Function exited with status: $?" # Output: 10
```

---