Introduction to Unix Shell

Michel Jouvin

michel.jouvin@ijclab.in2p3.fr

I2I computing courses 2024

Why a command shell?

- Users need to interact with computers
  - Launch an application with its input parameters/options
  - Look at the results/messages from an application
  - Manage files
- Basic/simplest way: a graphical user interface with a keyboard + a mouse
  - A double click to start an application
  - An application allow to navigate through a hierarchy of folders and files
  - A keyboard to enter application parameters or to type shortcuts to menu items
- Not always convenient: in particular when you want to repeat the same commands or set of commands
  - Clicks can become cumbersome
  - Repeating “manual” actions is error prone
  - Difficult when not impossible to interact with remote machines like HPC computers or virtual machines in a cloud

What is a command Shell?

- A non-graphical application where you can enter a command and its parameters
  - A command is an application launched by the shell: can be very simple or very complex
  - Parameters/options are specific to each command
  - General syntax for specifying them and accessing files is common to each command
  - Commands can be combined to chain the output of one command as the input of the next one
  - You can navigate the command history to recall some of them
- Shell scripts: special kind of application executing a set of commands with programming capabilities (variables, tests, loops…)
  - The real power of a command shell
  - Allow to write repetitive actions in a file that will be executed as one new command
  - Can use variables to control the execution, i.e. loop over a set of files
  - Can receive parameters/options from the command line
  - Can call another script…

Many different shells

- Every operating system offers one or several command shells
  - 2 different flavours of operating system nowadays: Windows and Unix..
  - But several variants per operating systems with different features and syntax!
- Windows: legacy CMD, PowerShell
- Unix: 2 families with several variants per family!
  - sh (the first Unix shell), ksh, zsh, **bash**
  - csh, tcsh
- This course will concentrate on bash
  - Available on all Unix distributions, many advanced features for scripting
  - Also available on Windows through Git for Windows (https://gitforwindows.org)
- If you have a Windows machine, you should have installed Git for Windows before the course…

Shell window: main components

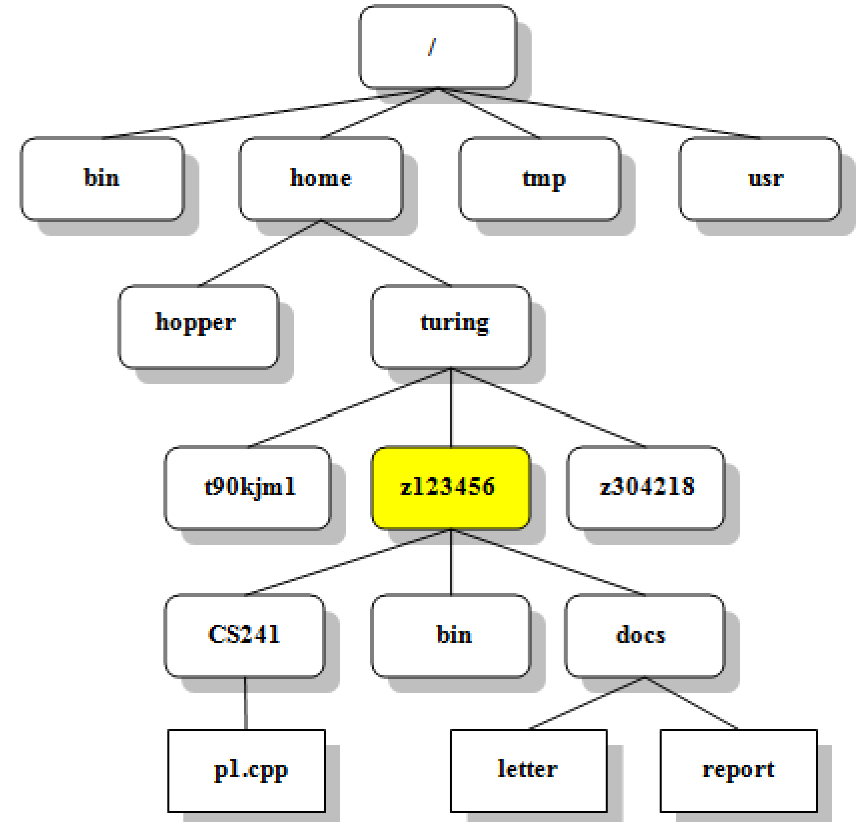
- Only one window per shell instance
- One line per command: typing <return> after a command triggers its execution
- Each line starts by a “prompt”: the command is entered after the prompt
  - The prompt can be configured to reflect your name, the directory you are in or other parts of your environment
  - By convention, in documentation, we refer to the prompt with the “$ “ string: it must never be entered (it is displayed by the shell)
  - The text you type with the keyboard will be entered at the insertion point shown by a “blinking cursor” (typically a blinking rectangle)
- If the shell is running on a machine with a mouse/trackpad, it is possible to select text previously entered and paste it at the insertion point
  - In Unix shell, selecting text implies copying it and is done by clicking the leftmost button; paste is done by clicking the rightmost button (right click, CTRL-click on MacOS)

