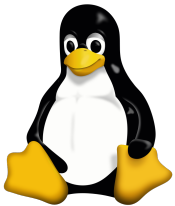
Linux Shell and Shell Scripting

Hessisches Kompetenzzentrum für Hochleistungsrechnen (HKHLR)

Tim Jammer, Nikolas Luke

15.03.2021



HKHLR is funded by the Hessian Ministry of Sciences and Arts

Schedule for today

- 09:00AM - 12:00AM Linux and Shell Scripting
- 12:00AM - 02:00PM Break / Individual time to work on exercises
- 02:00PM - 03:00PM Exercise discussion, Q&A time

Table of Contents

Linux is a family of Unix-like operating systems

We have to know Linux Shell to compute on **clusters**

- **Most** of super computing clusters are controlled by Linux
- Command line is the **only one** way to work on a cluster
- Large computations are run on clusters by **shell scripts**

Goal 1: To get initial knowledge on command line

- Data organization
- Usage of programs
- Computations on the cluster

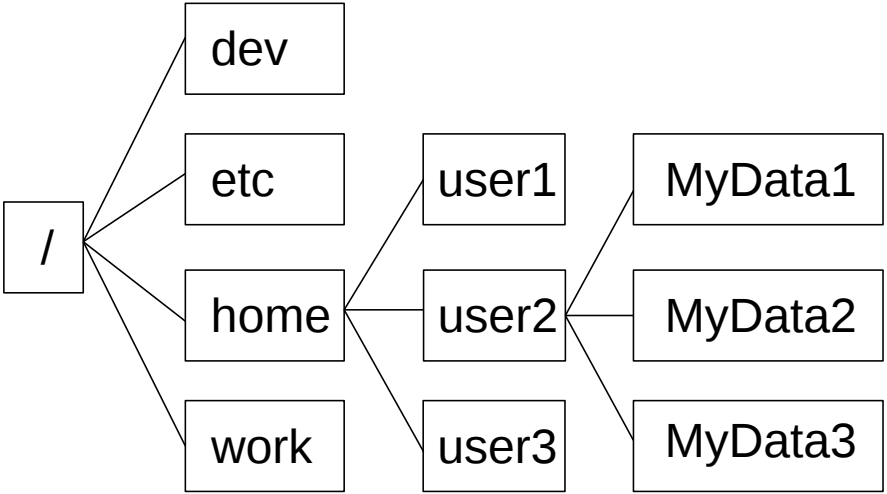
Goal 2: To learn basics of shell scripts

- To understand shell scripts
- To write own shell scripts

PART 1: Introduction of Shell

here: bash shell

Hierarchy of Files and File System



Shell is an implementation of the command line interface

Shell prompt invites to enter commands

```
username@computername:~>
```

```
echo $SHELL
```

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

```
exit
```

Basic Structure of a command

Basic structure

```
name [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 test 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

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

```
ls
```

To print content of **other directory**

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

Exercise for the break

- Download the `init_hands_on.sh` script on your computer or a Cluster (login with local documentation)
- follow along with the first page of the `exercies.pdf` file

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
```

Special directories for forming **relative paths**

- . is the reference to the current directory
- .. is the reference to the parent directory

Exercises: Special Directories and Relative Paths

To change the current working directory

```
cd <another_directory>
```

- Name of the directory
- Absolute or relative path to the directory

A simple way to change to the user home directory

```
cd              # same as cd ~
```

Create Directory

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

```
mkdir <new_directory>
```

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

Exercises: 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

To check the file type

```
file <some_file>
```

To check type of "other things"

```
type <command|alias|function>
```

Copy Files

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

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

To copy file into **another directory**

```
cp <file_name> <path/to/file_copy>
```

The system will **NOT** ask about overwriting (by default)

Use the `-i` option for prompting: `cp -i`

Copy Directories

To copy directory into **another directory**

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

The system will **NOT** ask about overwriting (by default)

Use the `-i` option for prompting: `cp -r -i`

Exercises: Copy Files and Directories

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** ask about overwriting (by default)

Use the `-i` option for prompting: `mv -i`

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
```

Exercises: Move and Rename Files and Directories

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

```
rm <file_name>

rmdir <empty_directory>
rm -r <directory_name>
```

The system will **NOT** ask about deleting (use the -i option)

There is **NO** concept of Recycle/Trash bin to recover data

Exercises: Delete Files and Directories

To search for files/directories in a directory

