Exploring Shell

Prof. Sérgio Dias

Summary

- Unix File System Structure.
- Commands for file manipulation, examination, searching.
- Redirection, Pipes and Processes.
- Persistent settings for bash shell.
- User accounts, groups,File permissions.
- Shell Programing.

Unix File System Structure

directory	**description**
/	root directory that contains all others
/bin	programs that came with the system
/dev	hardware devices
/etc	system configuration files
/home	users' home directories
/mnt	disks that have been "mounted" for use on this computer
/proc	currently running processes (programs)
/usr	temporary files
/tmp	user-installed programs

Links

Command	**description**
ln	create a link to a file
unlink	remove a link to a file

- **Hard Link**: two names for the same file.
  - *ln orig other*
    - the above command links other as a duplicate *name* for *orig*.
- **Soft/Symbolic Link**: A reference to another existing file.
  - *ln -s orig nickname*
    - creates a reference *nickname* to the file *orig*.

File examination

Command	**description**
cat	output a file's contents on the console
more or less	same as before but displays one page at a time
head, tail	shows in the screen the first or last few lines of a file
wc	count words, characters, and lines in a file
du	report disk space used by a file(s)
diff	compare two files and report differences

Searching and sorting

Command	**description**
grep	search a file for a given string
sort	convert an input into a sorted output by lines
uniq	strip duplicate (adjacent) lines
find	search for files within a given directory
locate	search for files on the entire system
which	shows the complete path of a command

Keyboard shortcuts

Key	description
Up arrow	repeat previous commands
Ctrl + r command name	search through your history for a command
Home/End	move to start/end of current line
Tab	auto-completes a partially typed file/commar
Ctrl + c	terminates the currently running process
Ctrl + z	suspends (pauses) the currently running proc

Shell History

- The shell remembers all the commands you've entered.
- Can access them with the *history* command.
- Can execute the most recent matching command with *!command name* .
  - Ex: *!cat* will search backwards until it finds a command that starts with cat, and re-execute the entire command line

Output redirection

command > *filename*

- run command and write its output to a filename instead to the console.
  - Its like an arrow going from the command to the file.
  - Atention: if the file already exists, it will be overwritten.
  - » appends rather than overwriting, if the file already exists.
  - Ex: *ls -l* > *teste.txt*

Input redirection

command < filename

- run command and read its input from filename instead of console
  - whenever the program prompts the user to enter input (such as reading from a file in C), it will instead read the input from a file.
  - some commands don't use this; they accept a file name as an argument.
  - note that this affects user input, not parameters
  - useful with commands that can process standard input or files
  - Ex:*Mail -s "Subject" to-address < Filename*

Pipes

command1 | *command2*

- run command1 and send its console output as input to command2
  - Ex: *less sort teste3.txt* | *uniq*

Processes

- Process: a program that is running.
- When you run commands in a shell, it launches a process for each command.
  - Process management is one of the major purposes of an OS.
  - A process is identified by a PID: 1324 Name: *ls*.

Process commands

Command	**description**
ps	list processes running by a user
top	show which processes are using CPU/memory
kill	terminate a process by PID
killall	terminate several processes by name
&	runs a command in the background

Aliases

alias name=command

- Must wrap the command in quotes if it contains spaces.
- Do not put spaces on either side of the = .
- Ex: alias *ll="ls -la"*.

.bash_profile and .bashrc

- Every time you log in to bash remotely, the commands in ~/.bash_profile are run.
  - You can put any common startup commands you want into this file.
  - Useful for setting up aliases and other settings for remote login.
- Every time you launch a non-login bash terminal, the commands in ~/.bashrc are run.
  - Useful for setting up persistent commands for local shell usage.
  - Alias, useful functions and configuration parameters.
- Atention : a dot (.) in front of a filename indicates a normally hidden file, use *ls –a* to see it.