Available commands

- No fixed/predefined list: every executable found in a *path*
  - Executable (Unix): every file with the `x` permission
- Path: a list of directories assigned to variable `PATH`
  - Syntax: *export PATH=/dir/1:/dir/2:/dir/3* (no space allowed)
  - To add something to the path: *export PATH=${PATH}:/new/dir*
  - To see the current path: *echo $PATH*
- To find the list of available commands
  - *ls [-l] directory*: list files present in a directory with their permissions if `-l` option is used
  - Autocompletion: type the first letters of the command followed by TAB key (if no letter entered, give the list of all commands, can be very long depending on the path!)

Files and directories

- File system: a set of directories and files, organized in a hierarchical way
  - Generally represented as an upside down tree
  - On Unix/MacOS/Windows Git, the root directory is named /
  - On Unix/MacOS, file and directory names are **case sensitive**
- Every user has a home (= personal) directory named ~
  - Directory you are in after logging in
- Navigating the file system tree: *cd*
  - *cd /home/turing/t90kmjl*: absolute path
  - *cd docs/report*: path relative to current directory (z123456)
  - . = current directory, .. = parent directory: *cd ../z304218*
- What is the current directory: *pwd*



Some commands related to files and directories…

- ***** (called *wildcard*) in a file/directory name replace any character
  - *m*e* will match *myfile*, *mine*, *male*…
- *ls*: list files in a directory
  - *ls*: list files in current directory
  - *ls /dir/ec/tory*: list files in /dir/ec/tory
  - *ls –l*: list additional information on files (permissions, creation date, owner, size…)
  - *ls –tr*: list file sorted by date (-*t*, rather than by names), in reverse order (-*r*, oldest first)
  - *ls –d*: display directory names rather than directory contents (the default)
- *mv* and *cp*: move (rename) or copy (duplicate) a file
  - *mv myfile ../other/dir:* move *myfile* to *../other/dir*
  - *cp myfile ../other/dir/newfile:* duplicate *myfile* into *../other/dir* with the copy named *newfile*
  - *cp –r mydir ../other/dir:* duplicate directory *mydir* and its contents into *../other/dir*
- cat/less *myfile: display the content of a text file, with* paging *(less)*
  - *more* is the ancestor of *less*

… Some commands related to files and directories

- *mkdir directory*: create a new directory
  - *directory* can be an absolute or relative path
  - *directory* cannot contain a wildcard
- *rm*: remove a file or directory
  - **Deleting is forever**… No trash, no way to recover a delete (except a backup)
  - *rm myfile*: delete *myfile*
  - *rm –r mydir:* delete directory *mydir* and its contents. **Be cautious!**
- *touch myfile*: create *myfile* as an empty file if it doesn't exist
  - Also change the modificated date to the current time (first purpose in fact!)
- Editing a file: several editors, matter of personal preference…
  - One easy to use: *nano*, commands prefixed by CTRL key with the available commands at the bottom of the window

