Linux (Unix) shell basics

Grigory Shamov

UManitoba HPC workshop, Nov 1, 2021

Why command line shell?

- Command line is about “texts”. Text is powerful and expressive
    - Many people actually think in text
    - Computer code is also text
    - You can “build text upon text”, making very complex systems from smaller components
- Shell is lightweight and easy to learn
    - In a day or two you can be functioning in shell
    - Documenting text interfaces is easy (in another text) as compared to graphics, motions etc.; try to learn to dance by pictures or words
- It is easy to implement
    - Found everywhere, from IOT devices to Supercomputers and clouds
- It is easy to use shell remotely, both in space and time

Goals for this session

- Understand what is shell , and what is it good for
    - Hopefully, start appreciating the UNIX way of computing
- Get an idea of how to use BASH shell
    - (BASH stands for Bourne-Again SHell)
    - Focusing on the 3 key moments:
        - Navigating Filesystem
        - Work with Environment
        - Manage processes
- Get background for rest of this workshop, as we will use concepts like ***command***, ***script***, ***environment variable*** heavily
- Get an idea where to look for more info on Unix shells, should you need it.

What computer sciences should already know (OS)

- Computers do not do “services”. They compute.
- Computers are made of Hardware and Software
  - Hardware :
    - Processors (Compute)
    - Memory/Storage (Keep data and code)
    - Input/Output Devices (Keyboard, Monitor = Console), network cards, printers,
  - Software
    - Operating system kernel: take hardware and present it to applications as “resources” to use
    - OS also to manage applications as “processes”, manage “users” and their “data” (as files)
    - OS to provide some kind of User Interface: CLIs and GUIs

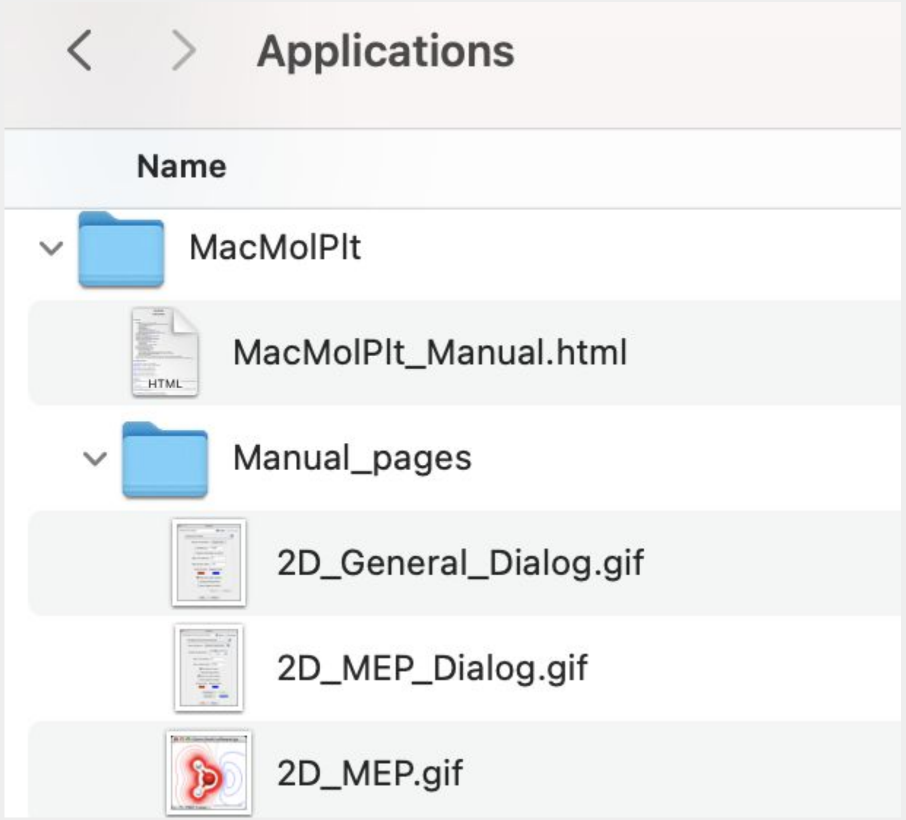
User interfaces (interface between User and the OS)

- Graphical (GUI). Windows, MacOS X
- Command line , shell. On UNIX derived OS (Linux, MacOS X) it is usually BASH or TCSH. There are other Unix shells. Windows has its own.

- What the Shell does? Lets users interact with OS to
  - Manage your data on filesystem, as files (create, copy/move, delete, find) etc.
  - Organize processes that do something with the data (start them, stop, orchestrate etc.
  - Query what OS does with the resources (free memory, disk, etc.)
  - Sysadmins would also change the OS settings (install software, tune etc.)
  - All of the above by simple texts in command line.

