Unit I – System Software and Assemblers

- Introduction to System Software Concepts
- Presented by: Akshada Dighe & Swati Chitale
- Course: System Programming
- Sem :B tech Sem III

What is System Software?

- Definition: System software is a type of computer program designed to run a computer's hardware and application programs.
- Acts as an intermediary between user applications and hardware.
- Examples: Operating systems, compilers, assemblers.

Components of System Software

- Operating System (OS): Manages hardware and software resources.
- Compiler: Converts high-level code to machine code.
- Assembler: Translates assembly language to machine code.
- Linker: Combines object files into a single executable.
- Loader: Loads executable into memory.
- Debugger: Helps in identifying and fixing bugs

Evolution of System Software

- 1st Generation: Manual machine coding.
- 2nd Generation: Use of assembly language and assemblers.
- 3rd Generation: Introduction of high-level programming languages.
- 4th Generation: GUI-based OS, multi-tasking, embedded systems.
- 5th Generation and Beyond: Cloud computing, AI-integrated OS, virtual machines.

Language Translators

- Purpose: Convert source code written in one language into another.
- Types:
- - Assembler: Assembly language → Machine language
- - Compiler: High-level language → Machine code (all at once)
- - Interpreter: High-level language → Machine code (line by line)

Machine Structure

- Central Processing Unit (CPU): Executes instructions.
- Registers: Fast storage for temporary data.
- Control Unit: Directs operations of the processor.
- ALU (Arithmetic Logic Unit): Performs calculations and logic.
- Memory: Stores data and programs.
- Input/Output Devices: Facilitate user interaction.

Machine Language

- Consists of binary code (0s and 1s).
- Hardware-dependent.
- Fastest execution but difficult to write and understand.
- Prone to errors.

Assembly Language

- Uses symbolic code (mnemonics) instead of binary.
- One-to-one correspondence with machine instructions.
- Easier to debug and modify than machine code.
- Still hardware-specific.

High-Level Language (HLL)

- Abstracts hardware complexities.
- Closer to human languages (e.g., Python, Java, C++)
- Portable and maintainable.
- Requires translator (compiler or interpreter) to convert to machine code.

Language Translation Basics

- Lexical Analysis: Breaks source code into tokens.
- Syntax Analysis: Checks the grammar and structure.
- Semantic Analysis: Validates the logic and meaning.
- Intermediate Code Generation: Translates into intermediate code.
- Optimization: Improves efficiency.
- Code Generation: Produces final machine

Summary

- System software is vital for running and managing hardware.
- Assemblers, compilers, and interpreters translate code.
- Understanding the layers from machine code to HLL is essential.
- Translation process involves multiple stages.

Unit II – Assembler and Macro Processor

Structure of an Assembler

- • Converts assembly language to machine code
- • Consists of:
- - Symbol Table
- - Opcode Table
- - Intermediate Representation
- - Output Generation

Design of a Single Pass Assembler

- • Scans source code once
- • Performs symbol definition and instruction translation in a single pass
- • Fast but limited in handling forward references

Design of a Two Pass Assembler

- • Pass 1: Constructs symbol table and identifies addresses
- • Pass 2: Translates instructions into machine code
- • Better handling of forward references

Macro Language and Macro Processor

- • Macro Language: Allows definition of macros (code templates)
- • Macro Processor: Expands macros before assembly
- • Improves code reuse and readability

Macro Instructions

- • A macro instruction represents a sequence of instructions
- • Defined once, can be invoked multiple times
- • E.g., MACRO ADDX A, B => ADD A, B; INC A

Features of Macro Facility

- • Reusability
- • Parameterization
- • Conditional Expansion
- • Nested Macro Calls
- • Modular Programming Support

Macro Instruction Arguments

- • Positional arguments: Replaced based on position
- • Keyword arguments: Named arguments for clarity
- • Example: MACRO SWAP &ARG1, &ARG2

