Advanced CLI: I/O & Text Processing

Session 5

Advanced CLI: I/O & Text Processing

Session 5

Objectives:

- Understand standard input, output, and error (stdin, stdout, stderr)
- Redirect input and output using >, >>, and <
- Chain commands with pipes |
- View and filter text using cat, less, more, head, tail
- Use grep for searching
- Use wc, sort, cut, and uniq for text analysis
- Practice combining commands through hands-on labs

What is stdin, stdout, stderr

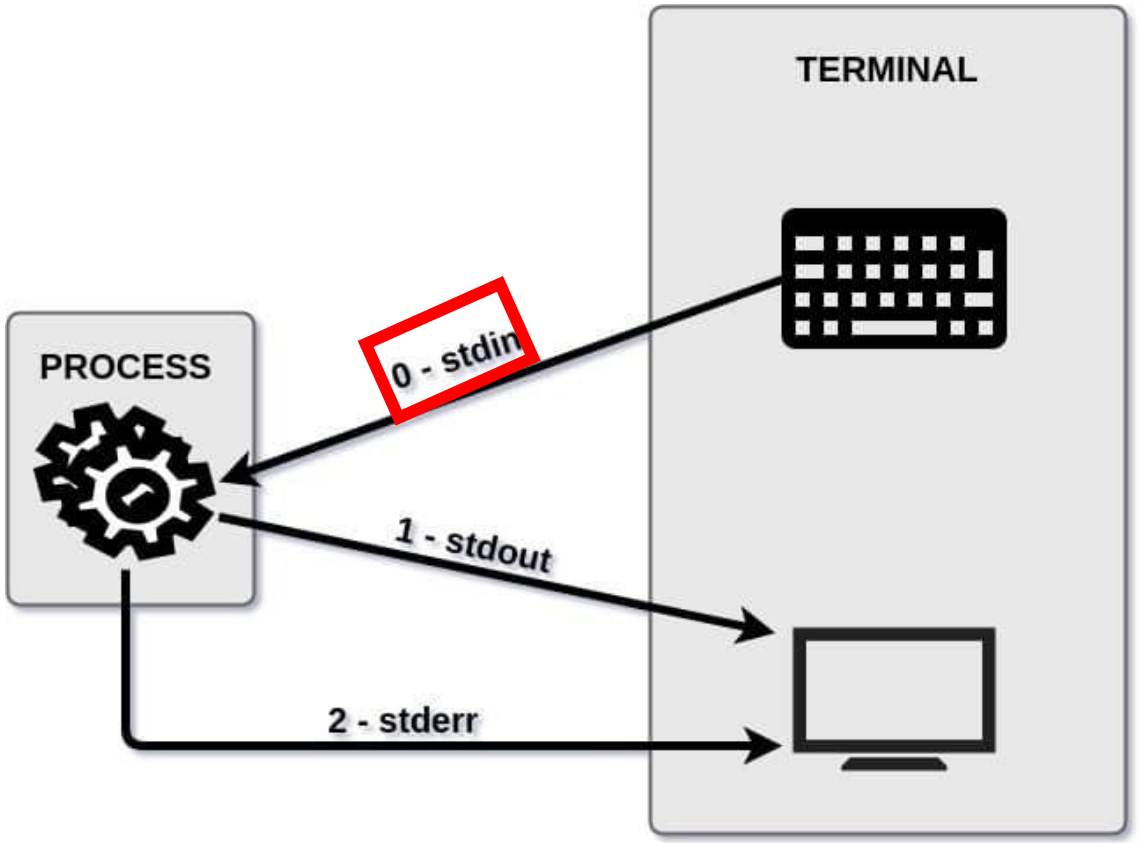
Linux has an interesting concept where basically all input and output (which are text) are actually streams of data/text. Like plumbing pipes where you can connect and disconnect sections to redirect water to different places, so you can you connect and disconnect streams of data.

Standard streams are input and output (I/O) communication channels between a program and it's environment.

