

# Linux Fundamentals

---

Virtualization and Cloud Computing - ay 2023/2024

Enrico RUSSO <enrico.russo@dibris.unige.it>

Giacomo LONGO <giacomo.longo@dibris.unige.it>

# Preliminary remarks

- we take as a reference Linux **distribution** the Ubuntu 22.04 Long-term support (LTS), codename Jammy Jellyfish, **server version** (Ubuntu Server)
  - **distributions** are **customized versions** of Linux, and the most significant differences are
    - the **software package manager** (e.g., apt, dnf, pkg, ...)
    - **tools for configuring services** and keeping persistent their configurations (e.g., the network configuration manager)
    - some **paths** on the filesystem of the **tools/applications configuration files**
  - **server distros** are tailored for networks and services
    - they **do not include** a Graphical User Interface (**GUI**) and their installation involves a **minimal number of software packages**
    - they are deployed in datacenters (where servers run headless)

# Preliminary remarks

- we administer each Linux server using the **shell** and a (virtual) **terminal**
  - the **shell** is a program that takes **commands** from the **keyboard** and gives them to the operating system to perform
    - many shell implementations exist but we take **bash** (or **sh**) as our reference
  - a **terminal** lets you **interact** with the **shell**
    - a **console** is generally a terminal in the **physical** sense, i.e., the primary terminal directly connected to a machine
    - a **virtual terminal** uses specific **network protocols** (e.g., telnet, ssh, ...) to connect to a remote machine and allow users to interact with its shell

# man and tldr.sh

man command in Linux is used to display the user **manual** of any **command** that we can run on the terminal

- `man [command]` (e.g., `man sshd`)

**tldr pages** simplified and community-driven man pages.

- <https://tldr.sh/>

# Users and accounts

# Users and groups

Linux is a **multi-user** operating system and it can deal with several users simultaneously.

- each user needs an **account**, i.e., a login and (when required) a password
- users have a **personal environment** (e.g., a home directory, a shell, ...), which can be accessed only by them (and the system administrator and everyone knowing the password)
- inside the system the user is identified by the **user ID (UID)** and one or more **group IDs (GID)**

# Accounts

There are three types of accounts on a Linux system:

- **root** account: is also called superuser and would have complete and unfettered control of the system (in recent distributions often does not have a password for interactive login)
- **system** accounts: are those needed for the operation of system-specific components
- **user** accounts: provide interactive access to the system for users and groups of users

# User administration files

There are four main user administration files

- **/etc/passwd**: keeps the user account information
- **/etc/shadow**: holds the encrypted password of the corresponding account
- **/etc/group**: contains the group information for each account
- ~~**/etc/gshadow**: contains secure group account information~~



# Becoming root

To become **root** user **from** an **unprivileged** (normal user account) account when no root password is set, we can use the *substitute user do* command

- **sudo -s** (enter the password of your Ubuntu user when requested)

According to its configuration file (/etc/sudoers), in Ubuntu distros users that are **members of the sudo group** can become root user.

```
# (%)user(/group) hostname=(runas-user:runas-group) command
# members of group sudo, logged in to any hostname, may run, as any user
# or group, any command
%sudo    ALL=(ALL:ALL) ALL
```



# Filesystem

# Filesystem Structure

Linux uses

- a **hierarchical** file system structure (much like an upside-down tree)
- with **root** (/) at the base of the file system and all other directories spreading from there
- **directories** have **specific purposes** and hold the same types of information following a hierarchy standard, namely **Filesystem Hierarchy Standard (FHS)**

# File types

The following file-system objects can be found

- **normal** (text-)files
- **executable** files (binary files or shell scripts)
- **directories**
- **device** files: all physical devices (hard disks, DVD, USB, ...) are denoted by specific files
- symbolic or hard **links**: references to files
- **sockets**: used for inter-process communication (similar to TCP/IP sockets)
- (named) **pipes** (see later)

# Filesystem Hierarchy Standard (FHS)

Path	Description
/etc	configuration files (disk configuration, valid user lists, groups, network configuration, hosts...)
/dev	device files: special files that provide an interface to a device driver (e.g., disk partitions, printers, and serial ports)
/bin	executable files available to all users
/sbin	executable files (usually for system administration)
/lib	shared library files
/usr	additional commands and data files