Working with files/folders

- Filesystems provide a hierarchical structure, tree-like.
- Files and folders have text names
- CS grows trees upside down. The topmost directory is called root, or "/"
- On UNIX systems, a "full path" to a file consists of all the directories from it to the root: /home/gshamov/dev/readme.txt
- Shell works in the context of "current", or "working" directory
- It provides command to "walk" and examine the filesystem tree



Filesystem commands : directories

```
 $>   pwd   # print working directory
/home/gshamov

$> cd dev1  # change directory one level down to dev1 no output if success
$> pwd
/home/gshamov/dev1

$> mkdir test1 # makes a new directory test1
$> mkdir -p level1/level2/Level3 # creates a tree, if level1 and or level2 do not
                                 # exist  they will be created with the  -p option
$> cd level1/level2
$> ls         # lists directory content. Note that names are case sensitive!
Level3
$> cd ..  ; pwd  # .. is one level up; you can have commands separated by ;
```

Filesystem commands : directories and files

$> **cd** ***/home/gshamov/dev1*** # an absolute path, starts from **"/"**
$> **cd ./test1** # a path relative to current dir which is "."
$> **cd ../commandline; pwd** # a path relative to one level up

$> **ls -la** # lists all files and directories in the current dir , with all attributes (large output skipped).

- Directories have d- preceding them (drwxr-xr-x)
- Files and directories have "user" and "group" ownership
- Files and directories have "mode" , or permissions.
    - Owner-group-other, **r**ead **w**rite e**x**ecute/search
    - Try **chmod** command to change them. chmod o-rx

$> **cp filesource.txt filedestination.txt # c**o**p**y source to destination
$> **mv filesource.txt filedestination.txt # m**o**v**e source; original deleted
$> **rm filedestination.txt # r**e**m**ove the file (caution!)

Filesystem commands : glob patterns

What if we want to affect all files, or some of the files, in bulk?
Glob patterns make it easy. Work in current working directory, special symbols will “expand” as follows:

- ‘*’ (star) expands as any number of characters.
- ‘?’ expands as any single character
- [abc123] expands as any one of the set 1,2,3,a,b,c
- {aa,bb,cc,dd} expands as aa, bb, cc, dd

```
$> cp *.txt  ./level2  # copy all .txt files to the directory level2

$> rm file*.txt                          # remove the files which names starts with “file”
                                         #  and ends with .txt (caution!)
$> mkdir level2/{one,two,three}   # creates level2/one level2/two level2/three
```

Intermezzo: BASH REPL

BASH is an interpreter. It takes commands from the console input (STDIN) and prints the results to console output (STDOUT) and errors to STDERR. It is called REPL (read-eval-print loop).

- **Save on typing:** the UP key gives previous command ; **history** prints all the history with line numbers, which can be repeated with **!number** ; Ctrl-R searches through history as you type; TAB tries to complete commands.
- Terminate running commands: usually Ctrl-C or in some cases ‘q’ would end a commands execution
- Freeze commands with Ctrl-Z; foreground/background them, etc.
- Some of the commands are separate programs; some are BASH builtins
- Commands usually have options of the form like -v or --version

File commands : printing and editing text files

$> **cat example.txt**   # prints content of the example.txt. (Why cat?)

$> **cat file1.txt file2.txt file3.txt**   # print the files in this order to STDOUT

$> **head -n 10 example.txt**   # prints first 10 lines of example.txt

$> **tail -n 10 example.txt**   # prints first 10 lines of example.txt

$> touch example1.txt   # if the example1.txt does not exist, it will be created

File commands : viewing and editing text files

$> **cat example.txt**   # prints content of the example.txt. (Why cat?)

$> **cat file1.txt file2.txt file3.txt**   # print the files in this order to STDOUT

$> **head -n 10 example.txt**   # prints first 10 lines of example.txt

$> **tail -n 10 example.txt**   # prints first 10 lines of example.txt

$> touch example1.txt   # if the example1.txt does not exist, it will be created

$> **grep Grigory example.txt** # finds all the lines with Grigory in the file

```
$> vi example.txt                # there are
$> nano example.txt              # several Linux
$> mc -ae example.txt            # text-mode editors to edit files
```

File commands : redirects and pipes

Output channels STDIN, STDOUT are special files and can be redirected to regular files. Often used to save result of a work.

