Linux Basics II:
Text editing and programming

Adriano Angelone, Graziano Giuliani

Course Outline

- UNIX/Linux Basics
- **Intermediate shell commands**
- **Editing and compiling source code**
- Text file manipulation
- Basic shell scripting

Download slides and exercise files with the command

```
git clone https://github.com/AA24KK/LinuxBasics.git
```

or download a ZIP archive at

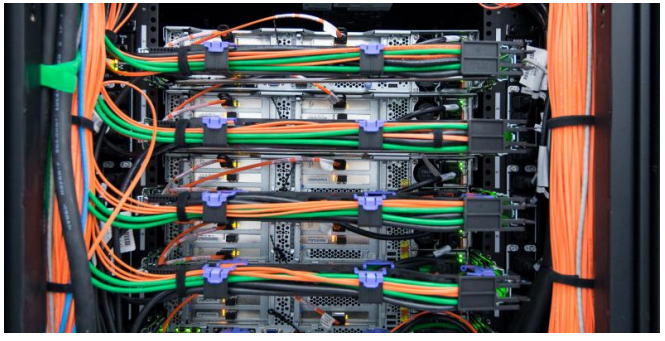
```
https://github.com/AA24KK/LinuxBasics/archive/master.zip
```

Adriano: **aangelon@ictp.it**, Room 263, ICTP
Graziano: **ggiulian@ictp.it**

Science with the Computer

You can do Science with a computer !

- Text Editors and WYSIWYG programs for writing
- Tools and libraries for data handling and visualization
- Data acquisition and storage
- Modelling and numerical algorithms



It can get to such complexities that a whole new Science has emerged:

Computer Science : study of computers and computational systems

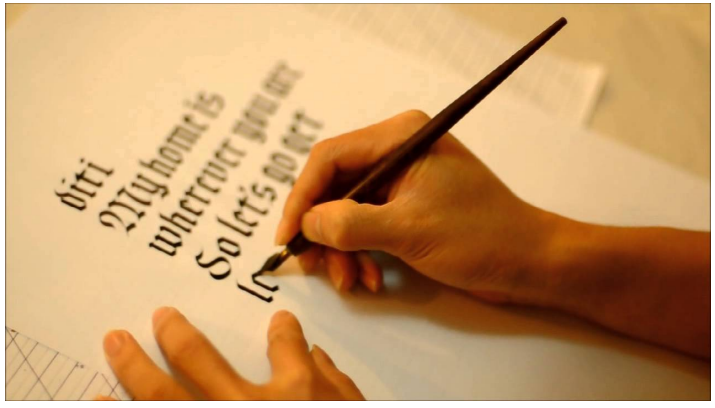
Text editing

Pick your choice !

- Local file: every desktop environment has a text editor
  - GNOME : gedit, geany
  - KDE (Plasma) : kwrite
  - Xfce : mousepad

  ...
- Text console: religious wars !
  - vi(m) : Unix pure and true !
  - emacs : I love GNU !
  - nano : I hate both of the above

  ...



Unix vi - 1

Starts by: `vim filetoedit.ext`

- Modes
  - **command mode**: Editor starts in command mode. Cursor movement, text deletion, pasting is possible. Can close/open files, save and quit editor.
  - **insertion mode**: Begins upon entering an insertion or change command.
- The [ESC] key returns the editor to command mode.
- Commands are executed by pressing the return key.
- To quit:
  - Saving the file: `:x`
  - Without saving the file: `:q!`

Unix vi - 2

- To enter insert mode:
  - `i` insert in the current position
  - `I` insert at line beginning
  - `a` append after character
  - `A` append at end of the line
  - `r` overwrite one character
  - `R` enter replace mode
  - `o` new line inserted below
  - `O` new line inserted above
- Move around in command mode:
  - `h,j,k,l` or arrows in insert mode: left-down-up-right
  - `w,e,b` next/previous word beginning or end
  - `(,)` next/previous sentence
  - `{,}` next/previous paragraph
  - `0,$` beginning/end of line
  - `gg,G` beginning/end of file

