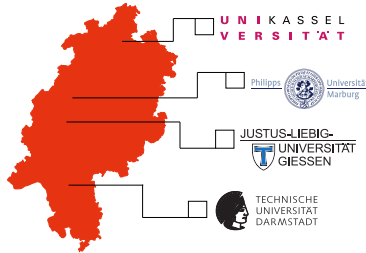


# Linux Shell and Shell Scripting

Hessisches Kompetenzzentrum für Hochleistungsrechnen (HKHLR)

HKHLR

18.02.2025



HKHLR is funded by the Hessian Ministry of Sciences and Arts



- 1 Linux Shell
- 2 Shell Scripting

## Linux

Unix-like operating systems built around the Linux kernel. Several Linux distributions are available.

## Familiarity with Linux command line environment is required for HPC

- ▶ **Most** of super computing clusters run a Linux distro (e.g. CentOS)
- ▶ Command line is (mostly) the **only** way to work on a cluster
- ▶ Large computations are run on clusters based on **shell scripts**

## Goal 1: Get familiar with the command line

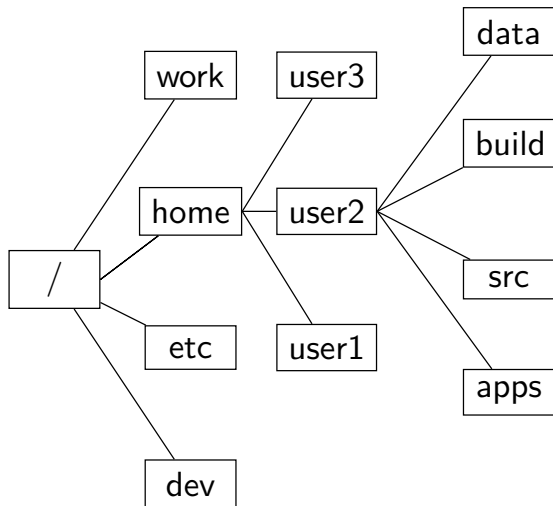
- ▶ Data organization
- ▶ Usage of programs
- ▶ Run computations on the cluster

## Goal 2: Learn basic shell scripting

- ▶ Understand structure and use cases of shell scripts
- ▶ Learn to write your own shell scripts

# PART 1: Introduction of Shell

here: bash shell  
(bash: Bourne Again Shell)



**Shell** is an implementation of the command line interface. Commands can only be passed as text.

**Shell prompt** invites to enter commands

```
username@computername:~>  
echo $SHELL
```

To close shell, press `ctrl+D` or use the command

```
exit
```

## Basic structure

```
<name-of-command> [options] [parameter1] [parameter2] ...
```

- ▶ Name of the command
- ▶ Options (nearly all commands offer `--help` option)
- ▶ Parameters (e.g. target file)

## example

```
cp -r -i directory /target/path/copy_of_directory
```



## Find help for a command

- ▶ Use `--help` option  
e.g. `cp --help`
- ▶ Use `man` command  
e.g. `man cp`

- ▶ Within the `man` command one can press `h` for help on how to navigate the manpage
- ▶ There is autocomplete with the `TAB` key for every command and file
- ▶ `apropos` makes full text search in man pages

There is **NO** File Browser to navigate through directories

### Current working directory

It is the current location in the file system

To show the **current working directory**

```
pwd
```

## Absolute path to file/directory

It is the location of the file/directory in the file system tree. An absolute path always starts at the root `/`.

To print content of the **current working directory**

```
ls
```

To print content of **other directory**

```
ls /path/to/directory
```

- ▶ Download the `init_hands_on.sh` script on your computer or a HPC Cluster (login with local documentation).
- ▶ While going through the next slides we will also go through the exercise sheet (file `exercises.pdf`)
- ▶ The content of the first page of the exercise sheet is related to the following slides.

Hidden files and directories are **NOT** shown by default

To show **ALL** files and directories of the current directory

```
ls -a  
ls -la          # short form of ls -l -a, -l options also shows symbolic links
```

### Special directories for forming relative paths

- ▶ `.` references the current directory (`ls -la .`)
- ▶ `..` references the parent directory of the current directory (`ls -la ../`)

## To change the current working directory

```
cd <path_to_another_directory>
```

► Use the *absolute* or *relative* path to the directory

## Simple ways to change the directory

```
cd          # change to the $HOME directory (same as cd ~)
cd -        # changes back to the previous directory (refer to $OLDPWD)
```

## To create directory in the **current working directory**

```
mkdir <new_directory>          # creates a new directory within the current directory  
mkdir -vp <nested_directory_path> # use the -p (--parents) option to create nested directories
```

Name of new directory must be **unique** in the current directory

Exercises: Absolute and Relative Paths, Change and Create Directories

To create text file in editor `nano`

```
nano <text_file>
```

## Basics of nano

- 1 Type any text in the field
- 2 Press `ctrl+O` to save
- 3 Press `Enter` to confirm
- 4 Press `ctrl+X` to exit



Files are created by programs but **NOT** by shell. The shell only parses the command used to create the file.