Users

- Linux is a multi-user operating system.
- Every program/process is run by a user.
- Every file is owned by a user.
- Every user has a unique integer ID number (UID).
- Different users have different access permissions, allowing user to:
  - Read or write a given file.
  - Browse the contents of a directory.
  - Execute a particular program.
  - Install new software on the system.

Groups

- A collection of users, used as a target of permissions.
  - A group can be given access to a file or resource.
  - A user can belong to many groups.
  - See who's in a group using grep <groupname> /etc/group.
  - Use groups to check to which group you belong.
- Every file has an associated group.
  - Owner of a file can grant permissions to the group.
- Every group has a unique integer ID number (GID).

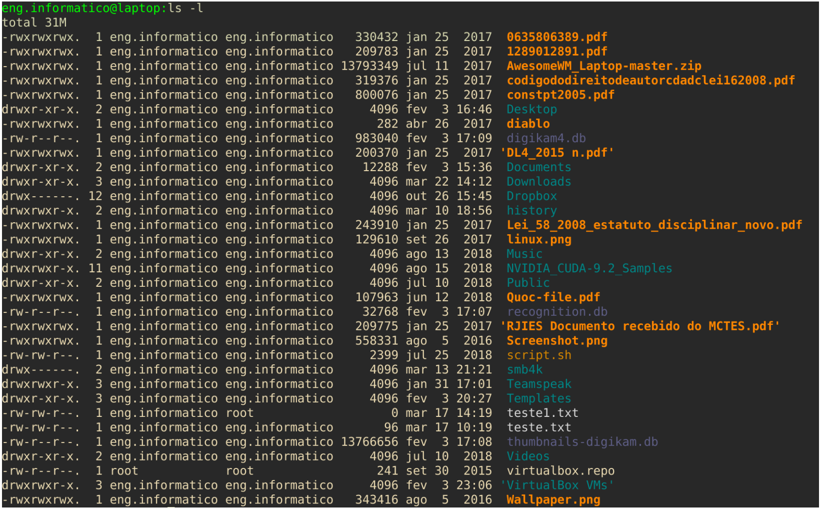
File permissions

- Types : read (r), write (w), execute (x).
- User : owner (u), group (g), others (o).
  - Permissions are shown when you type *ls -l*.
  - Ex: *d rwx rwx rwx*
  - if d is in the line description : is a directory.

Users & Permissions(Files - Directory)

- Users: each user fits only in one of three permission sets:
  - Owner (u) – if you create the file you are the owner, the owner can also be changed through chown command.
  - Group (g) – by default a group is associated with each file.
  - Others (o) – everyone other than the owner and the user who are in the particular group associated with the file.
- Permissions: For regular files, permissions work as follows:
  - Read (r) – allows file to be open and read.
  - Write (w) – allows contents of file to be modified or truncated.
  - Execute (x) – allows the file to be executed

