SWitCH – Sistemas de Computação e Redes (SCOMRED)

2020/2021

Laboratory 03

BASH command line.

BASH programming – Algorithmics.

1. The filesystem

Operating systems organize mass storage devices (e.g. disks) by establishing partitions or volumes over them, volumes/partitions may be then formatted. Formatting means creating a new and empty filesystem, ready to store objects, some most notorious object types every filesystem supports are files and folders (directories).

Unlike with volumes and partitions that follow some well-established standards and are recognized by every operating system, filesystems are specific for each operating system, currently, Windows operating systems use **NTFS** (NT filesystem) and most Linux distributions use **EXT4** (fourth extended filesystem).

1.1. Folders or directories

Folders or directories are containers of filesystem objects, they are very useful to organize the filesystem and keep related objects together, and because a folder may contain one or several other folders (often referred to as subfolders or subdirectories) the filesystem ends up being a tree of folders. When a new filesystem is created (by formatting a partition/volume) it will then contain a single folder named as the **root folder** or **root directory**, this will be where the first files and folders will be created.

In NTFS and the older FAT filesystem, the root folder is represented by a back slash (\), in EXT4 and other Linux/Unix filesystems the root folder is represented by a forward slash (**/**). The same slash is used to separate folder names and other object's names in a path.

One other difference is that in Windows systems, each volume/partition is accessed through a different drive letter and has its own root folder and directory tree, whereas in Unix/Linux there's only one root folder and different partitions/volumes are integrated in the same single directory tree by the mounting concept.

Folders are created with the **mkdir** command and removed with the **rmdir** command. By default, the **rmdir** command requires the folder to be empty. Like any other object, folders may be renamed and moved with the **mv** command.

```
mkdir /tmp/scomred-folder
ls -l /tmp
mv /tmp/scomred-folder /tmp/scomred-dir
ls -l /tmp
rmdir /tmp/scomred-folder
ls -l /tmp
```

1.2. Files and file types

The file object type purpose is storing data, a file is a variable size sequence of bytes, each byte in a file has a well-known position often referred as offset, the first byte in a file has offset zero. The operating system provides to running applications a set of functions to use files.

Files are created by applications with the purpose of storing data, you may create an empty file with the **touch** command. The **touch** command updates the access and modification times of as file to the current time, if the file doesn't exist, creates an empty file.

Files may be renamed and moved with the **mv** command and removed with the **rm** command. With the **-R** option, the **rm** command will remove a folder and all its content. Files may be copied with the **cp** command. With the **-R** option the **cp** command is able to copy directories and its content.

Example:

```
mkdir -p /tmp/f1/f2/f3
touch /tmp/f1/f2/f3/file1.sample
cp -R /tmp/f1 /tmp/F1
ls -l /tmp/f1/f2/f3
rm -R /tmp/f1
ls -l /tmp
ls -l /tmp/F1/f2/f3
rm -R /tmp/F1
```

The **-p** option used above with the **mkdir** command, allows the creation of sequence of folders and subfolders with a single command, by default to create a directory with the **mkdir** command, the parent directory must exist in the first place.

The above example also highlights that objet names are case sensitive in UNIX/Linux, and the same goes for commands and commands' options.

The use applications make of files is up to each application. For the filesystem and the operating system, there's only one objet type file, however, depending on the content of the file (bytes stored in it) we may talk about different file types.

Unlike with Windows systems where the file type is established by looking at the filename's last characters, in Unix/Linux the file type is established by looking into its content, more precisely the content of bytes in some key offsets, this is called the **magic number**.

In the Unix/Linux command line, the **file** command may be user to classify an object, and if it's a file the file type by checking the magic number.

Let's try it to check the objects with a name started by the **r** letter, in folder **/etc/**:

```
file /etc/r*
```

The **/etc/** folder is used to store system configuration information, most files here are text files.

Let's now check a folder with the purpose of storing executable files (commands and applications):

```
file /bin/b*
```

Some are compiled binary files to be directly executed by the CPU, others are scripts (text files with source code to be executed by an interpreter).

