Operating Systems 2021/2022

TP Class 01 – Introduction, Linux and C programming

Vasco Pereira (vasco@dei.uc.pt)
Dep. Eng. Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

```
operating system
noun
the collection of software that directs a computer's operations,
controlling and scheduling the execution of other programs, and
managing storage, input/output, and communication resources.

Abbreviation:  OS

                                       Source: Dictionary.com
```

1 2 9 0
Departamento de Engenharia Informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Updated on
17 February 2022

INTRODUCTION

These classes…

- Outline of TP lessons
  - Practice-oriented topics
    - Technologies and tools related with assignments
  - Discussion of assignments
  - Resolution of problems and questions
  - Support and assistance to students assignments
  - Resolution of exercises from T Classes

LINUX

How to get Linux in your computer

- Install a Linux distribution
  - Any PC/server flavor will work
- In Windows… use Cygwin*might need some tweaking
- In Mac… you might not need it! *might need some tweaking
- Use a LiveCD
- Use a Virtual Machine (preferred solution if you don't want to risk breaking something!)
  - VMWare, VirtualBOX, …

How to get Linux in your computer

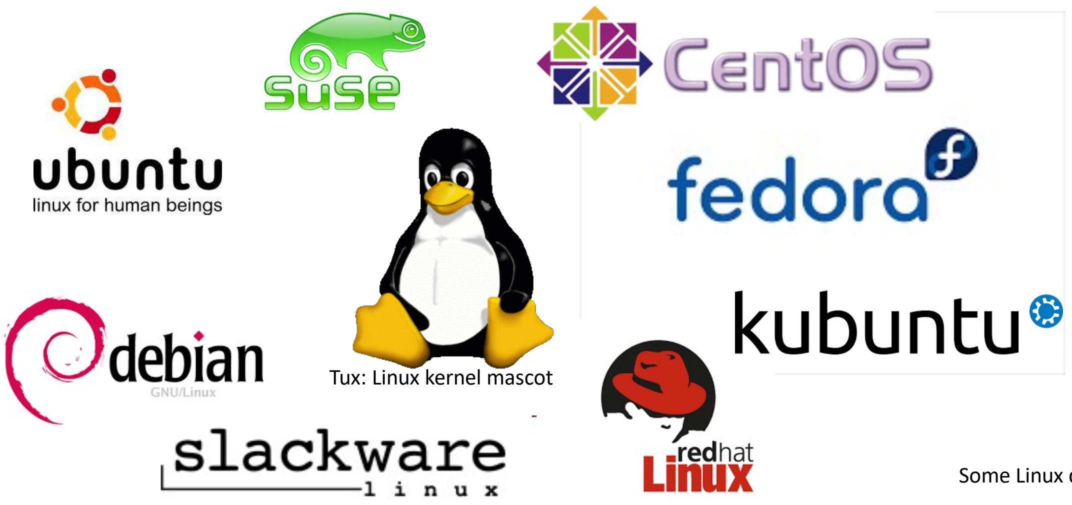
- However, for classes…
  - Ubuntu/LUbuntu was chosen for classes
  - Some options:
    - Install Linux as your main operating system or in dual boot
    - Use a Virtual Machine with a LUbuntu/Ubuntu distribution
      - Many software hypervisors available (e.g. VMWare, VirtualBOX)
      - Preferred solution if you don't want to risk breaking something!
      - Student's host system continues available
    - Create your own Virtual Machine with Linux
      - You should try to install a operating system from scratch
      - A Tutorial is given to help you
    - Use the Linux VM supplied - has all course sw already installed
    - Use student**2**.dei.uc.pt

Best option for classes!

Warning: the project code submitted must run in student2 or in the Linux VM supplied!

Introducing Linux

- Linus Trovalds created the Linux core, the kernel
  - Additional software (utilities, libraries, GUI, etc.) programs and documentation bundled together with a Linux core make different Linux distributions



