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

November 25, 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

KFA MFF UK

ÚČJF MFF UK

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

Further essential commands

- echo - Write arguments to the standard output. very simple but useful command. Especially with pipe.

```
echo Our first echo command # will write the arguments to the stdout, including a \newline character.
echo -n No newline # newline is not added
echo '$+-/*.]!@*(#*' # if the arguments contain "special" characters,
                     # which have reserved meaning in the scripting language (bash)
                     # than the safest way is to put single quote(' ').
echo $HOME "$HOME" '$HOME' # a good example
# how to print out single quotes?
echo \'text in single quotes\' # backslash \ turns off the special meaning
echo $HOME \$HOME. # special meaning of $ turned off (see later)
```

- tr - translate or delete. An utility which takes the standard input (supplied via pipe) and either translates some characters (called SET) to others (e.g. small letters to capital ones) or deletes some characters from SET. Now we will see why 'cat' is so useful.

```
# translation
cat /my/file | tr "[a-z]" "[A-Z]" #  (or  tr "[:lower:]" "[:upper:]"), lower to upper case
# data stream can be generated by echo too (not only cat /some/file)
echo "text   with  spaces" | tr "[:blank:]" " " # all horizontal space changed to single space
echo "text with numbers 2019" | tr "[:digit:]" "?" # digits changed to "?"
echo "text with Xsome Wunusual Qletters" | tr "XQW" "O"
# deleting
echo "text with numbers 2019" | tr -d "[:digit:]" # delete all digits
                              #   -d "[:alpha:]" # delete all letters
                              #   -d "xyz" # deletes all occurences of x,y,z
```

- xarg - build and execute command lines from standard input. In its basic usage, takes the standard input and puts it as arguments to command it calls. Very powerful, see man xargs!

```
echo str1 str2 | xargs command # this is equivalent to 'command str1 str2'
echo /dev/sda1 /dev/sda2 | xargs df -h # prints the size and free space on the
                                       # two indicated device (partitions)
                                       # can be useful when having a file with
                                       # each line as future argument for a command
```

- awk - scans patterns in the input and processes it. AWK is a separate programming language callable from command line. We will learn only a few basic usages. For more info e.g. http://www.grymoire.com/Unix/Awk.html.

```
# suppose to have a file with columns separated by unspecified whitespace (sometimes tab or multiple space)
# and one wants to take only some of the columns
# cut -d" " will not work, because it considers a specific delimiter
cat /my/file/with/columns.txt | awk '{print $1" "$3" "$4}' # this prints columns 1, 3 and 4 separated
                                                           # by single space
cat /my/file.txt | awk  'NR % 2 == 0' # print even lines
cat /my/file.txt | awk '/'pattern'/ {print $2}' # print the second columns
                                                # of lines containing the 'pattern'
```

Further commands

- dirname - returns the directory part of the path if the argument is a file. If the argument is directory, returns the parent directory's path

```
dirname /path/to/my/file # returns /path/to/my
dirname /path/to/my/dir # returns /path/to/my
dirname relative/path/to/something # returns  relative/path/to
```

- basename - returns the file part of the path, or the last dir, if the path is for directory. Optionally removes the suffix from filename.

```
basename /path/to/my/file # returns file
basename /path/to/my/dir  # returns dir
# with two arguments, 2nd as suffix to remove
basename /path/to/my/file.txt .txt #  returns file
```

KFA MFF UK

ÚČJF MFF UK

Command line calculators

- expr - the most common expression evaluator used for integer and logical operations. See man expr for the full list of operators.

```
expr \( 100 + 1 - \( 50 / 10 \) \) % 7 \* 1000
    # spaces between relation and numbers
    # (,),* must be backslashed because they have a different meaning in bash
expr \( 100 \> 101 \) \& \( "xyz" = "xy" \) # >,<,& must be backslashed too in logical expressions
```

- bc - An arbitrary precision calculator language. Much more capable than expr because you can set arbitrary precision. The default is integer precision. Moreover, life is easier with bc as no backslashes needed.

```
echo " expression " | bc # expression can be aritmetic or logical one
echo "((1+2)*3/4)-15+sqrt(100)" | bc # sqrt is  builtin function
-3
# to use the matlib (extended support of builtin functions and float point precision)
# use bc -l
echo "s(2)*c(2) / 2.0" | bc -l # s - sinus, c - cosinus (see man bc for others)
-0.18920062382698206283
# use bc for number base conversion
echo "obase=2;100" | bc -l # Convert Decimal to Binary
echo "obase=16;ibase=2;1011011010000101010110" | bc -l # Convert Binary to Hexadecimal
echo "a=1;b=2;a*b" | bc # user can define variables
```