1.3. Objects' owner in Linux

In Unix/Linux, each object has an owner (a valid user) and a group of users associated to it, the owner of an object is the user that creates the object and the group of the object is the primary group of the user that created it.

Only the **root** user is allowed to change the owner of an object, with the right permissions, a user may change the group of an object but only if is a member of the new group.

Let's check this by creating a **/testing123** folder:

```
mkdir /testing123
```

To see the owner, group and permissions of the objects:

```
ls -l /
```

So, as expected, the new folder belong to the root user and the group associated is the root group. Now let's use the **chown** command to make the **ubuntu** user the owner, and then list it again:

```
chown ubuntu /testing123
ls -l /
```

Now the owner is **ubuntu**, but the group is still **root**, to change the group we can either use the **chgrp** command or do it with **chown**, **chown** does both, possibly at the same time, for instance:

```
chown games:cdrom /testing123
ls -l /
```

The folder is now owned by the **games** user and the group is **cdrom**.

Some additional experiments:

```
chown :staff /testing123
ls -l /
chown ubuntu: /testing123
ls -l /
```

The last **chown** command line is changing the owner, and at the same time, the group to match the new owner's primary group.

1.4. Objects' permissions in Linux

In a filesystem, each object, among other properties, has an ACL (Access Control List), the ACL is a list of entities (e.g. users and groups), and for each, the permissions that entity has over the object.

The standard ACL in Unix/Linux has three fixed entries: the **owner** (u), the **group** (g), and **others** (o). Others stands for every user except the owner and the members of the group.

Regarding the permissions an entity may have over the object, there are three permissions: **read** (r), **write** (w), and **execute** (x). Permissions have a slightly different meaning depending on the object type:

	Files	Folders
read (r)	Read the file's content	List the folder's content (object names), to be able to see details about each object (e.g. owner, group, and permissions) the execute permission over the folder is also required.
write (w)	Write bytes into the file (append or overwrite), remove the content of the file	Change the properties of objects stored in the folder (name, owner/group and permissions). However, this is valid only with the execute permission over the folder.
execute (x)	Execute the file, meaning loading its content into a new process. If it's a script (interpreted program) the read permission is as well required.	Access the folder, to be able to access a folder, the execute permission is required on every folder starting from the root folder.

Some point to take notice:

- To change the name or other properties of an object, the required permission is write over the folder holding the object, not the object itself.
- Without the execute permission over a folder, every object stored in that folder and subfolders is inaccessible.

The properties of objects are shown by the **ls** command with the **-l** option (long listing), check the objects in the root folder:

```
ls -l /
```

For each object listed, the first character represents the object type, for instance **d** for a directory, **l** for a symbolic link, and **-** for a regular file. Next 9 characters represent the ACL, first three characters **rwx** for the owner, next for the group and the last three for others.

To change permission the **chmod** command may be used to change the ACL, to each entity (u, g, o), permissions (r, w, x) may be granted (+) or removed (-). A list of comma separated changes may be enforced in a single command line.

Check the following examples:

```
ls -l /
chmod u-r,g+w,g-x,o+w /testing123
ls -l /
chmod o-rwx /testing123
ls -l /
```

1.5. Text files

One of the most common file type is text files, in a text file each byte represents a character by following the standard ASCII table, and most byte values in this table represent letters, digits and punctuation marks.

Some byte values in the ASCII table have special meanings, some most relevant are **CR** (carriage return) and **LF** (line-feed) which implement on text files key feature: **being organized in variable length lines**.

In Unix/Linux operating systems, text file lines are terminated by **LF**, in Windows/DOS operating systems, text files lines are terminated by **CR** followed by **LF**.

Text files are used for many purposes, to start they are used to store source code, if such source code is to be run by an interpreter these files are usually referred as **scripts**. Most configuration files in a Unix/Linux system are text files, we already know several of them, many stored in folder **/etc/**.

Because text files are so common, in Linux there are many commands to handle them, on thing to bear in mind is these commands expect the files to be text files, meaning they use the byte values in the ASCII table and no others.

Let's download a text file from the Internet, it's a document that establishes the HTTP protocol standard, these documents that establish Internet standards are known as RFCs (Requests for Comments).