```
find <location> -name 'search_pattern'
```

1. The `-maxdepth N` option to restrict the level of searching
2. The `-type f` option to search only files
3. The `-type d` option to search only directories

Exercises: Searching Files and Directories

To search for text fragments in files

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

Group of files can be determined by the wildcards (**NO** quotes)

The `-i` option is to ignore case in text pattern

Exercises: Searching Text Patterns in Files

Topics so far

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

Live-Quiz:

Questions via zoom poll

Wildcards and Name Patterns

To apply one command to a group of files or directories

Mechanism 1

Character '∗' is replaced by any **combination** of characters

Mechanism 2

Character '?' is replaced by any **single** character

Mechanisms '∗' and '?' can be applied together

Wildcards: Examples

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
```

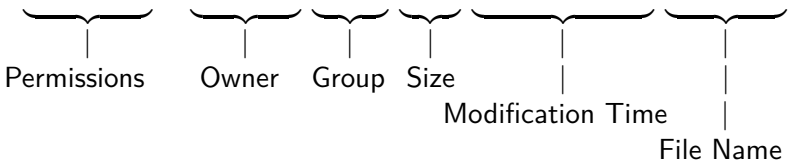
```
abc
abc123
abc1234567
abcdef
abcDEF1
aBc
a1c
```

Exercises: 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
```



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 e**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

```
drwxrwxrwx
-rwxrwxrwx
```

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
- Use '–' to forbid action

Example

```
chmod u+x my_script.sh
```

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 terminate program ("soft" way)

```
kill <PID>
```

To terminate program ("hard" way)

```
kill -9 <PID>
```

To stop foreground programs, press `Ctrl+C`

Many shell commands show output information on screen

Standard output is a special file linked to screen

Standard output is easily replaced by ordinary file

```
COMMAND >file_name                    #override

COMMAND >>file_name                   #append
```

Standard input (keyboard) and **standard error** (screen)

Pipeline is a series of commands divided by the operator "|"

Output of one command is input of another one

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

Example: Output of `ls` is viewed in the `less` program

```
ls -l | less
```

Move Examples: count, sort and "print end" of data

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

```
module avail my_soft
```

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

```
module list
```

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

```
module purge
```

Copy files from/to your local machine

- `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 shell

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

1. Write commands to 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`"

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

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

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

```
#This line is a comment.
```

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
```

Variables are used for storing information for shell

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 new **shell variables** (**NOT** environment)

```
MY_VARIABLE1='My first variable'
MY_VARIABLE2=2

echo $MY_VARIABLE1
echo $MY_VARIABLE2
```

There cannot be spaces around the = sign

To delete shell (environment) variable

```
unset VARIABLE_NAME
```

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

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
```

Use **double quotation marks** to allow expansion of variables

Use the output of a command as a variable

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

The command will be executed with all side effects!

such as deleting files

Example:

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

The basic syntax of the if-statement

```
if [ EXPRESSION1 ]; then
  BLOCK_COMMANDS1
elif [ EXPRESSION2 ]; then
  BLOCK_COMMANDS2
else
  BLOCK_COMMANDS3
fi
```

Comparison of integer values:

$a = b$: `a -eq b`	$a < b$: `a -lt b`
$a \neq b$: `a -ne b`	$a \geq b$: `a -ge b`
$a \leq b$: `a -le b`	$a > b$: `a -gt b`

The content of the script `my_script2.sh`

```
#!/bin/bash

a=10

if [ $a -gt 0 ]; then
  echo "a is greater than zero"
else
  echo "a is NOT greater than zero"
fi
```

```
chmod u+x my_script2.sh  #Make the file executable
./my_script2.sh  #Run the script
```

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

To sum up two integers stored in variables

```
a=10
b=5
c=$((a + b))
echo $c
```

For complex expressions, auxiliary tools are recommended

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

```
for VARIABLE [ in WORDS ]; do
  BLOCK_COMMANDS
done
```

The content of the script `my_script4.sh`

```
#!/bin/bash

for i in 1 2 3 10 11 hundred; do
  echo "Print i = $i"
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

Conclusions and Outlook

- The basic usage of the command line was discussed
- The fundamentals of shell scripting were studied
- You should be able to do a 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)

Exercise for the break

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

Thank you!