What is stdin, stdout, stderr

Three standard streams:

- **stdin** — standard input (keyboard) , the command line uses the stdin to receive inputs. If you type the command **ls** to list a directory content, you use the stdin.

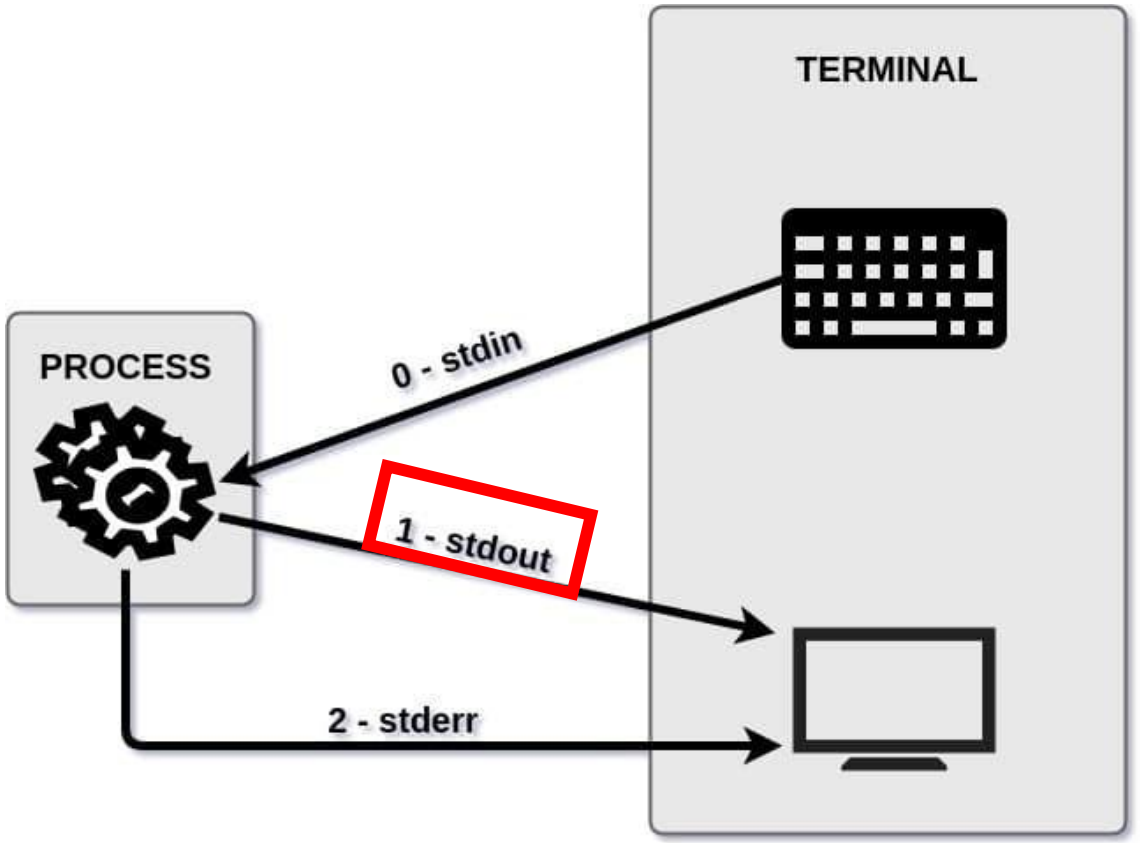


What is stdin, stdout, stderr

Three standard streams:

- **stdin** — standard input

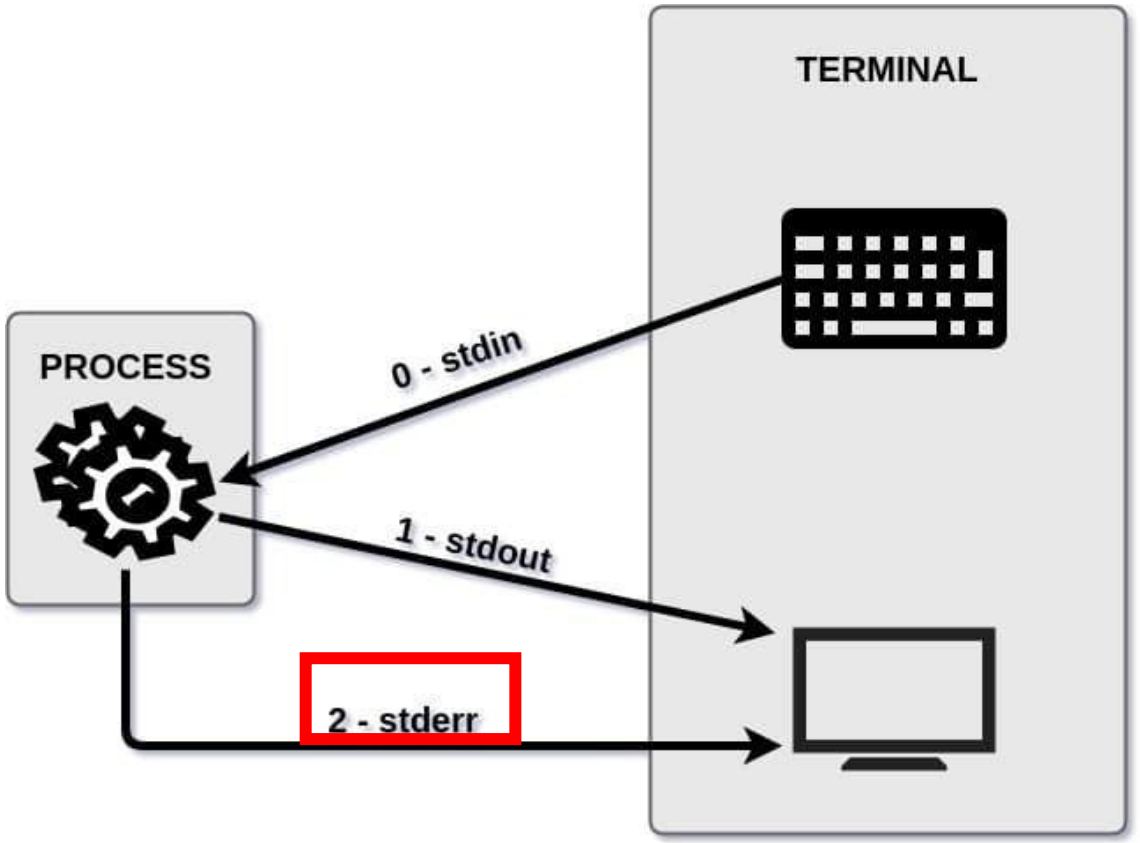
- **stdout** — standard output (terminal screen) , this is the counterpart of the stdin. For « ls » the listing of the directories will be displayed by the stdout channel in your screen.



What is stdin, stdout, stderr

Three standard streams:

- **stdin** — standard input
- **stdout** — standard output
- **stderr** — standard error output (terminal screen) , this channel is also an output, but specific to any error messages or diagnostics the command may return. For example, if I run my ls command in a path that doesn't exist, the "No such file or directory" output will be returned through the stderr channel

