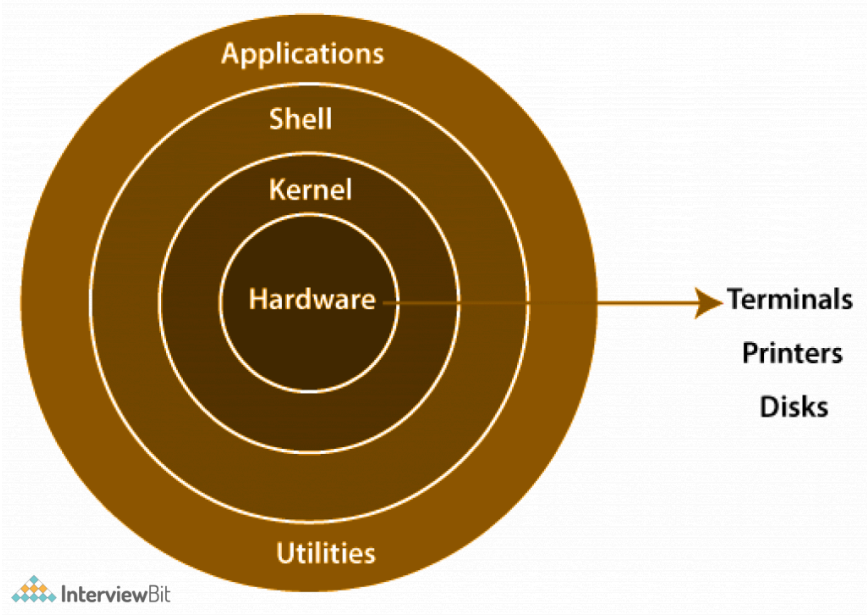
CH:3 Getting Started with Unix, Unix Shell Command

3.1 Unix Architecture



Hardware Layer in Unix

The **hardware layer** is the **lowest level** in Unix architecture. It contains the **physical components of the computer**. Unix itself does not work directly with users at this level; instead, the **kernel** sits on top of hardware and controls it.

The hardware layer mainly includes:

- **CPU (Processor)**
- **Main Memory (RAM)**
- **Secondary Storage (Hard Disk / SSD)**
- **Input Devices**
- **Output Devices**

Kernel Layer

The **kernel layer** is the **core (heart)** of the Unix operating system.
It sits **between hardware and users** and controls **all system resources**.

In simple words:
The kernel is the **manager** that decides
• which program uses the CPU
• how memory is allocated
• how files are stored
• how hardware devices are accessed

Shell Layer

The **shell layer** is the **interface between the user and the kernel**.
It allows users to **communicate with the Unix system** by typing commands.

In simple words:
The shell is a **command interpreter** that understands user commands and asks the kernel to execute them.

Command Interpretation

• Reads commands entered by the user
• Checks syntax and meaning
• Converts commands into system calls
• Example: mkdir

Application Layer

The **application layer** is the **topmost layer** in Unix architecture.
It consists of **programs and utilities** that users actually use to perform tasks.
This is the layer where **real work happens** – editing files, browsing data, compiling programs, etc.

3.2 Unix Features

Unix is popular because of its **powerful, stable, and flexible features**. These features make it suitable for **servers, development, and multi-user environments**.

---

1. Multiuser System

• Multiple users can use the system **at the same time**
• Each user has a separate account and permissions
• Users do not interfere with each other

Example:
In a college lab, many students log in to the same Unix server simultaneously.

---

2. Multitasking

• Unix can run **multiple programs at once**
• CPU time is shared among processes
• Improves system efficiency

Example:
Downloading a file while editing a document and listening to music.

---

3. Portability

• Unix is mostly written in **C language**
• Can run on different hardware platforms with minimal changes

Example:
Same Unix program can run on Intel, ARM, or other processors.

---

4. Security

• Strong **user authentication** (username and password)
• File permissions: read, write, execute
• Protects system from unauthorized access

Example:
A student cannot access another student's private files.

---

5. Powerful Shell

• Shell acts as a **command interpreter**
• Supports scripting and automation
• Allows command chaining using pipes

3.3 Types Of Shell (C, Bourn, Korn)

C

The **C Shell (csh)** is a Unix shell designed with **syntax similar to the C programming language**.
It was created to make shell scripting easier for programmers who already know C.
C Shell provides **interactive features** and **programming-style control structures**, which were advanced at the time it was introduced.

