Introduction to Linux: **POSIX and Bash Syntax**

Philipp Fruck & Pius Walter

POSIX vs Bash

Goal of this lecture

- Learn core Bash scripting syntax
- Understand differences between POSIX and Bash-specific features

Why?

- Scripts need to run reliably across different shells
- Core syntax knowledge is the foundation for writing scripts

Use Cases:

- Portable automation scripts
- Configuration and setup scripts
- Writing reusable shell functions

POSIX vs Bash

POSIX is a standard, Bash is a shell implementation

- `/bin/sh` -> POSIX-compliant shell
- `/bin/bash` -> Bash-specific features, not always POSIX

POSIX vs Bash

POSIX is a standard, Bash is a shell implementation

- `/bin/sh` -> POSIX-compliant shell
- `/bin/bash` -> Bash-specific features, not always POSIX

> On Fedora, /bin/sh is a symlink to /bin/bash, but other distros like Debian ship more minimal shells like dash

POSIX vs Bash

POSIX is a standard, Bash is a shell implementation

- `/bin/sh` -> POSIX-compliant shell
- `/bin/bash` -> Bash-specific features, not always POSIX

> On Fedora, /bin/sh is a symlink to /bin/bash, but other distros like Debian ship more minimal shells like dash

POSIX standard includes:

- Standard shell syntax (`if`, `while`, `for`, `case`)
- Conditionals (`[ ... ]`, with `=`, `-f`, `-n`, etc.)
- Variables and expansions
- Redirection
- Command substitution

POSIX vs Bash

POSIX is a standard, Bash is a shell implementation

- `/bin/sh` -> POSIX-compliant shell
- `/bin/bash` -> Bash-specific features, not always POSIX

> On Fedora, /bin/sh is a symlink to /bin/bash, but other distros like Debian ship more minimal shells like dash

POSIX standard includes:

- Standard shell syntax (`if`, `while`, `for`, `case`)
- Conditionals (`[ ... ]`, with `=`, `-f`, `-n`, etc.)
- Variables and expansions
- Redirection
- Command substitution

Bash adds arrays, enhanced conditionals (`[[ ... ]]`) and some more syntax.

POSIX vs Bash

POSIX is a standard, Bash is a shell implementation

- `/bin/sh` -> POSIX-compliant shell
- `/bin/bash` -> Bash-specific features, not always POSIX

> On Fedora, /bin/sh is a symlink to /bin/bash, but other distros like Debian ship more minimal shells like dash

POSIX standard includes:

- Standard shell syntax (`if`, `while`, `for`, `case`)
- Conditionals (`[ ... ]`, with `=`, `-f`, `-n`, etc.)
- Variables and expansions
- Redirection
- Command substitution

Bash adds arrays, enhanced conditionals (`[[ ... ]]`) and some more syntax.

> The shellcheck utility allows linting your shell script.
> It determines the correct shell syntax using the **shebang**.

Shebang

How-to

- The **shebang** line tells the system *which interpreter* should execute the script.
- Must appear as the first line of the script
- Starts with `#!` followed by the interpreter path
- Allows running `./script` instead of `bash ./script`
  - Ensures correct interpreter is used even if no file extension is present
  - Don't forget to `chmod +x ./script` (details in the permissions lecture)

Examples

```
#!/bin/sh
```

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

```
#!/usr/bin/env bash
```

```
#!/usr/bin/env python3
```

Shebang

How-to

- The **shebang** line tells the system *which interpreter* should execute the script.
- Must appear as the first line of the script
- Starts with `#!` followed by the interpreter path
- Allows running `./script` instead of `bash ./script`
  - Ensures correct interpreter is used even if no file extension is present
  - Don't forget to `chmod +x ./script` (details in the permissions lecture)

Examples

```
#!/bin/sh
```

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

```
#!/usr/bin/env bash
```

```
#!/usr/bin/env python3
```

Tip: Using /usr/bin/env makes your script more portable across systems (/bin/sh is a special case as it is required by the POSIX standard). Use of #!/bin/bash, #!/usr/bin/python etc. is discouraged.

More advanced: Watch the first minutes of https://media.ccc.de/v/all-systems-go-2026-440-binfmtbpf (especially p. 13-14)

POSIX Variables

```
1 # Assign a variable. Use quotes to include spaces
2 name=Alice
3 fullname="Bob the builder"
4
5 # Use a variable with $ or use brackets to define
  the start and end of variables
6 echo "Hello $name"
7 echo "${name}.lastname@example.com"
8
9 # Read-only
10 readonly pi=3.14
11
12 # Per command vs global environment variables
13 foo=bar mylocalscript
14 export PATH="/usr/local/bin:$PATH"
```

