APPS@UCU

Linux course

Shell

Morhunenko Mykola

APPLIED SCIENCES FACULTY

Contents

What is command shell?

Command Shell

- Command Shell is a computer program which provides user with a (CLI) command line interface to control the computer using keyboard, without GUI (Graphical user interface), for communication with the Linux system
- If you use Linux, you have definitely seen the command prompt. Usually it looks like $ or, probably, [username@hostname path] $
- From the very beginning it looks like GUI is faster, but it is totally false: CLI just has high entry threshold. But it allows to write scripts (files with shell commands) that can automate routine, which is impossible in GUI
- Much more programs provide only CLI. If you want to use servers and connect to other computers via ssh, then being a real programmer is knowing shell
- You can check your shell by the command `$ echo $SHELL`
- Most likely you have bash, the most popular and stable one
- If you are done, you can exit shell with exit command, or by pressing the Ctrl+d in the terminal emulator window

What is Bash

- Bash stands for "Bourne(born)-again-shell"
- It is the default shell for most Linux distros
- POSIX standard has a full description of shell. Bash implements all this features, plus something of its own, known as bashism
- Bash is a standard shell for the majority of Linux distros, but it doesn’t mean it is the best one

ZSH

- Zsh stands for Z-shell
- Like Bash, it derives from Bourne family of shells, in everyday usage zsh is the same, as bash, but has a lot of extensions and other configuration files' syntax (default - ~/.zshrc)
- There is an entire eco-system of configuration tools and themes called oh-my-zsh which is very popular
- There is a huge amount of extensions on github, that can make everyday usage easier - ranging from auto-complete to syntax highlighting
- There are differences in scripts, we'll pick this up later.

<-- It all starts here.

Linux

Path

- Path - one of the most important terms in this topic (and in understanding how programs work)
- Current path - or current directory, working directory - the directory, from where you are working, launching programs/scripts
- Paths can be absolute or relative
- Absolute
  - / - also known as root path, all paths that starts from it are absolute. Other examples:
  - /home/username
  - /usr/local/share/zsh/site-functions/
- Relative
  - Relative paths don’t start from the root. Shell interpreter always run all programs with respect to the current path. They never begin with /
  - .zshrc
  - Documents/UCULinux/presentations

Paths

- Names are case-sensitive, which means /home/UserName and /home/username are different names
- There are also special paths
  - ./ stands for the current path
  - ../ stands for the path one step back. For example, if the ./ is /home/username, the ../ will be /home
  - ~/ stands for the directory of current user. For example, for root user ~/ will be /root, and for username it will be /home/username

Bash intro

Syntax

All default commands (programs) have very similar syntax

Examples

```
program_name [option]... [arguments]...
```

Options starts from - or --
To see, how to use any command

Examples

```
program_name -h
or program_name --help
behind are synonyms, but parameters are in short and long forms
man program_name - it provides full documentation about the command
```

Base commands

- pwd - print working directory - show your current path
- ls <path> - list - show what is inside the directory
- cd [path] - change directory - change your current directory

Now it is possible to get something

Examples

```
username$ pwd
> /home/username
username$ ls
> Desktop Documents Downloads Music Pictures Videos
username$ cd Downloads
username$ pwd
> /home/username/Downloads
```

Base commands

Introducing ls

- ls -a - list all, including names starting with the dot symbol "."
- ls -l - list using a long list format, to display more information including permissions, size and important dates
- ls -r - list in reversed order
- ls -R - list current directory and all subdirectories recursively
- ls -S - sort by file size, larger first. They can be combined as well.

The most commonly used are:

- ls -la - list all with full info. The command has path as an argument so the following command is also valid
- ls -la /etc/systemd/system

There are much more options, that can be found using ls --help

Base commands

Now it is possible to move around the directories and see, what is around. In order to work on labs or projects it's usually required to create something and see what is inside being able manipulate it somehow:

- mkdir [dirname] - make directory - create a new directory
- touch [filename] - create a file with filename or update the date of file's last modification to the current date
- date - print current date and time
- echo [text] - print text to the standard output
- cat [filename] - short for concatenate - show the file's content
- cp [source...] [destination] - copy - copy all from source (all arguments except the last one) to the destination (the last argument). -r option required for directories
- mv [source...] [destination] - move - move all from source to destination
- rm [path] - remove - remove the file. -rf required for directories