# Filesystem Hierarchy Standard (FHS)

Path	Description
/var	variable-length files (e.g., log files).
/home	home directories
/boot	files for booting the system.
/tmp	temporary files.
/mnt	used to mount other temporary file systems.
/run	run-time variable data (e.g., running daemons)

# FHS: /proc

**/proc** (process information pseudo-file system) is a **virtual** filesystem.

- it does not contain real files but **runtime system information** (e.g., system memory, devices mounted, hardware configuration, ...)
- a lot of system utilities are simply calls to files in this directory (e.g., `lsmod` prints `/proc/modules`)
- by altering files located in this directory (`/proc/sys` or `/sys`) you can even read/change kernel parameters (see also `sysctl` command) while the system is running (see `/etc/sysctl.conf`).



# File information

While using `ls -lai` command, it displays various information related to file

inode	type	permissions	# links/dir	user	group	size	date	name
1984883	d	rw-rwxr-x	3	enrico	enrico	4096	set 25 13:59	.
1969658	d	rw-r-x---	24	enrico	enrico	4096	set 25 13:10	..
1984888	b	rw-rw-rw-	1	root	root	247,0	set 25 13:05	block_device
1984887	c	rw-rw-rw-	1	root	video	246,0	set 25 13:05	char_device
1984885	d	rw-rwxr-x	2	enrico	enrico	4096	set 25 13:53	dir
1984884	-	rw-rw-r--	2	enrico	enrico	10	set 25 13:53	file
1984884	-	rw-rw-r--	2	enrico	enrico	10	set 25 13:51	hard_link
1966437	-	rw-rw-r--	1	enrico	enrico	0	set 25 13:10	.hidden_file
1984886	l	rw-rwxrwx	1	enrico	enrico	4	set 25 13:03	link -> file
1984889	p	rw-rw-r--	1	enrico	users	0	set 25 13:06	pipe
1581	s	rw-rw-rw-	1	root	root	0	set 30 16:38	snspd.socket

# File ownership/permissions

- Every file/directory **belongs** to a specific **user** or a **group** of users
- Every user/group may have **permissions** to **read**, **write**, and/or **execute**

user	group	others
rwX	rwX	rwX

	File	Directory
r	the file can be read	the directory's contents can be shown.
w	the file can be modified	the directory's contents can be modified ( <u>requires the execute permission to be also set</u> )
x	the file can be executed	the directory can be accessed with cd

# File ownership/permissions: commands

- `chown [user.group] [file]`: **changes ownership** of a file or a directory
- `chmod` **changes** the `rwX mode` bits of a file or directory
  - `+/ -`: **adds** or **removes** the mode bits
  - `u`: sets the permissions for the **owner**
  - `g`: sets the permissions for the **group** that of the owner belongs to
  - `o`: sets the permissions for the **other** users
  - `a`: sets the permissions for **all**

For example, `chmod g+w file` (add the write permission to the group owner of file)

	u	g	o							
type	permissions			# links/dir	user	group	size	date		name
d	rw	xrwx	r-x	2	enrico	enrico	4096	set 25	13:53	dir
-	rw	-rw	-r--	2	enrico	enrico	10	set 25	13:53	file

# Special permissions

Run programs with **temporarily elevated privileges** in order to perform a specific task:

- user +s (or **SUID**): a file with SUID always **executes** as the **user who owns the file**, regardless of the user passing the command
- group +s (or **SGID**):
  - If set on a **file**, it allows the file to be **executed** as the **group that owns the file** (similar to SUID)
  - If set on a **directory**, any **files** created in the directory will have their **group ownership** set to that of the **directory owner**

**Sticky bit:**

- other +t: at the **directory level**, it **restricts file deletion**, i.e., Only the **owner** (and **root**) of a file can remove it within that directory

# Test SUID

SUID and SGID are forbidden on shell scripts

```
#include <stdio.h>
#include <unistd.h>

int main () {
    int real = getuid();
    int euid = geteuid();
    printf("The REAL UID =: %d\n", real);
    printf("The EFFECTIVE UID =: %d\n", euid);
}
```

```
gcc -o setuid setuid.c
```

The effective user ID determines what level of access the current process has.

