Comprehensive Notes on Linux and Shell Scripting

Based on Lectures by Dr. Tariq Mahmood

Contents

1 The History and Philosophy of Linux

The story of Linux is one of collaboration, innovation, and a drive to create free and open-source software. It begins with its predecessor, Unix.

1.1 The Unix Roots (1969 – 1980s)

- **Creation:** In 1969, Ken Thompson and Dennis Ritchie at AT&T Bell Labs created Unix. Initially written in assembly, it was later rewritten in C, a language also developed at Bell Labs. This rewrite made Unix portable across different hardware platforms, a revolutionary concept at the time.
- **Features:** Unix was a powerful multiuser and multitasking operating system.
- **Adoption and Licensing:** It quickly gained popularity in academic and research institutions. However, AT&T's licensing restrictions meant that Unix was not free software, which limited its accessibility.

1.2 The GNU Project (1983)

- **Vision:** In 1983, Richard Stallman launched the GNU Project with the ambitious goal of creating a completely free and open-source, Unix-like operating system. "GNU" is a recursive acronym for "GNU's Not Unix."
- **Contributions:** The GNU Project successfully developed many essential components of an operating system, including:
  - The GNU Compiler Collection (GCC).
  - The GNU Shell, which became **bash** (Bourne Again Shell).
  - Core utilities like `cp`, `ls`, `grep`, etc.
- **The Missing Piece:** Despite these successes, the GNU Project's own kernel, the GNU Hurd, faced development delays and was not ready.

1.3 The Birth of the Linux Kernel (1991)

- **Linus Torvalds' Project:** In 1991, Linus Torvalds, a Finnish student, began working on a new operating system kernel as a personal hobby. He was inspired by Minix, a teaching OS.
- **The Announcement:** On August 25, 1991, he famously announced his project on a Usenet newsgroup, stating, "I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu)."
- **First Release:** Linux version 0.01 was released in September 1991.

1.4 Linux + GNU = A Complete OS (1992–1994)

The combination of the stable Linux kernel with the mature GNU toolset created a fully functional, free operating system. This is why the system is often referred to as **GNU/Linux**.

- **Licensing:** In 1992, Linux adopted the GNU General Public License (GPL), legally ensuring that it and its derivatives would remain free and open-source.
- **Rise of Distributions:** In the mid-1990s, developers started packaging the Linux kernel with GNU software and other applications, creating **Linux distributions (distros)**. Early pioneers included Debian (1993) and Red Hat (1994).

1.5 The Linux Philosophy

The design and culture of Linux are guided by a few core principles:

- **"Everything is a file":** In Linux, nearly all system resources, including hardware devices, processes, and configuration settings, are represented as files in the filesystem.
- **"Do one thing and do it well":** Linux promotes the use of small, specialized tools that can be combined (often using pipes) to perform complex tasks.
- **Open-source, community-driven development:** The development model is collaborative and transparent, relying on contributions from thousands of developers worldwide.

2 Linux Architecture

Linux has a layered architecture that separates the hardware from the user applications, with the kernel acting as the central mediator.

2.1 Modes of Operation: Kernel vs. User Space

- **Kernel Mode:** The privileged mode where the kernel runs. It has unrestricted access to all hardware and system resources.
- **User Mode:** A restricted mode where user applications run. They cannot access hardware directly and must request services from the kernel.
- **System Calls:** The mechanism by which a user-mode application requests a service from the kernel. This involves a switch from user mode to kernel mode, execution of the service, and a return to user mode.

2.2 The Core Architectural Layers

1. **Hardware Layer:** The physical components of the system (CPU, RAM, storage, network devices).

2. **Kernel Layer:** The core of the OS. It manages the hardware, runs in privileged mode, and provides an abstraction layer for user programs. It is composed of several key subsystems.

3. **System Call Interface (SCI):** The bridge between the user space and the kernel space. The GNU C Library (`glibc`) provides a user-friendly wrapper around the SCI.

4. **User Space:** The environment where user applications and shells run in a less-privileged mode.

2.3 Detailed Kernel Components

The Linux kernel is modular and consists of five major subsystems:

Process Management Creates, schedules, and terminates processes. It handles multitasking by switching the CPU between different running processes.