Key Features of C Shell

• **C-like Syntax**
The syntax of C Shell resembles the C programming language.
This makes it easier for C programmers to learn and write shell scripts.

• **Command History**
C Shell stores previously executed commands.
Users can recall and reuse commands instead of typing them again.

• **Job Control**
Supports running programs in the background and foreground.
Users can suspend, resume, or terminate jobs easily.

Bourn

The **Bourne Shell (sh)** is one of the **earliest and most important Unix shells**.
It was developed by **Stephen Bourne** and became the **standard shell** for early Unix systems.

The Bourne shell is mainly used for **shell scripting and system-level tasks**, rather than advanced interactive features.

Key Features of Bourne Shell

• **Simple and Structured Syntax**
Bourne shell uses a clean and simple command structure.
It is easy to read and suitable for writing scripts.

• **Powerful Shell Scripting**
Designed mainly for writing scripts to automate tasks.
Widely used in system startup and administrative scripts.

• **Control Structures**
Supports programming constructs like if–else, case, for, while, and until.
Helps in writing logical and conditional scripts.

Korn

The **Korn Shell (ksh)** is a **powerful and advanced Unix shell** developed by **David Korn** at AT&T Bell Labs.
It was designed to **combine the best features of Bourne Shell (sh) and C Shell (csh)**.

Korn Shell is widely used for **both scripting and interactive work**, especially in commercial Unix systems.

Key Features of Korn Shell

• **Backward Compatibility with Bourne Shell**
Korn shell supports almost all Bourne shell scripts.
This allows easy migration from sh to ksh.

• **Advanced Scripting Capabilities**
Provides functions, arrays, and arithmetic operations.
Makes scripts more structured and powerful.

• **Command History**
Stores previously executed commands.
Users can easily recall and edit commands.

3.4 Unix File System

What is Unix File System?

The **Unix File System** is a method used by Unix to **store, organize, and manage files and directories**.

It follows a **hierarchical (tree-like) structure**, where everything starts from a single root directory `/`.

In Unix, **everything is treated as a file** – data files, directories, and even devices.

Basic Structure of Unix File System

• Root directory `/` is the top of the hierarchy
• All files and directories are organized under `/`
• No separate drive letters like Windows (C:, D:)
• File system looks like an inverted tree

Important Directories in Unix File System

`/` Root Directory

• Topmost directory
• Contains all other directories
• Only system administrator can modify critical files

`/bin`

• Contains essential user commands
• Commands needed in single-user mode
• Examples: `ls`, `cp`, `mv`, `cat`

`/home`

• Contains home directories of users
• Each user has a separate directory
• Example: `/home/johnDoe`

`/dev`

• Contains device files
• Devices are treated as files
• Example: keyboard, hard disk

3.5 Types of Files

In Unix, **everything is treated as a file**.
Based on their purpose and behavior, files are mainly classified into

1) **ordinary files,**
2) **directory files,**
3) **device files**.

---

1. Ordinary Files

Ordinary files are the most commonly used files in Unix.
They store **user data or program instructions** and contain information in text or binary form.

These files are created by users or applications and can be edited, copied, deleted, or executed depending on permissions.

Examples include text files, source code files, executable programs, images, and documents.

Example:
`notes.txt`, `program.c`, `a.out`

---

2. Directory Files

Directory files store information about **other files and directories**.
They act as containers that help organize files in a structured manner.

A directory file does not store actual file data; it stores **file names, locations, and inode references**.

Example:
`/home`, `/usr/bin`, `/etc`

---

3. Device Files

Device files represent **hardware devices** in Unix.
Unix treats hardware devices as files so they can be accessed using standard file operations.

These files are usually stored in the `/dev` directory and are managed by the kernel using device drivers.

Example:
Keyboard, hard disk, printer files in `/dev`

Comparison (Quick View)

• Ordinary files – store user data or programs
• Directory files – store file and directory information
• Device files – represent hardware devices

3.6 Unix File & Directory Permissions

Unix uses **permissions** to control **who can access a file or directory and how**.
This ensures **security and controlled sharing** in a multi-user environment.

