Faculty of Mathematics and Physics
CHARLES UNIVERSITY

NOFY077

Introduction to the Linux OS

Peter Huszár
KFA: Department of Atmospheric Physics

Pavel Řezníček
ÚČJF: Institute of particle and nuclear physics

December 21, 2023

Overview and Organization

Introduction to the Operation system Linux, focus on the command line, scripting, basic services and tools used in (not only) physics: tasks automation in data processing and modeling

Organization

- Graded Assessment (KZ): attendance to the lectures, worked out homeworks

Literature

- C. Herborth: Unix a Linux - Názorný průvodce, Computer Press, Praha, 2006
- D. J. Barrett: Linux - Kapesní přehled, Computer Press, Praha, 2006
- M. Sobell: Mistrovství v RedHat a Fedora Linux, Computer Press, Praha, 2006
- M. Sobell: Linux - praktický průvodce, Computer Press, Praha, 2002
- E. Siever: Linux v kostce, Computer Press, Praha, 1999
- Number of online sources...

Study materials and homeworks

- http://kfa.mff.cuni.cz/linux

Syllabus

1. UNIX systems, history, installation, basic applications
2. Structure of the Linux OS, file systems, hierarchy of the file system
3. Command line, shells, remote access (ssh, ftp)
4. Processes and their administration, basic system commands, packages, printing
5. Users, file and directory permissions
6. Work with files and directories, file compression, links, partition
7. Text-file processing commands, redirection, pipeline
8. Regular expressions
9. Command line based text editors
10. User and system variables, output processing
11. Scripts: basic construction, conditionals, loops, functions, automation
12. Networking, server-client services: http, (s)ftp, scp, ssh, sshfs, nfs
13. Programming in Linux (examples of Fortran, C/C++, Python), version control systems, documents in Latex

Shell Variables

Variables

BASH variables

You can use variables as in any programming languages. There are no data types. A variable in bash can contain a number, a character, a string of characters. You have no need to declare a variable, just assigning a value to its reference will create it.

- Creation and assigning a variable

```
STR="Hello World!"
echo ${STR} # to refer to variable value, use $
MYVAR=1000000
echo ${MYVAR} MYVAR # this prints '1000000 MYVAR'
```

- There are system variables that control the behavior of the system/shell/GUI:
- The command `set` will list all the system/shell variables (and functions - see later)
- E.g. $HOME - the HOME directory, $LANGUAGE - the system language, $PS1 - the look of the prompt
- E.g. $PATH - the list of paths, where BASH looks for binary files
- User can define his own system variables by setting them in ~/.bashrc
- use `export MYVAR="value"` in order the variable behaves as global
- A variable can be in three states: defined with a value (MYVAR=value), defined with NULL value (MYVAR=) and unset. To unset a variable, use `unset MYVAR`.
- You can define new variables with existing ones: `NEWVAR=${OLDVAR1}${OLDVAR2}` (this example merges two strings)

Variables

Variable expansion

- Bash enables numerous operations on variable value and gathering information on the variable (besides "asking" for its value)

```
${#MYVAR} # the length of variable value
${!prefix*} # prints all variables with their names starting with "prefix"
${MYVAR#pattern} # removes the match for pattern from the beginnig of MYVAR value
${MYVAR%pattern} # same as above but from the end of value
${MYVAR/pattern/string} # replaces pattern in MYVAR with string
${MYVAR^^} and ${MYVAR,,} # makes variables characters upper/lower case
```

- In the above examples, variables are "expanded" to a new value, which can be written out (with echo) or just saved to different variable(s).
- In the followig example, we rename all jpg files in a directory to JPG

```
for f in *.jpg; do # we will learn later
 echo "Renaming $f ..."
 mv $f ${f/.jpg/.JPG}
done
```

For a full list of variable expansion possibilities, see https://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-Expansion.html

Variables

Command output substitution in a variable

- The output of any command can be assigned to a variable as value in two syntactical way:

```
MYVAR=$( mycommand ) # preferred
MYVAR=` mycommand ` # not preferred
# the first way enables nesting:
MYVAR=$( mycommand $( anothercommand )   ) # mycommand takes the output of another command as argument
```

- eval - is a built-in Linux command which is used to execute arguments as a shell command. It combines arguments into a single string and uses it as an input to the shell and execute the commands.

```
MYVAR="ls -l /mydir"
eval $MYVAR
MYVAR='$'
MYVAR2=value
eval echo ${MYVAR}MYVAR2
```

- Use variable as a (part of) name for another variable.

```
MYVAR_A="123"
i=A
echo $((MYVAR_$i))
eval MYVAR_$i="456"
echo $MYVAR_A
```

Variables

Arrays in BASH

- Bash supports 1-dimensional arrays with arbitrary integer indexing

```
MYARR= ( 1 2 a b ahoj abc) # definition of an array, in this case indexing is starting from zero
echo ${MYARR[0]} -> 1 etc.
MYARR[100] = value # we can define/add arbitraty index
MYARR=([7]=a [10]=b [100]=c) # possibility of defining arbitrary index
MYARR+=(newelement1 newelement2) # extenstion of array
```

- Different information can retreived of arrays, including its length, list of elemets, list of indexes

```
echo ${MYARR[*]} # prints all the elements
... ${#MYARR[*]} # number of elements
... ${!MYARR[*]} # the list of indexes
```

Scripts

Scripts (reminder)

Sequence of commands to be processed.