Memory Management Handles the allocation and deallocation of system memory (RAM). It implements virtual memory, which gives each process its own private address space.

File System Management Provides a hierarchical directory structure and manages access to files on storage devices. It uses a Virtual File System (VFS) to support many different file system types (e.g., ext4, XFS, NTFS).

Device Management Uses device drivers to manage and communicate with hardware devices. These devices are represented as files, typically in the `/dev` directory.

Networking Implements the full TCP/IP networking stack, allowing communication between processes, systems, and the internet.

3 Users, Groups, and Permissions

Linux is a multiuser operating system, which requires a robust system for managing access to resources.

3.1 Users and Groups

- **Users:** Every person or process is assigned a user account. There are three types:
    1. **Root User (Superuser):** The administrator account with full privileges. Very powerful but also dangerous.
    2. **Normal Users:** Standard user accounts with limited access, typically unable to modify system files.
    3. **System Users:** Accounts created automatically for running services (e.g., `www-data` for a web server).

- **Groups:** A collection of users. Groups make it easier to manage permissions for multiple users at once. A user belongs to a primary group and can be a member of multiple secondary groups.

3.2 The Permissions Model

Every file and directory in Linux has permissions assigned to three categories of users:

Owner (User): The user who created the file.

Group: All users who are members of the file's group.

Others (World): Everyone else.

There are three basic types of permissions:

Read (r): View the contents of a file or list the contents of a directory.

Write (w): Modify a file or create/delete files within a directory.

Execute (x): Run a file (if it's a script/program) or enter (cd into) a directory.

3.3 Symbolic and Octal Representation

Permissions are commonly represented in two ways:

- **Symbolic:** A 10-character string, e.g., `-rwxr-xr--`.
  - The first character indicates the file type (`-` for file, `d` for directory).
  - The next three (`rwx`) are the owner's permissions.
  - The next three (`r-x`) are the group's permissions.
  - The last three (`r--`) are the permissions for others.
- **Octal (Numeric):** A three-digit number where each digit represents the permissions for owner, group, and others, respectively. The value is calculated by summing the numbers for each permission type: **r=4, w=2, x=1**.
  - `rwx` $= 4 + 2 + 1 =$ **7**
  - `r-x` $= 4 + 0 + 1 =$ **5**
  - `r--` $= 4 + 0 + 0 =$ **4**
  - Therefore, `rwxr-xr--` is represented as **755**.

4 Introduction to Shell Scripting

Shell scripting is the primary way to automate tasks and manage systems in Linux.

4.1 History and Comparison of Shells

A shell acts as an interface between the user and the kernel. It interprets commands and executes them.

Shell	Origin	Syntax Style	Status Today
sh	S. Bourne, 1977	Minimal, simple	Still used for portability (`/bin/sh`)
bash	GNU, 1989	Bourne + extras	Most common, Linux default
csh	Bill Joy, 1978	C-like	Mostly obsolete
ksh	David Korn, 1983	Bourne + advanced	Niche, used in some enterprises

4.2 What is Shell Scripting Used For?

Shell scripting automates repetitive tasks, manages system operations, and streamlines workflows.

- **System Automation:** Backups, cron jobs, cleanup tasks.
- **Monitoring:** CPU/disk alerts, service health checks.
- **Development/Deployment:** Build and test automation (CI/CD), environment setup.
- **Data/ETL:** Cleaning logs, merging CSV files, pre-processing data.
- **Glue Work:** Orchestrating multiple tools and commands to work together.

5 BASH Scripting Fundamentals

5.1 The Shebang and Executing a Script

- **The Shebang (`#!`):** The first line of a script must be a shebang, which tells the operating system which interpreter to use to run the script. For Bash scripts, this is `#!/bin/bash`.
- **Making a Script Executable:** Before a script can be run directly, it must be given execute permissions using the `chmod` command.

```
# Give the user execute permission
chmod u+x myscript.sh

# A common alternative for owner, group, and others
chmod +x myscript.sh

# Run the script
./myscript.sh

```

5.2 Variables and Positional Parameters

- **Variables:** Declared with `name="value"`. No spaces around the equals sign. Accessed with a dollar sign, e.g., `$name`.
- **Positional Parameters:** Special variables that hold the arguments passed to a script.
  - `$0`: The name of the script itself.
  - `$1`, `$2`, ...: The first, second, etc., arguments.
  - `$@`: All arguments as separate strings.
  - `$#`: The total number of arguments.