# Filesystem commands and links

- create/remove file/dir: touch, rm, rmdir, mkdir, ...
- edit files: nano, vi (to exit: press `<Esc>`, Press `:` and use `q!` or `wq`)
- view files: cat, less, more, head, tail (use `tail -f` with logs), grep...
- find files: `find <options> <starting/path> <expression>`
  - `find / -name passwd`: find all files with name passwd starting from /
  - `find /etc -name passwd -exec wc -l {} \;`: find files with name passwd starting from /etc and for each found file count lines; use `{}` within the command to access the filename
  - `locate <filename>` (apt install mlocate and update the db with `updatedb`)
- `ln -s <path> <linkname>`: create a symbolic link

# Mounting a file system

A file system must be **mounted** in order to be usable by the system.

- `mount`: see what is currently mounted (available for use) on the system.
- `mount -t <file_system_type> <device_to_mount> <directory_to_mount_to>`: mount a file system contained in the `<device_to_mount>` to the directory `<directory_to_mount_to>`
  - e.g., `mount -t iso9660 /dev/cdrom /mnt/cdrom` (mounting a cdrom).
- `umount <device/directory_mounted>`: unmount a filesystem
  - e.g., `umount /dev/cdrom`

# Compress and extract files: tar (gzip/bzip)

- `tar -cvzf name-of-archive.tar.gz /path/to/directory-or-file1 .. /path/to/directory-or-fileN`: compress one (or multiple) entire directory or one (or multiple) file on Linux (c: create, v: display progress, z: compress with gzip, f: specify archive file name)
  - use j for compressing with bzip
  - `--exclude=/path/to/directory-or-file1`: exclude directory or file
- `tar -xvzf name-of-archive.tar.gz`: extract archive
  - `--strip 1`: strip off the first directory and extract the rest (`tar -xvz --strip 1 -f <archive>.tar.gz`)



# Filesystem-related commands

Command	Description
<code>pwd</code>	prints current working directory
<code>ls</code>	lists the contents of a directory
<code>cd</code>	change the current path to the destination directory
<code>mkdir</code>	makes a new directory
<code>rmdir</code>	removes an empty directory
<code>cp</code>	copy file or directory
<code>mv</code>	move/rename file or directory

# Filesystem-related commands

Command	Description
wc	word, line, character, and byte count
more	paging through text one screenful at a time
less	improved version of more allows backward/forward movement
head	display first lines of a file
tail	display last lines of a file
grep	print lines in a file matching a pattern

# Processes

# Processes

*“On a UNIX system, everything is a file; if something is not a file, it is a process.”*

— Machtelt Garrels, [Introduction To Linux: A Hands On Guide](#)

- Whenever you execute a program (a command), Linux starts a new **process**
- The operating system tracks processes through a unique five-digit ID number known as the **pid** (or the process ID)
- show processes:
  - `ps -f`: processes of current user
  - `ps -ef`: all processes, `ps -aux`: all processes, BSD style
  - `top`: realtime

# Processes [2]

- **foreground** processes: by default, every process that you start runs in the foreground. It gets its **input** from the **keyboard** and sends its **output** to the **screen**
- **background** processes<sup>1</sup>: a background process runs **without being connected to the terminal** (if it requires an input, it waits). Adding an ampersand & at the end of the command starts it as a background process
  - jobs (list background processes executed from the current shell)
  - fg <jobid>: put the job in foreground
  - bg or CTRL-Z: put the job in background
  - kill %<jobid> or CTRL-C: kill the job
- kill <pid>: kill a process (If a process ignores a regular kill command, add -9)

<sup>1</sup> **daemons**: processes that run in the background and are not interactive (they have no controlling terminal)

