Lecture 2.1: Linux Scripting

Shell scripting originated in the early 1970s with the development of Unix.

Stephen Bourne created The Bourne Shell (sh) - one of the first to lay the groundwork for modern shell scripting.

Then, more shells were developed, including C Shell (csh), Korn Shell (ksh), and Bourne Again Shell (bash). Each brought enhancements and features for scripting.

A comparison:

Shell	Origin	Syntax Style	Good For	Status Today
sh	Bourne, 1977	Minimal, simple	Portable scripts	Still used (default `/bin/sh`)
bash	GNU, 1989	Bourne + extras	Everyday scripting, Linux default	Most common
csh	Bill Joy, 1978	C-like	Interactive use (historical)	Mostly obsolete, replaced by `tcsh`
ksh	David Korn, 1983	Bourne + advanced	Unix enterprise systems	Niche but alive

Usage:

Shell scripting is used to automate repetitive tasks, manage system operations, and streamline workflows. We can execute commands, manipulate files, and control system processes.

Usage Area	Examples
System Automation	Backups, cron jobs, cleanup
Monitoring	CPU/disk alerts, service watchdog
File/Text Handling	Bulk rename, log parsing, formatting
Development	Build/test automation, CI/CD
Deployment	Environment setup, software installs
Networking	Uptime checks, downloads, transfers
Data/ETL	Cleaning logs, CSV merging, pre-Hadoop prep
Security	Audit permissions, login monitoring
Education	OS concepts, scripting practice
Glue Work	Orchestrating multiple tools

Usage to Come:

Setup:

Instead of typing 10 Docker commands, one script can:

- Pull images.
- Create networks,
- Start containers with proper volumes and ports.

#!/bin/bash

 # Start Hadoop master and worker nodes

docker network create hadoop-net

docker run -d --name namenode --net hadoop-net -p 9870:9870 hadoop-namenode

docker run -d --name datanode1 --net hadoop-net hadoop-datanode

docker run -d --name datanode2 --net hadoop-net hadoop-datanode

Data Ingestion:

Automate process of moving datasets into Hadoop/Spark containers.

- Upload data from host → container.
- Load into HDFS automatically.

#!/bin/bash

docker cp dataset.csv namenode:/tmp/

docker exec namenode hdfs dfs -put /tmp/dataset.csv /input/

Job Submission:

Running jobs (MapReduce, Hive queries, Spark scripts) inside containers.

Scripts make this repeatable.

#!/bin/bash

docker exec namenode hadoop jar /opt/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-.jar wordcount /input /output*

Monitoring & Logging

Scripts can simplify monitoring cluster health.

- Check container status.
- Tail Hadoop/Spark logs.
- Alert if a container crashes.

```
#!/bin/bash

for c in namenode datanode1 datanode2; do

 echo "---- Logs for $c ----"

 docker logs --tail 5 $c

done
```

Automation of Experiments

Repeat experiments with:

- Different dataset sizes.
- Different cluster sizes (1 master, 2 slave→ 1 master, 4 slave).
- Different job configurations.

```
#!/bin/bash

for size in 1GB 2GB 5GB; do

 echo "Running wordcount for dataset $size"

 docker exec namenode hdfs dfs -put /datasets/$size.txt /input/

 docker exec namenode hadoop jar /opt/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar wordcount /input /output_$size

done
```

Cluster Management

Start/stop all containers in one command.

Clean up containers and networks after lab sessions.

```
#!/bin/bash
```

docker stop $(docker ps -q)

docker rm $(docker ps -aq)

docker network rm hadoop-net

Integration with Cron

Schedule jobs with cron inside containers.

Example: Run a Hive query every night at 2 AM.

Teaches **automation + scheduling** in a distributed environment.

Alpine:

Alpine Linus is a lightweight Linux distro based on musl libc and busybox. It was released in 2005. It focuses on simplicity, resource-efficiency and security.