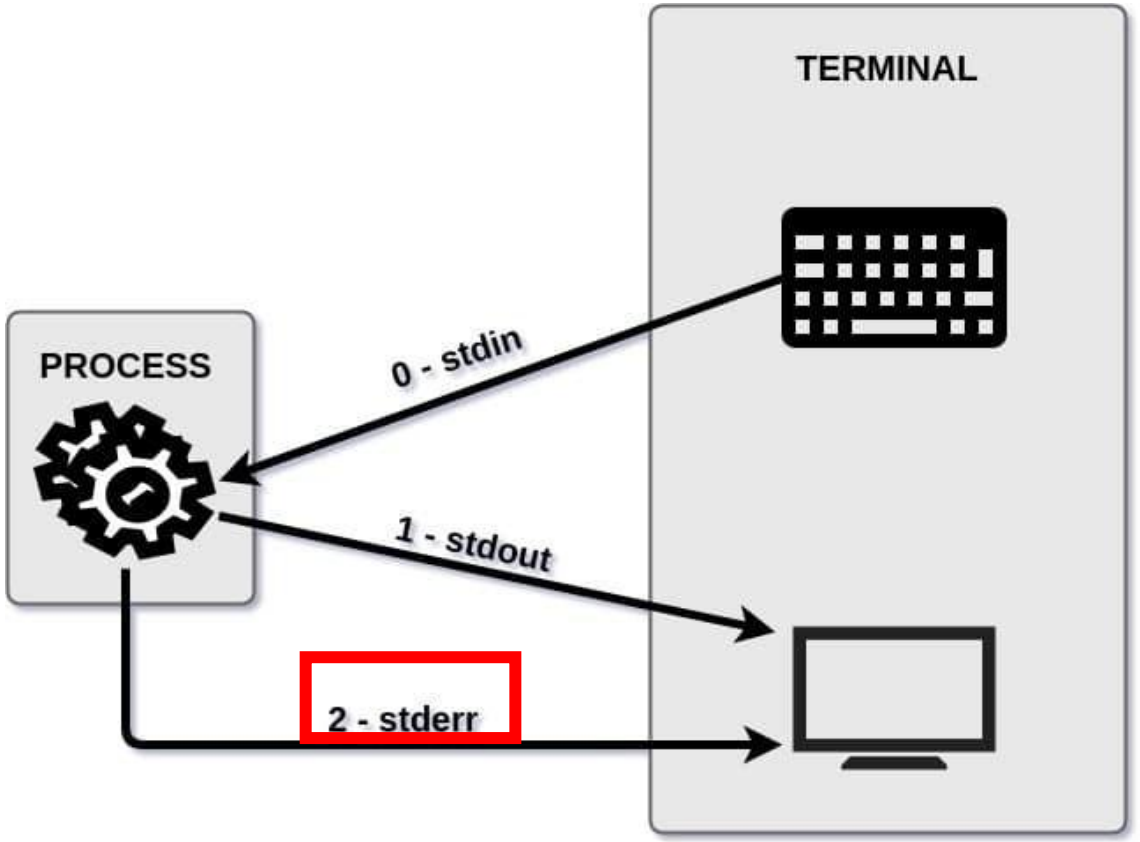


Output redirections

Three standard streams:

- **stdin** — standard input
- **stdout** — standard output
- **stderr** — standard error output (terminal screen) , this channel is also an output, but specific to any error messages. If I run « ls » in a path that doesn't exist, the "No such file or directory" output will be returned through the stderr channel

Despite being two different channels, stdout and stderr are displayed in the same way on the terminal.



Stream redirection

Stream output can be redirected to another stream, file, or command:

- **>** is frequently used in shell scripting to redirect a stream (typically stdout) into a file or another destination. If the target is a file, it will overwrite its contents.

- **>>** behaves similarly but appends the output to the file instead of replacing it.

- **<** is used in the opposite direction: it redirects input from a file to a command, substituting for keyboard input (stdin).

Stream redirection

Examples:

```
ls > list.txt      # overwrite the file
ls >> list.txt     # append to the file
wc -l < list.txt   # count lines by reading from file as stdin
command > output.txt 2>&1# Combine stdout and stderr into one file
```

- Write "Welcome" into greeting.txt (overwrites if it exists) using echo
- Append another line (“to the course”) to the same file
- Redirect standard error separately into error.log. Use ls and non existing file
- Read a greeting as input for wc command
- Try to use ls and combine stdout and stderr into 1 file

Output redirection

Examples:

```
ls > list.txt      # overwrite the file
ls >> list.txt     # append to the file
wc -l < list.txt   # count lines by reading from file as stdin
```

- Write "Welcome" into greeting.txt (overwrites if it exists) using echo

```
echo "Welcome" > greeting.txt
```