Some other useful Unix commands

- *cat myfile*:
- *echo “text”*: display a text
- *printenv*: list all defined environment variables
- *grep string myfile*: look for a string in file *myfile*
- *man command*: display the manual (documentation) for a command
- *history*: the list of the previously executed commands
  - Command history can also be navigated with up and down arrow keys
  - A previous command can be navigated and edited with the left and right arrow keys
  - *!number:p*: recall the command *number* in the history without executing it (*:p*)
  - *CTRL/R pattern*: search the history for *pattern*, multiple CTRL/R to find an older occurrence of *pattern*
- Most commands accept `-h` and/or `--help` to display their syntax and possible options
  - Command options often have a `**--**long-form` and a `**-**l` (short form)

Redirections and pipes

- It is often handy to redirect the output of a command to a file for later processing
  - Called *redirection*
  - Done by adding **>** *file_name* after the command: **replaces** *file_name* (if it exists) by the output of the command
  - To append the output to an existing file: ***>>*** *file_name*
  - 2 output channels: output and error. To redirect error only: *2>*. To redirect both: *>&*.
- Pipe: using the output of command as input of another one, without an intermediate file
  - A key and powerful feature of Unix shells: allow chaining/combining commands
  - Done by separating commands by | character:

    ls | less: paging a long file listing

    *wc –l * | sort –n |head –n 5*: 5 first entries of the file list sorted by number of lines
  - Command input file are generally omitted when using a pipe as input: *sort* and *head* previously
- Pipes and redirection can be combined
  - *wc –l * | sort –n |head –n 5 > /tmp/5_smallest_files*

Environment variables

- Useful when writing scripts but also to configure the SHELL environment
  - Example: PATH
- 2 types of variables: shell variables and environment variables
  - Shell variables: internal to shell, not seen by applications. By convention, lowercase.
  - Environment variables: passed to applications/scripts. By convention, uppercase.
  - When in doubt, define an environment variable: shell variables mainly used in scripts and loops
- Value assignment different for each type
  - Shell variables: *var=value* (no space allowed, between “” if the value contains spaces)
  - Environment variable ***export*** *VAR=value*
- Displaying/using the variable identical for both types
  - To use the value: ***$var*** or ***${var}*** (variable name case sensitive): variable interpolated in ***“****string****”*** *(double quotes)* but not in **‘**string**’** (single quotes)
  - To display the value: *echo $var*

Shell scripts

- A shell script is a text file containing a sequence of commands to execute
  - Commands are written the same way as if they were executed interactively
  - Advanced programming features for controlling the execution with variables, tests, loops…
  - The shell script becomes a new command: executed by entering its name as the command
  - Requires to add the execution permission to the script file: *chmod a+x script_file*
  - Everything after a # is interpreted as a comment (ignored by the shell)
- Assigning the result of a command to a variable: *myvar=$(command)*
  - *command* can be simple command or a pipe
- Every command exits with a status stored in variable *$?*
  - The status is a number, 0 in case of success: see command documentation for non zero status
- First line of shell script is called the *shebang* and tells the shell to use to execute it
  - Shebang syntax: *#!/path/to/shell*, for example *#!/usr/bin/env bash*
- Parameters on the command line passed as variables *$1*, *$2*, *$3*… (*$** for all parameters)

Tests for conditional execution

- Bash has if/then/else if/else construct with syntax shown
  - Indentation is optional but recommended for readability
- Conditions are typically comparisons between to values…
  - Strings: = (equality) or != different
  - Numbers: *-eq*, *-ne*, *-gt*, *-lt*, *-ge*, *-le*
  - Example: *[“${myvar}” = “test”]*
- … but also operators to test file existence or type : -e, -f, -x…
- … and variable “existence”:
  - *-n “${var}”* (non empty)
  - *-z “${var}”* (empty)

if *[condition]*
then
        some commands
elif *[condition]*
then
        another set of commmands
else
        other commands
fi