# Background processes: example

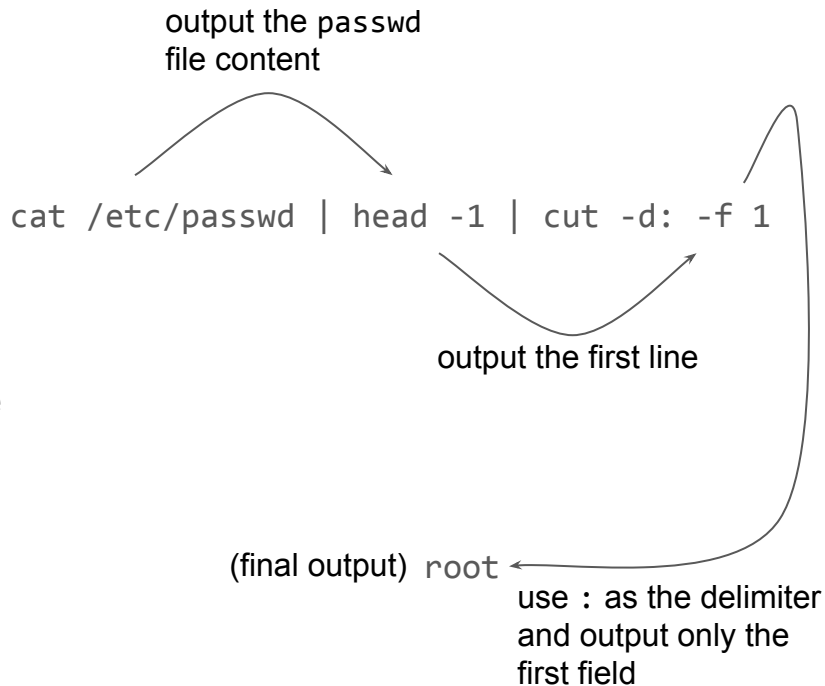
- `./loop.sh` (start in foreground and sends its output to the screen)
- CTRL-Z: put the job in background (but stopped)  
[1]+ Stopped ./loop.sh
- `bg`: run the process in background (it sends its output to the screen)  
[1]+ ./loop.sh &
- `jobs`: list jobs  
[1]+ Running ./loop.sh &
- `kill %1`: kill the job [1]  
[1]+ Terminated ./loop.sh

```
#!/bin/sh

# loop.sh
while true
do
    echo "hello"
    sleep 4
done
```

# Pipes and Filters

- You can connect two commands together so that the output from one program becomes the input of the next program. Two or more commands connected in this way form a **pipe**
- To make a pipe, put a vertical bar `|` on the command line between two commands
- When a program takes its input from another program, it performs some operation on that input, and writes the result to the standard output. It is referred to as a **filter**
- **named pipes** provide such communication to processes using a special file (it is subjected to permission checks)



# Packages and services



# Advanced Package Tool (APT)

A packaging system is a way to provide programs and applications for installation (without building a program from source). Debian derivatives use the dpkg format and apt for interacting with the packaging system.

- `apt update`: updates the database of available packages
- `apt install <package-name>`: install <package-name>
- `apt search <str>`: search a package having <str> in the name or description
- `apt remove <package-name>`: remove <package-name>

Find the package providing a specific file: `dpkg -S <file>`, list packages: `dpkg -l`

# systemd

systemd<sup>1</sup> is a Linux **initialization system** and **service manager**

- `systemctl`: show services status
- `systemctl start/stop/status <unit_name>`: start/stop/view a service

`journalctl` is a utility for **querying** and **displaying logs** from `journald`, the logging service of `systemd`.

- `journalctl -u <unit_name>`: message from specific unit (`journalctl --field _SYSTEMD_UNIT` list availables units)
- `journalctl path/to/executable`: message from specific executable
- `journalctl -f`: follow new messages (like `tail -f`)

<sup>1</sup><https://freedesktop.org/wiki/Software/systemd/>

# Networking

# Networking

`ip` command<sup>1</sup>: show/manipulate routing, devices, policy routing and tunnels

- `ip a`: show addresses
- `ip r`: show routes
- `ip route add default via 192.168.53.2`: add default gw
- `ip addr add 192.168.53.100/24 dev ens33`: add new address
- `ip addr del 192.168.53.100/24 dev ens33`: delete address

Configurations with the `ip` command are not persistent!

<sup>1</sup>[https://access.redhat.com/sites/default/files/attachments/rh\\_ip\\_command\\_cheatsheet\\_1214\\_jcs\\_print.pdf](https://access.redhat.com/sites/default/files/attachments/rh_ip_command_cheatsheet_1214_jcs_print.pdf)

# Netplan

netplan is a utility for easily configuring networking on a linux system (<https://netplan.io/>). It reads network configuration from `/etc/netplan/*.yaml` (see <https://netplan.io/examples/> for examples).