Conditional Macro Expansion

- • Allows condition-based macro expansion
- • Uses directives like IF, ELSE, ENDIF
- • Example: IF EQ &X, &Y

Macro Call within Macros (Nested Macros)

- • A macro can invoke another macro
- • Enables layered code abstraction
- • Must ensure no naming conflicts or infinite loops

Macros Defining Other Macros

- • Meta-macros: Macros that generate other macro definitions
- • Advanced technique for flexible code generation

Microprocessor Design

- • Involves the design of CPU components:
- - ALU
- - Control Unit
- - Registers
- - Instruction Set
- • Implemented using digital logic and assembly-level operations

Summary

- • Assemblers translate symbolic code to machine code
- • Macro processors enhance code reuse and modularity
- • Understanding assembler design helps in low-level programming and compiler development

Unit III – Linkers and Loaders

Loader Scheme

- • A loader is a system program that loads an executable file into memory.
- • Types of loaders:
- - Absolute Loader
- - Relocating Loader
- - Direct-Linking Loader

Absolute Loaders

- • Loads executable directly into specified memory locations.
- • Simple and fast.
- • Cannot handle address relocation.
- • Suitable for small programs.

Subroutine Linkages

- • Facilitates modular programming.
- • Link subroutines during program execution or loading.
- • Requires proper address referencing and control transfer.
- • Methods: Static and dynamic linking.

Relocating Loaders

- • Modifies object code to run at different memory locations.
- • Adjusts addresses based on load-time location.
- • Enables flexibility and memory reuse.

Direct Linking Loaders

- • Performs relocation and linking during program loading.
- • Combines multiple object modules.
- • Builds and updates symbol tables.

Other Loader Schemes

- • Binders: Combine independently compiled modules.
- • Linking Loaders: Combine and relocate modules at load time.
- • Overlays: Load parts of program as needed to save memory.

Dynamic Binders

- • Perform linking at runtime.
- • Enable dynamic linking of libraries and modules.
- • Efficient memory usage and flexibility.

Design of an Absolute Loader

- • Reads object file.
- • Loads instructions/data into specified memory locations.
- • No relocation or linking done.
- • Simple logic and implementation.

Design of a Direct-Linking Loader

- • Pass 1: Builds external symbol table.
- • Pass 2: Loads code and performs relocation and linking.
- • Supports modular programs.

Dynamic Link Libraries (DLLs)

- • Libraries loaded at runtime.
- • Shared among multiple programs.
- • Reduce memory usage and support updates without recompilation.

Summary

- • Loaders and linkers manage memory and module connections.
- • Various types serve different needs from static to dynamic linking.
- • Understanding loaders helps optimize program execution.

Unit IV: Compiler

Overview of Compiler Design Concepts

Basic Compiler Function

- A compiler translates source code into machine code.
- Functions include:
- - Lexical Analysis
- - Syntax Analysis
- - Semantic Analysis
- - Optimization
- - Code Generation
- - Code Linking and Assembly

Compiler Phases

- 1. Lexical Analysis
- 2. Syntax Analysis
- 3. Semantic Analysis
- 4. Intermediate Code Generation
- 5. Code Optimization
- 6. Code Generation
- 7. Code Linking

Lexical Analysis

- • Converts character stream to tokens
- • Removes whitespace and comments
- • Recognizes keywords, identifiers, literals, and operators

Syntax Analysis (Parsing)

- • Validates syntax of token sequence
- • Builds a syntax tree
- • Detects errors in structure

Role of Parser

- • Takes tokens from lexical analyzer
- • Produces parse tree or abstract syntax tree (AST)
- • Ensures source code follows grammar rules

Top-down Parsing

- • Starts from root and tries to rewrite to input string
- • Recursive Descent Parser
- • Predictive Parser (LL)

Bottom-Up Parsing

- • Starts from input and attempts to reach the start symbol
- • Builds parse tree from leaves to root
- • Includes LR, SLR, and LALR parsers

Operator Precedence Parsing

- • Handles operators with different precedence
- • Builds correct syntax tree based on precedence and associativity