Note: Every variable is a string

Be careful with special variables like PATH and PS1

POSIX Variables

```
# Assign a variable. Use quotes to include spaces
name=Alice
fullname="Bob the builder"

# Use a variable with $ or use brackets to define the start and end of variables
echo "Hello $name"
echo "${name}.lastname@example.com"

# Read-only
readonly pi=3.14

# Per command vs global environment variables
foo=bar mylocalscript
export PATH="/usr/local/bin:$PATH"
```

Note: Every variable is a string

Be careful with special variables like PATH and PS1

```
readonly name=Bob
fullname="$name the builder"
# Variables are replaced in double quotes
echo "Fullname: $fullname"
# But not in single quotes:
message='$name is a nice name!'
echo $message
# readonly cannot be changed
name="Beep   "
```

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

```
Fullname: Bob the builder
$name is a nice name!
/home/runner/work/_temp/nix-shell.0FChq9/.presentermvcAilm/script.sh: 9: name: is read only
```

POSIX Variables

```bash
# Assign a variable. Use quotes to include spaces
name=Alice
fullname="Bob the builder"

# Use a variable with $ or use brackets to define the start and end of variables
echo "Hello $name"
echo "${name}.lastname@example.com"

# Read-only
readonly pi=3.14

# Per command vs global environment variables
foo=bar mylocalscript
export PATH="/usr/local/bin:$PATH"
```

Note: Every variable is a string

Be careful with special variables like PATH and PS1

```bash
readonly name=Bob
fullname="$name the builder"
# Variables are replaced in double quotes
echo "Fullname: $fullname"
# But not in single quotes:
message='$name is a nice name!'
echo $message
# readonly cannot be changed
name="Beep   "
```

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

```
Fullname: Bob the builder
$name is a nice name!
/home/runner/work/_temp/nix-shell.0FChq9/
.presentermvcAilm/script.sh: 9: name: is
read only
```

Variable scoping: local variables inside functions vs global outside.

g export allows variables to be used by other processes launched from the current session.

POSIX Conditionals

If-else logic

The `if` statement runs commands conditionally based on exit status. Do not forget spaces around `[` and `]`.

```
# hint: Do not use this in prod :)
if [ "$USER" = "root" ]; then
    echo "Welcome, root user!"
elif [ "$USER" = "admin" ]; then
    echo "Hello, admin."
else
    echo "Access denied."
fi

# negation works with != or ! in beginning
if [ ! "$USER" = "root" ]; then echo "No!"; fi
if ! grep MyText myfile.txt; then echo Nope; fi
```

POSIX Conditionals

If-else logic

The `if` statement runs commands conditionally based on exit status. Do not forget spaces around `[` and `]`.

```
# hint: Do not use this in prod :)
if [ "$USER" = "root" ]; then
    echo "Welcome, root user!"
elif [ "$USER" = "admin" ]; then
    echo "Hello, admin."
else
    echo "Access denied."
fi

# negation works with != or ! in beginning
if [ ! "$USER" = "root" ]; then echo "No!"; fi
if ! grep MyText myfile.txt; then echo Nope; fi
```

Concatenation (`if ...; then` is optional)

```
[ "$USER" = "root" ] || [ "$USER" = "admin" ] && echo
"Woohoo"
```

POSIX Conditionals

If-else logic

The `if` statement runs commands conditionally based on exit status. Do not forget spaces around `[` and `]`.

```
# hint: Do not use this in prod :)
if [ "$USER" = "root" ]; then
    echo "Welcome, root user!"
elif [ "$USER" = "admin" ]; then
    echo "Hello, admin."
else
    echo "Access denied."
fi

# negation works with != or ! in beginning
if [ ! "$USER" = "root" ]; then echo "No!"; fi
if ! grep MyText myfile.txt; then echo Nope; fi
```

Concatenation (`if ...; then` is optional)

```
[ "$USER" = "root" ] || [ "$USER" = "admin" ] && echo "Woohoo"
```

Common test conditions

Operator	Meaning	Example
`-e`	Path exists	`[ -e "$file" ]`
`-f`	Regular file exists	`[ -f "$file" ]`
`-d`	Directory exists	`[ -d "$dir" ]`
`-r`	Path is readable	`[ -r "$file" ]`
`-w`	Path is writable	`[ -w "$file" ]`
`-x`	Path is executable/searchable	`[ -x "$file" ]`
`-s`	File exists and is non-empty	`[ -s "$file" ]`
`-L`	Path is a symbolic link	`[ -L "$file" ]`
`-z`	String is empty	`[ -z "$var" ]`
`-n`	String is non-empty	`[ -n "$var" ]`
`=`	Strings are equal	`[ "$a" = "$b" ]`
`!=`	Strings are different	`[ "$a" != "$b" ]`

Bash-Specific Conditionals

Safer comparison

Bash's `[[ ... ]]` supports safer string comparisons (`==`) that prevent word splitting or path name expansion and logical operators.

```
if [[ "$USER" == "root" ]]; then
    echo "Welcome, root user!"
elif [[ "$USER" == admin* ]]; then
    echo "Hello, admin."
else
    echo "Access denied."
fi

[[ "$USER" != "admin" ]] && echo "no!"
```