Creating Links

In Linux there are two types of links: hard and symbolic

- Inode - a data structure in the Unix-style file systems, that describes a file-system object
- Inode can have any number of hard links, and the inode will persists on the system until all hard links disappear. Changing in one file applies these changes also to all its hard links

Going deeper

- ls -i command is used to list all files with it’s inodes
- ln [source] [destination] is used to create hard links

Examples

```
username $ ln file1 file2
username $ ls -i
> 9700529 file1 9700529 file2
```

it is noticeable that both files have similar inodes

Examples

```
username $ echo "Hello World!" >> file1
username $ cat file2
> Hello World!
```

Changes in one file apply these changes to the other one

Going deeper

What about symbolic links?

- symbolic links (symlinks) are used more often. This is a special type, and the link refers to another file by name, not by inode.
- deleting the source file will make the symlink broken
- ln -s - used to create the symbolic link

Examples

```
username $ ln -s file1 file3
username $ ls -l
> -rw-rw-r– 2 username groupname 18 Apr 17 00:47 file1
  -rw-rw-r– 2 username groupname 18 Apr 17 00:47 file2
  lrwxrwxrwx 1 username groupname 5 Apr 17 00:54 file3 -> file1
```

- Symlinks can be created for any type of file system objects
- Can be used to point to an object from another file system

Wildcards, Globs

- In case, there is a folder with 25 test files, but it is necessary to delete first 8 of them... there are few ways how to deal with that

Examples

```
username $ rm test1 test2 test3 test4 test5 test6 test7 test8
  or...
username $ rm test[1-8]
  or if you want to delete all tests...
username $ rm test*
```

- So, `*` wildcard stands for all matches, any number of any symbol
- `?` stands for any one symbol
- `[]` wildcard stands for ranges, so `[abc]` means "any of a, b, c", the same for [a-c]
- `!` stands for non-match, so [!a] stands for any symbol except ’a’

Important about wildcards

- Be careful while using wildcards
- Bash preprocess all input to extend it with respect to wildcards
- so if you want to use one of such symbols just as symbols, you can either escape character with \ symbol, or use single quotes

Examples

username $ echo [fo]* > ./new_file
in this case you will add names of all files starts with 'f' or 'o'
username $ echo '[fo]*' > ./new_file
username $ echo \[fo\]* > ./new_file
both approaches above are correct

- All bash commands are here

Searching

- So as for now we know how to create files, directories, move them and remove. But how to find them?
- The Linux file system is well-structured, so it’s very easy to navigate it, but still, there are thousands of files and it’s impossible to remember all of the locations. There is entire presentation about the Linux File system hierarchy.
- find [path] -name ["filename"] , and in the name can globs can be used (but they must be escaped)
- But if you don’t know for sure the filename or dirname, the better way is find <path> -regex ["filename_regex"]
- Be careful! All files in the system have their own permissions. In order to search somewhere outside the /home/username folder, you must run the program in the privileged mode

Redirections

- As it will be covered in the File systems overview lecture, one of the defining features of Unix is that everything is a file
- Even Shell is a group of files
- By default, in Unix systems programs read the input from a so called stdin (input from the keyboard), write the output to the stdout and write errors to the stderr
- Redirections allows to change the input/output files
  - 0<filename or <filename – input from the filename
  - 1>filename or >filename – output to the filename , rewrite the file content
  - 1>>filename or >>filename – output to the filename , add to the file content
  - 2>filename – error to filename , rewrite the file content
  - 2>>filename – error to filename , add to the file content
  - &>filename – both output and errors to filename
  - 2>&1 – errors to stdout

```
username $ ls >> some_file # No output to the
username $ cat some_file # show the content of some_file
> Documents Downloads Music Pictures Programs
```

Pipes

- Sometimes it’s required to give one program output of the other program (some kind of composition)
- There are pipes for such cases
- Pipe can be made by the `|` symbol
- `wc` - Word Count program, used to... count words (also letters, lines etc)

```
username $ ls -la | wc -l
> 23
```