One funny thing is to download the file we are going to use HTTP. We could use **lynx** or **curl**, but we are going to use yet another HTTP client, the **wget** command.

Download the text file:

```
wget https://www.ietf.org/rfc/rfc2616.txt
```

Let's check the files properties and print its content on the terminal.

```
ls -l rfc2616.txt


file rfc2616.txt


cat rfc2616.txt
```

The **cat** command simply copies the entire content to the terminal, to be able to see the content screen by screen or line by line the **more** command may be used instead:

```
more rfc2616.txt
```

(use q to exit)

The **less** command allows the user roll up and down the content:

```
less rfc2616.txt
```

(use q to exit)

Other two useful commands are **head** and **tail**, they present respectively the first and the last lines of the content of a text file, by default the first 10 lines and the last 10 lines:

```
head rfc2616.txt

tail rfc2616.txt
```

How many lines are there in this text file?

The **wc** command (stands for word count) gives you the answer, despite is name it also counts lines, bytes and characters, it may also determine the longest line length, By default it prints the number of lines, the number of words, and the number of characters:

```
wc rfc2616.txt
```

The **sort** command sorts a text file content, there are several interesting options, try these command lines:

```
sort rfc2616.txt

sort -n rfc2616.txt

sort -n -u rfc2616.txt

sort -r rfc2616.txt
```

The **cut** command is used to extract parts of each line of a text file.

For instance, to extract the first 20 characters of each line:

```
cut -c 1-20 rfc2616.txt
```

For instance, to extract the usernames and UID of each user in the /etc/passwd file:

```
cut -d ":" -f 1,3 /etc/passwd
```

2. Global Regular Expression Print - grep

The grep command is a very useful whenever we want to search for something in a text content. Given a matching pattern, every line matching such pattern is printed. The matching pattern may be a simple sequence of characters, for instance:

```
grep "http" rfc2616.txt

grep "3" rfc2616.txt
```

For a simple sequence of characters, the matching is for every line than contains that sequence, however the matching pattern may also be a **regular expression**, a regular expression is as well a sequence of characters, but it uses some special characters with special meanings under matching point of view. Examples:

```
grep "^3" rfc2616.txt

grep "3$" rfc2616.txt
```

As you may have guessed, the ^ character matches the beginning of the line and the $ character the end of the line.

This also presents a challenge when by hazard we want to match literally such a character, for instance to match every line containing the ^ character you must protected it from being interpreted by a double backslash:

```
grep "\\^" rfc2616.txt
```

Otherwise, of course, you will get all the lines because every line has a beginning.

Regular expressions have many applications (e.g. data validation) and they are supported by almost every programming language, however, they are not fully standardized and you may find some small differences in the list of special characters between different implementations.

The grep itself supports some variants, from the manual page:

```
Pattern Syntax
       -E, --extended-regexp
              Interpret PATTERNS as extended regular expressions (EREs, see below).
       -F, --fixed-strings
              Interpret PATTERNS as fixed strings, not regular expressions.
       -G, --basic-regexp
              Interpret PATTERNS as basic regular expressions (BREs, see below).  This is the default.
       -P, --perl-regexp
              Interpret  PATTERNS as Perl-compatible regular expressions (PCREs).  This option is experimental when combined
with the -z (--null-data) option, and grep -P may warn of unimplemented features.
```

The use of the egrep command is equivalent to use grep -E …

The grep command manual page contains extensive information about supported regular expressions features, some most relevant are:

\\	Avoid the next character's interpretation.
^	Match the beginning of the line.
$	Match the end of the line.
.	Match one character (any character).
*	Match the previous character zero or more times.
[abc…]	Match one occurrence of one of the characters in the list, may include ranges, e.g.: [a-z,0-9].
[^abc…]	Match one occurrence of one character not in the list.

With the **-E** command line flag (extended regular expressions), some additional features are:

?	Match the previous character zero or one time.
+	Match the previous character one or more times.
{n}	Match the previous character n times.
(A|B)	Match the occurrence of either A or B, both A and B may be themselves patterns.