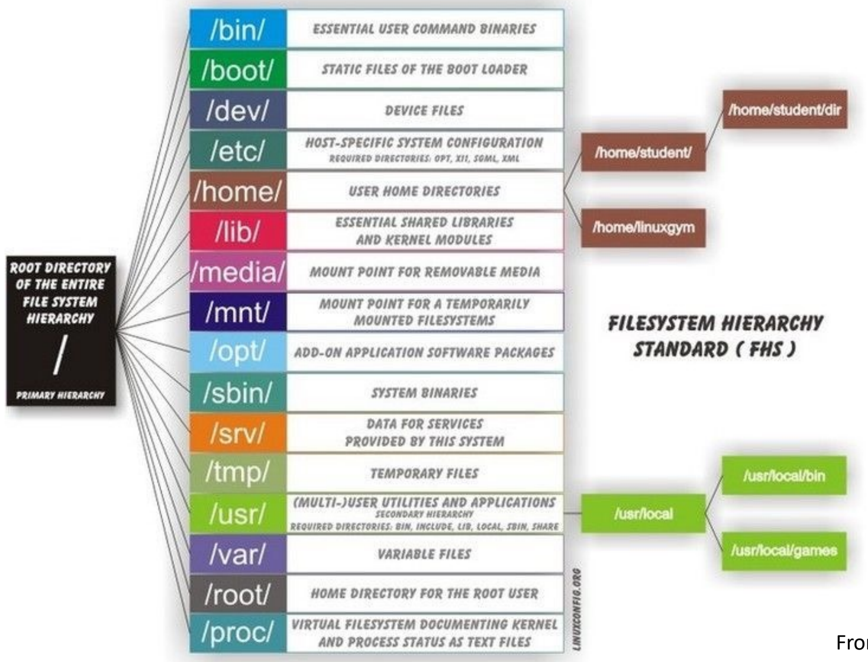
Some Linux distributions

Introducing Linux

- Linux is multitask and multiuser
- Linux components
  - Kernel: OS core
  - Filesystem: deals with files
  - Shell: the command interpreter

Introducing Linux
Filesystem

- Standard hierarchy in a Linux file system


/bin/
ESSENTIAL USER COMMAND BINARIES
/boot/
STATIC FILES OF THE BOOT LOADER
/dev/
DEVICE FILES
/etc/
HOST-SPECIFIC SYSTEM CONFIGURATION
REQUIRED DIRECTORIES: OPT, X11, SGML, XML
/home/
USER HOME DIRECTORIES
/lib/
ESSENTIAL SHARED LIBRARIES AND KERNEL MODULES
/media/
MOUNT POINT FOR REMOVABLE MEDIA
/mnt/
MOUNT POINT FOR A TEMPORARILY MOUNTED FILESYSTEMS
/opt/
ADD-ON APPLICATION SOFTWARE PACKAGES
/sbin/
SYSTEM BINARIES
/srv/
DATA FOR SERVICES PROVIDED BY THIS SYSTEM
/tmp/
TEMPORARY FILES
/usr/
(MULTI-)USER UTILITIES AND APPLICATIONS
SECONDARY HIERARCHY
REQUIRED DIRECTORIES: BIN, INCLUDE, LIB, LOCAL, SBIN, SHARE
/var/
VARIABLE FILES
/root/
HOME DIRECTORY FOR THE ROOT USER
/proc/
VIRTUAL FILESYSTEM DOCUMENTING KERNEL AND PROCESS STATUS AS TEXT FILES
ROOT DIRECTORY OF THE ENTIRE FILE SYSTEM HIERARCHY
/
PRIMARY HIERARCHY
/home/student/
/home/student/dir
/home/linuxgym
/usr/local
/usr/local/bin
/usr/local/games
FILESYSTEM HIERARCHY STANDARD (FHS)
LINUXCONFIG.ORG

From: https://linuxconfig.org/filesystem-basics

Introducing Linux
Filesystem

- Each directory has, at least, 2 files
  - "." – current directory
  - ".." – parent directory
  - "~" – accesses user home directory
    - E.g.

```
cd ~/so          ->      cd /home/vasco/so
cd ~vasco/so     ->      cd /home/vasco/so
```

- Files are case-sensitive
- Everything is a file
- Every program executed by the shell starts with 3 open file descriptors (0,1,2) = (*stdin*, *stdout*, *stderr*)

Introducing Linux

Filesystem

- File types
  - **Regular files** – files that contain user/programs data, programs, text, etc.
  - **Directories** – files with information about files or other directories below
  - **Special files** – support block devices, character devices, named pipes, Unix domain sockets. Enables access to devices such as terminals, printers, disks, etc,. and are accessed in the same way as other files

Introducing Linux

Filesystem

- Filesystems supported
  - Linux supports different filesystems including physical, virtual and network ones
    - Examples:
      - NFS – Network File System
      - ext2, ext3, ext4 – Extended filesystem
      - XFS
      - FAT
      - …
  - The LUbuntu 20.04 uses *ext4* filesystem