Each file or directory has:
• **Owner (user)**
• **Group**
• **Others**

And three types of permissions:
• **Read (r)**
• **Write (w)**
• **Execute (x)**

Permission Representation

Permissions are shown as a **10-character string**:

Example:
`-rwxr-xr--`

Meaning:
• `-` → ordinary file (`d` for directory)
• `rwx` → owner permissions
• `r-x` → group permissions
• `r--` → others permissions

Meaning of Permissions for Files

• **Read (r)**
Allows viewing file content

• **Write (w)**
Allows modifying file content

• **Execute (x)**
Allows running the file as a program

Example:
If a file has `rwx` permission, the user can read, edit, and execute it.

Meaning of Permissions for Directories

Permissions work **differently for directories**.

• **Read (r)**
Allows listing files inside the directory

• **Write (w)**
Allows creating, deleting, or renaming files

• **Execute (x)**
Allows entering the directory using `cd`

Important note:
Without execute permission, a directory cannot be accessed even if read permission exists.

Permission Groups Explained

• **Owner**
The user who created the file

• **Group**
Users belonging to the same group

• **Others**
All remaining users

Changing Permissions – `chmod`

Permissions can be changed using the `chmod` command.

Symbolic method:

- `chmod u+x file.txt`
- `chmod g-w file.txt`
- `chmod o=r file.txt`

3.7 Connecting Unix Shell : Telnet

What is Telnet?

Telnet is a **network protocol** used to **connect to a remote Unix system** and access its **shell (command line)** over a network.

In simple words:
Telnet allows a user to **log in to another computer remotely** and work on its Unix shell as if they were sitting in front of it.

How Telnet Works (Step-by-Step)

- User opens a terminal on local machine
- User types `telnet hostname` or `telnet IP_address`
- Telnet client sends a connection request
- Remote system responds
- User enters **username and password**
- After authentication, **Unix shell is provided**

Now the user can execute commands on the remote system.

Example Explanation

Suppose a student wants to access the **college Unix server** from home.

- Student runs telnet command
- Connects to college server
- Logs in using credentials
- Gets Unix shell prompt

All commands run on the **college server**, but input/output appears on the student's screen.

3.8 Login Commands

These **login-related commands** are used to **manage user sessions, identity, and system information** in Unix.

1. `passwd:` The `passwd` command is used to **change the password of a user account**. For security, Unix asks for the current password before allowing a new one.

2. `logout:` The `logout` command is used to **end the current login session**. After logout, the user is returned to the login screen.

3. `who:` The `who` command displays **all users currently logged into the system**. It also shows login time and terminal used.

4. `who am i:` The `who am i` command shows **details of the current logged-in user**. It is useful in multi-user environments.

5. `clear:` The `clear` command is used to **clear the terminal screen**.
It does not affect files or running programs.

6. `uname:` The `uname` command displays **system-related information** such as the operating system name.

3.9 Unix File / Directory Related Commands

These commands are used to **create, view, copy, move, search, and manage files and directories** in Unix systems.

1. `ls:` Lists files and directories in the current directory. Eg: ls

2. `cat:` Prints the entire file content on the terminal. Eg: cat file.txt

3. `cd:` Changes the current working directory. Eg: cd /home/student

4. `pwd:` Displays the full path of the current directory. Eg: `pwd`

5. `mv:` Moves or renames files and directories. Eg: mv old.txt new.txt

6. `cp:` Copies files or directories. Eg: cp file1.txt file2.txt

7. **`ln:`** Creates links between files. Eg: ln file1.txt link1.txt
8. **`rm:`** Deletes files Eg: rm file.txt
9. **`rmdir:`** Removes a directory only if it is empty. Eg: rmdir emptydir
10. **`mkdir:`** Creates a new directory. Eg: mkdir project
11. **`chmod:`** Changes file or directory permissions. Eg: chmod 755 script.sh
12. **`chown:`** Changes the owner of a file. Eg: chown user1 file.txt
13. **`chgrp:`** Changes the group of a file. Eg: chgrp students file.txt
14. **`find:`** Searches for files and directories. Eg: find /home -name "file.txt"
15. **`more:`** Displays file content page by page. Eg: more bigfile.txt
16. **`less:`** Allows forward and backward navigation. Eg: less bigfile.txt
17. **`head:`** Displays first few lines of a file By default 10 line. Eg: head file.txt
18. **`tail:`** Displays last few lines of a file. Eg: tail file.txt
19. **`wc:`** Counts lines, words, and characters. Eg: wc file.txt
20. **`touch:`** Creates an empty file. Eg: touch newfile.txt
21. **`stat:`** Displays detailed file information. Shows size, permissions, and timestamps. Eg: stat file.txt
22. **`alias:`** Creates a shortcut command. Eg: alias ll='ls -l'
23. **`type:`** Tells whether a command is built-in or external. Eg: type ls

