LW-04. LINUX SHELL. FILES GLOBBING & STREAMS REDIRECTION.

1. TARGET.

- Learn to use shell file globbing (wildcard);
- Learn basic concepts about standard UNIX/Linux streams redirections;
- Acquire skills of working with filter-programs.
- Get experience in creating composite commands that have a different functional purpose than the original commands.

2. ASSIGNMENTS.

NOTE. Start Your **UbuntuMini** Virtual Machine on your VirtualBox. You need only Linux Terminal to complete the lab tasks.

Before completing the tasks, make a Snapshot of your Virtual Linux. If there are problems, you can easily go back to working condition!

2.0. Create new User account for this Lab Work.

- Login as student account (**user with sudo permissions**).
- Create new user account, example stud. Use **adduser** command. (NOTE. You can use the command “userdel –rf stud” to delete stud account from your Linux.)

```
$ sudo adduser stud
```

- Logout from student account (**logout**) and login as stud.

2.1. Shell File Globbing Study.

2.2. File Globbing Practice. (Fill in a Table 1 and Table 2)

2.3. Command I/O Redirection Study.

2.4. Redirection Practice. (Fill in a Table 3 and Table 4)

3. Report.

The report is provided electronic form with Report Blank Form (use a docx).

Report for Lab Work 04: Linux Shell. Files Globbing & Streams Redirection

Student Name	Student ID (nV)	Date

3.1. Insert Completing Table 1. File globbing understanding.

3.2. Insert Completing Table 2. File globbing creation.

3.3. Insert Completing Table 3. Command I/O redirection understanding.

3.4. Insert Completing Table 4. Command piping understanding.

4. Guidelines.

4.1. Shell File Globbing Study.

4.1.1. File Globbing Wildcards Description.

File globbing (or dynamic filename generation) is a feature provided by the UNIX/Linux shell to represent multiple filenames by using special characters called wildcards with a single **file** name. A wildcard is essentially a symbol which may be used to substitute for one or more characters. See the man page of **glob(7)** for more information. (This is part of LPI topic 1.103.3. - Linux Professionals Institute).

* asterisk
? question mark
[ ] square brackets
! exclamation mark
a-z and 0-9 ranges
\, ', " preventing file globbing
{ } braces

1. Asterisk: *.

The asterisk * is interpreted by the shell as a sign to generate filenames, matching the asterisk to any combination of characters (even none). When no path is given, the shell will use filenames in the current directory.

```
$ ls
file1  file2  file3  File4  File55  FileA  fileab  Fileab  FileAB  fileabc
$ ls File*
File4  File55  FileA  Fileab  FileAB
$ ls file*
file1  file2  file3  fileab  fileabc
$ ls *ile55
File55
$ ls F*ile55
File55
$ ls F*55
File55
$
```

2. Question mark: ?.

The question mark **?** is interpreted by the shell as a sign to generate filenames, matching the question mark with exactly one character.

```
$ ls
file1  file2  file3  File4  File55  FileA  fileab  Fileab  FileAB  fileabc
$ ls File?
File4  FileA
$ ls Fil?4
File4
$ ls Fil??
File4  FileA
$ ls File??
File55  Fileab  FileAB
$
```

3. Square brackets: [and].

The square bracket **[** is interpreted by the shell as a sign to generate filenames, matching any of the characters between **[** and the first subsequent **]**. The order in this list between the brackets is not important. Each pair of brackets is replaced by exactly one character.

```
$ ls
file1  file2  file3  File4  File55  FileA  fileab  Fileab  FileAB  fileabc
$ ls File[5A]
FileA
$ ls File[A5]
FileA
$ ls File[A5][5b]
File55
$ ls File[a5][5b]
File55  Fileab
$ ls File[a5][5b][abcdefghijklm]
ls: File[a5][5b][abcdefghijklm]: No such file or directory
$ ls file[a5][5b][abcdefghijklm]
fileabc
```

```
$
```

4. Exclamation mark: !.

You can also exclude characters from a list between square brackets with the exclamation mark **!**. And you are allowed to make combinations of these **wild cards**.

```
$ ls
file1  file2  file3  File4  File55  FileA  fileab  Fileab  FileAB  fileabc
$ ls file[a5][!Z]
fileab
$ ls file[!5]*
file1  file2  file3  fileab  fileabc
$ ls file[!5]?
fileab
$
```

5. Ranges: A-Z, a-z, 0-9, …

The bash shell will also understand ranges of characters between brackets.

```
$ ls
file1  file3  File55  fileab  FileAB   fileabc
file2  File4  FileA   Fileab  fileab2
$ ls file[a-z]*
fileab  fileab2  fileabc
$ ls file[2-4]
file1  file2  file3
$ ls file[a-z][a-z][0-9]*
fileab2
$
```