Introducing Linux
Filesystem

- Permissions
  - The standard permissions and security mechanism in Linux is the same as that in Unix

```
user1@ubuntu:~/Desktop$ ls -la
drwxr-xr-x  3 user1 group1 4096 Sep 10 16:19 .
drwxr-xr-x 10 user1 group1 4096 Sep  4 02:22 ..
drwxrwxr-x  2 user1 group1 4096 Sep 15  2015 temp
-rwxrwxr-x  1 user1 group1 7383 Sep 10 16:19 trab01
-rw-rw-r--  1 user1 group1  558 Sep 10 15:31 trab01.c
1    2      3   4      5     6       7         8
```

1 - file type
2 - file permissions
3 - hard links to file
4 - file owner
5 - file group
6 - file size (bytes)
7 - time of last modification
8 - file name

```
 rwx  rwx  rwx
 111  111  111
user group others
```

Permissions: read (r) ; write(w) ; execute(x)

File types

–	: regular file
b	: block special file
c	: character special file
d	: directory
l	: symbolic link
n	: network file
p	: FIFO
s	: socket

Introducing Linux
Filesystem

- Permissions (cont.)
  - Change permissions
    - `chmod {ugoa}{+-}{rwx} {file}`
      - **u**=user who owns file, **g**=users in file group, **o**=other users, **a**=all
      - '+'= give permission; '-'= remove permission
    - `chmod {octal mode} {file}`
    - E.g.
      - `$ chmod 777 temp      # rwx rwx rwx`
      - `$ chmod 700 temp      # rwx --- ---`
      - `$ chmod 660 temp      # rw- rw- ---`
      - `$ chmod u+r temp      # adds owner r permission`
      - `$ chmod g+rwx temp    # adds group rwx permission`
      - `$ chmod g-wx temp# removes wx permission from group`
    - Example of octal conversion: 6 (octal) = 110 (binary) => rw-
    - Use `umask` to define a mask for new files

Introducing Linux

Basic shell commands (I)

- Getting help
  - **man** keyword

```
# online manual
# shows them command page for the keyword
#specified

# Navigation in manual:
#     'b': previous page
#     'space': next page;
#     '/' search
#     Keyboard arrows: up/down 1 line
#     'q' quit
```

Introducing Linux

Basic shell commands (I)

- Getting help (2)
  - **man** *[section number]* keyword
    - A manual is divided into multiple sections; commands with the same name can be found in different sections.
    - By specifying a section, only a specific section of a manual is shown.
    - Linux manual sections:
      - 1: General commands
      - 2: System calls
      - 3: Library functions
      - 4: Special files
      - 5: File formats and conventions
      - 6: Games and screensavers
      - 7: Miscellaneous
      - 8: System administration commands and daemons

Introducing Linux

Basic shell commands (I)

- Getting help (3)
  - **man** –k Keyword — # The option "-k" searches the given keyword as # a regular expression in all the manuals and # returns the manual pages with the section # number in which commands are found.
  - **apropos** *keyword* — # search the manual page names and # descriptions to find the right man page

Introducing Linux

Basic shell commands (II)

- Work with directories and files

pwd	# print working directory
ls *[-a] [wildcard]*	# list directory contents
cd *directory*	# change directory
mkdir *directory*	# make a new directory
rmdir *directory*	# remove a directory
cp *file1 file2*	# copy file(s)
mv *file1 file2*	# move file(s)
rm *file*	# remove file(s)
ln *[-s] file linkname*	# make a symbolic link to a file/directory

Introducing Linux

Basic shell commands (III)

cat *file*	# concatenate and print files to *stdout*
more *file*	# print files to *stdout* page by page
grep *[-n] "string" file*	# print lines from a file that match a pattern
diff *file1 file2*	# compare two files
sort *file*	# sort a file
cut -c list} {-f list -d }	# cut parts from a file
wc *{-lwc} file*	# line, word, character count
file *file*	# determine the file type
head *file*	# shows the start of a file
tail *file*	# shows the end of a file
chmod *{ugoa} {+-} {rwx} file*	# change the permissions mode of a file
chown *[owner][:group] file(s)*	*# change file owner and group*

Introducing Linux

Basic shell commands (IV)

- Working with processes

ps [-a]	# process status
Kill -{*SIGNAL*} pid	# sends a signal to a process
	# (*SIGNAL*=-9 terminates a process)