LR, SLR, and LALR Parsers

- • LR: Left-to-right scanning, Rightmost derivation
- • SLR: Simple LR using Follow sets
- • LALR: Lookahead LR combining states with same items

Intermediate Code Generation

- • Produces an intermediate form between source and machine code
- • Facilitates optimization and portability

Three Address Code

- • Form of intermediate code
- • Uses at most three addresses per instruction
- • Examples: t1 = a + b, t2 = t1 * c

Intermediate Code Forms

- • Postfix notation
- • Syntax trees
- • Directed acyclic graphs
- • Three-address code

Compiler Generation Tools

- • LEX: Lexical analyzer generator
- • YACC: Yet Another Compiler Compiler, generates parsers

Interpreters

- • Directly executes instructions written in a programming language
- • Does not produce machine code
- • Slower than compiled code

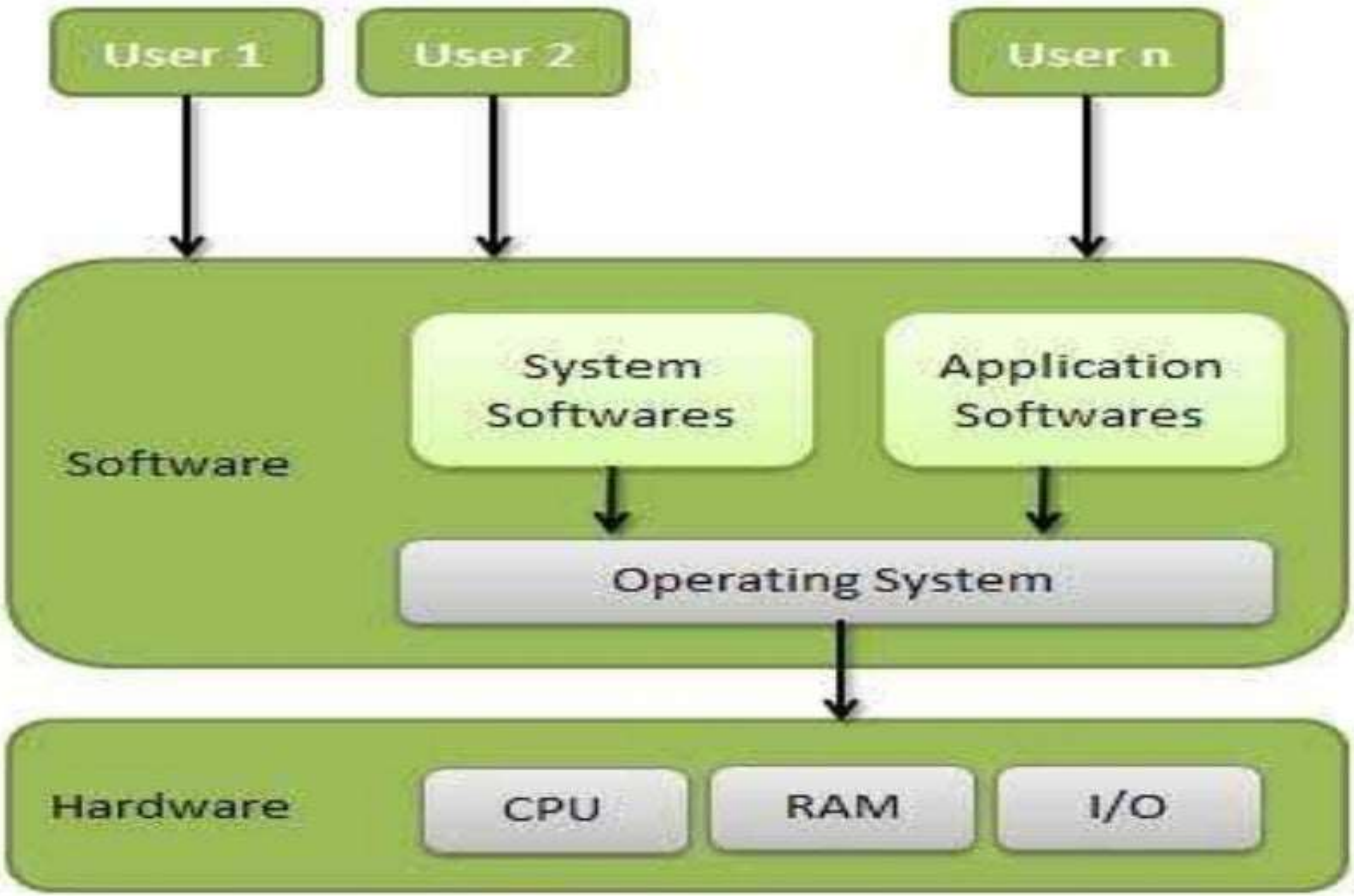
Unit V: Operating System

- Overview of key concepts in Operating Systems and Shell Scripting.

System Concept

- • Manages computer hardware
- • Acts as an intermediary between user and hardware
- • Provides an environment to execute programs