- Append another line (“to the course”) to the same file

```
echo "to the course" >> greeting.txt
```

- Redirect standard error separately into error.log. Use ls and non existing file

```
ls non_existing_file 2> error.log
```

- Read a greeting as input for wc command

```
wc -w < greeting.txt
```

- Try to use ls and combine stdout and stderr into 1 file

```
ls /etc /doesnotexist
```

Stream redirection

Examples:

- Create a file name.txt with three name:

Theo
Adele
Daniela

- Try sort names.txt

- Try sort < names.txt

Stream redirection

Examples:

- Create a file name.txt with three name: Adele, Daniela, Theo

cat > name.txt
Adele
Daniela
Theo

Or echo -e "Adele\nDaniela\nTheo" > name.txt.

Stream redirection

Examples:

- Create a file name.txt with three name: Adele, Daniela, Theo
- Try sort names.txt
- Try sort < names.txt

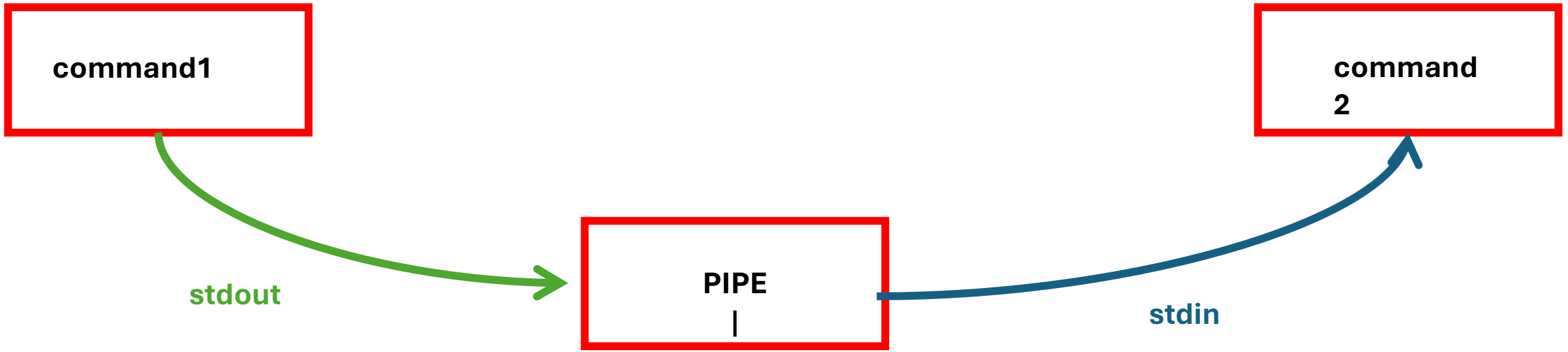
It is the same!!

Command	Reads from	Useful when…
sort names.txt	File directly	Simpler, preferred for file input
sort < names.txt	Standard input	For scripting, pipelines, stdin-based tools

Pipe |

Pipe |

Pipes allow you to connect the output of one command directly as input to another. This is a core concept in building powerful command-line workflows:



Pipe |

Exemples:

- ls /etc | less
- cat /etc/passwd | wc -l
- cat name.txt | sort
- ls /usr/bin | tail -n 5

Pipe |

Exemples:

- ls /etc | less # View long directory listings one page at a time
- cat /etc/passwd | wc -l # Count how many lines (users) in passwd file
- cat name.txt | sort # Sort the lines of file.txt
- ls /usr/bin | tail -n 5 # Show the last 5 entries in /usr/bin

Text viewer: less, cat, more

- **cat** — display full content

- **less, more** — scrollable viewers

Examples:

- cat file.txt
- less file.txt
- more file.txt

Text viewer: head, tail, wc

- **head** — first lines
- **tail** — last lines
- **wc** — count lines, words, characters

Examples:

- head –n 5 file.txt (print the first 5 lines)
- wc -l file.txt, wc -c file.txt

Sort, cut, uniq

- **sort** — sort lines alphabetically or numerically
- **uniq** — remove duplicates (use with sort)
- **cut** — extract columns/fields:
  - -d Specifies the delimiter (e.g., : or ,)
  - -f Specifies the field number(s) to extract
  - -c Extracts characters by position

Examples:

- cut -d":" -f1 /etc/passwd | sort | uniq

Sort, cut, uniq

- **sort —** sort lines alphabetically or numerically
- **uniq —** remove duplicates (use with sort)
- **cut —** extract columns/fields:
  - -d Specifies the delimiter (e.g., : or ,)
  - -f Specifies the field number(s) to extract
  - -c Extracts characters by position

Examples:

- cut -d":" -f1 /etc/passwd | sort | uniq
  - Reads the file /etc/passwd
  - Uses : as the field delimiter
  - Extracts the **first field** from each line, which is the **username**
  - **| sort**
  - Sorts the usernames alphabetically
  - **| uniq**
  - Removes any duplicates (although /etc/passwd usually doesn't have username duplicates)

Search with grep

- **grep —** is for filtering/searching'

Examples:

- grep "root" /etc/passwd and cat /etc/passwd | grep root
- Create a logs.txt with

“Line 1: Reading configuartion... done.
Line 2: ERROR: Falied to parse 'config.ini' — missing '=' on line 23.
Line 3: error connecting to the databse. Retrying...
Line 4: ERROR: Timeout while waiting for responce.
Line 5: Disk check: 1 warning, 3 errors found.
Line 6: error: Invalid user input detected.
Line 7: ERROR 403: Forbiden acces to /admin/panel.
Line 8: error in script.sh line 45: unexpected `fi`.
Line 9: ERROR: Unable to locate dependency 'libsys.so'.
Line 10: error writing log fle — permission denied.
Line 11: Retrying... error persists.
Line 12: All retries failed due to critical ERROR.

- grep -i "error" logs.txt (What does the -i option?)

Search with grep

- **grep —** is for filtering/searching'

Examples:

- grep "root" /etc/passwd and cat /etc/passwd | grep root

- grep -v "localhost" /etc/hosts # Show lines not containing 'localhost'
- grep -n "root" /etc/passwd # Show matching lines with line numbers
- grep -c "/bin/bash" /etc/passwd # How many line match
- grep -r "TODO" ~/projects # Search recursively in a folder
- grep -l "password" *.conf # List only files containing 'password'

Linux commands

awk	allows manipulation of text
bg	place suspended job into background
cal	display calendar
cat	view contents of a file
cd	change directory
chmod	change permissions on a file/directory
cp	copy a file
cut	extract a field of data from text output
diff	compare files line by line
echo	output text to the terminal or to a file
emacs	text editor
fg	bring suspended job to foreground
file	display file type
find	search for files
grep	search a file or command output for a pattern
head	view beginning of file
history	display list of most recent commands
less	scroll forward or back through a file
ln	create a link to a file
ls	list files in a directory
man	view information about a command
mkdir	make directory
more	scroll through file a page at a time
mv	change the name of a file (move)
nano/pico	text editors
printenv	display shell variables
ps	show current process information
pwd	print current working directory
rm	delete or remove a file
rmdir	delete or remove a directory
sed	stream editor
sleep	pause
sort	perform a sort of text
tail	view end of the file
touch	create an empty file or update timestamps
tr	character substitution tool
uniq	remove identical, adjacent lines
vi/vim	text editor
wc	print number of lines, words or characters
which	shows full path of a command
whoami	displays username

Linux exercices

- Certain characters have special meanings or functions in the shell

Character	Meaning	Example
' (Single Quote)	Used to enclose **literal text**. No variable expansion or special characters inside.	'Hello $USER' → Displays Hello $USER (no variable expansion).
" (Double Quote)	Similar to single quotes, but allows **variable expansion** and **command substitution**.	"Hello $USER" → Displays Hello <username> (variable expansion happens).
`cmd` (Backticks)	Used for **command substitution**, allowing the output of a command to be part of another.	`date` → Prints the current date and time by running the date command.
`	` (Pipe)	**Pipes** the output of one command to another command.
> (Redirect)	**Redirects** the output of a command to a file, overwriting the file if it already exists.	echo "Hello" > hello.txt → Writes Hello into hello.txt.
; (Semicolon)	**Separates multiple commands** on the same line. The commands are executed sequentially.	echo "Hello"; ls → Prints Hello and then lists files sequentially.

