Introduction to Linux: **Scripting**

Philipp Fruck & Pius Walter

Scripting

Goal of this lecture

- Learn practical scripting pitfalls and best practices
- Get familiar with editors and process/job control

Why?

- Real-world scripts must handle errors and edge cases safely
- Editors and job control are daily tools for any Linux user

Use Cases:

- Writing robust, production-ready scripts
- Managing long-running or background tasks
- Editing files directly on remote servers

Functions in POSIX / Bash

Functions allow reusable blocks of code

- Functions improve code clarity and reduce repetition.
- Parameters passed to a function or script are accessed via `$1`, `$2`, ...
  - `$0` is the invoked executable (remember C's `argv[0]`?)
  - `$#` shows the parameter count
  - `$@` shows all parameters (starting at `$1`)
  - `shift` drops the first n parameters (by default 1)

```
#!/bin/sh
myfunc() {
    echo "Hello from $0"
    echo "$# args: $@"
    echo "First: $1" ; shift ; echo "New first: $1"
}
myfunc foo bar
```

———————————— [finished] ————————————

```
Hello from
/home/runner/work/_temp/nix-shell.0FChq9/.presentermS
d3o94/script.sh
2 args: foo bar
First: foo
New first: bar
```

Functions in POSIX / Bash

Functions allow reusable blocks of code

- Functions improve code clarity and reduce repetition.
- Parameters passed to a function or script are accessed via `$1`, `$2`, ...
  - `$0` is the invoked executable (remember C's `argv[0]`?)
  - `$#` shows the parameter count
  - `$@` shows all parameters (starting at `$1`)
  - `shift` drops the first n parameters (by default 1)

```
#!/bin/sh
myfunc() {
    echo "Hello from $0"
    echo "$# args: $@"
    echo "First: $1" ; shift ; echo "New first: $1"
}
myfunc foo bar
```

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

```
Hello from
/home/runner/work/_temp/nix-shell.0FChq9/.presentermS
d3o94/script.sh
2 args: foo bar
First: foo
New first: bar
```

Function return values

```
 1 #!/usr/bin/env bash
 2
 3 # the function keyword only exists in
   bash
 4 function add() {
 5     echo $(( $1 + $2 ))
 6 }
 7
 8 # Functions can return plaintext
 9 sum=$(add 3 5)
10 echo "Sum = $sum"
11
12 # And functions can return status
   codes
13 check_file() {
14     [ -f "$1" ] # same as return
   0/1/255
15 }
16
17 check_file ../README.md && echo
   Exists!
```

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

```
Sum = 8
Exists!
```

Asking for Input

Besides parameters, scripts can also ask the user questions at runtime:

```
#!/bin/sh
read -r -p "Your name: " name
echo "Hello, $name!"
```

Asking for Input

Besides parameters, scripts can also ask the user questions at runtime:

```
#!/bin/sh
read -r -p "Your name: " name
echo "Hello, $name!"
```

`read` stores a line of input in a variable:

- `-p` : show a **p**rompt
- `-r` : **r**aw, do not interpret backslash as an escape character
- `-s` : **s**ilent, do not echo the input (e.g., for passwords)

For Loop

POSIX for loop

```
# Iterate over numbers
for i in 0 1 2 3 4 5 6 7 8 9 10; do
    echo -n "$i "
done
echo
# Loop over files (pipe output into head)
for file in *.md; do
    echo "File: $file"
done | head -n 5
```

———————————— [finished] ————————————

```
0 1 2 3 4 5 6 7 8 9 10
File: 00-overview.md
File: 01-core-commands.md
File: 02-system-tools.md
File: 03-shell-basics.md
File: 04-posix-and-bash-syntax.md
```

For Loop

POSIX for loop

```
# Iterate over numbers
for i in 0 1 2 3 4 5 6 7 8 9 10; do
    echo -n "$i "
done
echo
# Loop over files (pipe output into head)
for file in *.md; do
    echo "File: $file"
done | head -n 5
```

———————————— [finished] ————————————

```
0 1 2 3 4 5 6 7 8 9 10
File: 00-overview.md
File: 01-core-commands.md
File: 02-system-tools.md
File: 03-shell-basics.md
File: 04-posix-and-bash-syntax.md
```

Bash set notation

While not strictly related to for loops `bash` also supports set notation:

```
echo {1..10}
echo {A..C}{1..3}
for i in {1..10}; do echo -n $i,; done
```