DHCP	Static
<pre>network:   version: 2   ethernets:     enp3s0:       dhcp4: true</pre>	<pre>network:   version: 2   ethernets:     enp3s0:       addresses:         - 10.10.10.2/24       routes:         - to: default           via: 10.10.10.1       nameservers:         search: [mydomain, otherdomain]         addresses: [10.10.10.1, 1.1.1.1]</pre>

# Test network configuration

- apply netconf configuration

- `netplan apply`

- check connectivity

- `ping [address]`

- check nameservers

- `host [name]`

# Register names (locally)

The `/etc/hosts` is a plain text file that maps hostname to ip addresses.

```
127.0.0.1 localhost
```

```
127.0.1.1 vcc
```

```
192.168.58.2 gw gw.my.net
```

```
enrico@vcc:~$ ping gw.my.net
```

```
PING gw (192.168.58.2) 56(84) bytes of data.
```

```
64 bytes from gw (192.168.58.2): icmp_seq=1 ttl=128 time=0.179 ms
```

```
64 bytes from gw (192.168.58.2): icmp_seq=2 ttl=128 time=0.360 ms
```

```
64 bytes from gw (192.168.58.2): icmp_seq=3 ttl=128 time=0.397 ms
```

\* Windows: `C:\Windows\System32\drivers\etc\hosts`

# Remote terminal



# SSH client

## Windows

- openssh client
  - from **powershell** (as administrator): `Get-WindowsCapability -Online | ? Name -like 'OpenSSH*' (check) and Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0 (install)`

connect with `ssh user@[address]`

- putty
  - <https://www.putty.org/>



# SSH pubkey authentication

Public key authentication is a way of logging into a remote account using a cryptographic key rather than a password.

## Client side (WINDOWS)

- generate a keypair: `ssh-keygen`
- copy the pub key to the server (using secure-copy from ssh): `scp id_rsa.pub enrico@192.168.58.100:.`

## Server side (LINUX)

- create a `.ssh` directory (`mkdir ~/.ssh`)
- move `id_rsa.pub` in a new `~/.ssh/authorized_keys` file (`mv id_rsa.pub ~/.ssh/authorized_keys`)

# Shell scripting

# Shell scripting

A shell script is a program designed to be run by the Linux shell.

```
1.  #!/bin/bash
2.  # print current dir
3.  pwd
4.  # list files
5.  ls
6.  echo "end"
```

Line 1: **shebang** construct, specify the interpreter (sh shell)

Line 2,4: comments

Line 3,5: commands (listed in the order of execution)

Line 6: print the "end" string

# Variables

The name of a variable can contain only letters (a to z or A to Z), numbers ( 0 to 9) or the underscore character ( \_).

1. `#!/bin/bash`
- 2.
3. `NAME="Enrico" # assign value`
4. `echo $NAME # print value (use $)`
5. `A=10`
6. `OP=$(expr $A '*' 2) # assign the output (man expr) to OP`
7. `TMP=`ls -l` # you can also use `` (backtick) instead of $()`

# Environment variables

- Every shell has a set of attached variables
  - **system-defined** (e.g., \$PATH contains an ordered list of paths that Linux will search for executables when running a command)
  - **user-defined**: export command **promotes** a shell **variable** to an **environment variable**
- They are defined for the **d** and are inherited by any child shells or processes

# Special variables

Name	Description
\$0	The filename of the current script
\$n	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is \$1, the second argument is \$2, and so on)
\$#	The number of arguments supplied to a script
\$*	All the arguments are double quoted. If a script receives two arguments, \$* is equivalent to "\$1 \$2"
\$@	All the arguments are <i>individually</i> double quoted. If a script receives two arguments, @\$ is equivalent to "\$1" "\$2"
\$?	The exit status of the last command executed
\$\$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing
\$!	The process number of the last background command

# Operations

We can use external programs to perform basic (arithmetic, boolean, string, file test) operations.