Some examples:

```
egrep "^.4.{3}A" rfc2616.txt

grep "http.*HTTP" rfc2616.txt
```

Try to express these matching criteria in natural language.

3. The `find` command

If you want to locate objects within the filesystem, the **find** command is a powerful tool you may use.

The **find** command searches recursively the directory tree starting from a given folder, a wide range of matching criteria is supported: object name, object type, size, permissions, owner, last change timestamp …

For each match, several options are available, including a fully programmable printing of the object's properties, and the execution of a command.

A simplified vision of the command's syntax is:

```
find [STARTING-FOLDER] [SEARCH-OPTIONS] [MATCHING-OPTIONS] [ACTION-OPTIONS]
```

The default **STARTING-FOLDER** is the current working directory (./). By default every object is a match and the default action is **-print** which stands for printing the object's name.

Examples of some MATCHING-OPTIONS

By default, multiple matching options are linked by the **and** logical operator (**-a** option), to specify alterative criteria for match use the **-o** option (**or** logical operator). To reverse (negate) the sense of a match option precede it with the **-not** option.

-name NAME	Match the object's name, wildcards may be used. With wildcards, very simple regular expressions may be created, the * character matches any sequence of 0 or more characters, and the ? character matches one character.
-user USERNAME	The object is owned by the user USERNAME.
-type T	The object if of T type, T is: b block (buffered) special c character (unbuffered) special d directory p named pipe (FIFO) f regular file l symbolic link s socket
-writable	The user running the command has the write permission over the object.
-regex PATTERN	The object's name is matched the PATTERN containing a regular expression.
-size [+|-]N[U]	The objects size (rounded up) matches the N value. The + prefix stands for size greater than, the - prefix stands for size smaller than. The default unit is blocks of 512 bytes, the U suffix may be used to specify other units, e.g. k, M, G.

Examples of some ACTION-OPTIONS	
-delete	Remove the object.
-print	Print the full object's name (path from STARTING-FOLDER).
-printf FORMAT	Customized print about the object in a C style printf format.
-exec COMMAND ;	Execute a custom command. This command ends with a semicolon, if started from a shell, the semicolon must be protected by a backslash. Any occurrence of {} is replaced by the matched object name.
-execdir COMMAND ;	Similar to the previous, but unlike the former the command is executed with the CWD matching the directory where the matched object was found. For the -exec action, the CWD is the STARTING-FOLDER.

Examples:

```
find /usr -size +20M -printf "%s bytes -> File: %p\n"


find /etc -type f -name "*net*" -exec ls -l {} \;
```

4. Processes management

In UNIX/Linux, a process is a running application, when at the command line an external command, like for instance find, is executed, a new process is created and the find application (binary file) is loaded into that new process and executed, in the case of the find command it will not take long to complete its mission so it will exit and the process is destroyed.

As with users and groups, processes also have a unique identifier, known as PID (Process ID), processes always have a parent (the process who created it). This is true except for the first process created when the kernel is loaded on the operating system boot, all processes running in the system are descendants of this initial process. The initial process is usually loaded from **/sbin/init** or **/usr/bin/systemd**, and is assigned with PID 1.

The **ps** command (process status) shows the status of running processes in the system, several command line options are available to change the processes that are listed and the printed details about each listed process. By default, the **ps** command lists only processes that are owned by the current user and associated current terminal. The **-e** option lists all processes.

Try it:

```
ps


ps -e
```

The pstree command presents a very interesting view of the running processes and their parenthood relationships, try it:

```
pstree
```

Every running process has an owner, initially it’s the user who created it, but if started by root the processes is allowed to change the UID, meaning to impersonate any user.

Let’s see the owner of the processes:

```
ps xau
```

4.1. Signals

One way to interact with a running process is through signals, only the owner of a process or the **root** user are allowed to **send signals to a process**.

When a process receives a signal it pauses temporarily what is doing and executes a function known as signal handler, once the handler execution ends (the function returns), the process returns back to what it was doing.

