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 16, 2025

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

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, same as `[ EXPRESSION ]`

`[[ EXPRESSION ]]`: more versatile version of `[ ]`

`(( ))`: arithemtic tests (e.g. comparision of numbers)

- 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
-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
```

Tests (cont'd)

`[ ] vs [[ ]]`: Using the `[[ ... ]]` test construct, rather than `[ ... ]` can prevent many logic errors in scripts. For example, the `&&`, `||`, `<`, and `>` operators work within a `[[ ]]` test, despite giving an error within a `[ ]` construct. Arithmetic evaluation of octal or hexadecimal constants takes place automatically within a `[[ ... ]]` construct.

```
[[ -L $file && -f $file ]]   works in [[ ]]
[ -L "$file" ] && [ -f "$file" ]
[[ a < b ]]: lexicographical comparison
[ a \< b ]: Same as above. \ required or else does redirection like for any other command.
[[ a = a && b = b ]]: true, logical and
[ a = a && b = b ]: syntax error, && parsed as an AND command separator cmd1 && cmd2
[[ (a = a || a = b) && a = b ]] vs. [ ( a = a ) ]: syntax error, () is interpreted as a subshe
```

`(( EXPRESSION ))`

```
     (( 5 > 4 ))
     (( 5 == 5 ))
     (( t = 40 < 45?7:11 ))   # C-style trinary operator.
     echo "If 40 < 45, then t = 7, else t = 11."
```

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
```

Conditions - case

Equivalent of `if/then/elif/elif/.../else/fi` statements chain

- Can use shell pattern matching (e.g. `*`)
- Use `|` for OR of matches
- On match the sequence of commands is run till `;;`
- `*)` typically used for safety `else` with an error message that there was no match

```
case $varname in
  pattern1)
    command1
    ;;
  pattern2|pattern3|pattern4)
    command2
    ;;
  *)
    command_error_no_match
esac
```

Loops - while/until/do/done

Keep looping (un)till EXPRESSION is valid

- Assuming the arguments in the EXPRESSION are changing during the sequence of commands in the loop, thus allowing the loop to stop at some point
- Can immediately stop the loop with `break` command
- Can immediately jump to next iteration with `continue` command

While

Stop looping if EXPRESSION becomes false

```
while [ EXPRESSION ]
do
   commands
   if [ ... ] ; then break ; fi     # alternative way to stop the loop
done
```

Until

Stop looping when EXPRESSION becomes true

```
until [ EXPRESSION ]
do
   commands
   if [ ... ] ; then break ; fi     # alternative way to stop the loop
done
```

For cycle

Loop over predefined list of items

- The list of items to cycle over is space-separated
- Can immediately stop the loop with `break` command
- Can immediately jump to next iteration with `continue` command
- `seq 1 100` to generate list of indexes from 1 to 100

```
for var_i in item1 item2 item3
do
   commands
   if [ ... ] ; then break ; fi     # possible way to stop the loop prematurelly
done
```

Space separation in list

- Potentially dangerous when list contains items with space, e.g. weird filenames
- For files use `find` command instead of `for` cycle
- Or replace spaces by a defined string and inside the loop revert this replacement:

```
# Would not work for files with space
for i in $(ls -1) ; do
  echo $i
done

# Works:
for ii in $(ls -1 | sed 's, ,__mezera__,g') ; do
  i=$(echo $ii | sed 's,__mezera__, ,g')
  echo $i
done
# Works
find . -maxdepth 1 -name '*' -exec echo {} \;
# Works
find . -maxdepth 1 -name 'a*' | while read i ; do echo $i ; done
```

For cycle

Loop over predefined list of items - cont'd

- The list of items to cycle over can be defined alternatively like:

```
for i in {1..5};do echo $i ;done
# from BASH v4.0+,  {START..END..INCREMENT} syntax
for in {0..10..2};do echo $i ;done
# control the width of the loop item:
for i in {001..500};do echo $i ;done
# or combining with other character and multiple ranges
for i in a{001..500} {700..999};do echo $i ;done

## The C-style Bash for loop
for (( initializer; condition; step ))
for (( c=1; c<=5; c++ ));do echo $c ;done
```