———————————— [finished] ————————————

```
1 2 3 4 5 6 7 8 9 10
A1 A2 A3 B1 B2 B3 C1 C2 C3
1,2,3,4,5,6,7,8,9,10,
```

For Loop

POSIX for loop

```
# Iterate over numbers
for i in 0 1 2 3 4 5 6 7 8 9 10; do
    echo -n "$i "
done
echo
# Loop over files (pipe output into head)
for file in *.md; do
    echo "File: $file"
done | head -n 5
```

———————————— [finished] ————————————

```
0 1 2 3 4 5 6 7 8 9 10
File: 00-overview.md
File: 01-core-commands.md
File: 02-system-tools.md
File: 03-shell-basics.md
File: 04-posix-and-bash-syntax.md
```

Bash set notation

While not strictly related to for loops `bash` also supports set notation:

```
echo {1..10}
echo {A..C}{1..3}
for i in {1..10}; do echo -n $i,; done
```

———————————— [finished] ————————————

```
1 2 3 4 5 6 7 8 9 10
A1 A2 A3 B1 B2 B3 C1 C2 C3
1,2,3,4,5,6,7,8,9,10,
```

While Loop

POSIX compliant

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

[finished]

```
Count 1
Count 2
Count 3
Count 4
Count 5
```

Bash specific

```sh
#!/usr/bin/env bash
counter=1
while (( counter <= 5 )); do
    echo "Count $counter"
    ((counter++))
done
```

[finished]

```
Count 1
Count 2
Count 3
Count 4
Count 5
```

Word Splitting

```
files="file1.txt file2.txt file3.txt"

# Unquoted: splits into words
for f in $files; do
    echo "File: $f"
done
```

——————————— [finished] ———————————

```
File: file1.txt
File: file2.txt
File: file3.txt
```

Word Splitting

```bash
files="file1.txt file2.txt file3.txt"

# Unquoted: splits into words
for f in $files; do
    echo "File: $f"
done
```

——————————— [finished] ———————————

```
File: file1.txt
File: file2.txt
File: file3.txt
```

```bash
files="file1.txt file2.txt file3.txt"

# Quoted: treated as single string
for f in "$files"; do
    echo "File: $f"
done
```

——————————— [finished] ———————————

```
File: file1.txt file2.txt file3.txt
```

Word Splitting

```bash
files="file1.txt file2.txt file3.txt"

# Unquoted: splits into words
for f in $files; do
    echo "File: $f"
done
```

——————————— [finished] ———————————

```
File: file1.txt
File: file2.txt
File: file3.txt
```

```bash
files="file1.txt file2.txt file3.txt"

# Quoted: treated as single string
for f in "$files"; do
    echo "File: $f"
done
```

——————————— [finished] ———————————

```
File: file1.txt file2.txt file3.txt
```

Again, shellcheck to the rescue :)

Playing with Fire: Strings as Code

`eval` takes a string and executes it as a shell command:

```
cmd="echo Hello from eval"
eval "$cmd"
```

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

```
Hello from eval
```

Playing with Fire: Strings as Code

`eval` takes a string and executes it as a shell command:

```
cmd="echo Hello from eval"
eval "$cmd"
```

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

```
Hello from eval
```

Imagine a script that asks for an IP address and pings it.

```
# Intended use
ip="127.0.0.1"
eval "ping -c 1 -q $ip" | tail -3
```

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

```
--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss,
time 0ms
rtt min/avg/max/mdev = 0.044/0.044/0.
```

Playing with Fire: Strings as Code

`eval` takes a string and executes it as a shell command:

```
cmd="echo Hello from eval"
eval "$cmd"
```

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

```
Hello from eval
```

Imagine a script that asks for an IP address and pings it.

```
# Intended use
ip="127.0.0.1"
eval "ping -c 1 -q $ip" | tail -3
```

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

```
--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss,
time 0ms
rtt min/avg/max/mdev = 0.044/0.044/0.
```

Everything in the string is interpreted, including things you did not intend:

```
# "Creative" user input
ip="127.0.0.1; whoami; hostname"
eval "ping -c 1 -q $ip" | tail -3
```

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

```
rtt min/avg/max/mdev =
0.047/0.047/0.047/0.000 ms
runner
runnervm0hi8g
```

Playing with Fire: Strings as Code

`eval` takes a string and executes it as a shell command:

```
cmd="echo Hello from eval"
eval "$cmd"
```

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

```
Hello from eval
```

Imagine a script that asks for an IP address and pings it.

```
# Intended use
ip="127.0.0.1"
eval "ping -c 1 -q $ip" | tail -3
```

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

```
--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss,
time 0ms
rtt min/avg/max/mdev = 0.044/0.044/0.
```

Everything in the string is interpreted, including things you did not intend:

```
# "Creative" user input
ip="127.0.0.1; whoami; hostname"
eval "ping -c 1 -q $ip" | tail -3
```

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

```
rtt min/avg/max/mdev =
0.047/0.047/0.047/0.000 ms
runner
runnervm0hi8g
```

eval is evil!