6. Preventing file globbing: \, ‘ ’, “ ”.

The screenshot below should be no surprise. The **echo *** will echo a * when in an empty directory. And it will echo the names of all files when the directory is not empty.

```
$ mkdir test42
$ cd test42
$ echo *
*
$ touch file42 file33
$ echo *
file33 file42
$
```

Globbing can be prevented using quotes or by escaping the special characters, as shown in this screenshot.

```
$ echo *
file33 file42
$ echo \*
*
$ echo '*'
*
$ echo "*"
*
$
```

7. Braces: { }.

Is often mentioned in conjunction with shell search patterns, even though it is really just a distant relative. In general, a word on the command line that contains several comma-separated pieces of text within braces is replaced by as many words as there are pieces of text between the braces, where in each of these words the whole brace expression is replaced by one of the pieces.

```
$ ls
red.txt yellow.txt blue.txt black.txt
$ ls {red,yellow,black}.txt
red.txt yellow.txt black.txt
```

This replacement is purely based on the command line text and is completely **independent of the existence or non-existence** of any files or directories - unlike search patterns (before), which always produce only those names that actually exist as path names on the system.

You can have more than one brace expression in a word, which will result in the cartesian product, in other words all possible combinations (Cartesian Product):

```
$ ls
a1.dat a2.dat a3.dat a4.dat b1.dat b2.dat b3.dat c1.dat c2.dat c3.dat d1.dat d2.dat
$ ls {a,b,c}{1,2,3}.dat
a1.dat a2.dat a3.dat b1.dat b2.dat b3.dat c1.dat c2.dat c3.dat
$
```

This is useful, for example, to create new directories systematically; the usual search patterns cannot help there, since they can only find things that already exist:

```
$ mkdir -p revenue/200{8,9}/q{1,2,3,4}
$ tree revenue
$ rm -rd revenue
$
```

4.1.2. File Globbing Practice.

4.1.2.1. Fill in the table.

Table 1. File globbing understanding.

Nr	Task Description	Your Answer
0	Create a test directory and enter it. Create the following files: `prog.c prog1.c prog2.c progabc.c prog` `p.txt p1.txt p21.txt p22.txt p22.dat` Which of these file names match the search patterns?	Write directory and files creation commands
1	prog*.c	
2	prog?.c	
3	p?*.txt	
4	p[12]*	
5	p*	
6	*.*	

4.1.2.2. Fill in the table.

Table 2. File globbing creation.

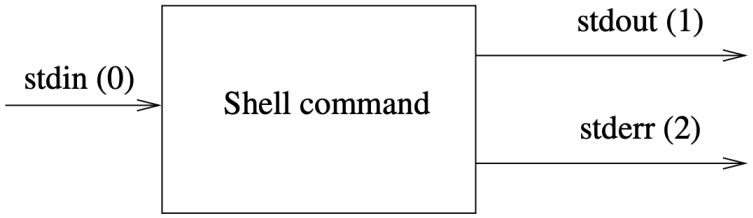
Nr	Task Description	Your Answer
0	Create a test directory and enter it. Create the following files: `file1` `file10` `file11` `file2` `File2` `File3` `file33` `fileAB`	Write directory and files creation commands

	`` filea fileA fileAAA file(file 2 #(the last file name has 6 characters including a space) ``` What search patterns match these names?	
1	List (with ls) all files starting with file	
2	List (with ls) all files starting with File	
3	List (with ls) all files starting with file and ending in a number	
4	List (with ls) all files starting with file and ending with a letter	
5	List (with ls) all files starting with file and having a digit as fifth character	
6	List (with ls) all files starting with file and having a digit as fifth character and nothing else	
7	List (with ls) all files starting with a letter and ending in a number	
8	List (with ls) all files that have exactly five characters	
9	List (with ls) all files that start with f or F and end with 3 or A	
10	List (with ls) all files that start with f have i or R as second character and end in a number	
11	List (with ls) all files that do not start with the letter F	
12	List (with ls) all files that do not ending with a letter or number	
13	Create new directories tree systematically for CurrentYear/MonthsNumber **use composite command construction from elements: mkdir, `date +%G` and {1,2,3…,12}**	
14	Create new directories tree systematically for CurrentMonthsName/DaysNumbers	

4.2. COMMAND I/O REDIRECTION STUDY.

4.2.1. COMMAND I/O REDIRECTION DESCRIPTION.

The bash shell, and the other shells employed by Linux allow the user to specify what should happen to the input and output of programs that run. The bash shell has three basic streams; it takes input from **stdin** (stream **0**), it sends output to **stdout** (stream **1**) and it sends error messages to **stderr** (stream **2**). The drawing below has a graphical interpretation of these three streams.