5.3 Operators and Tests

BASH provides different operators for tests and evaluations:

- `[ expression ]` or `test expression`: The original test command. Good for string and file tests, but can be clumsy with numbers.
- `[[ expression ]]`: An extended test available in Bash. Supports more advanced features like regex matching and string comparisons.
- `(( expression ))`: Used for purely arithmetic evaluation and comparison.

6 Practical Shell Scripting Examples

6.1 Basic Constructs

6.1.1 Variables and Arithmetic

```bash
#!/bin/bash
name="Alpine"
echo "Welcome to $name Docker!"

a=5
b=10
sum=$((a + b))
echo "Sum: $sum"
```

Listing 1: Variables and Arithmetic

6.1.2 Conditional Statements (if)

```bash
#!/bin/bash
num=10
# -gt means "greater than"
if [ "$num" -gt 5 ]; then
  echo "$num is greater than 5"
else
  echo "$num is less than or equal to 5"
fi
```

Listing 2: If Statement

6.1.3 Loops (for and while)

```bash
#!/bin/bash
# For loop
echo "For loop:"
for i in {1..5}; do
  echo "Iteration $i"
done

# While loop
echo "While loop:"
count=1
```

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

Listing 3: For and While Loops

6.1.4 Functions

```bash
#!/bin/bash
greet() {
  # $1 is the first argument passed to the function
  echo "Hello, $1!"
}

greet "World" # Call the function
```

Listing 4: Functions

6.2 File and System Operations

6.2.1 Checking for a File's Existence

```bash
#!/bin/bash
FILE="test.txt"
# -f checks if it's a regular file
if [ -f "$FILE" ]; then
  echo "File '$FILE' exists."
else
  echo "File '$FILE' does not exist."
fi
```

Listing 5: File Existence Check

6.2.2 Reading User Input

```bash
#!/bin/bash
echo "Enter your name:"
read name
echo "Hello, $name!"
```

Listing 6: User Input

6.2.3 Reading a File Line by Line

```bash
#!/bin/bash
FILENAME="log.txt"
# IFS= prevents trimming whitespace
# -r prevents backslash interpretation
while IFS= read -r line; do
  echo "Processing line: $line"
done < "$FILENAME"
```

Listing 7: Reading a File

6.3 Scripting in a DevOps Context

Shell scripting is essential for automating infrastructure and application deployment, especially with tools like Docker.

6.3.1 Example: Hadoop Cluster Setup Script

A script can automate the entire setup of a multi-container application.

```bash
#!/bin/bash
# Start Hadoop master and worker nodes

# Create a dedicated network
docker network create hadoop-net

# Start the namenode (master)
docker run -d --name namenode --net hadoop-net -p 9870:9870 hadoop-
    namenode

# Start worker nodes
docker run -d --name datanode1 --net hadoop-net hadoop-datanode
docker run -d --name datanode2 --net hadoop-net hadoop-datanode

echo "Hadoop cluster started."
```

Listing 8: Hadoop Docker Setup

6.3.2 Example: Automating Job Submission

Scripts make running complex jobs repeatable and less error-prone.

```bash
#!/bin/bash
INPUT_FILE="dataset.csv"
HDFS_INPUT_DIR="/input"

# Copy data from host to the container's temp directory
docker cp "$INPUT_FILE" namenode:/tmp/

# Move the data into HDFS
docker exec namenode hdfs dfs -put "/tmp/$INPUT_FILE" "$HDFS_INPUT_DIR"

# Run the wordcount MapReduce job
docker exec namenode hadoop jar \
  /opt/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar \
  wordcount "$HDFS_INPUT_DIR" /output

echo "Wordcount job submitted."
```

Listing 9: MapReduce Job Submission