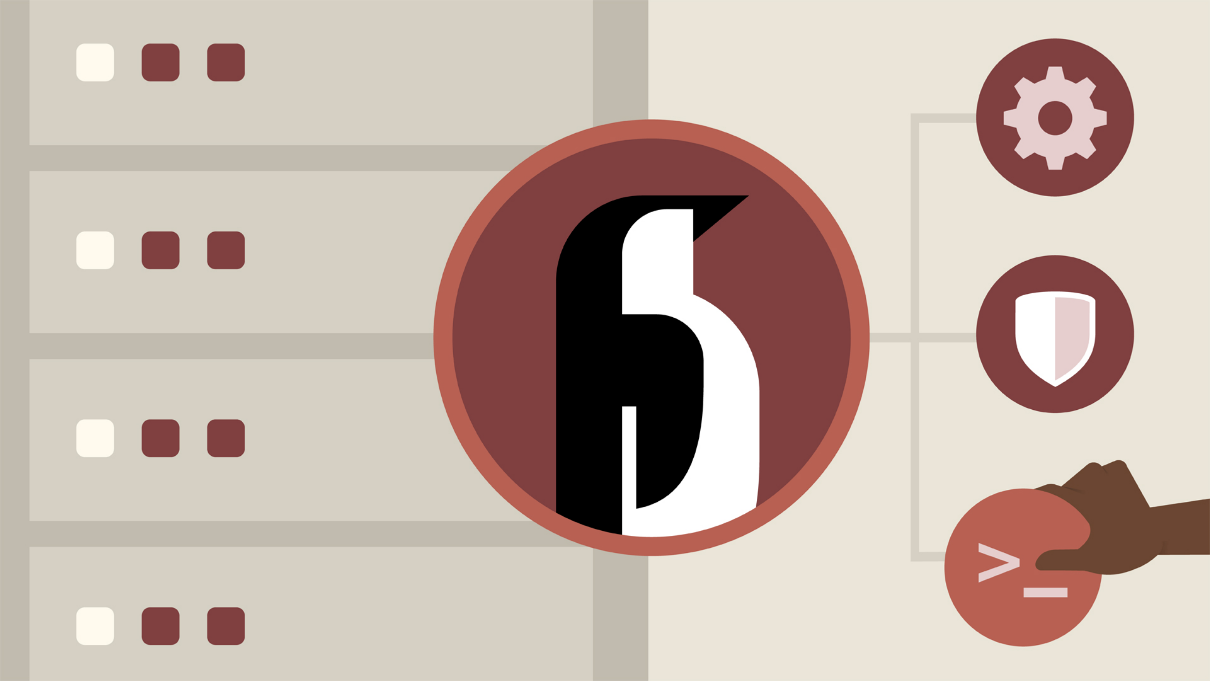
- That means that there are 23 lines of output
- Pipes are powerful, and one of must-know and must-use instruments

History

- All shell commands are saved to the .bash_history or .zsh_history
- If you want to make "anonymous" command (not store it to the history file) just start the command from the space
- There are important environment variables:
- HISTSIZE indicates how many commands from your history file are loaded into the shell’s memory
- SAVEHIST indicates how many commands your history file can hold
- All important variables are here
- ZSH has special extensions for a better work with the history
  - zsh-autosuggestions – suggests you the last command from the history
  - zsh-z - cd command alternative, but with some history analysis

Hot keys

- Usually people use only Ctrl + Shift + C, Ctrl + Shift + V, Ctrl + C, Ctrl + Z , Though there are a lot of other useful hotkeys:
  - Ctrl + n or arrow-up – next command in the history
  - Ctrl + p or or arrow-down – previous command in the history
  - Ctrl + c – SIGINT signal to the program (SIGnal INTerrupn, usually stops the process)
  - Ctrl + l – clear the screen, call clear program
  - Ctrl + x; Ctrl + e – open the $EDITOR to change the inputted command; if there is no command, just open the $EDITOR
  - Ctrl + z – freeze current program
  - Ctrl + Shift + C – copy text to the global clipboard
  - Ctrl + Shift + V – paste text from the global clipboard
  - Tab - completion/suggestions of the command
- Full list available here



Extensions

- The Linux world doesn't need file extensions
- The operating system doesn't use them to determine how to open a file
- But extensions are used by some parts of the OS to determine which program to use to open file
- How does the operating system find out, which is the file and how to deal with it?