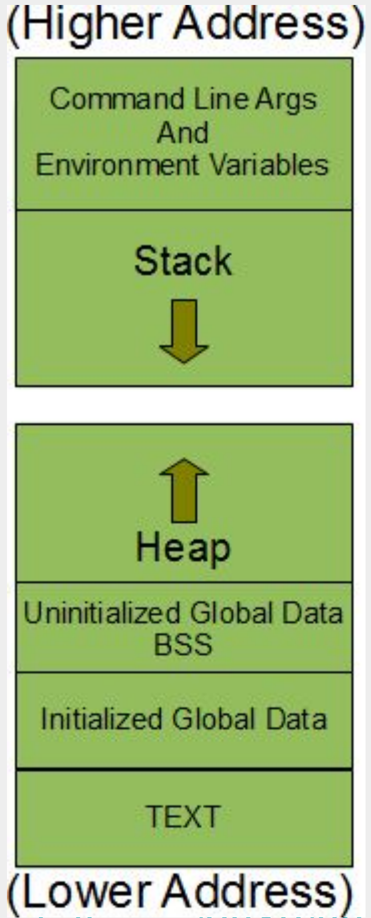
```
$> cat  file1.txt file2.txt file3.txt > all.txt  # concatenate files, save to all.txt

$> grep Grigory example.txt > foundG.txt  # finds all the lines with Grigory in
                                          #the file example.txt, and saves them to another file
$> cat < filein.txt > fileout.txt       # read from one file. Save to the other
```

Output channels STDIN, STDOUT can be piped with **“|”**

```
$> cat example.txt | less                            # scroll/view the file (q to quit)
$> cat  file1.txt file2.txt file3.txt | tee all.txt  # concatenate files, save to all.txt
                                                     #and print with the tee command
$> find /home/gshamov/dev1 | grep gaussian  # find all files but filter for ones
                                            # related to gaussian somehow
```

Processes and Environment

- A program that runs, and is allocated computing resources by the OS kernel, is a "process".
- This includes code text, data and stack spaces in memory.
- Each process has its "Environment"
  - Set of key=value pairs
- Shell uses Environment as both general programming and for communication between processes.
- Shell itself is a process, of course.
- OS/shell use special environments like PATH, USER, HOME, CPATH, PS1, etc.


(Higher Address)
Command Line Args And Environment Variables
Stack
Heap
Uninitialized Global Data BSS
Initialized Global Data
TEXT
(Lower Address)

(picture from https://www.thegeekstuff.com/2012/03/linux-processes-memory-layout/)

Environment variables in BASH shell

Variables have”name” and “value”. They usually are strings for shell (but can also be arrays or numbers depending on the context). BASH requires no spaces around the **‘=’** sign :

```
$> name=”value”   # creates a variable called  name with “value”
$> value=”value”   # creates a variable called value with value “value”
$> a=”1 2 3 4”       # creates a variable a with this string
```

To “expand” or get value , **$**name is used. Or more safe syntax of **${**name**}**. Echo command echoes a string. Depending on quotes, variables get expanded or not.

```
$> echo ‘$name Thats not my name’
$name Thats not my name
$> echo “$name, Thats not my name”
value, Thats not my name
```

Environment variables in BASH shell

Variables can be created, **export**ed to child processes and **unset**. The **env** command prints all of the variables of the current shell.
Why using variables?

- Indirection; some parts of the string can be not known ahead of time
- Make more generic scripts, better programming
- Communicate with processes and the user by conventional variables

Some of them:

PATH=/usr/bin:/usr/local/bin:$HOME/bin # a comma separated list where
# executables can be found by OS

USER , **HOME** # current username and home directory, cf. above

TMPDIR # place for temporary files; PWD # current directory

SLURM provides many variables related to nodes, cores, etc. to its jobs (try?)

Config files in BASH shell

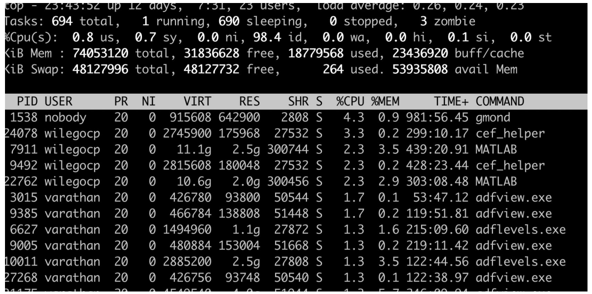
Special files under $HOME are used to store various per-user setting.

- **.bash_profile , .bashrc**   # used to set environment and run session startup
- **.icewm/** # stores per-user menus for iceWM window manager
- **.local/**   # directory for many user-installed software like Python and R libraries

The ls command does not show .files, unless asked explicitly with -a option.

Users can customize their PATH and other settings (Language, Prompt looks, etc.) in .bashrc or .bash_profile . (A wrong customization can break things: these files must have no errors).