### Create a file

```
touch <path_to_some_file>
```

### To check the file type

```
file <path_to_some_file>
```

## To copy file in the **current working directory**

```
cp <file_name> <file_copy>
```

## To copy file to **another directory**

```
cp <file_name> <path/to/file_copy> # Path can be relative or absolute
```

The system will **NOT** prompt before an overwrite! Use the `-i` option.

⇒ `cp -i <file_name> <path_to_file_copy>`

By default the `cp` command will not copy the *full* content of a directory by recursing into every subdirectory.

### Recursive copy

```
cp -r <directory_name> <path/to/another_directory>
```

### Archive mode copy

Use the `-a` option to copy recursively and to preserve file attributes (comes later in this course).

To move file from the **current working directory**

```
mv <file_name> <path/to/another_directory>
```

To move file with **renaming**

```
mv <file_name> <path/to/another_directory>/<new_name>
```

The system will **NOT** prompt before overwriting unless we use `-i`

⇒ `mv -i <file_name> <path/to/another_directory>`

To move directory from the **current working directory**

```
mv <directory_name> <path/to/another_directory>
```

To move directory with **renaming**

```
mv <directory_name> <path/to/another_directory>/<new_name>
```

## To delete **file/directory** in the **current working directory**

```
rm <file_name>  
rmdir <empty_directory> # Recommended! Will prompt in case the directory is not empty.  
rm -r <directory_name> # Please be *very* careful with this command!
```

- ▶ The system will **NOT** ask about deleting (use the -i option)
- ▶ There is **NO** concept of recycle/trash bin to recover data

Exercises: Copy, Move, Rename, and Delete Files and Directories

## To search for files/directories in a directory

```
find <location> -name 'search_pattern' # <location> is a relative or absolute path.
```

- ▶ The `-maxdepth N` option to restrict the level of searching
- ▶ The `-type f` option to search only files
- ▶ The `-type d` option to search only directories
- ▶ The `-exec` option to execute commands on the results of the search

## Example

```
find . -type f -name "*.txt" -exec ls -la {} \;
```

To search for text fragments in files

```
grep 'text_fragment' file1 file2...
```

- ▶ Multiple files can be specified with wildcards (e.g. `grep <pattern> file*`)
- ▶ The `-i` option is for case-insensitive search
- ▶ `grep` can also deal with regular expressions (see the `-E` option)

Exercises: Searching Text Patterns in Files



## Topics covered so far

- ▶ Filesystem layout
- ▶ Basic navigation in the directory tree
- ▶ Structure of Linux commands
- ▶ Help for commands
- ▶ Manipulation of the filesystem (e.g. create, delete or move)

Wildcards and patterns act as placeholders for characters in e.g. file or directory names.

### Wildcard characters

- ▶ Character '\*' is replaced by any **combination** of characters
- ▶ Character '?' is replaced by any **single** character
- ▶ Mechanisms '\*' and '?' can be applied together

Example 1: `ls -l abc*`

```
abc
abc123
abc1234567
abcdef
abcDEF1
```

Example 2: `ls -l a?c`

```
abc
aBc
a1c
```

### Example 3: All files from Example 1 and Example 2

```
ls -1 a?c*           # -1: one column output
```

```
abc  
abc123  
abc1234567  
abcdef  
abcDEF1  
aBc  
a1c
```

Question: What is the difference between `ls -1` and `ls -1 *`?

## To list files and directories with their **attributes**

```
ls -l /etc
```

```
drwxr-xr-x 2 mo48xoxi group 512 13. Nov 13:47 Dir_tmp
-rw-r--r-- 1 mo48xoxi group 177 13. Nov 12:59 myfile.txt
-rw-r--r-- 1 mo48xoxi group 1387 13. Nov 13:11 README
-rwx----- 1 mo48xoxi group 15 13. Nov 12:59 script1
```

Diagram illustrating the components of the `ls -l` output:

drwxr-xr-x	2	mo48xoxi	group	512	13. Nov 13:47	Dir_tmp
-rw-r--r--	1	mo48xoxi	group	177	13. Nov 12:59	myfile.txt
-rw-r--r--	1	mo48xoxi	group	1387	13. Nov 13:11	README
-rwx-----	1	mo48xoxi	group	15	13. Nov 12:59	script1

Labels below the output:

- Permissions (drwxr-xr-x)
- Owner (mo48xoxi)
- Group (group)
- Size (512)
- Modification Time (13. Nov 13:47)
- File Name (Dir\_tmp)