Unix vi - 3

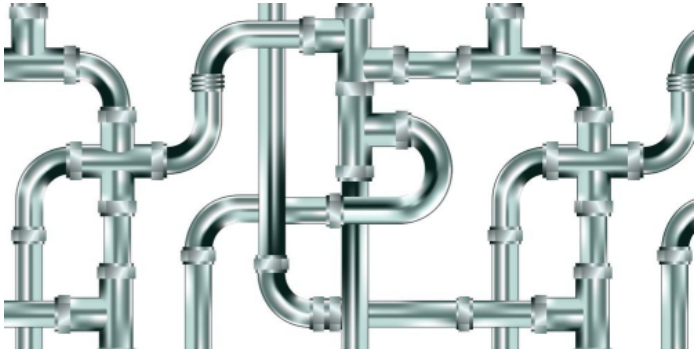
- change text
  - `C` change to the end of line
  - `Ncw` change N words
- Delete text
  - `xX` delete character to right/left
  - `D` delete to the end of the line
  - `dd,:d` delete the whole line
  - `Ndd` delete N lines
  - `Ndw` delete N words
- Copy text
  - `yy,:y` copy the line
  - `Nyy` copy N lines
- Paste text
  - `pP` paste the line(s) after/before current

Unix vi - 4

- Search
  - `/string` Search string ahead
  - `?string` Search string backward
  - `n,N` Next item ahead/backward
- Substitute
  - `:s/pattern/string/` substitute pattern with string
  - `:s!/path/subst!/new/path/!` for a file path
  - `:s/pattern/string/g` all occurences in line
  - `:M,N s/pattern/string/g` all in lines M to N
  - `:1,$ s/pattern/string/g` all occurences in file
- Undo, redo, join, capitalize
  - `.` Repeat last change
  - `u,U` undo the last/all changes in line
  - `Ctrl+r` Redo the change
  - `J` Join with next line
  - `~` change case

Writing your own program - 1

Use the building blocs of existing programs and create a complex **pipeline** of stages to reach the desired processing



- **Pros** : No programming in the general sense involved, just carefully examination of the input and output of existing system programs to create the required processing. The REAL UNIX way of using a computer.
- **Cons** : Limited by the possible processing allowed by system programs, generally related to text file manipulation, non portable across different systems

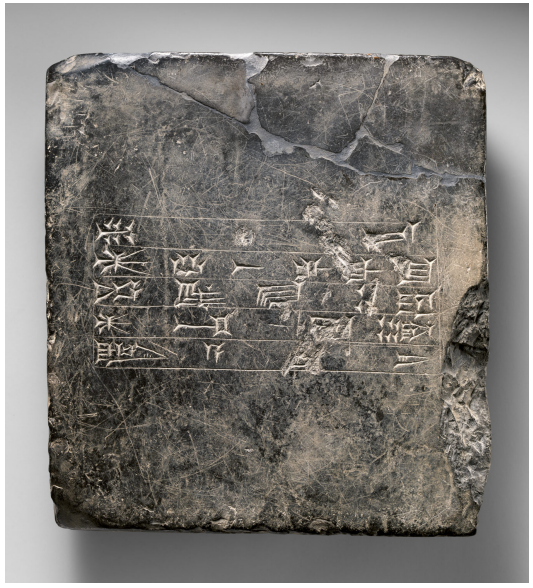
Example:

```
ls -al | grep $USER | tr -s ' ' | cut -d " " -f 5 > sizes.txt
```

Writing your own program - 2

Use generic **scripting language** interpreters which can more flexibly allow runtime evaluation of a processing

- **Pros** : More flexible, eventually the shell itself can be used, can use specialized libraries for compute intensive tasks, rapid prototyping
- **Cons** : Need to learn a programming language, not as fast as a system binary can be.