Loops

- Loops: allow to repeat an action on a set of objects/values
    - Often used in scripts but can also be used interactively for executing actions on multiple objects as they can be written on one line
- Base syntax (*list_of_value* is a list of values separated by a space, can be a variable):

```
for var in list_of_value
do
    echo ${var}      # or other commands
done
```

- Can also be written as one line: *for var in list_of_value; do echo ${var}; done*
- Special commands than can be used in loops:
    - *break* can be used to exit prematurely the loop
    - *continue* to go immediately to next iteration (without executing the remaing commands)
- To execute a loop with indices: *for i in $(seq first last); do echo $i; done*

Script example

```
#!/usr/bin/env bash

# Illustration of quoting effect
echo Input parameters with globbing: $*
echo "Input parameters: $*"
echo 'Input parameters: $*'

if [ -z "$1" ]
then
   echo "File name required"
   exit 1
elif [ "$1" = "*" ]
then
   echo "Warning: too many files, not supported"
   exit 2
fi
```

```
files=$(wc -l $1 2> /dev/null| sort -n | tail -5)

echo "5 largest files:"
saved_ifs=$IFS
IFS=$'\n'
for file in ${files}
do
   echo ${file}
done
IFS=${save_ifs}
```

Launch script with a parameter like *'g*'* (or something matching files in your directory)

Searching files

- Often necessary to find all the files that contains a certain string or to find all the lines that contain a word: *grep* and *find*
- *grep* looks for all occurences of a pattern in a file(s) (or command output with a pipe)
  - Base syntax: *grep [options] pattern file_or_files*
  - *pattern* can be a simple string or use the *regex* syntax (use *egrep* rather than *grep* in this case)
  - Option –*v*: all lines except those matching the pattern
  - Option –*l*: list file names containing (or not containing if –*v*) the pattern rather than the lines
  - Option –*r*: if *file_or_files* is a directory, process all files in it
- *find* return all files that matches some criteria
  - Powerful but syntax potentially complicated: *man find* for details
  - Simple example: find */my/dir –name '*.txt'* will list all files with extension *.txt* in */my/dir*

Miscellaneous commands

- *alias newcmd="string"*: define a command *newcmd*
  - *alias ll="ls –l"*: define command *ll* as the ls command with its option *-l*
- *sort [-n] file*: sort a list of line alphabetically or by number if *–n*
  - *file* can be omitted if used in a pipe: *cat /etc/passwd | sort*
- *cut –d delim –f field_number file*: retrieved the *field_number* element on each line where an element is delimited by character *delim*
  - *cut –d: -f3 /etc/passwd*: retrieve the 3d element of each line (UID), with *:* as a field delimiter
- *wc –l file*: count the number of lines in the file *file*

Execution in the background

- A script or command can require a long time to run
  - Not convenient to remain connected to the machine running it, e.g. remote machine
  - Risk of loosing the whole work if the current shell is stopped
- A “long” command can be executed in the background by adding *&* at the end of the command line
  - Output, **if not redirected**, is going the current window and is lost if the windows is closed
  - If input is required the command is blocked until set again in the background
  - If a command was not put in the background, it is possible to move it to the background by clicking *CTRL/Z* and then entering the command *bg*
  - *ps* gives the list of commands in the background: *fg %n* puts the command back in the foreground
- Ensuring that a command is not stopped if the current shell is closed: prefix it with `nohup`, typically with redirection
  - *nohup my_very_long_script script_parameters > /tmp/myscript.out &*
  - Must not require any input from the keyboard

Useful links

- https://swcarpentry.github.io/shell-novice
  - This course largely based on it
  - Very progressive learning of Unix shell features
  - See https://carpentries-incubator.github.io/shell-extras/ for more advanced topics