To send a signal to a process, the **kill** command is used, the command name comes from one of the signals it may send, the **SIGKILL**, this signal is a last resource mean to force the termination of a process and is handled by the kernel, not a handler within the process. There are many signals with different purposes, you can list them with the **kill** command:

```
kill -l
```

To send a signal to a process, the PID must be used as argument, by default the sent signal is **SIGTERM** which is used to ask the process to orderly terminate, unlike **SIGKILL** it’s handled by the process itself. **SIGKILL** should be used only when a process fails to exit with the **SIGTERM** signal.

4.2. Running commands in background

So far we have been running commands in **foreground**, this means the calling program (the shell) suspends its execution and waits for the command to end, and meanwhile all command line interactions take place with the running command and not with the calling program.

However, a command may be started in **background**, this means it will run in parallel with the calling program that will not suspend its execution. To run a command in background you simply append to the command line the ampersand (**&**).

Let’s try it with the **sleep** command which runs for a number of specified seconds and then exits:

```
sleep 55555 &
```

The **sleep** command is now running in background for 55555 seconds, two numbers were printed, the **job**, within brackets, and the **PID**. The job is a shell specific way to manage background processes within a session, probably this will be your job 1.

Now, let’s see the process status:

```
ps
```

And the jobs status using the **jobs** command:

```
jobs
```

Jobs management by the shell allows some interesting options, we can transfer a job from background into foreground by using the **fg** command:

```
fg 1
```

(Assuming is job 1)

Now the **sleep** command is no longer running in background, it's running in foreground.

Press **CTRL+Z**, this sends the **SIGTSTP** to the foreground process making it suspend the execution, now it's not running, neither in foreground nor in background. Let's check:

```
jobs
```

We can continue the process execution in foreground or in background, to continue the execution in foreground we would use the **fg** command as before, to continue the execution in background we use the **bg** command instead:

```
bg 1
```

(Assuming is job 1)

Let's check:

```
jobs
```

Send to it the **SIGTERM** signal, for that you must know its PID:

```
kill PID
```

The process has gracefully terminated by itself. If it fails to do so, you could always use **SIGKILL**:

```
kill -SIGKILL PID
```

In the case of a foreground process, we can use **CTRL+C** to send to it the **SIGINT** signal to the process, in that case the process should exit. Bear in mind that unlike **SIGKILL**, both **SIGTSTP** and **SIGINT** may be intercepted by a specific handler in the process and fail the expected behaviour.

5. Processes' input/output - file descriptors

In UNIX/Linux, processes are pretty much like closed boxes, a process is allowed to interact directly with resources within the process only. Interactions with the system or with other processes are always indirect by using kernel system-calls, one of such mean of interaction between processes is signals.

File descriptors are the main mean by which a process interacts with the external world resources, even though they are called file descriptors, they may or not provide access to regular files, in UNIX/Linux most resources are managed as if they were files.

File descriptors are non-negative integer numbers, unique within each process, that allow the interaction with an external resource.

Among many other resources and devices that are treated as files, the command line terminal is itself managed as a file. The command line shell has three file descriptors open, and because open file descriptors are inherited by child processes, any command started from the shell has these same file descriptors open, they are:

File descriptor	Designation	Resource
0	Standard Input (stdin)	Read-only file descriptor associated to the terminal input (the keyboard). Used to read input data.
1	Standard Output (stdout)	Write-only file descriptor associated to the terminal output (the screen). Used for regular prints.
2	Standard Error (stderr)	Write-only file descriptor associated to the terminal output (the screen). Used for errors reporting.

For most text handling commands we have seen so far (**cat; more; less; head; tail; wc; sort; cut; grep**) the default source of the text to process is descriptor 0 (stdin), when no filename is provided at the command line.

5.1. Input/output redirection

When running a command from the command line, the three descriptors are directed to the terminal:

- Keyboard: descriptor 0
- Screen: descriptor 1 and descriptor 2

But they can be redirected to regular files (descriptor 1 and descriptor 2) or from regular files (descriptor 0).

The < signal is used to redirect descriptor 0 from a regular file and the > signal to redirect descriptor 1 to a regular file. When redirecting the output, > stands for rewrite, meaning it will remove the content if the destination files already exists, to append >> is used instead.