Show permissions of user in directory/file

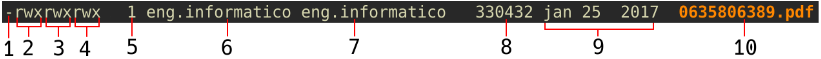


```
eng.informatico@laptop:ls -l
total 31M
-rwxrwxrwx.  1 eng.informatico eng.informatico   330432 jan 25  2017  0635806389.pdf
-rwxrwxrwx.  1 eng.informatico eng.informatico   209783 jan 25  2017  1289012891.pdf
-rwxrwxrwx.  1 eng.informatico eng.informatico 13793349 jul 11  2017  AwesomeWM_Laptop-master.zip
-rwxrwxrwx.  1 eng.informatico eng.informatico   319376 jan 25  2017  codigododireitodeautorcdadclei162008.pdf
-rwxrwxrwx.  1 eng.informatico eng.informatico   800076 jan 25  2017  constpt2005.pdf
drwxr-xr-x.  2 eng.informatico eng.informatico     4096 fev  3 16:46  Desktop
-rwxrwxrwx.  1 eng.informatico eng.informatico      282 abr 26  2017  diablo
-rw-r--r--.  1 eng.informatico eng.informatico   983040 fev  3 17:09  digikam4.db
-rwxrwxrwx.  1 eng.informatico eng.informatico   200370 jan 25  2017 'DL4_2015 n.pdf'
drwxr-xr-x.  2 eng.informatico eng.informatico    12288 fev  3 15:36  Documents
drwxr-xr-x.  3 eng.informatico eng.informatico     4096 mar 22 14:12  Downloads
drwx------. 12 eng.informatico eng.informatico     4096 out 26 15:45  Dropbox
drwxrwxr-x.  2 eng.informatico eng.informatico     4096 mar 10 18:56  history
-rwxrwxrwx.  1 eng.informatico eng.informatico   243910 jan 25  2017  Lei_58_2008_estatuto_disciplinar_novo.pdf
-rwxrwxrwx.  1 eng.informatico eng.informatico   129610 set 26  2017  linux.png
drwxr-xr-x.  2 eng.informatico eng.informatico     4096 ago 13  2018  Music
drwxrwxr-x. 11 eng.informatico eng.informatico     4096 ago 15  2018  NVIDIA_CUDA-9.2_Samples
drwxr-xr-x.  2 eng.informatico eng.informatico     4096 jul 10  2018  Public
-rwxrwxrwx.  1 eng.informatico eng.informatico   107963 jun 12  2018  Quoc-file.pdf
-rw-r--r--.  1 eng.informatico eng.informatico    32768 fev  3 17:07  recognition.db
-rwxrwxrwx.  1 eng.informatico eng.informatico   209775 jan 25  2017 'RJIES Documento recebido do MCTES.pdf'
-rwxrwxrwx.  1 eng.informatico eng.informatico   558331 ago  5  2016  Screenshot.png
-rw-rw-r--.  1 eng.informatico eng.informatico     2399 jul 25  2018  script.sh
drwx------.  2 eng.informatico eng.informatico     4096 mar 13 21:21  smb4k
drwxrwxr-x.  3 eng.informatico eng.informatico     4096 jan 31 17:01  Teamspeak
drwxr-xr-x.  3 eng.informatico eng.informatico     4096 fev  3 20:27  Templates
-rw-rw-r--.  1 eng.informatico root                   0 mar 17 14:19  teste1.txt
-rw-rw-r--.  1 eng.informatico eng.informatico       96 mar 17 10:19  teste.txt
-rw-r--r--.  1 eng.informatico eng.informatico 13766656 fev  3 17:08  thumbnails-digikam.db
drwxr-xr-x.  2 eng.informatico eng.informatico     4096 jul 10  2018  Videos
-rw-r--r--.  1 root            root                 241 set 30  2015  virtualbox.repo
drwxrwxr-x.  3 eng.informatico eng.informatico     4096 fev  3 23:06 'VirtualBox VMs'
-rwxrwxrwx.  1 eng.informatico eng.informatico   343416 ago  5  2016  Wallpaper.png
```

Show permissions of user in directory/file

- □ Permissions are displayed as a sequence of 10 dashes or letters at the beginning of each line.
- □ First Column tells the file type.
- □ The others represent the permission keys.
- □ When a key is activated it appears, when it is inactive, a dash appears.

Show permissions of user in directory/file



```
-rwxrwxrwx  1 eng.informatico eng.informatico  330432 jan 25  2017  0635806389.pdf
```

1 File Type

2 Permissions applied to the owner

3 Permissions applied to the group

4 Permissions applied to others

5 Number of absolute links

6 Owner

7 Group

8 Size(kb)

9 Last date of modification

10 Name

File type (field 1)

File Type	Description
-	Common file
d	Directory
|	Symbolic Link
c	Character Devices
b	Block Device
s	Sockets
=	Pipes

Changing file/directory permissions

- Letter codes: *chmod who(+-)what filename*
  - *chmod u+rw teste3.txt* (allow owner to read/write).
  - *chmod +x script* (allow everyone to execute the script file).
  - *chmod ug+rw,o-rwx teste4.xls* (owner/group can read and write; others nothing)
- Octal (base-8) codes: *chmod nnn filename*
  - Three numbers between 0-7, for owner (u), group (g), and others (o).
  - Each gets +4 to allow read, +2 for write, and +1 for execute.
    - *chmod 600 file.txt* (owner can read/write (rw)).
    - *chmod 664 test.dat* (owner rw; group rw; other r).
    - *chmod 751 banner* (owner rwx; group rx; other x).