- apcalc - another Arbitrary Precision CALCulator - the most powerful with much more builtin functions. The number of digits in the output is unlimited (limited only by the computer's internal memory).

```
echo "2**1000" | calc -pd # ** means exponent, power of 1000 - to demonstrate the unlimited precision/digits
echo "comb(100,4)" | calc -pd # combinatorial number 100!/4!(100-4)! # some more "exotic" functions
```

  List of apcalc builtin functions: "calc help builtin"

- Simple aritmentic expression as variables (see later in details)

```
var=$(( 10*80-5+1+(2+3) )) # This evaluates the expression in the double-paranthesis ((expression))
                           # and saves the value in variable 'var' (bash variables to come later)
```

Date/time related commands

Date/time related commands

Commands to work with datetime values/calendars

- cal - displays a calendar

```
cal # displays the calendar for the actual month
cal 2025 # displays the calendar for 2025
cal 11 2025 # displays the calendar for November 2025
```

- date - a date/time "calculator"

```
date # shows the current date in the default format
date +"Today date is %Y-%m-%d, and time is: %H:%M:%S" # see man date
date --date="1620-11-08" +%A # info o konkretnim datumu
date --date="-100000 days" # what happened 100000 days ago
date --date="+1000000 hours" # what is the date 1000 000 hours from now
# it can handle complicated date/time calculations
date --date="2000-01-01 12:00 +10 days -1000 seconds + 9898 hours - 1 year" +"%Y-%m-%d,%H:%M:%S"
```

- time - an utility to measure the amount of time it takes a program to execute. It also measures CPU usage and displays statistics.

```
time command arguments # basic usage
time find ~ -name '*.pdf' # this will find all pdf files in the home directory
                          # and the time command will write out how long it took
```

File/directory search

Search commands

Commands to search files and directories

- locate - search files based on a pre-built database by updatedb. Locate searches the whole directory tree but search in a database of files which is updated e.g. once-per-week.

```
locate name # locate all files with 'name' in their name ()
locate -c name # prints the number of found files
locate -e name # prints only those files which really exists in the moment of search.
```

- find - a powerfull search engine for files and directories with the possibility to logically combine search criteria

```
# basic usage
find /my/dir "search criteria" # search files/directories in /my/dir (and deeper)
                               # based on the search criteria
find /my/dir -name '*jpg'      # search based on name, files ending with jpg.
find /my/dir -type d -name 'a*' # search directories starting with "a" (or files only -type f)
find /my/dir -empty # find all empty files/directories
find /my/dir -perm 664 # find files/directories with permissions 664 (-perm u+rw)
find /my/dir -user student # find files/directories with "student" as owner
find /my/dir -size 5M # files larger than 5Megabytes
# you can combine search criteria
find /my/dir -size 5M -and -perm 664 -or  -empty -and -not -name '*jpg' # -or,-and,-not
```

Further actions on found items

```
find /my/dir "search criteria" -exec rm -rf {} \; # this will find files/dirs and applies
                                                  # the command after the -exec for each file found
                                                  #  (one-by-one!!)
find /my/dir -iname "*dvorak*mp3" -exec mplayer {} \; # listen to all dvorak mp3-s, mplayer is started
                                                      # separately for each file found
find /my/dir "search criteria" -exec command {} + # start the command only once for all files as parameters
find /my/dir -iname "*dvorak*mp3" -exec mplayer {} + # mplayer is started once and plays all files found
```

- whereis/which - find the whole path to the called binary/command. Can be useful if one has multiple instalations of a program and the specific binary can be executed from different directory. In this case, it is good to know which one is executed.

```
whereis python3 # this will show the full path for the ls command
which   python3 # shows exactly which binary is executed if more "python3"-s are installed
```

Pattern search in texts - regular expressions

- grep - One of the most useful and versatile commands in a Linux terminal environment is the "grep" command. The name "grep" stands for "global regular expression print". This means that grep can be used to see if the input it receives matches a specified pattern.

```
cat /my/input/file(s) | grep "pattern" # this will print all lines with the word 'pattern'
 # This is of course equivalent to grep "pattern" /my/input/file(s)
cat /my/input/file(s) | grep --color "pattern" # occurences are 'colored'
cat /my/input/file(s) | grep -o "pattern" # print only matches (-c will print the count)
 # useful options
 # -i - case insensitive, -v invert search; -l -- prints only files with matches
 # -L - print files without match
```

- However, the real power of grep comes with the introduction of regular expressions!!!

We see that with grep, we can search for some characters, words, but what about more complicated patterns???
For example:

- words that start to/end/contain a specific set of letters
- words starting with capitals or having certain number of characters
- email addresses
- IP address
- special numbers (e.g. real numbers)
- specific parts of computer code
- webpage address ...
- The above examples cannot be searched with simple `grep "word" /my/file`
- The solution is "regular expressions"
- A quick example: regular expression and search for a valid email address within a textfile

```
grep -E ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$ /my/file
```