Running processes/scripts



```
top - 23:43:52 up 12 days,  7:31, 23 users,  load average: 0.26, 0.24, 0.23
Tasks: 694 total,   1 running, 690 sleeping,   0 stopped,   3 zombie
%Cpu(s):  0.8 us,  0.7 sy,  0.0 ni, 98.4 id,  0.0 wa,  0.0 hi,  0.1 si,  0.0 st
KiB Mem : 74053120 total, 31836628 free, 18779568 used, 23436920 buff/cache
KiB Swap: 48127996 total, 48127732 free,      264 used. 53935808 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1538 nobody    20   0  915608 642900   2808 S   4.3  0.9 981:56.45 gmond
24078 wilegocp  20   0 2745900 175968  27532 S   3.3  0.2 299:10.17 cef_helper
 7911 wilegocp  20   0   11.1g   2.5g 300744 S   2.3  3.5 439:20.91 MATLAB
 9492 wilegocp  20   0 2815608 180048  27532 S   2.3  0.2 428:23.44 cef_helper
22762 wilegocp  20   0   10.6g   2.0g 300456 S   2.3  2.9 303:08.48 MATLAB
 3015 varathan  20   0  426780  93800  50544 S   1.7  0.1  53:47.12 adfview.exe
 9385 varathan  20   0  466784 138808  51448 S   1.7  0.2 119:51.81 adfview.exe
 6627 varathan  20   0 1494960   1.1g  27872 S   1.3  1.6 215:09.60 adflevels.exe
 9005 varathan  20   0  480884 153004  51668 S   1.3  0.2 219:11.42 adfview.exe
10011 varathan  20   0 2885200   2.5g  27808 S   1.3  3.5 122:44.56 adflevels.exe
27268 varathan  20   0  426756  93748  50540 S   1.3  0.1 122:38.97 adfview.exe
```

- Shell will try to run anything.
- Any unknown string is interpreted as a command / executable.
- But it needs to know how
  - Permissions on the executable should be **x**.
  - The file should be found (in PATH or explicitly called).
  - The beginning of the executable : #!/bin/bash
- When a new “child” process starts, it inherits the environment from the “parent”
- Commands like **‘ps’**, **‘top’** can be used to query running processes
- Multitasking/blocking: some processes might run in foreground, some in the background
  - (command like ‘jobs’, ‘fg’, ‘bg’, ‘wait’ to deal with background processes)

Running processes and scripts

PATH is a colon-separated list of paths where executables will be found.

```
$> which xeyes   # checks if “xeyes” is found in the PATH and prints where
/usr/bin/xeyes
```

```
$> export PATH=$PATH:/opt/bin  # adds a directory to the existing PATH
$> echo $PATH                  # prints the PATH value
```

```
$> echo ‘echo “Hello”’ > hello.sh
$> ./hello.sh                  # a error will result
$> chmod u+rwx ./hello.sh      # make it readable and executable (for scripts)
$> ./hello.sh; bash hello.sh   # the second would have worked w/o chmod
```

#!/bin/bash on top of the file tells kernel it is a BASH script, to be run by /bin/bash

Can be any other ingterpreter, like **#!/usr/bin/env python**

BASH as a programming language

A “Script” lets you repeat REPL commands in a sequence. So that you don’t have to type them every time.

Commands in a script also can communicate with help of environemnt variables, files and pipes, forming workflows without human intervention.

Special variables of interest:

- **$0, $1, $2, $3, ..$9, ${10},**.. are the command line parameters. $0 is name of the script. $> ./myname.sh is gshamov
- **$@** and **$*** : all the arguments or all elements of an array
- **$?** : exit code of a command (failed or not). Btw., one can check the commands exit status by **“&&”** and **“||”** logical operators.

Capturing output of a command into a file by **“>”**
Capturing output of a command into a variable with ` command ` or $(command) expansion. **export PATH=$PATH:`pwd`**

BASH as a programming language

Besides simple sequential commands execution, BASH provides flow control operators:

Conditional / branching

```
If condition
then
  … command ..
fi
```

Iterations / loops

```
for variable in list
do
  … commands on $variable
done
```

“Do nothing” , or comment is **#** ; can be used for human reader, or to pass info for non-shell programs, for example

- **#!/bin/bash** passes info on the shell interpreter to the kernel
- #SBATCH directives pass information to SLURM scheduler

More info / shell tutorials

- **man command ; command --help; command -?; info command**

  $> man man

- Wikipedia had commands described (but now it is corrupted by large influx Windows/Dos shell info less relevant to Unix systems)
- Clean and concise intro, CS focused:
  - https://matt.might.net/articles/basic-unix
  - https://matt.might.net/articles/settling-into-unix
  - https://matt.might.net/articles/bash-by-example
- https://LearnXinYminutes.com/docs/bash has a quick intro
- Carpentries format:: https://www.hpc-carpentry.org/hpc-shell