- Working with users

who or **w**	# display who is logged in the system
whoami	# who am I
passwd	# change user password

- Editors

nano *file*	# file editor
vi *file*	# file editor
emacs *file*	# file editor

Introducing Linux

Basic shell commands (V)

- Other

date	# print date & time
cal *month year*	# calendar
find *dir {-name file -print}*	# find a file starting at directory *dir*
sleep *seconds*	# suspend execution for n seconds
lpr *-Pprinter file*	# send a job to the printer
echo *'string'*	# echo arguments to the standard output
clear	# clear the terminal screen
csh *script*	# C-shell command interpreter
uname	# print system information
du	# estimate file space usage
df	# report file system disk space usage
	# use –*K* for 1K blocks
	# use –*T* to include the file system type

Introducing Linux

Basic shell commands (VI)

Ctrl-c	# cancel a foreground process
Ctrl-z	# suspend a foreground process
command **&**	# put a process in background
jobs	# list the background processes
fg	# resume to foreground a stopped process
fg %job_number	# bring a background job into the foreground
bg	# put in background a stopped process

- Redirection (> output < input)

command **>** file	# redirect *stdout*
command **>>** file	# redirect *stdout* but append
command **>&** file	# redirect *stdout* and *stderr*
(command > file1) > &file2	# redirect *stdout* to file1 and *stderr* to file2
command **<** file	# redirect *stdin*
command1 **|** command2	# pipe between two commands

Introducing Linux
Basic shell commands (VII)

- Some metacharacters
  - *****   # file substitution wildcard; zero or more characters
  - **?**   # file substitution wildcard; one character
  - **[]**   # file substitution wildcard; any character between brackets
  - ;   # command separator

  - E.g.

```
$ ls
a  aaa.c  aa.c  aba.c  aea.c  bb  cde.c  dd.txt
$ ls *.c
aaa.c  aa.c  aba.c  aea.c  cde.c
$ ls a*
a  aaa.c  aa.c  aba.c  aea.c
$ ls a?a.c
aaa.c  aba.c. aea.c
$ ls a[eb]a.c
aba.c  aea.c
$ ls ??.*
aa.c  dd.txt
```

Introducing Linux
Basic shell commands (VIII)

- sudo
  - Gives root previleges to any particular command that a user wants to execute; it demands the user password; not all users have permissions to use sudo
- su (substitute user)
  - Is used to switch from one account to another. The user will be prompted for the password of the user switching to.
  - If user types only 'su' without any option then It will be considered as root and user will be prompted to enter root user password.

Introducing Linux

Basic shell commands (IX)

- Handling packages
  - Installing new packages
    - sudo apt install {package}
  - Remove packages but leave configuration files
    - sudo apt remove {package}
  - Remove packages and configuration files
    - sudo apt purge {package}
  - Cleaning old packages
    - To delete downloaded packages already installed (and no longer needed)
      - sudo apt clean
    - To remove all stored archives in cache, for packages that can not be downloaded anymore:
      - sudo apt autoclean
    - To remove all unnecessary packages
      - sudo apt autoremove
    - To delete old kernel versions
      - sudo apt remove --purge linux-image-X.X.XX-XX-generic
    - If you don't know which kernel version to remove
      - dpkg --get-selections | grep linux-image
  - Note:
    - Before Ubuntu 16.04 use **`apt-get`** command instead of **`apt`**

BASIC SHELL PROGRAMMING

Basic Shell Programming

- Linux Shell
  - The shell is the command interpreter
  - Some shells used in Linux:
    - sh: Bourne shell
    - csh: C shell has a syntax modeled after C programming language
    - bash: Bourne-again shell, a new version of the Bourne Shell

Basic Shell Programming

- Basic Shell Programming in `/bin/bash`
    - A shell script is a series of commands written in a plain text file
    - A shell script can take input from user, file and output them on screen
  - Why writing shell scripts?
    - Useful to create our own commands
    - To automate some tasks and save time

Basic Shell Programming

- Bash shell initialization files
  - /etc/profile
    - List of commands to be executed in users login. Usually sets variables such as PATH, USER or HOSTNAME.
  - ~/.bash_profile
    - Used to configure users environments individually
  - ~/.bash_login
    - Setting that are only executed when the user logs into the system
  - ~/.profile
    - In the absence of ~/.bash_profile and ~/.bash_login, which are bash specific, ~/.profile is read. It can hold the same configurations, which are then also accessible by other shells. It was the original profile configuration for the Bourne shell.
  - ~/.bashrc
    - Script executed when a new shell is opened
  - ~/.bash_logout
    - Contains specific instructions for the logout procedure

Individual user initialization files

Basic Shell Programming

- Changing shell initialization files
  - When changing files the user must reconnect to the system (make a new login) or **source** the changed file so that changes are applied.
  - E.g.:
    - `source  .bashrc`

- **NOTE**:
  - Not all user individual files always exist by default. If you need, create them. Use `ls -a` command to see all files in you home directory.