Example:

- Shell scripting
- Python language
- R statistical language

Writing your own program - 2

Use a low level **programming language** which is parsed by a program called *compiler* to create a system binary program

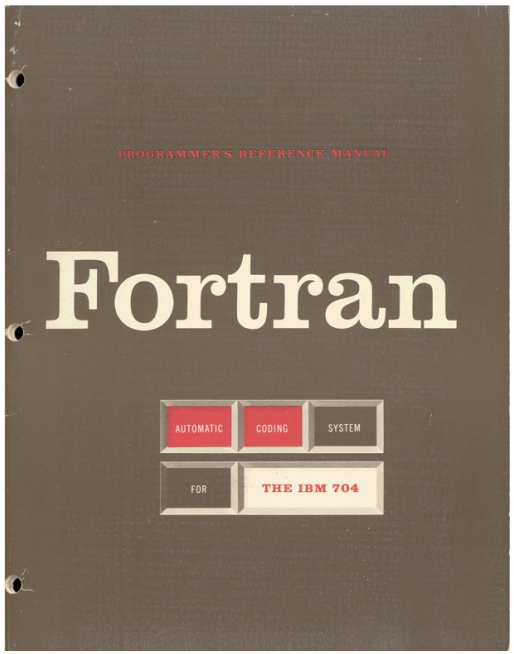
- **Pros** : Fast execution time, tailored processing to the problem to solve
- **Cons** : Need to learn a programming language, not as flexible as a scripting language, may require writing code even for very simple and common tasks best approached by generic system programs.

Example:

- Fortran Programming Language
- C/C++ Programming Language

Fortran program

Fortran source files are text files



- The User writes the source file:

```
program myprog
 print *, 'Hello world'
end program myprog
```

- A compiler parses source files and create binary object files:

```
gfortran -o myprog myprog.f90
```

- Objects are linked with other objects or libraries to create executables:

```
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000010 0003 003e 0001 0000 06f0 0000 0000 0000
0000020 0040 0000 0000 0000 1a78 0000 0000 0000
0000030 0000 0000 0040 0038 0009 0040 001d 001c
0000040 0006 0000 0004 0000 0040 0000 0000 0000
```

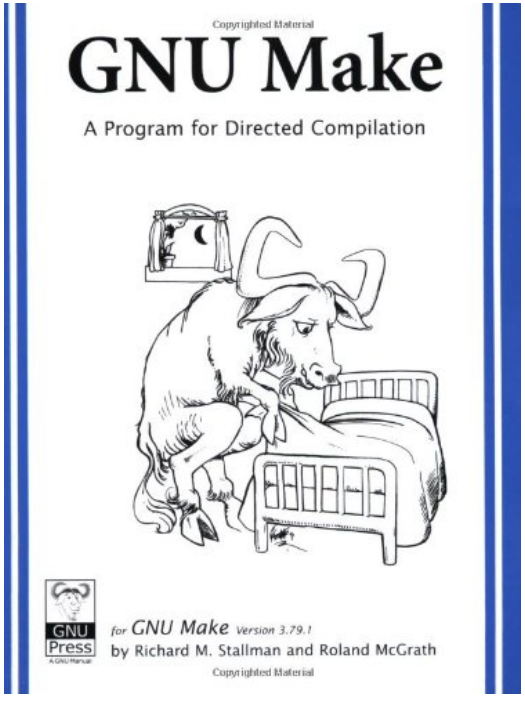
Compiler flags

The compiler is a program and accepts command line arguments

- `-g`
  - include debugging information
- `-Wall`
  - Enables commonly used warning options pertaining to usage recommend avoiding and that are easy to avoid
- `-pedantic`
  - check program for Fortran 95 standard conformance
- `-fbacktrace`
  - print the whole trace of the error
- `-fcheck=all`
  - perform all available run-time checks
- `-Ofast`
  - Optimize for fast execution time

Make program

The traditional way to manage a project code is the `make` program



- `[Mm]akefile`
  - For each directory in a project, you provide a Makefile.
  - The makefile contains:
    - **Targets** : things you can ask to be made
    - **Dependencies** : order of things to be made
    - **Variables** : useful to store options
    - **Conditionals** : select how to do on variable value
  - A hyerarchy of Makefiles can be built
  - For very complex projects Makefiles can be generated through other tools
  - Newer projects use different build helpers, but you can count on make be present on UNIX.