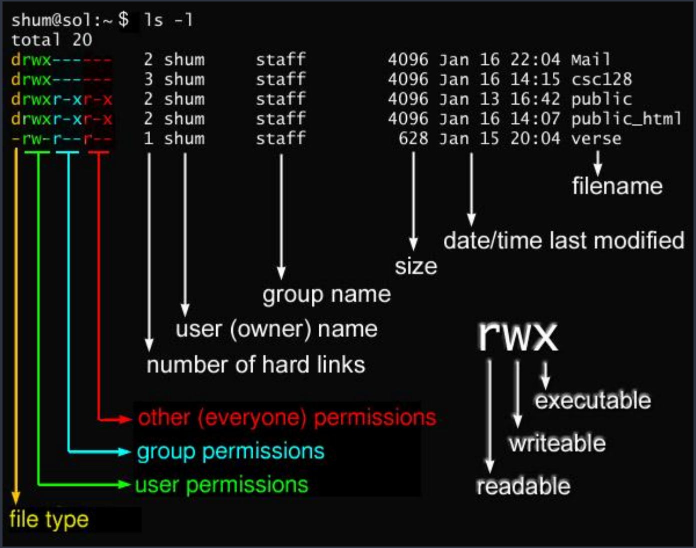
Permissions

- When executing the ls -l command, at the very beginning of every line there is 10 characters and then two words

```
username $ ls -la
> drwxrwxr-x 10 username groupname 4096 Apr 20 02:21 dir
  -rw-rw-r-- 10 username groupname 4096 Apr 20 02:21 textfile
  -rwxr-xr-x 10 username groupname 4096 Apr 20 02:21 binary_file
```

- They are not just letters, there is a lot of information behind these 10 characters (or, actually, 3 decimal numbers)

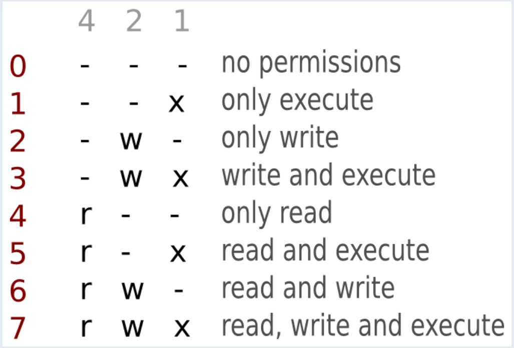
Permissions



```
shum@sol:~$ ls -l
total 20
drwx------   2 shum     staff        4096 Jan 16 22:04 Mail
drwx------   3 shum     staff        4096 Jan 16 14:15 csc128
drwxr-xr-x   2 shum     staff        4096 Jan 13 16:42 public
drwxr-xr-x   2 shum     staff        4096 Jan 16 14:07 public_html
-rw-r--r--   1 shum     staff         628 Jan 15 20:04 verse
```

Permissions

- The very first letter stands for the file type (this topic will be covered in the File systems presentation)
- To change permissions, there is a command `chmod`
- To see all possible parameters, use `chmod --help` . (But some explanations for the beginners are on the next page)
- All triplets of permissions `rwx` have theirs number correspondences



	4	2	1	
0	-	-	-	no permissions
1	-	-	x	only execute
2	-	w	-	only write
3	-	w	x	write and execute
4	r	-	-	only read
5	r	-	x	read and execute
6	r	w	-	read and write
7	r	w	x	read, write and execute

Permissions

- The outputoutput of `chmod --help` may have some confusing stuff in it
- In `[ugoa]` - User, Group, Other users, All
- In `[rwxXst]` - Read, Write, eXecute
- eXecute only if the file is a directory or already has execute permission for some user
- Set user or group ID on execution
- Save program text on swap device (a performance enhancer)

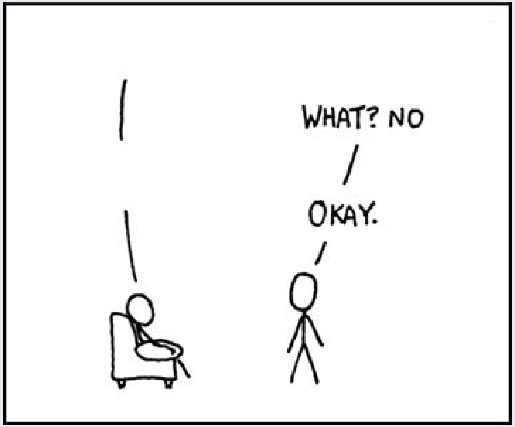
sudo

Sudo

- By default, every user have permissions to play around only his /home/username directory
- To somehow modify the system (either install programs or modify global settings), you need a so called superuser mode
- If there's an error starting with permission denied , then it's necessary to run the program with privileges, for example, using sudo