Input redirection example:

```
wc -l < rfc2616.txt
```

Notice that this is not the same as:

```
wc -l rfc2616.txt
```

Output redirection:

```
wc < rfc2616.txt > results.text


ls -l /tmp >> results.text
ls -l /doesnt-exist >> results.text
```

The **results.text** file will contain the results of the commands:

```
cat results.text
```

Another example is creating a copy of a text file without using the **cp** command:

```
cat rfc2616.txt > rfc2616-copy.txt
```

But not for the last command, because the result is an error and is being sent to descriptor 2. To redirect descriptor 2, we can use **2>** or **2>>**. We can also redirect both descriptor 1 and descriptor 2 to the same file by using **&>** or **&>>**. One other option is redirecting one descriptor to the other, for instance **2>&1** redirects descriptor 2 to descriptor 1, which in turn may be redirected do a file.

So to append both the standard out and the standard error of the execution of a command to a file we have several options:

```
echo "Testing stdout and stderr" > 2ndRes.text
ls -l /tmp /Ups >> 2ndRes.text 2>> 2ndRes.text
ls -l /tmp /Ups &>> 2ndRes.text
ls -l /tmp /Ups >> 2ndRes.text 2>&1
cat 2ndRes.text
```

The **echo** command, used above copies the provided text to descriptor 0 (prints the text).

5.2. Pipelining input/output

Often we may want the output of a command to be the input of another command, for instance:

```
ps -e > tmp.txt

grep apache < tmp.txt > tmp1.txt

wc -l < tmp1.txt

rm tmp.txt tmp1.txt
```

The use of pipes avoids the need of temporary files to store data, a pipe is represented by **|** and means the descriptor 1 (stdout) of the left side command is piped into the descriptor 0 (stdin) of the right side command, this means the same goal of the above command lines sequence can be attained by:

```
ps -e | grep apache | wc -l
```

If you want to pipe both descriptor 1 (stdout) and descriptor 2 (stderr), then **|&** may be used instead:

```
ls /tmp /Ups |& sort
```

Some more pipelining examples:

```
cat rfc2616.txt | tr " " "_"

cat rfc2616.txt |tr "[:upper:]" "[:lower:]"

cat /etc/passwd | tr ":" "\t"
```

The **tr** command translates characters from standard input, writing to standard output. A character matching the first list is replaced by the corresponding character in the second list. While translating it may optionally squeeze (replace sequences of repeated character with a single character). Instead of translating it may be used to delete specific characters.

6. Getting help about commands (manual pages)

Nobody is supposed to know the syntax of every command, what a BASH user and script developer is required to know is about the commands that exist and may be useful to him. Once the command name is known, the man command may be used to get the exact syntax (arguments usage).

Manual pages are organized in sections (check man man), some sections are about commands, others for instance about C functions, some pages exist in different sections because names are the same.

For instance **printf**, it's an external command, the manual page is in section 1, however there's also a printf library C function available at section 3. By default the man command starts by searching the manual page in section 1 (external commands).

One other issue regarding commands manual pages is about internal and external commands, the manual pages are about external commands, to get information about internal commands the manual page of the shell must be used, for instance for BASH **man bash**.

For the sake of efficiency many external commands have internal commands that replace them, usually they will have the same features and syntax.

External commands can be directly called by specifying the full path, the full path of an external command can be known by using the **which** command (as far as the executable file is on the PATH).

Try it:

```
which printf
```

This returns /usr/bin/printf, that's the external command, now try:

```
man printf
```

This is the manual page of the **printf** external command, extensive information can be also attained by using the command's **--help** option:

```
/usr/bin/printf --help
```

This is the internal BASH version:

```
printf --help
```

The internal version doesn't support the --help option. To get help you should use the BASH manual page (**man bash**), though they should be similar.

7. BASH scripts

We already know a BASH script is an executable text file (with the execute permission) that starts with **#!/bin/bash** line. The operating system will then recognise this file as an interpreted program to be executed by **/bin/bash** (the interpreter). The file should contain a sequence of lines with commands the BASH is able to execute, they will all get executed one after the other, line by line, starting from the first line.