The keyboard often serves as **stdin**, whereas **stdout** and **stderr** both go to the display. Redirections with pipes (|) are processed first, and other redirections (with > and <) are processed from left to right.

The following redirections are possible

Redirection	Effect of redirection
command > file	Output of command goes to file
command > /dev/null	Send output to null. (Discard the output.)
command > /dev/tty1	Send output to terminal number one. (Require root permission)
command 2> file	Errors and diagnostic messages from the command go to a file
command >> file	Output of a command is added to a file (append)
command 2>> file	Errors/diagnostic of a command is added to a file (append)
command > file 2>&1	Output and errors/diagnostics go to a file
command >& file	
command &> file	
command < file	Command reads input from a file
command << here document	Command reads input from document-is-here construction
command <<< string here	Command reads input from string-is-here construction
command1 | command2	Output from command1 is input for command2 (piping)
command | tee file	Output of a command is copied to stdout and to one or more files

1. Stdout: >.

Stdout can be redirected with a **greater than** sign. While scanning the line, the shell will see the **>** sign and will clear the file.

The **>** notation is in fact the abbreviation of **1>** (**stdout** being referred to as stream **1**).

```
$ echo It is cold today!
It is cold today!
$ echo It is cold today! > winter.txt
$ cat winter.txt
It is cold today!
$
```

Note that the bash shell effectively **removes** the redirection from the command line before argument 0 is executed. This means that in the case of this command: "echo hello > greetings.txt", the shell only counts two arguments (echo = argument 0, hello = argument 1). The redirection is removed before the argument counting takes place.

2. Noclobber prevention of erased exist output file.

Output file is erased

While scanning the line, the shell will see the > sign and **will clear the file**! Since this happens before resolving **argument 0**, this means that even when the command fails, the file will have been cleared!

```
$ cat winter.txt
It is cold today!
$ abracadabra It is cold today! > winter.txt
-bash: abracadabra: command not found
$ cat winter.txt
$
```

Noclobber set - unset

Erasing a file while using > can be prevented by setting the **noclobber** option.

```
$ cat winter.txt
It is cold today!
$ set -o noclobber
$ echo It is cold today! > winter.txt
-bash: winter.txt: cannot overwrite existing file
$ set +o noclobber
```

Overruling noclobber with >|

```
$ set -o noclobber
$ echo It is cold today! > winter.txt
-bash: winter.txt: cannot overwrite existing file
$ echo It is very cold today! >| winter.txt
$ cat winter.txt
It is very cold today!
```

3. Append: >>.

Use **>>** to **append** output to a file.

```
$ echo It is cold today! > winter.txt
$ cat winter.txt
It is cold today!
$ echo Where is the summer ? >> winter.txt
$ cat winter.txt
It is cold today!
Where is the summer ?
$
```

4. Stderr: 2>.

Redirecting **stderr** is done with **2>**. This can be very useful to prevent error/diagnostic messages from cluttering your screen.

The screenshot below shows redirection of **stdout** to a file, and **stderr** to **/dev/null**. Writing **1>** is the same as >.

```
$ find / > allfiles.txt 2> /dev/null
$
```

5. Stderr and stdout: 2>&1.

To redirect both **stdout** and **stderr** to the same file, use **2>&1**.

```
$ find / > allfiles_and_errors.txt 2>&1
$
```

Note that the order of redirections is significant. For example, the command

```
ls > dirlist 2>&1
```

directs both standard output (file descriptor 1) and standard error (file descriptor 2) to the file dirlist, while the command

```
ls 2>&1 > dirlist
```

directs only the standard output to file dirlist, because the standard error made a copy of the standard output before the standard output was redirected to dirlist.