- `expr` (man `expr`)
  - `expr 10 '/' 3`
- `test` (man `test`)
  - `test -f /etc/passwd; echo $?:` test if `/etc/passwd` is a file; the exit status is 0 for true
  - `test -d /tmp/testdir || mkdir /tmp/testdir:` create `/tmp/testdir` if it does not exist
  - `test -d /tmp/testdir && rmdir /tmp/testdir:` remove `/tmp/testdir` if it exists
  - `test 10 -gt 50; echo $?:` test if 10 is greater than 50



# Redirect Input/Output

- `cat /etc/passwd > /tmp/users:` **redirects the output** of the command in the `/tmp/users` file
- `echo "test" >> /tmp/users:` **appends the output** in an existing file
- `wc -l < /tmp/users:` count the number of lines in the file by **redirecting the standard input** of the `wc` command from the file `/tmp/users`
- `command > /dev/null:` **discard the output**
- `command > /dev/null 2>&1:` **discard** both **output** of a command and its **error output** (2 represents `STDERR` and 1 represents `STDOUT`)
- `echo message 1>&2:` display a message on `STDERR` by **redirecting** `STDOUT` into `STDERR`

# if..then..else

The base for the 'if' constructions is:

```
if [ expression ]; then  
    code if 'expression' is true.  
else  
    code if 'expression' is false.  
fi
```

```
1.  #!/bin/bash  
2.  a=100  
3.  if [ $a -gt 50 ]; then  
4.    echo "yes"  
5.  fi  
  
6.  b="enrico"  
7.  if [ $b == "Enrico" ]; then  
8.    echo "equal"  
9.  fi  
  
10. ls /tmp/nonexistent  
11. if [ $? == 0 ]; then  
12.   echo "yes"  
13. else  
14.   echo "no"  
15. fi
```

# while

The base for the '**while**' constructions is:

```
while [ expression ];  
do  
  code if 'expression' is true.  
done
```

```
1.  #!/bin/bash  
2.  i=0  
3.  while [ $i -lt 10 ];  
4.  do  
5.    echo "i: $i"  
6.    i=$(expr $i '+' 1)  
7.  done
```

```
1.  #!/bin/bash  
  
2.  a=0  
3.  while true # infinite loop  
4.  do  
5.    echo "now: $a"  
6.    sleep 1  
7.    if [ -f /tmp/exit ]; then  
8.      echo "bye."  
9.      exit  
10.   fi  
11.   a=$(expr $a '+' 1)  
12. done
```

# for

The base for the **'for'** constructions is:

```
for variable in [ expression ]  
do  
  code using $variable  
done
```

Some examples of [ expression ]:

- `1 2 3 4 5..N` (a numeric range)
- `string1 string2..stringN` (strings)
- `$(a_cmd_here)` (the output of a command)
- `{0..10..2}` (a range with a step)

```
1.  #!/bin/bash  
2.  for i in {0..10..2}  
3.    do  
4.      echo "Welcome $i times"  
5.    done  
  
6.  for e in $(ls -1 /etc)  
7.    do  
8.      if [ -d "/etc/$e" ]; then  
9.        echo "$e is a directory"  
10.     fi  
11.  done
```

# for (example)

```
if [ $# -lt 1 ]; then
    echo "This command count the entries of a list of directories."
    echo
    echo "$0 [listofdir]"
    echo
    echo "Please specify a list of directories."
    exit
fi

for e in "$@"
do
    tmp=$(ls -1 $e | wc -l)
    echo "$e has $tmp entries."
done
```

# Useful commands

sed	stream editor for filtering and transforming text
cut	cut out fields from `stdin` or files
tr	translate characters
sort	sort lines of text files
wc	count lines, words, or bytes
uniq	output the unique lines from the given input or (sorted) file
xargs	execute a command with piped arguments coming from another command
egrep	return lines that contain a pattern matching a given regular expression
logger	log a message to syslog

# Exercises

1. create a script that accepts a filename *f*, a color *c* (red, blue, yellow, white or green) and an integer *i* as args and return *true* iff *c* appears *i* times in *f* (check if args are valid!).

A sample file follows:

```
red
blue
yellow
red
green
green
red
```

2. create a script that accepts an optional arg *a*. if *a* is the string 'empty' shows all the Linux accounts without password or show others otherwise.
3. create a script that accepts a program name *n* and a message *m* as args. It loops checking if *n* is running. If *n* is not running, logs for 3 times the message *m*.

# Further Readings

- Ubuntu Server Guide (<https://ubuntu.com/server/docs>)
- The Linux Command Line (<http://linuxcommand.org/tlcl.php>)
- GNU Bash Manual (<https://www.gnu.org/software/bash/manual/>)
- Advanced Bash-Scripting Guide (<https://tldp.org/LDP/abs/html/>)
- 25 Free Books To Learn Linux For Free  
(<https://itsfoss.com/learn-linux-for-free/>)