Sudo

- To run any program as a superuser, use sudo <program_name> <program_parameters> , where sudo stands for "super user do"
- To login into the shall as a superuser, both su root and sudo -i can be used. When you type a password, it stays invisible
- DO NOT RUN GUI AS ROOT! instead use gksu (it is deprecated for as 2021, but still in AUR) or alternatives (kdesu, sux)



Process Control

- To stop the process use Ctrl + z keybinding
- To unstop the process fg program is used (foreground)
- To unstop the program but leave it in the background, bg program is used (background)
- If you want some program to run at the background, & symbol is used

```
username $ clion &
> [1] 117306
```

- This command will start a clion from the current environment, but the command line still will be available for the input
- Be careful. After such command all clion output will still be printed to the stdout (if not redirect it)
- jobs -l – list all background jobs

Process Control

- Signals - one of possible ways how different processes communicate with each other in the POSIX systems
- To sent signal to the process ID (that 5-6 digits numbers) kill command can be used
- SIGTERM - kill sends it by default, which means stopping the program (TERMinate)
- SIGINT - Interrupt the process, the hotkey is Ctrl + C
- But jobs program shows only processes started from the current session
- ps program used to see all processes, their PIDs and other info

```
username $ ps ax
> PID TTY       STAT    TIME COMMAND
    1 ?         Ss      0:02 /sbin/init
    2 ?         S       0:00 [kthreadd]
    3 ?         I<      0:00 [rcu_gp]
     .....
```

- So to stop the process immediately, just run kill [PID]

Scripts

Scripts

- Script - just file with bunch of commands that can be interpreted (e.g. python, lisp scripts, or bash scripts)
- They can be used both for automation some small routine tasks and as large programs
- Shell scripts names ends with .bash , .sh , zsh
- But as far as we know, the Linux system doesn't use the extensions to identify the type of the file. How to run a script as a usual program?
- One of possible ways to run the script is to give it as a parameter to the command interpreter

```
username $ bash ./my_first_script.sh
> some output
```

Scripts

- Or make the file executable (add x or 111 flag)

```
username $ chmod +x ./my_first_script
username $ ./my_fist_script
> some output
```

- Actually, there is a mistake. If you do it in the following way, the script will be run from the running shell. But we want to make Python, Ruby. Scheme also executable. Unfortunately, the bash interpreter can not execute them...

Shebang

- Shebang - the name of the character sequence at the very first line of the script (line #1, that is important!) which specifies the absolute path to the interpreter.

```
#!/bin/bash
echo Hello world
```

- Or also possible variant (to search for the interpreter in PATH (about PATH in next presentation))

```
#!/bin/env python3
print("Hello world")
```

Variables

- User can define and use variables at the environment:

```
username $ my_var="This is my var"
```

- There is no space allowed on either side of the "=" sign
- There are local and global variables
- When we "export" the var, it becomes available in all applications run from the current session, which means global
- For programs running from the shell, global variables are the same as environment variables
- To get the variable, ${} syntax is used:

```
username $ echo ${my_var}
> This is my var
```

Script arguments

- Scripts (actually, all programs) are useless without some input from the user
- Just to remember, the default program calling syntax:

Examples

```
program_name [option]... [arguments]...
```

- So to access the arguments:
  - ${@} - all arguments
  - ${#} - number (length) of all arguments
  - ${0} - script name
  - ${1} , ${2} ... - other script parameters

Quoting

- As far as there are a lot of special characters, how to use them if you want to print them?
- Quoting or escaping is used to make a regular symbol from a special one
- \ is used in a lot of programming languages

```
username $ echo \$SHELL
> $SHELL
username $ echo ${SHELL}
> /bin/zsh
```

- ' - single quotes, make "escaped" everything inside
- " - double quotes, allow variables expansion

Conditionals. If

- The standard if with both one and many branches

```
if [ condition ]; then
    action;
elif [ condition 2 ]; then
    action 2;
...
else ; \newline
    actionx
fi
```

- Example:

```
if [ "$(whoami)" != 'root' ]; then
    echo "Operation not permitted for non superusers"
    exit 1;
fi
```