## Permissions regulate access to file/directory for action

- ▶ To **r**ead file *or* to show content of directory (e.g. `ls`)
- ▶ To **w**rite file *or* to change content of directory (e.g. `rm`)
- ▶ To **x**ecute file *or* to enter directory (e.g. `cd`)

- ▶ **U**ser = creator of file or directory (owner)
- ▶ **G**roup = group of users who owns file or directory
- ▶ **O**ther = other users
- ▶ **A**ll = all users

The first column of the output

```
d rwx rwx rwx  
- rwx rwx rwx  
  1   2   3
```

1 - for **u**ser

2 - for **g**roup

3 - for **o**thers

To change permissions for access to file or directory

```
chmod <access_rule> <file_name>
```

### Access rule is written in symbolic notation

- ▶ Set access rights for user, group, other, or for **all**
- ▶ Set access rights separately to read, write, execute
- ▶ Use '+' to give access, or '-' to deny access

### Example

```
chmod u+x my_script.sh          # Make script executable  
chmod u=rwx,g=,o= my_script.sh # Give file owner 'rwx' permissions while none to all others
```



**Process ID** or **PID** are used to refer to running program

To show PIDs of processes started in **one** sessions

```
ps
```

To show PIDs of processes started in **all** sessions

```
ps x
```

- ▶ If PID is known, the program can be terminated
- ▶ To stop foreground programs, press `Ctrl+C`

To terminate program the "soft" and the "hard" way

```
kill <PID>      # this is the soft way  
kill -9 <PID>   # this is the hard way
```

Terminate by name: `killall` command

```
killall <name_of_program> # Terminates all processes running the command to be terminated
```

- ▶ Many shell commands show output information on screen
- ▶ **Standard output, error** are special files linked to screen

`stdout` and `stderr` can be redirected to files

```
<command> > <file_name>          # overwrite
<command> >> <file_name>          # append
<command> 2> error.log 1> out.log  # stderr --> error.log, stdout --> out.log
<command> &> <file_name>          # stderr and stdout to <file_name>
```

A **Pipeline** is a series of chained commands divided by the *pipe operator* "`|`"

Output of one command is input of another one

```
COMMAND1 | COMMAND2 | COMMAND3 | ...
```

## Examples

- ▶ Output of `ls` is viewed in the `less` program

```
ls -l | less
```

- ▶ Count, sort output of other command or inspect it's tail:

```
... | wc -l      ... | sort      ... | tail
```

## Module system commands

```
module avail my_soft
```

```
module load my_software/x.y
```

```
module list
```

```
module unload my_software/x.y
```

```
module purge
```

- ▶ `scp` same as `cp` but copy to/from remote host possible
- ▶ `rsync` same but recognizes existing identical files and has interesting options:
  - ▶ `-a` transfer as an archive (keeping owners, rights and timestamps)
  - ▶ `-v` be verbose (otherwise `rsync` will not print any info)
  - ▶ `-z` transfer data compressed
  - ▶ `-c` exclude some files like `*.o`

## Example

```
rsync -avzC ~/cluster/ username@lcluster13.hrz.tu-darmstadt.de:/home/username
```

## Windows

- ▶ There are also gui applications for filetransfer, like Filezilla

## PART 2: Shell Scripting

A **shell script** is a file which consists of commands executed by a shell.

### Why are shell scripts useful?

- ▶ To perform repeated actions multiple times
- ▶ To avoid mistakes in long and complex commands
- ▶ To start computations on super computing cluster



## General process of using a shell script

- 1 Write commands into a text file (e.g. by the `nano` editor)
- 2 Make the text file executable (by the `chmod` command)
- 3 Execute the script by the command `./script_name`

## Be careful

When making a script executable and you execute it by accident bad things can happen (depends on what your script does). To be on the safe side skip the `chmod \+x <script_name>` step and invoke the script by passing it to the appropriate shell, e.g. `bash <script_name>.`

First Line: The name of interpreter starts with **shebang** “#!”

```
#!/bin/bash # This line tells the kernel which shell to use.
```

A comment starts with '#' (will be **NOT** executed)

```
# This line is a comment. Such lines will *not* be interpreted by the shell.
```

Commands and comments can be written in one line

```
echo 'Hello world!' #This text is also a comment :)
```

## Check the attributes of the script file

```
ls -l my_script1.sh
```

## Make the script executable

```
chmod u+x my_script1.sh
```

## Execute the script like any other program:

```
./my_script1.sh
```

## Shell supports use of variables

**Variables** are used for storing "information".

## Two types of variables

- ▶ **Shell variables**
- ▶ **Environment variables**

## Main rules for naming variables

- 1 Only letters, digits and underscores are allowed
- 2 Variables begin with a letter or the '\_' character
- 3 Do **NOT** create system variables (e.g. PATH, HOME)

## To create/access **shell variables** (**NOT** environment)

```
MY_VARIABLE1='My first variable'  
my_variable1=2                # Shell is case sensitive.  
  
echo $MY_VARIABLE1  
echo $my_variable1
```

There **cannot** be spaces around the = sign!