Alpine uses **Almquist Shell (ash)**, a lightweight shell which is POSIX-compliant. The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

Many Unix and Linux distros are **POSIX** and are interchangeable. You can also install bash if needed.

The **apk** is the Alpine Package Keeper for package management for installation of packages. Some common uses of scripting in Alpine are *Docker container management, system monitoring (DevOps), deployment scripts (DevOps) and configuration management*.

About BASH:

A shell language is a programming language used as a command-line interface (CLI) for interacting with the OS. It provides a way for users to execute commands, run programs, and manage files and directories. It is an interpreter between the user and OS, allowing them to communicate and perform tasks. It is basically used to write scripts.

A few facts about bash:

• Bash (Bourne Again SHell) is the default shell on most Linux distributions

• Its syntax is based on the Bourne shell (it shares many similarities with other shell languages)

• Bash variables are created and assigned values using the syntax "variable_name=value"- case-sensitive by default, and their values can be accessed using the syntax "$variable_name".

• Bash also supports environment variables, which are global variables that are available to all processes running on the system - can be accessed and modified using the "export" command.

• Offers a range of features for command-line editing and history – command completion, automatic word and path completion, and history expansion. These features make it easy to navigate and edit commands, saving time and effort.

• Bash keeps a history of executed commands, allowing users to recall and reuse commands quickly using shortcuts or by searching through the history.

• Bash offers a wide range of built-in commands, control structures, and features such as arrays, functions, and arithmetic operations

• Bash scripts can be used for various purposes, from simple automation tasks to complex system administration workflows.

• Bash, being the most popular shell language, has a large user community and extensive support - powerful tool for automation and system administration.

A bit about Shebang:

A shebang (also called hashbang) is the character sequence: **#!**

It is placed at the very beginning of a script file.

It tells the operating system **which interpreter should be used to run the script**.

How it works:

- When you run a script (e.g., ./myscript.sh), the kernel looks at the first line.
- If it starts with #!, the kernel uses the specified interpreter to run the file.

- Example:
    - #!/bin/bash
    - echo "Hello World"
    - #!/bin/bash tells the OS: “Use the **Bash shell** to execute this script.”

Common Examples:

- *#!/bin/bash* → Run with Bash shell.
- *#!/bin/sh* → Run with Bourne shell (or its equivalent, often dash in modern systems).
- *#!/usr/bin/env python3* → Run with Python 3 (using env to find it in PATH).
- *#!/usr/bin/perl* → Run with Perl interpreter.

Imp:

- If no shebang is present, the script will run with the **default shell** (often /bin/sh) when invoked directly.
- Shebang only matters when executing the script as a program (./myscript.sh), not when you explicitly call an interpreter (bash myscript.sh).

A bit about chmod:

Used to make a file executable: chmod +x filename

chmod → change file **mode** (permissions).

+x → add the **execute permission**.

Try out the following:

Create script: *echo -e '#!/bin/bash\necho "Hello World"' > myscript.sh*

Check permissions: *ls -l myscript.sh*

Should get: *-rw-r--r-- 1 user group 24 Sep 10 08:20 myscript.sh*

Make executable: chmod +x myscript.sh

Should get: -rwxr-xr-x 1 user group   24 Sep 10 08:20 myscript.sh

Run: *./myscript.sh*

Scripting with BASH:

Go into alpine: *docker run -it alpine*

Install bash as follows:

• Update: *apk update*

• Add bash: *apk add bash*

• Check version: *bash --version*

Install nano (editor): *apk add nano*

Go to /home and: *mkdir scripts* and *cd scripts*

Create your first script: *touch first.sh*

Nano: *nano first.sh*

Add the Shebang and Content:

#!/bin/bash

echo "Hello, World!"

Ctrl+s and Ctrl+x

Make executable: *chmod +x first.sh*

Run: *./first.sh*

Operators:

- [ ] → for string & file tests, but clumsy with numbers.
- [[ ]] → extended test, supports regex, string comparison.
- (( )) → purely arithmetic evaluation.