Operating System Diagram :-

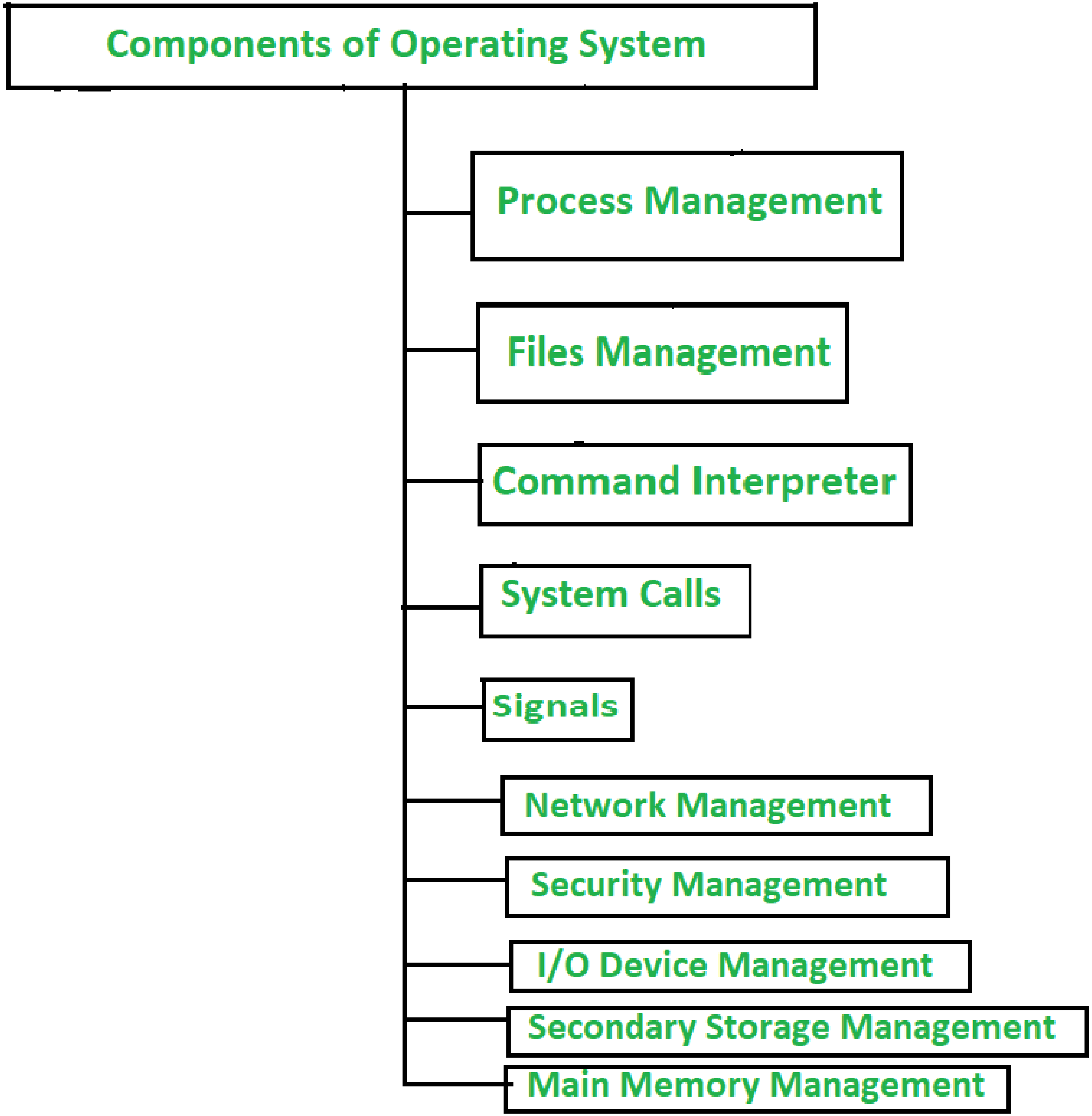


Operating System Structure

- • Monolithic Systems
- • Layered Approach
- • Microkernels
- • Modules
- • Hybrid Systems