## Delete shell (environment) variable

```
unset VARIABLE_NAME
```

- ▶ Shell variables are only valid within the current shell
- ▶ If a child process needs a variable, it **must be** an environment variable

To **convert** shell variable into environment or **create** it directly

```
export VARIABLE_NAME
```

```
export VARIABLE_NAME=VARIABLE_VALUES
```

To remove environment variable

```
export -n VARIABLE_NAME
```

Use quotation marks to include text with whitespaces

Compare the output of two commands with system variable

```
echo 'Hello, I am $USER' #The single quotation marks
```

```
echo "Hello, I am $USER" #The double quotation marks
```

## Expansion of shell variables

When using shell variables in *double quotation marks* their content will be expanded. This does not happen in case single quotation marks are used.

### Example

```
datapath_prefix="$HOME/data/my_project"    # $HOME will be expanded to /home/<username>  
datapath_current="$datapath_prefix/files"  # expands content of datapath_prefix
```



Capture the output of a command as a variable

```
VARIABLE=$(command)
```

**The command will be executed with all side effects!**

This also applies to deleting files.

Example:

```
file1=my_filename  
File_contents1=$(cat $file1)
```

## The basic syntax of the if-statement

```
1 if [ EXPRESSION1 ]; then
2   BLOCK_COMMANDS1
3 elif [ EXPRESSION2 ]; then
4   BLOCK_COMMANDS2
5 else
6   BLOCK_COMMANDS3
7 fi
```

### Comparison of integer values:

$a = b$ :    `a -eq b`

$a \neq b$ :    `a -ne b`

$a \leq b$ :    `a -le b`

$a < b$ :    `a -lt b`

$a \geq b$ :    `a -ge b`

$a > b$ :    `a -gt b`

./scripts/my\_script2.sh

```
1 # SPDX-FileCopyrightText: 2021 Competence Center for High Performance Computing in Hessen (\
   # HKHLR)
2 # SPDX-License-Identifier: MIT
3
4
5 #!/bin/bash
6
7 my_super_variable=-1
8
9 if [ $my_super_variable -gt 0 ]; then
10     echo "my_super_variable is positive"
11 else
12     echo "my_super_variable is negative"
13 fi
```

To evaluate integers, **compound command** `((...))` is applied

To sum up two integers stored in variables

```
1 a=10
2 b=5
3 c=$((a + b))
4 echo $c
```

The shell has limited support for arithmetic operations. For more complex expressions (e.g. with floating point numbers), auxiliary tools are recommended (e.g. `bc` or `awk`).

## The basic syntax of the for-loop (traditional shell form)

```
1 for ITEM in [ LIST OF ITEMS ]; do
2     BLOCK_COMMANDS
3 done
```

## The shell supports two "types" of for-loops

- ▶ Counter-based for loops, e.g.: `for ((idx=0; idx<5; ++idx)); do echo $idx; done`
- ▶ "Range-based" for loops, e.g.: `for idx in $(seq 1 4); do echo $idx ; done`

./scripts/my\_script4.sh

```
1 # SPDX-FileCopyrightText: 2021 Competence Center for High Performance Computing in Hessen (\
   # HKHLR)
2 # SPDX-License-Identifier: MIT
3
4
5 #!/bin/bash
6
7 for var in {1..5} some_word; do
8     echo "Print var = $var"
9 done
```

## bash debug option

use debug option `-x` in order to see what actually will be executed

```
bash -x my_script.sh
```

### More precise way:

- ▶ `set -x` will turn on debug output for the following statements
- ▶ `set +x` will turn them off again

**Always test your scripts first!**

## Topics so far

- ▶ Create and run shell scripts
- ▶ Basic syntax
- ▶ Create, print and export variables
- ▶ if- and for-statement
- ▶ Debugging with bash



- ▶ The basic usage of the command line was discussed
  - ▶ The fundamentals of shell scripting were studied
  - ▶ You should be able to do your daily work on a cluster
- 
- ▶ Many commands have a lot of useful options (see man pages!)
  - ▶ A lot of useful commands of shell are worth knowing (e.g. `cut`, `tr`, `tar/zip`, `sort`, `sed`, `awk`, `wc`)
  - ▶ Many tools are available (e.g. Midnight Commander)

- ▶ Download the `init_hands_on.sh` script on your computer or a HPC Cluster (login with local documentation).
- ▶ Have fun with the `exercises.pdf`.
- ▶ Try the other Linux commands from the handout yourself and collect some questions.

Thank you!