3.10 Operators in Redirection & Piping

In Unix, **redirection and piping operators** control **input and output flow** between commands and files. <, >, <<, >>, |

1. `<` (Input Redirection): The `<` operator is used to **take input from a file instead of the keyboard**. The command reads data directly from the specified file. Eg: cat < input.txt

2. `>` (Output Redirection): The `>` operator is used to **send output to a file** instead of displaying it on the screen. If the file already exists, its content is **overwritten**. Eg: ls > files.txt

3. `<<` (Here Document): The `<<` operator allows **multiple lines of input directly from the terminal**. Input continues until a specified delimiter is reached.

Eg: cat << EOF

Hello

Unix World

EOF

4. `>>` (Append Redirection): The `>>` operator is used to **append output to an existing file**. It does not overwrite previous content. Eg: date >> log.txt

5. `|` (Pipe Operator): The `|` operator connects **output of one command to input of another**. It allows commands to work together. Eg: ls | wc -l The output of `ls` becomes input to `wc -l`, which counts the number of files.

Key Points to Remember

• Redirection controls **input and output**
• `<` takes input from file
• `>` writes output (overwrite)
• `>>` appends output
• `<<` allows multiline input
• `|` connects commands

3.11 Finding Patterns in Files

In Unix, **pattern-finding commands** are used to **search text inside files**.
They are very useful for **logs, source code, and large text files**.

1. `grep` (Global Regular Expression Print): `grep` searches for a **pattern (word or regular expression)** in a file and prints matching lines. Eg: grep "error" logfile.txt

2. `fgrep` (Fixed grep): `fgrep` searches for **fixed strings only**, not regular expressions. Special characters are treated as **normal characters**. Eg: fgrep "a+b" math.txt

3. `egrep` (Extended grep): `egrep` supports **extended regular expressions**, which are more powerful and expressive. Eg: egrep "error|fail" logfile.txt

3.12 Working with Columns and Fields

In Unix, **column and field processing commands** are used to **extract, combine, and relate data** from files.
They are especially useful when working with **tabular data**, and CSV-like files.

1. `cut:` The `cut` command is used to **extract specific columns or fields** from each line of a file.

Eg: cut -d "," -f 1 names.csv
Explanation: Extracts the **first column** from a comma-separated file.

Eg: cut -c 1-5 file.txt
Explanation: Displays characters from position 1 to 5 in each line.

2. `paste:` The `paste` command is used to **merge files line by line**. It places columns from different files side by side.
Eg: paste file1.txt file2.txt
Explanation: Combines lines from both files into **columns**.

3. `join:` The `join` command is used to **combine two files based on a common field**. It is similar to **JOIN operation in databases**.

Eg: join file1.txt file2.txt
Explanation: Joins records from both files having the same first column.

3.13 Comparing Files in Unix : `cmp`, `comm`, `diff`

Unix provides **file comparison commands** to check **similarities and differences** between files. These are useful in **programming, version control.**

1. `cmp` (Compare)

The `cmp` command compares **two files byte by byte**. It tells **whether files are identical or where the first difference occurs**.

Eg: cmp file1.txt file2.txt

Output (if different): file1.txt file2.txt differ: byte 15, line 2

Explanation: This means the first difference is found at **byte 15 on line 2**.

2. `comm` (Compare Sorted Files Line by Line)

The `comm` command compares **two sorted text files** line by line.
It divides output into **three columns**.

Important rule:
Both files **must be sorted** before using `comm`.

Eg: comm file1.txt file2.txt

Output format:
- Column 1 – lines only in file1
- Column 2 – lines only in file2
- Column 3 – lines common in both files

Example explanation:
If a name appears in both files, it will appear in **column 3**.

3. `diff` (Difference Between Files)

The `diff` command compares **text files line by line** and shows **exact differences**.

Eg: diff file1.txt file2.txt

Output:

2c2

< Hello World

> Hello Unix

Explanation:

Line 2 in file1 is changed to line 2 in file2.