Super-user (root)

su

  - Start a shell with root privileges.

- Super-user: An account used for system administration.
  - Full privileges on the system.
  - Represented as a user named root.
- Most users have more limited permissions than root

Tar Files

- tar : create or extract .tar archives (combines multiple files into one .tar file).
- Utility that allow the combination of multiple files into a single .tar file.
- Create a single file from multiple files:
  - *$ tar -cf filename.tar archive*
    - -c creates an archive.
    - -f read to/from a file.
    - archive - can be a list of filenames or a directory
- Extracting files from an archive:
  - $ tar -xf filename.tar
    - -x extracts files from an archive.

Compressed files

Command	**description**
gzip, gunzip	create or extract .zip compressed archives
zip, unzip	terminate a process by PID

- Compress a file:
  - *$ gzip filename* produces: filename.gz
- Uncompress a file:
  - *$ gunzip filename.gz* produces: filename
- Using zip and unzip commands is similar to gzip and gunzip.

Shell scripts

- Shell script: short program meant to perform a target task.
  - Set of commands combined into one executable file.
- To write a bash script:
  - Type one or more commands into a file and save it.
  - Type a special header in the file to identify it as a script.
  - Enable execute permission on the file.
  - Run it.

Shell script syntax

\# !interpreter

- Written as the first line of an executable script.
- Tells to the shell that the file must be treated as a script and to be run by the given interpreter.

- Make the script executable and then run it
  - *chmod u+x myscript.sh*
  - *./myscript.sh*
- Example myscript.sh: A script that lists all files of a directory:

```
#!/bin/bash
ls -l
```

Shell variables

name=value (declaration)

- Written EXACTLY as shown; no spaces allowed.
- Often given all-uppercase names by convention.
- Once set, the variable is in scope until unset (within the current shell)
  - AGE=64
  - NAME="Michael Young"
- $name (usage)
  - echo "$AGE years old"
  - 64 years old

- Misspell a variable's name, a new variable is created
- NAME=Ana is diferent from variable Name=Rob
- When you use an undeclared variable, an empty value is used.

Shell variables - continuation

- Store a multi-word string, must use quotes.
  - NAME=John Anderson -> Won't work
  - NAME="John Anderson" -> Correct
- Do not use $ during assignment or reassignment
  - $string="Hi" -> Wrong
  - string="Hi" -> Right
- Forgetting echo to display a variable

```
$name
echo $name
```

Shell variables - continuation

- Variables are stored as strings(operations on variables are done as string operations, not numeric)
- To make integer operations we have to use the let word:

```
x=42
y=23
let z="$x + $y"
```

- Integer operators: + - * / %
- if a non-numeric variable is used in numeric context, you'll get 0

Shell special variables

Variable	**description**
$HOSTNAME	name of computer you are using
$HOME	your home directory
$PATH	list of directories holding commands to execute
$PS1	the shell's command prompt string
$PWD	your current directory
$SHELL	full path to your shell program
$USER	your user name

- Automatically defined for you in every bash session

Capture command output

variable=$(command)

  - Captures the output of command into the given variable

- Example

```
FILE=$(ls *.txt)
echo $FILE
```

Single vs. Double quotes

- Single quotes do not expand variables/execute commands in $()

  ```
  NAME="Charlie Brown"
  echo ’Hi $NAME! Today is $(date)’
  Hi $NAME! Today is $(date)
  ```