Lab

Objective: Build a complete command-line pipeline using redirection, piping, and text processing tools to extract structured information from real or synthetic system logs.

Scenario: Your team is helping a system administrator understand the content of a large text log. You will create a report that includes search, summary statistics, and file redirection.

- **Challenge-Based Instructions:** Your working folder should be: ~/cli_project

Output files to use: errors.txt, warnings.txt, summary.txt, final_report.txt

1. Copy a log file to your working folder.
2. Extract all lines that contain the words error and warn, and save them to two different files.
3. Find how many lines are in each file. Then, try counting the number of words and characters too.
4. Identify the 5 earliest and 5 latest log messages for errors and warnings.
5. From each set, extract the timestamps (assume they are the first 2 words per line). Count how often each timestamp occurs.
6. Combine the results into a file called final_report.txt. Add a line showing the total number of warn messages.
7. Bonus: Try doing step 5 in one line using only pipes and no intermediate file

Lab

❓ **Questions to Answer:**

How many lines contain error and warn?

What is the most frequent timestamp for error messages?

What is the longest line found in errors.txt?

What is the difference between using > and >> in your summary file?

Which options of grep and wc did you find most useful and why?

What did using a pipeline help you do more efficiently?

Lab

Objective: Build a complete command-line pipeline using redirection, piping, and text processing tools to extract structured information from real or synthetic system logs.

Scenario: Your team is helping a system administrator understand the content of a large text log. You will create a report that includes search, summary statistics, and file redirection.

- **Step-by-step Tasks (Explore Options!):**

1.Prepare Environment:

```
mkdir ~/cli_project
cd ~/cli_project
cp /var/log/syslog .
```

2.Search for key terms:

```
 grep -i "error" syslog > errors.txt
grep -i "warn" syslog > warnings.txt
```

Lab

3.Play with wc and make stats:

```
wc -l errors.txt > summary.txt
wc -w errors.txt >> summary.txt
wc -c errors.txt >> summary.txt
wc -L errors.txt >> summary.txt
```

Bonus .Play with grep:

```
 grep -v "info" syslog > without_info.txt   # Exclude matches
grep -n "warn" syslog > numbered_warns.txt # Show line numbers
grep -c "warn" syslog >> summary.txt       # Count matches
```

4.Inspect and slice data

```
head -n 5 errors.txt > first_errors.txt
tail -n 5 warnings.txt > last_warnings.txt
```

5. Extract and sort log timestamps (assumes timestamp in fields 1 and 2)

```
cut -d' ' -f1,2 errors.txt | sort | uniq -c > error_times.txt
cut -d' ' -f1,2 warnings.txt | sort | uniq -c > warning_times.txt
```

Lab

6.Make the final report:

```
cat summary.txt > final_report.txt
cat error_times.txt >> final_report.txt
echo "" >> final_report.txt
cat warning_times.txt >> final_report.txt
```

7. Optional (Bonus): Use pipeline instead of intermediate files:

```
grep -i error syslog | cut -d' ' -f1,2 | sort | uniq -c | sort -nr | head -n 5
```

❓ **Questions to Answer:**

1.How many total lines mention "error" and "warn"?
2.What are the five most frequent timestamps associated with errors?
3.What is the longest line (in characters) found in errors.txt?
4.Which grep option did you find most useful? Why?
5.Which tools in this lab could be reused for analyzing other files?
6.What is the difference between using a pipeline and using intermediate files?