Introduction to Linux: **Shell Basics**

Philipp Fruck & Pius Walter

Shell Basics

Goal of this lecture

- Introduction to shell scripting

Why?

- Automate repetitive tasks
- Improve/speedup personal workflow

Use Cases:

- Backup scripts
- Deployment automation
- Log parsing and monitoring
- System maintenance tasks

Useful (scripting) tools

Tool	Description	Example Command
`tr`	Translate or delete characters	`echo $PATH | tr : '\n'`
`sed`	Manipulate text (in place)	`sed -i 's/old/new/g' file.txt`
`cut` / `awk`	Text selection, e.g. print first word	`cut -d ' ' -f 1, awk '{print $1}'`
`basename`	Extracts the file name from a path.	`basename /path/to/file.txt`
`dirname`	Extracts the directory part of a file path.	`dirname /path/to/file.txt`
`sort` / `uniq`	Sorts lines and removes duplicate lines.	`sort file.txt | uniq`
`base64`	Encodes and decodes data in Base64 format.	`echo "text" | base64`
`jq`	Parse and format JSON data.	`cat file.json | jq .`
`yq`	Parses and formats YAML data.	`cat file.yml | yq .`
`date`	Show/set system date and time.	`date -d @1730502000`
`time`	Measure execution time of a command	`time ls -l`
`sleep`	Wait for `n` seconds	`sleep 10`
`diff/delta`	Show difference between two files	`delta first.txt second.txt`

If no jq is installed: `cat file.json | python3 -m json.tool` might help

`uniq` only works with sorted input, always use `<command> | sort | uniq` to avoid mistakes

Exit Codes

On Linux, every command returns an **exit code** (`$?`) indicating success or failure.

An exit code is always in the range `0-255`.

Only 0 indicates success, every other exit code indicates a failure.

```
true
echo Output of true: $?

false
echo Output of false: $?
```

```
ls /tmp
echo "Exit code of ls: $?"

grep "foo" file.txt
echo "Exit code of grep: $?"
```

```
──────────────────── [finished] ────────────────────
```

```
Output of true: 0
Output of false: 1
```

> The exit command can be used to quit an interactive shell

> See man errno or man sysexits.h for common C exit codes

Concatenating Commands

The POSIX standard allows chaining commands in multiple ways:

```
# Sequential execution: commands runs regardless of
result
false; echo Hello 1
true; echo Hello 2
# Conditional execution: run next only if previous
succeeds
true && echo Hello 3
# Conditional execution: run next only if previous
fails
false || echo Hello 4
# Run commands in sub-group
true && (false; echo Hello 5)
```

```
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
```

[finished]

```
# Command substition: Get line count of all slides
wc -l $(find . -type f)
# legacy substitution syntax, just so you know
echo "Running in `pwd`"
```

```
   255 ./01-core-commands.md
   636 ./08-services-and-remote-access.md
   526
./09-storage-and-system-security.md
   938 ./10-markup-languages.md
   255 ./02-system-tools.md
   311 ./03-shell-basics.md
   491 ./00-overview.md
   362 ./06-basic-permissions.md
   370 ./07-advanced-permissions.md
   415 ./04-posix-and-bash-syntax.md
   471 ./05-scripting.md
  5030 total
Running in
/home/runner/work/lecture-introduction-to
-linux/lecture-introduction-to-linux/slid
es
```

[finished]

File Descriptors

Manage input/output streams

```
# Redirect stdout to a file (overwrite content)
echo "Hello" > output.txt

# Append stdout to a file
echo "Hello 2" >> output.txt

# Redirect stderr to hide messages
ls non_existing_file 2> /dev/null

# Redirect both stdout and stderr
command &> combined.txt

# Create custom FD and write to it
exec 3> output.txt
echo "Hello, World!" >&3
exec 3>&- # Close the FD

# Bash feature: Use command output as file input
diff <(ls first) <(ls second)
```

File descriptor IDs

- 0 → stdin
- 1 → stdout
- 2 → stderr
- Custom FDs start at 3.

Writing to a file without an editor

```
cat > newfile.txt <<EOF
this
is
the
content
of my
textfile
EOF
```

Instead of <<EOF, you can use cat > newfile.txt and hit ^C to stop writing to the file

Pipes

The output of one command can be piped into another command to work with it (instead of a file):

```
ls / | grep bin
```

———————— [finished] ————————

```
bin
sbin
```

Pipes only capture stdin:

```
ls /root/ | grep --only Perm
```

———————— [finished with error] ————————

```
ls: cannot open directory '/root/':
Permission denied
```

File descriptors to the rescue!

```
ls /root/ 2>&1 | grep --only Perm
```

———————— [finished] ————————

```
Perm
```

Infinite piping possible!

```
curl https://api.github.com/users/p-fruck/keys 2
>/dev/null | jq .[0].key | tr -d '"' | ssh-keygen -lf
-
```

Note: -f - means --file: stdin and reads the input from the pipe

```
# spying on our students on the remote system
last | grep '^s26' | cut -d ' ' -f 1 | sort | uniq |
wc -l
```

Globbing

Globbing (Filename Expansion)

Globbing lets you match filenames using wildcards:

Pattern	Matches
`*`	Any string, including empty
`?`	Any single character
`[abc]`	Any one of the listed characters
`[!abc]`	Any character not listed

Example:

```
# List all .txt files
ls *.txt

# List files starting with a or b
ls [ab]*.txt
```

Persisting Configuration

How it works

- Previous settings like `alias` are set on per-session basis
- All shells support configuring persistent settings using configs
- On Bash: `~/.bash_profile` and `~/.bashrc` are used
  - They load further files from `~/.bashrc.d/*`
  - Other shells like ZSH provide e.g. `~/.zshrc`
- System-wide config is provided via `/etc/profile`, `/etc/bashrc` and `/etc/profile.d/*.sh`
- Config files are just shell scripts
- You can load them manually by running `source script.sh` (Bash) or `. script.sh` (POSIX)

Differences

Feature	`~/.bash_profile` & `/etc/profile`	`~/.bashrc` & `/etc/bashrc`
Type of Shell	Login shells	Non-login interactive shells
When executed	At login (e.g., SSH, virtual terminal)	For every new interactive terminal
Common Use	Environment variables, session setup	Aliases, functions, interactive settings

Thank you!

Thank you for your attention!

Don't forget the feedback in Moodle please!