- Double quotes expand variable names and $() work

  ```
  NAME="Charlie Brown"
  echo "Hi $NAME! Today is $(date)"
  Hi Charlie Brown! Today is Tue Feb 12 12:23:54 PDT 2019
  ```

Shell Commands

- □ echo : prints output to console.
- □ read : reads value from console and stores it into a variable
- □ printf : prints complex formatted output to console
- □ # comment text
- □ Example: Prints the current directory.

```
#!/bin/bash
read -p "What is your name? " name
printf "%10s " $name
# Prints the current directory
echo "Your current dir is: $(pwd)"
```

Command-line arguments

- □ $0 : name of this script
- □ $1, $2, $3, ... : command-line arguments
- □ $#: number of arguments
- □ $@: array of all arguments
- □ example.sh argument1 argument2 argument3

```
#!/bin/bash
echo "Name of script is $0"
echo "Command line argument 1 is $1"
echo "there are $# command line arguments: $@"
```

Exit Status

- All Linux commands returns an integer code when finish, called its “exit status”
  - 0 usually* denotes success, or an OK exit status
- The status of the last command executed can be check in the variable $?
- Example

```
$ cat NotExist.txt
$ echo $?
1 # “Failure”
```

Operators

Operators	**description**
=, !=, <, >	compares two string variables
-z, -n	tests if a string is empty or not
-lt, -le, -eq	<, <=, ==
-gt, -ge, -ne	>, >=, !=
-e, -f, -d	tests whether a given file or directory exists
-r, -w, -x	tests whether a file exists and is readable/writable/executable

For loops

```
for name in value1 … valueN; do
        commands
done
```

- Note the semi-colon after the values!
- The pattern after in can be:
  - A hard-coded set of values you write in the script
  - A set of file names produced as output from some command
  - Command line arguments: $@
- Example:

```
        for file in *.txt; do
                mv $file $file2
        done
```

if/else

```
if [ condition ]; then      # basic if
      commands
fi


if [ condition ]; then      # if / else if / else
      commands1
elif [ condition ]; then
      commands2
else
      commands3
fi
```

- There MUST be spaces as shown in the code below.

while and until loops

```
while [ condition ]; do     # while condition is true
      commands
done

until [ condition ]; do     # while condition is false
      commands
done
```

case statement

```
case EXPRESSION in
    CASE1) COMMAND-LIST;;
    CASE2) COMMAND-LIST;;
    ...
esac
```

Arrays

```
name=(element1 element2 ... elementN)

name[index]=value      # set an element
$name                  # get first element
$name[index]           # get an element
$name[*]               # elements sep.by spaces
$#name[*]              # array's length
```

- Arrays don't have a fixed length and can grow as necessary.
- If you go out of bounds, shell will silently give you an empty string

Functions

```
function name() {    # declaration
      commands        # () optional
}

name # call
```

- □ Functions are called simply by writing their name (no parens)
- □ Parameters can be passed and accessed as $1, $2, etc.

Regular Expression

"[a-zA-Z_]+(([a-zA-Z_])+$\dot{)}$+[a-zA-Z]2,4"

- Regular expression(regex): description of a pattern in a text
  - Can test whether a string matches the expression's pattern.
  - Can use a regex to search/replace characters in a string
  - Extremely powerful but tough to read(above regex expression matches a email address)

Regex wildcards and anchors

"abc"

- A regex match a particular substring
- Its a pattern, not a string!
- . (a dot) matches any character except \n
- m̂atches the beginning of a line; $ the end
- \< demands that pattern is the beginning of a word;
- \> demands that pattern is the end of a word
- | means OR
- () are for grouping
- \starts an escape sequence
- * means 0 or more occurrences
- + means 1 or more occurrences

Regex wildcards and anchors

- ? means 0 or 1 occurrences
- {min,max} means between min and max occurrences
- [ ] group characters into a character set; will match any single character from the set
- Inside a character set, specify a range of characters with -