Conditionals

- There are a lot of conditions regarding different types:
- For numbers:
  - -lt - <
  - -gt - >
  - -le - <=
  - -ge - >=
  - -eq - ==
  - -ne - !=
- Logical operations
  - -a - and
  - -o - and
  - ! - not

Conditionals

- [-d FILE] - True if FILE exists and is a directory
- [-e FILE] - True if FILE exists
- [-f FILE] - True if FILE exists and is a regular file
- [-r FILE] - True if FILE exists and is readable
- [-s FILE] - True if FILE exists and has a size greater than zero
- [-w FILE] - True if FILE exists and is writable.
- [-x FILE] - True if FILE exists and is executable
- [-z STRING] - True of the length if ”STRING” is zero
- [-n STRING] - True if the length of ”STRING” is non-zero
- For strings the most popular cooperators are used
- [[ ]STRING1 == STRING2]]
- the same about <, >, !=, etc

Conditionals. Case

- If you have an experience in other programming languages, you should definitely know the case statement. Here is the example:

```
case [expression] in
  [pattern_1])
    actions
    ;;
  [pattern_2])
    actions
    ;;
    ...
  [pattern_n])
    actions
    ;;
  *)
    actions
    ;;
esac
```

Loops

If you know any programming language, you definitely know, what loop is. Here is bash syntax for them

```
for [var] in [some iterable]
do
    statements
done
```

```
while [ condition to be True ]
do
    statements
done
```

```
until [ condition to be False ]
do
    statements
done
```

Loop examples

```bash
#!/usr/bin/env bash
for thing in "$@"
do
    echo you typed ${thing}.
done
```

```bash
myvar=0
while [ $myvar -ne 10 ]
do
    echo $myvar
    myvar=$(( $myvar + 1 ))
done
```

More for loop examples

```bash
# Three-expression for loop
for ((i = 0; i < 5; i++)); do
    echo $i
done

# infinity loop
for (( ; ; )); do
    ...
done

# break and continue
for ((i = 0; i < 5; i++)); do
    if ((i % 2 == 0)); then continue; fi
    if ((i > 7)); then break; fi
    echo "$i"
done
```

Arithmetic

- In bash arithmetic is a little bit tricky
- To evaluate the arithmetic expression, let can be used

```
username $ a=4+5
username $ echo $a
> 4+5
username $ a=((4+5))
username $ echo $a
> 9
```

- +, -, *, /, ** (power), %, ++, – commands can be used

Arrays

- As all other programming languages, Bash has arrays
- Here is the syntax of using and introduction of the arrays (in ZSH it is a little different)

```bash
# Syntax:
array_name=(el1 el2 el ...) # values are space-separated
# Example:
array_name=('Apple' 'Lemon') #
array_name[6]='Orange' # You can index elements even
# "out of bounds" all elements between them will be just empty
array_name[-1]='Milk' # reverse indexing is also possible,
# as in Python
echo $array_name[*]
```

```
username $ ./script.sh
> Apple Lemon Milk
```

Arrays

- There are a lot of ways how to iterate through the array and other more important commands

```
for i in ${!array_name[*]}; do # iteration on the array indexes
    echo ${array_name[i]}
done
for i in ${array_name[*]}; do # iteration on the array elements
    echo $i
done
echo "${#array_name[@]}" # print array's length
array_name+=(Melon) # append element to the array
```

- Full list can be found here

Functions

- Function declarations can be like this:

```
function_name() {
    echo $1, $2 # You can access function arguments by it's
    positions, not names
    echo $@ # number of arguments
}
function function_name { # One more way to declare same functions
    echo $1, $2
}
```

Namespace

- For example in C, the local variable (defined inside some scope) life cycle ends with the end of the scope
- On the opposite in Bash all declared variables overwrite the global one with the same names
- to prevent this, use local definitions

```
#!/usr/bin/env bash
myvar="hello"
globalvar="global hello"
myfunc() {
    local globalvar="four five six"
    myvar="one two three"
}
myfunc
echo $myvar
echo $globalvar
```

```
username $ ./start.sh
> one two three
  global hello
```

Sources

Sources

- Bash presentation for Operating systems course, UCU, Oleg Farenyuk (only from UCU domain)
- Linux basics from the founder of Gentoo, Daniel Robbins, Chris Houser, Aron Griffis
- Bash basics, Daniel Robbins, Chris Houser, Aron Griffis
- Scripting OS X