To create and edit text files a text editor is required, for beginners, either **nano** or the mc's text editor are two good alternatives. You may find **mc** (GNU Midnight Commander) interesting because it supports mouse interaction.

To use mc (GNU Midnight Commander), it must be installed first (optional):

```
apt install mc
```

Create the following BASH script:

```
#!/bin/bash
echo "I've a number between 0 and 100, it may even be 0 or 100"
echo "Try to guess it!"
NUM=$(shuf -i 0-100 -n 1)
TRIES=0
GUESS=101
while [ $NUM != $GUESS ]; do
 read -p "Enter your guess please: " GUESS
 TRIES=$(($TRIES+1))
 if [ $GUESS -gt $NUM ]; then echo "Sorry, too large!"; fi
 if [ $GUESS -lt $NUM ]; then echo "Sorry, too short!"; fi
done
echo "Very well, you have guessed it in $TRIES tries"
```

Create this content in a new file named **guessNumberGame** in your current working directory, for instance with **nano** type:

```
nano guessNumberGame
```

Warning: don't copy & past the above content, you must type it by hand, otherwise some characters won't be correctly copied.

Give it the execute permission:

```
chmod +x guessNumberGame
```

Test the script by playing the game a couple of rounds.

```
./guessNumberGame
```

Suggested practical exercises

1. List the contents of **/usr/bin** using the command **ls .** (which lists the contents of the current working directory). Use an absolute path to navigate into **/usr/bin**.

2. List the contents of **/usr/bin** using the command **ls .** (which lists the contents of the current working directory). Use only relative path(s) to navigate into **/usr/bin**.

3. Create the **/tmp/Exercise3** folder
   a) Remove all permissions for the group and other over the created folder.
   b) Copy the **/etc/passwd** file into the created folder.
   c) Rename the **passwd** file in the created folder to **users.txt**.
   d) Make the **ubuntu** user the owner of **users.txt**.
   e) Remove the **/tmp/Exercise3** folder and its content.

4. Create a single command line that tells:
   a) How many user groups exist in the system?
   b) How many users exist in the system?
   c) How many regular files are there in folder **/etc**?

5. Without using the **cp** command, copy the **/etc/services** text file to **/root/my-services-list**.
   a) Make the **ubuntu** user the owner of **/root/my-services-list**.
   b) Append to **my-services-list** the list of running processes.
   c) Append to **my-services-list** the sorted content of the **/etc/group** file.
   d) Append to **my-services-list** the first five usernames defined in the **/etc/passwd** file.

6. Use the **wget** command to download this text file:

   https://www.gutenberg.org/cache/epub/250/pg250.txt

   About the **pg250.txt** file:
   a) Print all lines started by **The Internet**.
   b) Show how many words are there in this text file.
   c) Print all sentences that are completely within quotes (start and end with “).
   d) Print the number of lines and the size of the longest line.
   e) Print the first 5 lines of each chapter.
   f) Print all lines that start with a number.

7. For every file in the **/var** folder and subfolders that doesn’t belong to the **root** user, print the owner’s name and the filename.

8. For every file in the **/etc** folder and subfolders that is newer than the **/etc/passwd** file (more recently changed than …), print the last access time and the filename.

9. For every file in the **/usr/bin** folder and subfolders with a size bigger than 10 Mbytes, use the **file** command to show its content type based on the magic number.

10. Execute the following command: **sleep 5h**

    a) Make the command run in background. Hint: stop it first.

    b) Send to the SIGTERM to the process that is running the command.

11. Create the following files:

    a) A file named **file1.txt**, containing **This is test 1**. The only command you are allowed to use is **cat**. Hint: end the input use **CTRL+D**.

    b) A file named **file2.txt**, containing **This is test 2**. The only command you are allowed to use is **echo**.

    c) A file named **file3.txt**, with the content of **file1.txt**, followed by the content of **file2.txt**.