Operating System Components

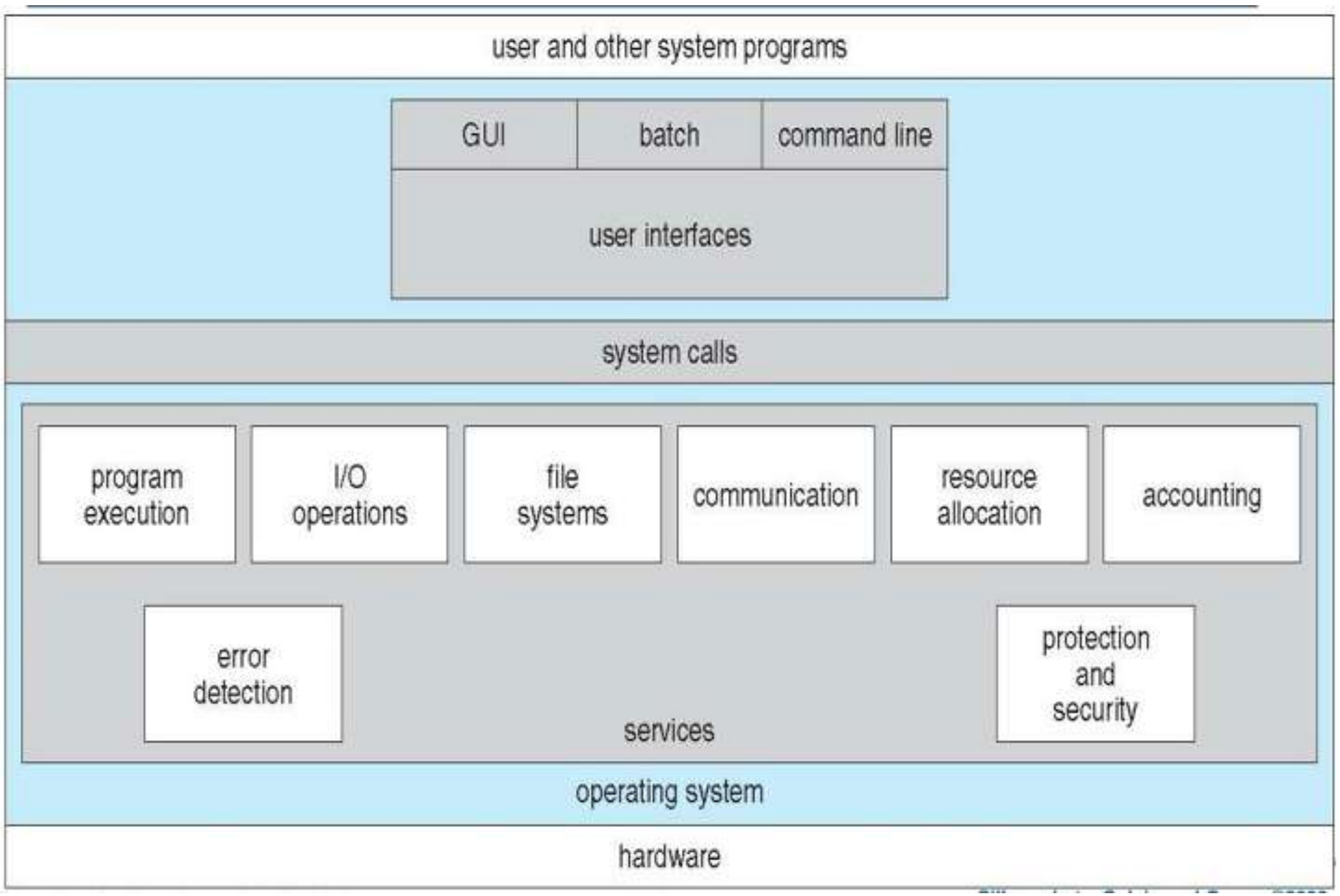
- • Process Management
- • Memory Management
- • File System Management
- • Device Management
- • Security & Protection