6. Output redirection and pipes: >, >>, |.

By default you cannot grep inside **stderr** when using pipes on the command line, because only **stdout** is passed.

```
$ rm file42 file33 file1201 | grep file42
rm: cannot remove ‘file42’: No such file or directory
rm: cannot remove ‘file33’: No such file or directory
rm: cannot remove ‘file1201’: No such file or directory
$
```

With **2>&1** you can force **stderr** to go to **stdout**. This enables the next command in the pipe to act on both streams.

```
$ rm file42 file33 file1201 2>&1 | grep file42
rm: cannot remove ‘file42’: No such file or directory
$
```

You cannot use both **1>&2** and **2>&1** to switch **stdout** and **stderr**.

```
$ rm file42 file33 file1201 2>&1 1>&2 | grep file42
rm: cannot remove ‘file42’: No such file or directory
$ echo file42 2>&1 1>&2 | sed 's/file42/FILE42/'
FILE42
$
```

You need a third stream to switch stdout and stderr after a pipe symbol.

```
$ echo file42 3>&1 1>&2 2>&3 | sed 's/file42/FILE42/'
file42
$ rm file42 3>&1 1>&2 2>&3 | sed 's/file42/FILE42/'
rm: cannot remove ‘FILE42’: No such file or directory
$
```

7. Joining stdout and stderr: &>.

The **&>** construction will put both **stdout** and **stderr** in one stream (to a file).

```
$ rm file42 &> out_and_err
$ cat out_and_err
rm: cannot remove ‘file42’: No such file or directory
$ echo file42 &> out_and_err
$ cat out_and_err
file42
```

8. Stdin redirection: <.

Redirecting **stdin** is done with < (short for 0<).

```
$ cat < text.txt
one
two
$ tr 'onetw' 'ONEZZ' < text.txt
ONE
ZZO
```

9. Document here: <<.

The **here document** (sometimes called here-is-document) is a way to append input until a certain sequence (usually EOF or other marker) is encountered. The **EOF** marker can be typed literally or can be called with Ctrl-D.

```
$ cat <<EOF > text.txt
> one
> two
> EOF
$ cat text.txt
one
two
```

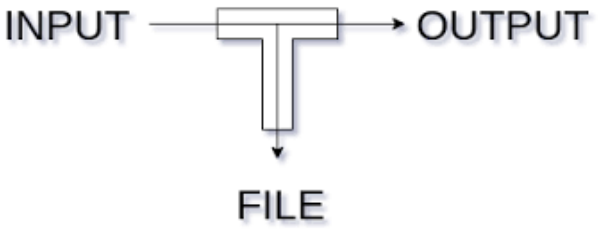
10. String here: <<<.

The **here string** can be used to directly pass strings to a command. The result is the same as using **echo string | command** (but you have one less process running).

```
$ base64 <<< http://sys.academy.lv
aHR0cDovL3N5cy5hY2FkZW15Lmx2Cg==
$ base64 <<< https://sys.academy.lv
aHR0cHM6Ly9zeXMuYWNhZGVteS5sdgo=
$
$ base64 -d <<< aHR0cHM6Ly9zeXMuYWNhZGVteS5sdgo=
https://sys.academy.lv
$
$ base64 -d <<< aHR0cDovL3N5cy5hY2FkZW15Lmx2Cg==
http://sys.academy.lv
$
```

See rfc 3548 for more information about **base64**.

11. tee.



Command tee reads data from standard input and writes them, firstly, to standard output, and secondly, to one or more files specified in the command; if there is no file, then it is created; if there is, then it is overwritten.

To add to the file, use the -a switch (from the word append - add): tee -a file.

```
$ ls | tee -a file
$ ls | tee file1 file2 > file3
```

4.2.2. Command I/O Redirection Practice.

4.2.2.1. Fill in the table 3.

When building pipelines (command1 | command2), for which of the 32 commands below the concepts are appropriate:

- both stdin and stdout;
- only stdin;
- only stdout;
- no stdin nor stdout?

```
pwd, cd, ls, mkdir, rmdir, rm, mv, ln, du, touch, cat, cp, find, more, tail, head, file,
banner, who, id, uname, date, cal, wc, chsh, exec, echo, export, readonly, set, unset, passwd.
```

Note, a construction below is not considered stdout:

```
$ command --help > file-out
```

Table 3. Command I/O Redirection understanding.

Have both stdin and stdout	Have only stdin	Have only stdout	No stdin nor stdout
cat		pwd	cd

4.2.2.2. Fill in the table 4.

For the commands (who, wc, pwd, cat, uname, id, echo, banner, passwd, set, more) analyze the existence of all possible combinations of pipe pairs of the forms: **command1 | command2.**

```
$ # Example for checkin
$ who
web      pts/0        2025-02-03 11:31 (192.168.111.1)
$ who | wc -l
1
$
```

Table 4. Command piping understanding.

c1\c2	who	wc	pwd	cat	uname	id	echo	banner	passwd	set	more
who	-	+	-	+	-	-	-	-	-	-	+
wc											
pwd											
cat											
uname											
id											
echo											
banner											
passwd											
set											
more											