Basic Shell Programming

- Variables in bash
  - Set or change the value of a variable
    - `var_name=value   (no spaces!)`
    - E.g.:
      - `VAR1="Operating Systems"`
      - `VAR2=10;`
  - Precede the variable with *$* to use it
    - E.g. :
      - `echo "This class is about" $VAR1`
      - `echo "This class is about" ${VAR1}`
  - Unset a variable
    - `unset var` *(no $ before the variable)*
    - E.g. :
      - `unset VAR1`

Basic Shell Programming

- Global variables
  - Global or environment variables are available in all shells. Use `env` or `printenv` to display environment variables.
- Local variables
  - Are only available in the current shell
  - NOTE: Use command `set` with no arguments to see all variables
- Export
  - A variable created in a shell with `var=value` is only available in that shell. Child processes will not be aware of the variable. In order to pass the variables to a subshell, use `export` command.
    - `export variable[=value]`
    - (changes made by child processes do not affect variable values in the parent process)

Basic Shell Programming

- Some of the reserved shell variables
  - **HOME**: current user directory
  - **PATH**: A colon-separated list of directories in which the shell looks for commands
  - **PS1**: The primary prompt string which is displayed before each command. The default value is "'\s-\v\$ '”
    - NOTE: Check the "PROMPTING" section of the Bash man page for a complete list of escape sequences.
      - `man bash`
      - `/PROMPTING` # to find PROMPTING in the man page
  - **PS2**: secondary prompt displayed when a command needs more input (e.g. a multi-line command).

Basic Shell Programming

- My first script

```
#!/bin/bash
# declare STRING variable
STRING="Hello World"
# print variable on a screen
echo $STRING
```

hello_world.sh

- Make the file executable

```
chmod +x hello_world.sh
```

- Execute the script

```
./hello_world.sh
```

Basic Shell Programming

- Some notes:
  - The first line “`#!/bin/bash`” indicates which interpreter should be used. If we don't know where our interpreter is located `which` command may be used to find out.
  - To insert comments in the script use `#`
  - Normally Bash scripts have the extension `.sh`
  - When we type a command on the command line, the system runs through a preset series of directories, looking for the program we specified. These directories are defined in the variable PATH. If the current directory is not in the PATH, it will not be looked. However, if a path is given, the file is directly accessed.
    - As “`.`” denotes the current directory, “`./hello_world.sh`” accesses the file in the current directory. An absolute path could also be given.

Basic Shell Programming

- My first script with command line arguments

```
#!/bin/bash
echo Script = $0
echo My name is $1 and I live in $2
echo $# arguments inserted: $*
```

arg_test.sh

  - **`$0`** - The name of the script
  - **`$1 – $9`** - Command line arguments given to the script
  - **`$#`** - number of command line arguments given to the script
  - **`$*`** - All of the command line arguments

- Execute the script (example)

```
./arg_test.sh Vasco Coimbra
```

Basic Shell Programming

- Back ticks
  - Execute shell commands from within a script

```
#!/bin/bash
# use back ticks " ` ` " to execute shell command
echo `uname -a`
# save command in a variable
lines=`ls| wc -l`
echo Number of lines is $lines
```

backticks.sh

Basic Shell Programming

- If-else-fi

```
if [ <some test> ]
then
<commands>
fi
```

Or (more compact)

```
if [ <some test> ]; then
<commands>
fi
```

- If-elif-else-fi

```
if [ <some test> ]
then
<commands>
elif [ <some test> ]
then
<different commands>
else
<other commands>
fi
```