Bash-Specific Conditionals

Safer comparison

Bash's `[[ ... ]]` supports safer string comparisons (`==`) that prevent word splitting or path name expansion and logical operators.

```
if [[ "$USER" == "root" ]]; then
    echo "Welcome, root user!"
elif [[ "$USER" == admin* ]]; then
    echo "Hello, admin."
else
    echo "Access denied."
fi

[[ "$USER" != "admin" ]] && echo "no!"
```

Combine tests using `&&` and `||`:

```
if [[ -f "$file" && -r "$file" ]]; then
    echo "File exists and is readable."
fi
```

Bash-Specific Conditionals

Safer comparison

Bash's `[[ ... ]]` supports safer string comparisons (`==`) that prevent word splitting or path name expansion and logical operators.

```
if [[ "$USER" == "root" ]]; then
    echo "Welcome, root user!"
elif [[ "$USER" == admin* ]]; then
    echo "Hello, admin."
else
    echo "Access denied."
fi

[[ "$USER" != "admin" ]] && echo "no!"
```

Combine tests using `&&` and `||`:

```
if [[ -f "$file" && -r "$file" ]]; then
    echo "File exists and is readable."
fi
```

Regex matching

Use `=~` to match a POSIX extended regular expression (`man 7 regex`).

`BASH_REMATCH` is an array: `[0]` holds the whole match, `[1]`, `[2]`, ... the capture groups.

```
input=abc123
if [[ "$input" =~ [0-9]+ ]]; then
    echo "Found number: $BASH_REMATCH"
else
    echo "Invalid input."
fi
```

———————— [finished] ————————

```
Found number: 123
```

Arithmetic

Bash

Bash supports `(( ... ))` for arithmetic comparisons.

A nonzero arithmetic result means success.

```
count="50"
if (( count > 10 && count < 100 )); then
    echo "Count is in range."
fi

if (( count % 2 == 0 )); then
    echo "Even number."
fi

((count++))
echo Count: $count

(( "1" > 2 )) || echo false
```

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

```
Count is in range.
Even number.
Count: 51
false
```

Arithmetic

Bash

Bash supports `(( ... ))` for arithmetic comparisons.

A nonzero arithmetic result means success.

```
count="50"
if (( count > 10 && count < 100 )); then
    echo "Count is in range."
fi

if (( count % 2 == 0 )); then
    echo "Even number."
fi

((count++))
echo Count: $count

(( "1" > 2 )) || echo false
```

[finished]

```
Count is in range.
Even number.
Count: 51
false
```

POSIX

POSIX only supports arithmetic via `$(( ... ))`.

Results are passed as string (no exit code).

```
# in POSIX, numbers must not be quoted!
echo $(( 1 + 2 ))
echo -n $(( 1 > 2 )) && echo '->yes'
echo -n $(( 1 < 2 )) && echo '->yes'

counter=0
# POSIX-compliant increment
counter=$(( counter + 1 ))
echo Counter: $counter
```

[finished]

```
3
0->yes
1->yes
Counter: 1
```

Selections

POSIX Case statement

`case` compares a value against multiple patterns

```
case "$arg" in
    start)
        echo "Starting service..."
        ;;
    stop)
        echo "Stopping service..."
        ;;
    restart|reload)
        echo "Restarting service..."
        ;;
    *)
        echo "Usage: $0 {start|stop|restart}"
        exit 1
        ;;
esac
```

Selections

POSIX Case statement

`case` compares a value against multiple patterns

```
case "$arg" in
    start)
        echo "Starting service..."
        ;;
    stop)
        echo "Stopping service..."
        ;;
    restart|reload)
        echo "Restarting service..."
        ;;
    *)
        echo "Usage: $0 {start|stop|restart}"
        exit 1
        ;;
esac
```

Bash Select

```
select choice in "foo" "bar" "foobar"; do
    echo "choice: $choice, reply: $REPLY"
    break
done
```

Selections

POSIX Case statement

`case` compares a value against multiple patterns

```
case "$arg" in
    start)
        echo "Starting service..."
        ;;
    stop)
        echo "Stopping service..."
        ;;
    restart|reload)
        echo "Restarting service..."
        ;;
    *)
        echo "Usage: $0 {start|stop|restart}"
        exit 1
        ;;
esac
```

Bash Select

```
select choice in "foo" "bar" "foobar"; do
    echo "choice: $choice, reply: $REPLY"
    break
done
```

Combining select and case

```
PS3="Choose an option: "
select choice in "Date" "List" "Quit"; do
    case $REPLY in
        1) date ;;
        2) ls -l ;;
        3) echo "Goodbye!"; break ;;
        *) echo "Invalid choice." ;;
    esac
done
```

select uses PS3 instead of PS1.

Thank you!

Thank you for your attention!

Don't forget the feedback in Moodle please!