Positional Parameters

- $0 → the name of the script itself.
- $1 → the **first argument** passed to the script.
- $2 → the **second argument**.
- $3 … $9 → third to ninth arguments.
- ${10}, ${11} … → for arguments beyond 9 (must use braces).

File Exists:

- `-e file` → file exists (any type: file, dir, socket, etc.).
- `-d file` → file is a directory.
- `-r file` → file is readable.
- `-w file` → file is writable.
- `-x file` → file is executable.

IFS

IFS tells the shell **how to split text into words/tokens**.

IFS=" <tab><newline>" (ie space, tab, and newline are separators.)

Simple Scripts with BASH:

Ex1: Define a variable name and use it with $ to display its value.

```
#!/bin/bash

name="Alpine"

echo "Welcome to $name Docker!"
```

Ex2: Arithmetic: (( )) used for arithmetic evaluation (also for conditional testing)

```
#!/bin/bash

a=5

b=10

sum=$((a + b))
```

```
echo "Sum: $sum"
```

Ex3: if statement to check conditions. -gt means "greater than".

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

Ex4: for loop to iterate through 1 to 5 and prints each iteration.

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

Ex5: while loop continues until the condition is false, incrementing the count each time.

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

Ex6: define a function greet that takes one argument ($1) and prints a greeting.

```
#!/bin/bash
greet() {
 echo "Hello, $1!"
}
greet "Alpine"
```

Ex7: The read command captures user input and stores it in the variable name.

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

Ex8: This checks if a file named test.txt exists using the -f flag.

```
#!/bin/bash
if [ -f "test.txt" ]; then
 echo "File exists."
else
 echo "File does not exist."
fi
```

Ex9: creates a directory at /tmp/mydir. The -p option prevents errors if the directory already exists.

```
#!/bin/bash
```

mkdir -p /tmp/mydir

echo "Directory created."

Ex10: creates a file and copies it to another location.

#!/bin/bash

echo "Creating a test file..."

echo "This is a test." > test.txt

cp test.txt /tmp/test_copy.txt

echo "File copied to /tmp/"

Ex11: The rm command deletes the test.txt file. The -f option forces the removal without prompt.

#!/bin/bash

rm -f test.txt

echo "Test file removed."

Ex12: output redirection: The first command creates a file; the second appends to it.

#!/bin/bash

echo "Logging output..." > log.txt

echo "This will go to the log file." >> log.txt

Ex13: This reads a file line by line using a while loop. IFS= read -r prevents leading/trailing whitespace from being trimmed.

#!/bin/bash

echo "Reading file..."

while IFS= read -r line; do

```
 echo "$line"
done < log.txt
```

Ex14: a simple menu-driven script: The case statement handles user choices, allowing listing files, creating a directory, or exiting.

```
#!/bin/bash
while true; do
 echo "1. List files"
 echo "2. Create directory"
 echo "3. Exit"
 read choice
 case $choice in
    1) ls ;;
    2) echo "Enter directory name:"; read dirname; mkdir -p "$dirname"; echo
    "Directory created." ;;
    3) exit ;;
    *) echo "Invalid option" ;;
 esac
done
```

Ex15: Checks if exactly two arguments are provided. If not, it prints usage instructions and exits.

```
#!/bin/bash
# Check for command-line arguments
if [ "$#" -ne 2 ]; then
 echo "Usage: $0 <arg1> <arg2>"
 exit 1
```

```
fi
echo "Argument 1: $1"
echo "Argument 2: $2"
```

Ex16: Demonstrates string manipulation using parameter expansion to convert to uppercase and lowercase.

```
#!/bin/bash
string="Hello, World!"
echo "Original: $string"
echo "Uppercase: ${string^^}"
echo "Lowercase: ${string,,}"
```