Basic Shell Programming

- Some possible tests (find more in the online manual)

```
! EXPRESSION             # The EXPRESSION is false.
-n STRING                # The length of STRING is greater than zero.
-z STRING                # The lengh of STRING is zero (ie it is empty).
STRING1 = STRING2        # STRING1 is equal to STRING2
STRING1 != STRING2       # STRING1 is not equal to STRING2
INTEGER1 -eq INTEGER2    # INTEGER1 is numerically equal to INTEGER2
INTEGER1 -gt INTEGER2    # INTEGER1 is numerically greater than INTEGER2
INTEGER1 -lt INTEGER2    # INTEGER1 is numerically less than INTEGER2

-d FILE        # FILE exists and is a directory.
-e FILE        # FILE exists.
-r FILE        # FILE exists and the read permission is granted.
-s FILE        # FILE exists and it's size is greater than zero (ie. it is not
empty).
-w FILE        # FILE exists and the write permission is granted.
-x FILE        # FILE exists and the execute permission is granted.
```

- Boolean operators
  - and - &&
  - or - ||

Basic Shell Programming

- E.g.:

```
#!/bin/bash
# tests if the number sent as an argument is >=1000
if [ $1 -ge 1000 ]      # ge = greater or equal
then
echo Hey it\'s a big number.
else
echo Does not reach 1000!
fi
```

Basic Shell Programming

- While

```
while [ <some test> ]
do
<commands>
done
```

  - E.g.:

```
#!/bin/bash
counter=1
while [ $counter -le 10 ]
do
echo $counter
counter=`expr $counter + 1`      # bash also supports ((counter++))
done                             # or let counter=$counter+1
```

Basic Shell Programming

- For

```
for var in <list>
do
<commands>
done
```

  - E.g.:

```
#!/bin/bash
names="Coimbra Lisboa Porto"
for city in $names
do
echo $city
done
```

COMPILING AND DEBUGGING C

Compiling C programs

- Syntax

  ```
  gcc -Wall file1.c file2.c -o trab01
  ```

  - `file1.c` and `file2.c` contain the source code
  - `trab01` is the name of the executable created

  - `-Wall` : includes most of the warnings
  - ATTENTION: Solve every error and warning that is shown by the compiler. Do not ignore warnings! Normally they result from poor and error-prone coding!

- `gcc -Wall ... -c`
  - Object files ".o"

Compiling with makefile

- Make

```
FLAGS   = -Wall -g
CC      = gcc
PROG    = trab02
OBJS    = trab02.o queue.o

all:     ${PROG}

clean:
         rm ${OBJS} ${PROG} *~

${PROG}:         ${OBJS}
         ${CC} ${FLAGS} ${OBJS} -o $@

.c.o:
         ${CC} ${FLAGS} $< -c

###########################

queue.o:  queue.h queue.c

trab02.o: queue.h trab02.c

trab02: queue.o trab02.o
```

Debugging

- Why use a debugger?
  - It helps the programmer to find and fix errors in programs.
- How to debug C in Linux?
  - Compile using option –g (debug information)
    - E.g. `gcc –Wall –g prog1.c –o prog1`
  - Use gdb (GNU debugger)
    - `gdb {executable}`   E.g. `gdb trab01`
    - `gdb --core={core}`
    - Core dump size
      - `ulimit –a` //get current maximum size
      - `ulimit -c {size/unlimited}` //set maximum core dump size (in Mac OS look in /cores/)
  - Some functionalities of gdb
    - Inspect program state
    - Create breakpoints in execution
    - Run step by step

Commands and more information: See “GNU debugger” tutorial and “Assignment 01”!

Class demos included

- C examples:
  - User input
  - Accessing files
  - …
- Shell scripts used in the slides

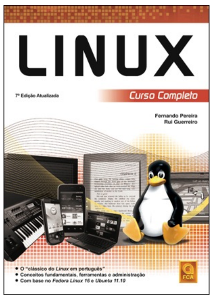
INTRODUCTION TO ASSIGNMENT 01 – “LINUX AND C PROGRAMMING”

INTRODUCTION TO ASSIGNMENT 02 – "INTRODUCTION TO LINUX"

References

- Linux online manual
- LUbuntu manual
  - https://manual.lubuntu.me/
- Bash Reference Manual
  - https://www.gnu.org/software/bash/manual/bash.pdf

Where to learn more?



- Linux Curso Completo
  7ª Edição", 2012
  Fernando Pereira, Rui Guerreiro
  FCA

- AT&T Archives: The UNIX Operating System
  https://www.youtube.com/watch?v=tc4ROCJYbm0

Thank you! Questions?

I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who.
—Rudyard Kipling