12. Execute the following command line:

```
shuf -i 1-100 -r -n 200 > numbers.txt
```

    This **shuf** command usage example outputs 200 random numbers ranging from 1 to 100.

    a) Present a numerically sorted list of the numbers in the **numbers.txt** file.

    b) With a command line print the answer to: How many different numbers are the in the **numbers.txt** file?

    c) With a command line print the answer to: How many single digit numbers are the in the **numbers.txt** file?

    d) With a command line print the answer to: Which is the biggest number in the **numbers.txt** file?

13. Add the following new functionality to the guessNumberGame developed during this class.

For each user the best score he has ever achieved should be stored persistently. When each round ends, he must be informed if he has or not beaten his previous best score, and a congratulations message should be presented if so.

Suggestions:

- To be persistent, the best score must be stored in a file, to be personal, it should be stored in the user's home folder, for instance **~/.guessNumberGame.topScore**
- To get the file's content into a variable use VARNAME=$(cat ~/.guessNumberGame.topScore), remember on the first round the file will not exist.
- To place a new value in the file just echo the variable's value redirecting the output to the file.

Test the game again with the new functionality you have implemented.

How could you cheat the game as if you have achieved a previous best score of 2 tries only?

14. The following BASH script lists every valid user’s login name and home folder:

```
#!/bin/bash
USERSLIST=$(getent passwd|cut -d ":" -f 1,6)
for USR in $USERSLIST; do
 echo "${USR%:*} - ${USR#*:}"
done
```

To list valid users and valid groups, the **getent** command should be used instead of listing the contents of files /etc/passwd and /etc/group. Valid users and groups in Linux may not be limited to those files contents. The output format of getent is the same as on those files.

The output of the getent command is sent (through a pipe) to the cut command to extract required data, the fields separator is the colon (-d “:”) and we want fields number 1 (username) and number 6 (home folder). Each element of the list will have the format USERNAME:HOMEFOLDER.

Afterwards, by using the variable expansion with suffix and prefix pattern removal, the two elements are isolated.

Create the script file with the above content, for instance named **usershomes** and test it.

Add a new functionality so that the script checks if the home folder exists and it’s a directory, if otherwise a conforming warning message should be presented.

Suggestion: check the **test** command, **man test**. Even though this manual page is about the external command, the internal **test** command should work the same way. You may also check it on the BASH manual page (**man bash**).

15. Create a BASH script that for each valid group (getent group) counts how many valid users (getent passwd) have that group as primary group.

Before proceeding let’s add some groups and some users having those groups as their primary group:

```
groupadd scomredgrp1

groupadd scomredgrp2
```

```
useradd -m -g scomredgrp1 scomredusr1
useradd -m -g scomredgrp1 scomredusr2
useradd -m -g scomredgrp2 scomredusr3
useradd -m -g scomredgrp1 scomredusr4
```

Notes:

- The –m option instructs the useradd command to create the new user’s home directory.
- The –g option sets the new user’s primary group.

  Users named **scomredusr1**, **scomredusr2**, and **scomredusr4** have primary group **scomredgrp1**, and user **scomredusr3** has primary group **scomredgrp2**.

Suggestions for the script implementation:

- The primary group of each user is a user account’s attribute (forth field), but is stored there as a GID (Group Identifier). To list every user’s primary group GID (forth field) you may use:

  getent passwd|cut -d “:” -f 4

- The group identifier (GID) of each group is an attribute of the group account (third field), to get both the group name and GID (first and third fields) for every group you may use:

  getent group|cut -d “:” -f 1,3

  To go through the list of groups, use a **for** loop. Each element of this list will have the form GROUP-NAME:GID

  If one of these elements is stored in variable GNAMEGID, then you can extract the group name by expanding it as ${GNAMEGID%:*}, and you can extract the GID by expanding ${GNAMEGID#*:}. In the first case it’s a prefix pattern removal and in the second case a suffix pattern removal.

- Once you have the GID of each group, then you can use the grep command to filter user accounts with that GID and pass the output (pipe) to the **wc** command to count the number of lines (**man wc** for details).

Before adding commands to the script, test those commands on the command line to see if they are working as expected.

The desired output for this script is a sequence of lines in the following format:

GROUPNAME - NUM users

Where NUM is the number of users having GROUPNAME as their primary group.