Ex17: Shows how to define and access elements of an array in Bash.

```
#!/bin/bash
fruits=("apple" "banana" "cherry")
echo "First fruit: ${fruits[0]}"
echo "All fruits: ${fruits[@]}"
```

Ex18: Reads a file line by line and processes each line.

```
#!/bin/bash
# Assuming a file named "data.txt" exists
while IFS= read -r line; do
 echo "Processing: $line"
done < data.txt
```

Ex19: The set -e command causes the script to exit if any command fails. The error is handled with a message.

```
#!/bin/bash

set -e # Exit immediately if a command exits with a non-zero status

cp source.txt destination.txt || { echo "Copy failed"; exit 1; }

echo "File copied successfully."
```

Ex20: This script creates a backup directory named with the current date and copies all .txt files into it.

```
#!/bin/bash

backup_dir="backup_$(date +%Y%m%d)"

mkdir -p "$backup_dir"

cp *.txt "$backup_dir"

echo "Backup completed to $backup_dir"
```

Ex21: Downloads a file using curl and checks for success or failure.

```
#!/bin/bash

url="http://example.com/file.txt"

if curl -O "$url"; then
 echo "Download successful."

else
 echo "Download failed."

fi
```

Ex22: Checks the HTTP status code of a website using curl.

```
#!/bin/bash
response=$(curl -s -o /dev/null -w "%{http_code}" http://example.com)
if [ "$response" -eq 200 ]; then
 echo "Website is up!"
else
 echo "Website is down! Status code: $response"
fi
```

Ex23: Checks if a file is readable and prints an appropriate message.

```
#!/bin/bash
file="my_script.sh"
if [ -r "$file" ]; then
 echo "$file is readable."
else
 echo "$file is not readable."
fi
```

Alpine Docker with Bash:

Make a Dockerfile and paste:

```
# Use the latest Alpine image
FROM alpine:latest
# Install Bash
RUN apk add --no-cache bash
# Set the default command to start Bash
CMD ["/bin/bash"]
```

Save and build: docker build -t alpine-bash .

Now run. It will directly enter Bash: docker run -it alpine-bash

Exercises:

• Make a scripting calculator (basic operations)

• Validate user input (keep on asking until the right argument is entered)

• Take a text file as input and count the total occurrences of a keyword (use grep)

• Output Fibonacci up to a certain number (argument)

• A script that outputs basic system information: uname, uptime, df etc.

• Webscraper – get URL through curl and then grep, sed or awk (more focused on programming language)

• CSV to JSON converter :)

Ubuntu Exercises:

Log Analyzer

Write a Bash script that:

- Reads /var/log/syslog.
- Extracts all unique services that generated error messages.
- Counts how many times each service reported an error.
- Stores the top 5 most error-prone services into a CSV file (errors.csv) with the format: *Service,Occurrences*

Custom Backup System

Write a script that:

- Takes a directory name as input.
- Compresses it into a .tar.gz file with today's date in the filename.
- Before compressing, it must skip files larger than 50 MB and log their names into skipped_files.txt.
- At the end, it should print the total size of the backup.

User Management with Constraints:

Write a script that:

- Reads usernames from a text file (users.txt).
- Creates each user with:
    - A home directory.
    - A default password (ChangeMe123).
- Then, disables login for users whose names start with "temp".

Process Monitor

Write a script that:

- Monitors processes every 5 seconds.
- If any process uses more than 30% CPU or 500 MB memory, log its PID, name, CPU%, and memory% into alerts.log.
- If the same process is logged 3 times consecutively, automatically kill it.

Parallel Download Manager

Write a script that:

- Reads a list of URLs from urls.txt.
- Downloads them in parallel (max 3 at a time).
- Logs failed downloads separately.

Customized ls command:

Write a script named my_ls.sh that:

- Prints files in a directory with:
    - File name
    - Size in KB
    - Owner
    - Last modified time
- And sorts it by size (descending).