Functions

Similar behaviour as in other programming languages

- Mostly to help organization/readabilty of the code
- Accept parameters, treated in similar way as input parameters of scripts (i.e. $1, $2, etc.)
- Output transfered via `echo` command or e.g. my modifying a ”global” variable

```
x=0

myfunc() {
  for i in $@ ; do
    echo $i
  done
  x=1
}

echo $x
myfunc a bb cc 123
echo $x
x=0
str=‘myfunc dd ee‘ # x is not changed, myfunc is run in separate shell !
echo $str
echo $x
```

Automatic script options

Use of getopt command

- Colon : after option letter specifies that the option is expecting an argument

```
while getopts 'ha:' OPTION; do
  case "$OPTION" in
    h)
      echo "Option h (does not expect argument)"
      ;;

    a)
      echo "Option a with value \"$OPTARG\""
      ;;

    ?)
      echo "script usage: $(basename $0) [-h] [-a somevalue]" >&2
      exit 1
      ;;
  esac
done
shift "$(($OPTIND -1))"

echo "Remaining input arguments: $@"
```

Scripts - exercises

- Exercise 1: How to compare floating-point numbers ? Hint `bc -l`, `python -c ... exit,print`
- Exercise 2: Loop through all links in current directory (and sub-directories), check the file really exists (link is valid)
- Exercise 3: Store script input parameters into variables array. Iteratively destroy input parameters one by one and print the remaining on the screen (try all `for`, `while` and `until` loops)

Scripts - exercises

- Exercise 1: How to compare floating-point numbers ? Hint `bc -l`, `python -c ... exit,print`
- Exercise 2: Loop through all links in current directory (and sub-directories), check the file really exists (link is valid)
- Exercise 3: Store script input parameters into variables array. Iteratively destroy input parameters one by one and print the remaining on the screen (try all `for`, `while` and `until` loops)
- Exercise 4: For cycle to generate N random numbers (N=1000 if no argument passed to the script) and print the highest value. Hint: `$RANDOM`.
- Exercise 5: Select random 500 lines from mcData.txt (make sure the lines do not repeat)
- Exercise 6: Loop through archives `backup*`, search for files named `Invariant_masses.txt`, join their content with `mcData.txt` and remove duplicated lines
- Exercise 7: Batch analysis: script triggering a computation jobs
  - Job = generate 100 random numbers with given seed in `rnd.txt`, sleep 1 sec between the generation
  - Run max. 5 jobs in parallel
  - Allow the script to run more than once without breaking the rule above
  - Hint: use flag-files or `ps axuf` to find out which jobs are running, which are finished

Scripts running after logout

nohup

- Most simple way to keep process running after logout (or killing mother terminal)
- Syntax: `nohup command arguments`
- Output goes to `nohup.out` file

screen

- More complex system, behaving as a virtual terminal, allowing to:
  - Detach and re-attach to running session
  - After re-attaching one can see the output of the session
  - Works better on remote machines with complex authentication
  - Can name sessions
  - `screen` allows to send command to a running detached session
- `screen` to start a session
  - *CTRL-a d* to detach from session
  - `screen -list` to list sessions (either attached or detached)
  - `screen -r` to attach to a sessions

tmux

- Similar functionality to `screen`, but more actively developed
- `tmux` to start a session
  - *CTRL-b d* to detach from sessions
  - *tmux ls* to list sessions
  - *tmux attach* to attach

Crontab - Reminder

CRON system:

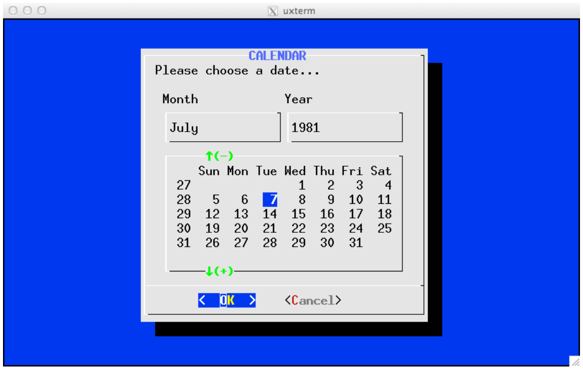
- /etc/crontab: basic file to run tasks per hour/day/week/month
- /etc/cron.hourly
- /etc/cron.daily
- /etc/cron.weekly
- /etc/cron.monthly
- /etc/cron.d: more complicated rules

```
# /etc/cron.d/renew_prak0x: crontab entries for reweval of the prak0x user home directories
# Execute only during the period of the exercises (01.Oct - 20.Jan)
# TODO ?: Add entry in between day in case of 2 excercises per single day

SHELL=/bin/bash

# m  h   dom   mon              dow  user  command
32   01   *    OCT,NOV,DEC,JAN  SUN  root  /home/prak_template/bin/reboot.cron.sh
# NO!!! (studenti by po rebootu nenasli sva data !)
#@reboot                             root  /home/prak_template/bin/renew_prak0x.cron.sh
12   03   *    OCT,NOV,DEC       *   root  /home/prak_template/bin/renew_prak0x.cron.sh
12   03  1-20  JAN               *   root  /home/prak_template/bin/renew_prak0x.cron.sh
```

Graphical interface to scripts

Programs to easily create simple graphics interfaces:

- Calendar
- File selection
- Forms
- Messages
- Lists
- Progress bars
- Text entry



```
                    CALENDAR
 Please choose a date...

  Month                Year
  July                 1981

      ↑(-)
      Sun Mon Tue Wed Thu Fri Sat
  27                1   2   3   4
  28    5   6   7   8   9  10  11
  29   12  13  14  15  16  17  18
  30   19  20  21  22  23  24  25
  31   26  27  28  29  30  31
      ↓(+)

         <  OK  >      <Cancel>
```

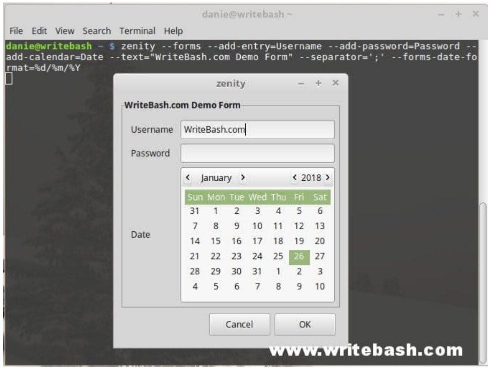
dialog

- Terminal-based graphics
- See number of exampes in `/usr/share/doc/dialog/examples`

Graphical interface to scripts

Programs to easily create simple graphics interfaces:

- Calendar
- File selection
- Forms
- Messages
- Lists
- Progress bars
- Text entry



```
danie@writebash ~ $ zenity --forms --add-entry=Username --add-password=Password --
add-calendar=Date --text="WriteBash.com Demo Form" --separator=';' --forms-date-fo
rmat=%d/%m/%Y
```

```
WriteBash.com Demo Form
Username  WriteBash.com
Password
Date      January  2018
          Sun Mon Tue Wed Thu Fri Sat
           31   1   2   3   4   5   6
            7   8   9  10  11  12  13
           14  15  16  17  18  19  20
           21  22  23  24  25  26  27
           28  29  30  31   1   2   3
            4   5   6   7   8   9  10
```

dialog

- Terminal-based graphics
- See number of exampes in `/usr/share/doc/dialog/examples`

zenity / gdialog

- Graphical windows (GTK)
- See examples at `https://help.gnome.org/users/zenity/3.32/`