File permissions

There are three basic attributes for plain file permissions:

- read
- write
- execute

They mean what you would expect. There are three classes of users:

- owner
- group
- other

For each of the three classes you have three possible attributes to set.

For directories:

- read : you can list the content
- write : you can create/remove files inside
- execute : you can access it and its content

Check permissions

See permissions: `ls -l`

```
~/example_dir » ls
file_1  file_2  file_3  file_4
-------------------------------------------------
~/example_dir » ls -l
total 0
-rw-r--r-- 1 nemesis3 users 0 Sep 25 00:58 file_1
-r-xr--r-- 1 nemesis3 users 0 Sep 25 00:56 file_2
-rw-rw-rw- 1 nemesis3 users 0 Sep 25 00:56 file_3
-rwxr-xr-x 1 nemesis3 users 0 Sep 25 00:56 file_4
```

First 3 chars: **read** (`r`), **write** (`w`), **execute** (`x`) permissions **for user**
Second 3 chars: read/write/execute **for group**
Last 3 chars: read/write/execute **for all users**

Useful if the system does not let you remove a file

Changing permissions: `chmod`

Change permissions: `chmod`

`chmod <who><+/-><what> <file>`

- `<who>`:
  `u` (user),
  `g` (group),
  `a` (everybody)
- `<+/->`: `+` to add, `-` to remove
- `what`: `r,w,x` as above

```
~/example_dir » ls -l
total 0
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_1
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_2
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_3
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_4
-------------------------------------------------
~/example_dir » chmod u+x file_2
-------------------------------------------------
~/example_dir » chmod u-w file_2
-------------------------------------------------
~/example_dir » ls -l
total 0
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_1
-r-xr--r-- 1 nemesis3 users 0 Sep 25 01:03 file_2
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_3
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_4
-------------------------------------------------
~/example_dir » chmod a+w file_3
-------------------------------------------------
~/example_dir » ls -l
total 0
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_1
-r-xr--r-- 1 nemesis3 users 0 Sep 25 01:03 file_2
-rw-rw-rw- 1 nemesis3 users 0 Sep 25 01:03 file_3
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_4
-------------------------------------------------
~/example_dir » chmod a+x file_4
-------------------------------------------------
~/example_dir » ls -l
total 0
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_1
-r-xr--r-- 1 nemesis3 users 0 Sep 25 01:03 file_2
-rw-rw-rw- 1 nemesis3 users 0 Sep 25 01:03 file_3
-rwxr-xr-x 1 nemesis3 users 0 Sep 25 01:03 file_4
```

Executing scripts and programs

`./<executable>`

launches an executable

```
~ » cat example_script
#!/bin/bash

echo 'hello'
-----------------------------
~ » chmod u+x example_script
-----------------------------
~ » ./example_script
hello
```

`./<executable> &`

executes **in the background**

(the shell is free during execution)

```
~ » cat example_script
#!/bin/bash

sleep 5
echo 'hello'
-----------------------------------------
~ » chmod u+x example_script
-----------------------------------------
~ » ./example_script&
[1] 32587
-----------------------------------------
~ » echo 'test1'
test1
-----------------------------------------
~ » hello

[1]  + 32587 done       ./example_script
```

Checking and killing processes

`ps`
shows running processes

Processes are identified by a code (**PID**)

`kill <PID>` or `pkill <name>`
stop processes

```
~ » ps
    PID TTY          TIME CMD
    532 tty1     00:00:00 startx
    554 tty1     00:00:00 xinit
    555 tty1     00:04:58 Xorg
    559 tty1     00:00:00 xf86-video-inte
    562 tty1     00:00:07 i3
    570 tty1     00:00:05 nm-applet
    571 tty1     00:01:11 pcloud
    572 tty1     00:00:08 cbatticon
    621 tty1     00:00:00 pcloud
    666 tty1     00:00:01 pcloud
    710 tty1     00:00:17 pcloud
    732 tty1     00:00:00 pcloud
  31509 pts/0    00:00:00 nvim
  31789 pts/1    00:00:05 nvim
  32181 pts/1    00:00:04 okular
  32895 pts/2    00:00:00 ps
```