- Never use `eval` with user input or any data you do not fully control
- In most cases there is a safer alternative (arrays, functions, `"$@"`, ...)

Error Handling

Shell scripts often need strict error handling:

```
#!/usr/bin/env bash
set -xeuo pipefail
```

This ensures predictable, safe scripts and easier debugging.

- `-x` : optional: show commands as e**x**ecuted, great for pipelines, debugging, ...
  - Uses `PS4` environment variable as prompt
- `-e` : **e**xit immediately on error
- `-u` : error on **u**nset variables
- `-o pipefail` : **pipe**line **fail**s if any command in the chain fails
  - previously existed only in Bash
  - added in POSIX.1-2024 [1]

Error Handling

Shell scripts often need strict error handling:

```
#!/usr/bin/env bash
set -xeuo pipefail
```

This ensures predictable, safe scripts and easier debugging.

- `-x` : optional: show commands as **e**xecuted, great for pipelines, debugging, ...
  - Uses `PS4` environment variable as prompt
- `-e` : **e**xit immediately on error
- `-u` : error on **u**nset variables
- `-o pipefail` : **pipe**line **fail**s if any command in the chain fails
  - previously existed only in Bash
  - added in POSIX.1-2024 [1]

> This could have saved Kyoto University multiple terabytes of valuable research data:
> https://www.youtube.com/watch?v=Nkm8BuMc4sQ
> 6:05 shows why the difference between mv and cp can be very important!

Temporary Files

Scripts often need a place for intermediate results.
`mktemp` creates one safely:

```
tmp=$(mktemp)
echo "scratch data" > "$tmp"
echo "Created $tmp"
cat "$tmp"
rm "$tmp"
```

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

```
Created
/home/runner/work/_temp/nix-shell.0FChq9/tmp.XqLOcXmr
jN
scratch data
```

Temporary Files

Scripts often need a place for intermediate results.
`mktemp` creates one safely:

```
tmp=$(mktemp)
echo "scratch data" > "$tmp"
echo "Created $tmp"
cat "$tmp"
rm "$tmp"
```

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

```
Created
/home/runner/work/_temp/nix-shell.0FChq9/tmp.XqLOcXmr
jN
scratch data
```

- Creates a file with a unique, unpredictable name (default in `/tmp`)
- `mktemp -d` creates a temporary **d**irectory
- Never hardcode names like `/tmp/myscript.tmp`: collisions and security issues
- Remember to clean up afterwards, more on that in a moment

Processes and Job Control

Key Commands

Command	Description	Example Usage
`&`	Run a command in the background	`./long_running_command &`
`Ctrl+Z`	Pause (suspend) a running process	Press `Ctrl+Z` while running a command
`bg`	Move a stopped job to the background	`bg %1` (background job 1)
`fg`	Bring a background job to the foreground	`fg %3` (foreground job 3)
`jobs`	List all background jobs	`jobs`

Processes and Job Control

Key Commands

Command	Description	Example Usage
`&`	Run a command in the background	`./long_running_command &`
`Ctrl+Z`	Pause (suspend) a running process	Press `Ctrl+Z` while running a command
`bg`	Move a stopped job to the background	`bg %1` (background job 1)
`fg`	Bring a background job to the foreground	`fg %3` (foreground job 3)
`jobs`	List all background jobs	`jobs`

Examples

```
# Run a command in the background
$ long_running_task &
[1] 12345  # Job 1, PID 12345

# Suspend the process with Ctrl+Z
$ ^Z
[2]+ Stopped                 long_running_task

# list background jobs
$ jobs
[1]-  Running                 long_running_task
[2]+  Stopped                 long_running_task
```

```
# Move the suspended process to the background
$ bg
[1]+ 12345 long_running_task &

# Bring the process to the foreground
$ fg
long_running_task

# Bring the second background process in the
foreground
$ fg %3
other_task
```

Signals

Signals are small messages the kernel (or other processes) send to a process.