- Allows functions, loops, conditions, call external commands
- Two ways how to run a script:
  - `./script.sh`: starts a new shell and runs the script in it (script file must be executable: `chmod +x ./script.sh`
  - `source ./script.sh` (or also `. ./script.sh`: runs the commands from the script one by one in the current shell → i.e. as if one would write them manually in the current terminal
  - `*.sh` used for *bash*-compatible scripts
  - `*.csh` used for *csh*-compatible scripts
- `#` are used for comments
- Special header "comment": `#!/usr/bin/zsh` instructs the script to be run by the `zsh` shell. Not only for shells, but also for interpreters like `python`
- `exit` [`number`] to quit script [and possibly return a *return code*]
  - Not needed at the very end of a script, it will end by itself
- `set -x` command inside a script instruct to show the commands being run by the script (i.e. for debugging)

Special characters (reminder)

- `''` (single quotes) do no interpret special chars, while `""` (double quotes) do
  - e.g. `echo '$i'` vs. `echo "$i"`
- ` `` ` (single backquotes) to insert output of command between the quotes
  - But better use `$(command)` instead
- `;` (semicolon) allows to put more commands on single line
  - e.g. `echo "ahoj" ; echo "abc"`
- `&` at the end of line to run program in the background, while continuing in the script
- `\` (backslash) cancels meaning of a special character
  - e.g. `echo "\$i"`
  - e.g. not to interpret space (`./script.sh ahoj\ abc` = `./script.sh "ahoj abc"`)
  - e.g. to allow quotes inside quotes (`echo "var = \"ahoj\""`)
  - at the end of line means wrapping - the line continues and the next line. Otherwise end-of-line is interpreted as delimiter of next command (equivalent of `;`)

```
echo \
"ahoj"


for myfile in filename1 \
              filename2 \
              filename3 \
do
  echo $myfile
done
```

Script special variables

Input arguments

The arguments passed with script are accessiable via special variables

- ./script.sh arg1 arg2 arg3 ...

```
$1, $2, $3, ...    Individual arguments on command line (positional parameters)
$#                 Number of command-line arguments
$*                 All arguments on command line ("$1 $2 ...")
$@                 All arguments on command line, individually quoted ("$1" "$2" ...)
$0                 Command name
```

- Use `shift` command to ”destroy” the first argument and shift the list of arguments to left, i.e. $1 becomes what was $2, $2 what was $3 etc., while original content of $1 is lost

Control of run commands in script (as well as in shell)

```
$?                 Return value (exit code) of the last preceding command
$!                 Process ID number (PID, see ’ps axuf’ of the last preceding command
$$                 Process ID number of the current process (the shell running the script)
```

Quick check of input variables content (script: $var replace by $1)

```
${var:-value}      Use var if set; otherwise, use value
${var:=value}      Use var if set; otherwise, use value and assign value to var
${var:?value}      Use var if set; otherwise, print value and exit
${var:+value}      Use value if var is set; otherwise, use nothing
```

Test expressions

`test EXPRESSION`: compare values, check file types

`[ EXPRESSION ]`: alternative notation

- Return code `$?` is `0` if true, `1` if false

```
( EXPRESSION )                 EXPRESSION is true
! EXPRESSION                   EXPRESSION is false
EXPRESSION1 -a EXPRESSION2     both EXPRESSION1 and EXPRESSION2 are true
EXPRESSION1 -o EXPRESSION2     either EXPRESSION1 or EXPRESSION2 is true
-n STRING                      the length of STRING is nonzero (also without -n)
-z STRING                      the length of STRING is zero
STRING1 = STRING2              the strings are equal
STRING1 != STRING2             the strings are not equal
INTEGER1 -eq INTEGER2          INTEGER1 is equal to INTEGER2
INTEGER1 -ge INTEGER2          INTEGER1 is greater than or equal to INTEGER2
INTEGER1 -gt INTEGER2          INTEGER1 is greater than INTEGER2
INTEGER1 -le INTEGER2          INTEGER1 is less than or equal to INTEGER2
INTEGER1 -lt INTEGER2          INTEGER1 is less than INTEGER2
INTEGER1 -ne INTEGER2          INTEGER1 is not equal to INTEGER2
```

```
FILE1 -nt FILE2                FILE1 is newer (modification date) than FILE2
FILE1 -ot FILE2                FILE1 is older than FILE2
-d FILE                        FILE exists and is a directory
-e FILE                        FILE exists
-f FILE                        FILE exists and is a regular file
-L FILE                        FILE exists and is a symbolic link (same as -h
-r FILE                        FILE exists and read permission is granted
-w FILE                        FILE exists and write permission is granted
-x FILE                        FILE exists and execute (or search) permission is granted
-s FILE                        FILE exists and has a size greater than zero
```

- … and other file flags (ownership, types)

Expressions

- Arguments in EXPRESSION typically contain output of commands

```
test $(cat /etc/passwd | cut -d: -f1 | wc -l) -gt 100
test `cat /etc/passwd | cut -d: -f1 | wc -l` -gt 100
```

- Be careful to treat cases when arguments in expression can contain spaces, better always use "" for string arguments (works for integers too though), especially when argument is an output of command with not-well predictable result ! (e.g. filenames can contain spaces...)

```
i="ahoj abc"
test  $i  = "ahoj abc"      # results in: bash: [: too many arguments
test "$i" = "ahoj abc"      # OK
```

Conditions - if/then/else

Use result of `test`

- Notation using square brackets `[ EXPRESSION ]`

```
if [ EXPRESSION ]
then
  command1
elif [ EXPRESSION ]
then
  command2
else
  command3
fi
```

```
if [ EXPRESSION ] ; then
  command1
elif [ EXPRESSION ] ; then
  command2
else
  command3
fi
```

- Short one-command condition using `&&` and/or `||`:

```
[ EXPRESSION ] && command1 || command2
```

- is equivalent to:

```
if [ EXPRESSION ] ; then command1 ; else command2 ; fi
```