Components of Operating System
Process Management
Files Management
Command Interpreter
System Calls
Signals
Network Management
Security Management
I/O Device Management
Secondary Storage Management
Main Memory Management

Operating System Services

- • Program execution
- • I/O operations
- • File system manipulation
- • Communication
- • Error detection

Operating System Services



System Calls

- • Interface between process and OS
- • Types: Process Control, File Management, Device Management, Information Maintenance, Communication

Shell Scripting

- • Bourne Shell (SH)
- • Bourne-Again Shell (BASH)
- • C-Shell (CSH)
- • TCSH
- • Korn Shell (KSH)

Shell Commands

- • Basics: ls, cd, cp, mv, rm
- • Pipelining: |
- • Background/Foreground: &, fg, bg
- • File Permissions: chmod, chown, chgrp

AWK Programming

- • Pattern scanning and processing language
- • Useful for reports and data extraction
- • Syntax: pattern { action }
- • Variables, loops, conditionals

Process Control

- • ps: Display process status
- • top: Real-time process monitoring
- • kill: Terminate process
- • nice/renice: Set process priority

Device Drivers

Overview, Anatomy, Types, and Comparative Study

Definition of Device Drivers

- Software that allows the operating system to communicate with hardware devices.
- Acts as a translator between the hardware and applications or OS.
- Essential for system stability and performance.

Anatomy and Types of Device Drivers

- Anatomy: Initialization, Data Handling, Communication, Error Handling.
- Types:
- - Kernel-mode Drivers
- - User-mode Drivers
- - Virtual Device Drivers
- - Bus Drivers, Function Drivers, Filter Drivers

Device Programming

- Involves writing code to control specific hardware devices.
- Uses specific commands, protocols, and registers.
- Often requires understanding of hardware datasheets and I/O operations.

Installation and Incorporation of Driver Routines

- Drivers are installed via OS-specific mechanisms (e.g., .inf files in Windows).
- Incorporation involves binding drivers with OS services and device managers.
- Requires proper signing and verification for security.

Basic Device Driver Operation

- Initialization during system boot or device connection.
- Interrupt handling and I/O operations.
- Communication with user applications and OS kernel.
- Handling device-specific commands and states.

Implementation with Line Printer

- Example: Writing a driver to communicate with a parallel port printer.
- Tasks include: Initialization, Data Transfer, Status Checking, Error Handling.
- Interaction with port-mapped I/O and interrupt requests (IRQs).

Comparative Study: Unix vs Windows Drivers

- UNIX:
- - Open-source, typically written in C.
- - Integrated into the kernel or as modules.
- - Use of device files (/dev).
- Windows:
- - Often proprietary, developed using WDK.
- - Uses INF files, services, and the registry.
- - Different driver models (WDM, KMDF, UMDF).