When you press `Ctrl+C` or `Ctrl+Z`, the terminal sends a signal to the running process.

Signal	Nr.	Typical trigger / use	Default action
`SIGHUP`	1	Terminal was closed	Terminate
`SIGINT`	2	`Ctrl+C`	Terminate
`SIGKILL`	9	`kill -9 PID`	Terminate, cannot be caught
`SIGTERM`	15	`kill PID` (default), polite "please quit"	Terminate
`SIGCONT`	18	`bg` / `fg`	Continue a stopped process
`SIGTSTP`	20	`Ctrl+Z` (used by job control)	Stop (suspend)

Signals

Signals are small messages the kernel (or other processes) send to a process.

When you press `Ctrl+C` or `Ctrl+Z`, the terminal sends a signal to the running process.

Signal	Nr.	Typical trigger / use	Default action
SIGHUP	1	Terminal was closed	Terminate
SIGINT	2	Ctrl+C	Terminate
SIGKILL	9	kill -9 PID	Terminate, cannot be caught
SIGTERM	15	kill PID (default), polite "please quit"	Terminate
SIGCONT	18	bg / fg	Continue a stopped process
SIGTSTP	20	Ctrl+Z (used by job control)	Stop (suspend)

`kill` lets you send signals to processes via their process ID (PID):

```
$ kill -l            # list all signals
$ kill -TERM 12345   # ask process 12345 to terminate
$ kill -9 12345      # force kill, process gets no chance to clean up
```

Signals

Signals are small messages the kernel (or other processes) send to a process.

When you press `Ctrl+C` or `Ctrl+Z`, the terminal sends a signal to the running process.

Signal	Nr.	Typical trigger / use	Default action
SIGHUP	1	Terminal was closed	Terminate
SIGINT	2	Ctrl+C	Terminate
SIGKILL	9	kill -9 PID	Terminate, cannot be caught
SIGTERM	15	kill PID (default), polite "please quit"	Terminate
SIGCONT	18	bg / fg	Continue a stopped process
SIGTSTP	20	Ctrl+Z (used by job control)	Stop (suspend)

`kill` lets you send signals to processes via their process ID (PID):

```
$ kill -l            # list all signals
$ kill -TERM 12345   # ask process 12345 to terminate
$ kill -9 12345      # force kill, process gets no chance to clean up
```

`pkill` lets you kill processes by name instead of PID: `pkill -9 firefox`

Cleaning Up on Exit

`trap` runs a command when the shell receives a signal, so a script can react to e.g., `Ctrl+C`:

```
#!/bin/sh
tmp=$(mktemp)

# EXIT: special pseudo-signal, runs whenever the
script ends
trap 'echo "Cleaning up $tmp"; rm -f "$tmp"' EXIT
# INT: Ctrl+C
trap 'echo "Interrupted!"; exit 130' INT

echo "working..." > "$tmp"
cat "$tmp"
```

———————————— [finished] ————————————

```
working...
Cleaning up
/home/runner/work/_temp/nix-shell.0FChq9/tmp.I6PRHF66
aH
```

Cleaning Up on Exit

`trap` runs a command when the shell receives a signal, so a script can react to e.g., `Ctrl+C`:

```
#!/bin/sh
tmp=$(mktemp)

# EXIT: special pseudo-signal, runs whenever the
script ends
trap 'echo "Cleaning up $tmp"; rm -f "$tmp"' EXIT
# INT: Ctrl+C
trap 'echo "Interrupted!"; exit 130' INT

echo "working..." > "$tmp"
cat "$tmp"
```

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

```
working...
Cleaning up
/home/runner/work/_temp/nix-shell.0FChq9/tmp.I6PRHF66
aH
```

- Typical use: remove temporary files, release locks, stop child processes
- `SIGKILL` (and `SIGSTOP`) cannot be trapped
- To restore the default `trap` behavior, use `-`:
  - `trap - EXIT` or `trap - INT`
- For complex cleanup procedures, trap a function

```
cleanup() {
    echo "Clean up"
    rm -rf "$tmp"
    ...
}
trap cleanup EXIT
```

Thank you!

Thank you for your attention!

Don't forget the feedback in Moodle please!

Sources

[1] https://pubs.opengroup.org/onlinepubs/9799919799/utilities/V3_chap02.html#tag_19_08_01