`top` **gives more info**
(e.g., CPU and RAM usage)

Press `P` to sort by CPU usage,
`M` to sort by RAM usage

```
top - 16:49:23 up  5:33,  1 user,  load average: 0.85, 0.64, 0.98
Tasks: 209 total,   2 running, 207 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.3 us,  0.4 sy,  0.0 ni, 98.1 id,  0.0 wa,  0.1 hi,  0.0 si,  0.0 st
MiB Mem :  15897.2 total,   9020.1 free,   2349.5 used,   4527.6 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  12964.5 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   3916 nemesis3  20   0 2088160 488888 227168 S   3.3   3.0  29:29.73 spotify
  33434 nemesis3  20   0  777856  46020  27336 S   3.3   0.3   0:00.62 pcmanfm
    555 nemesis3  20   0  495444 138288 122028 R   3.0   0.8   5:02.45 Xorg
    996 nemesis3  20   0 5498816 564460 117004 S   2.0   3.5  20:11.18 skypeforlinux
   2240 nemesis3  20   0 2746416 263252 138472 S   1.3   1.6   5:49.31 Web Content
   1629 nemesis3  20   0 3748828 437252 199028 S   0.7   2.7   7:42.99 firefox
   3877 nemesis3  20   0 4584432 232776 121944 S   0.7   1.4  10:30.81 spotify
      1 root      20   0   34164  10392   7752 S   0.3   0.1   0:17.34 systemd
    571 nemesis3  20   0 1412920 149700  86488 S   0.3   0.9   1:13.38 pcloud
    572 nemesis3  20   0  302196  36572  26668 S   0.3   0.2   0:08.60 cbatticon
    710 nemesis3  20   0  913584  86016  61924 S   0.3   0.5   0:18.02 pcloud
  31675 nemesis3  20   0  737920  39084  29872 S   0.3   0.2   0:09.42 sakura
  33395 nemesis3  20   0  738488  39764  30260 S   0.3   0.2   0:00.30 sakura
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.01 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-kblockd
```

Accessing remote computers

`ssh`: **access a remote computer**
(e.g., to use a CPU cluster)

```
ssh <user>@<remote machine>
```

`scp` allows file transfer

Clusters usually handle long calculations with a **workload manager**:
`slurm` is one of the most popular

You will have finite disk space and CPU time:
remember your limits

Don't take too many CPUs:
be mindful of others

Exercise 1

- Change directory into code.

```
$ > cd code
```

```
$ > ls
```

```
examplestart.f90 goodstart.f90 Makefile
```

- Use vim to examine the Makefile

```
$ > vim Makefile
```

- Type make
- Execute the examplestart program

```
$ > ./examplestart
```

- What does it mean?

Exercise 2

- Edit the Makefile, comment the FLAGS line, uncomment following

```
# FCFLAGS = -O2
FCFLAGS = -Wall -pedantic
```

- Make the program again
- Edit examplestart.f90 and modify it to fix warnings
- Execute the examplestart program

```
$ > ./examplestart
```

- What does it mean?

Exercise 3

- Edit the Makefile, comment the FLAGS line, uncomment following
  `# FCFLAGS = -O2`
  `# FCFLAGS = -Wall -pedantic` a
  `FCFLAGS = -Wall -pedantic -fcheck=all -fbacktrace -g -O0`
- Make the program again
- Execute the examplestart program
  `$ > ./examplestart`
- Edit examplestart.f90 and modify it to fix errors
- Execute the examplestart program
  `$ > ./examplestart`
- Compare with proposed *best* program:
  `$ > diff -Naurb examplestart.f90 goodstart.f90`
- Diffs?