Life in a Shell: Getting the Most out of Linux/ Unix

Thomas Nichols, PhD
University of Warwick

25 September, 2014

Motivation

- Linux is crucial for Scientific Computing
  - Fast clusters use it (buster, minerva, etc)
- Need mastery of the command-line & scripts
  - A command-line environment is excellent for manipulating large numbers of files
  - But without basic skills, the command-line can slow you down, result in errors, and can drive you nuts
  - Scripts crucial for efficiency & reproducibility

Long-Term Outline

- Life on the (tcsh) command line
  - Wildcards, pipes, essential commands,
- Basic bash Scripting
  - Variables, if/then, for … do
- Intermediate bash Scripting
  - Case statements, advanced tests
- Advanced bash Scripting
  - Parsing arguments, functions

Life on the Command Line

- Shell basics
- Fundamental commands
- Wildcards
- Input/Output redirect
- Shell variables (local vs. environmental)
- Essential commands

Linux: How do I get there?

- Windoze
  - `putty` ssh client and terminal
  - `Cygwin` – Linux command line suite
    - Still need a terminal; use Dos command window, or mintty
    - Then ssh to linux host
- MacOS – You’re already there!
  - Terminal terminal program
    - Bash shell by default
    - ssh to other Linux hosts

Shell Basics

- The Shell
  - Just another program
    - Accepts your keystrokes
    - Sends corresponding letter on terminal
    - Runs programs on your behalf
  - But shells are *also* scripting language
    - Text file lists of commands
    - Complicated, looped, conditional programs

Shell Basics

- Different types of shells
  - sh "Bourne Shell"
    - Written by Steve Bourne at Bell Labs, c. 1974
    - Not so friendly on command line
    - On linux, now same as bash
  - bash "Bourne-Again Shell"
    - More friendly on command line
    - Regarded as best-practice scripting shell language
  - csh "c-shell"
    - Former standard shell
    - On linux, now same as tcsh
  - tcsh "Enhanced c-shell"
    - Enhanced c-shell, with tabular completion

Which Shell to Use?

- Interactive, on command line
  - bash
    - Most common; previously, tcsh was dominant
    - It's the default
    - Changing the default is hard
- For scripting
  - bash
    - functions
    - Extensively used in FSL, elsewhere
    - See "Csh Programming Considered Harmful"

File Paths

- Hierarchical directory tree

`/`	“Root” directory
`/tmp`	Temporary files
`/home`	User files
`/etc`	System configuration files

- Special directories

`.`	(period) references current directory
`..`	(period^2) references parent directory
`~`	Your home (& initial) directory

Filenames

- Essentially no limit on filename length (256)
  - Though best to keep it reasonable <20 char
- Extensions meaningless to unix itself
- But use them for humans' sake
  - Text files `.txt`
  - Data file `.dat` (generic)
  - Shell script `.sh` (bash/sh)
- Best to *not* use extensions in directory names

How Shell Parses Your Commands

- Each line entered broken into **white-space separated tokens**
  - White space = 1 or more space or tabs
  - E.g. `cd/to/my/directory` Only 1 token!
  - E.g. `cd /to/My Programs/Desktop` 3 tokens!
- First token is the command
- Remaining tokens are arguments to command
  - E.g. `cd /to/my/directory`
    - "cd" first token, the command
    - "/to/my/directory", argument for command “cd”
  - E.g. `cd "/to/My Programs/Desktop"`

Copy & Paste Danger!
Smart quotes don't work!
‘ ’ “ ”
Must use plain quotes ' "

How Shell Parses Your Commands

- Arguments vs. Options
  - Convention has it that optional arguments are preceded by a minus sign
  - E.g. `ls /tmp` (Show contents of /tmp dir)
  - E.g. `ls -l /tmp` (Show detailed contents)

Fundamental Commands

- `pwd` “Print working directory”
  - You are always *somewhere*, from which
- `cd` “Change directory”
  - E.g. `cd ..` (go up one directory)
  - E.g. `cd ~/tmp` (go to my personal temp dir)
  - E.g. `cd ../../duh` (go up 2 dirs, then duh)
  - E.g. `cd ~` (go to your home directory)
  - E.g. `cd` (same)

Fundamental Commands

- `ls` “List files”
  - E.g. `ls` (list files in current directory)
  - E.g. `ls .` (same)
  - Optional Arguments
    - `-l` (minus ell) Long listing, showing date, size
    - `-a` Include files beginning with . (dot)
    - `-t` Order by time of last modification (best w/ -l)
    - `-d` Do not list subdirectory contents
    - E.g. `ls /home/essicd`
      Shows contents of the directory
    - E.g. `ls -d /home/essicd`
      Shows info on the directory itself

Fundamental Commands

- `mkdir` <dirname>
  - Create a directory
- `rmdir` <dirname>
  - Remove a directory; must be empty
- `rm` <file>
  - Remove files
  - Optional Arguments
    - `-i` Interactive – ask if you're sure for each file
    - `-r` Recursive, delete directories and conents

Fundamental Commands

- `cp file1 file2`
  `cp file1 file2 file3 … directory`
  - Creates a copy of a file *(first form)*
  - Copies one or more files to a directory (second form)
  - Optional Arguments
    - `-i` Interactive, warn about over-writing
    - `-r` Recursive, copies directories and contents
    - `-p` Preserve file modification times (otherwise timestamp on new file is now)
- `mv file1 file2`
  `mv file1 file2 file3 directory`
  - Renames a files (i.e. "moves" it) *(first form)*
  - Moves one or more files to a directory *(second form)*
  - Optional Arguments
    - `-i` Interactive, warn about over-writing

Fundamental Commands

- `more` Show file, one screen at a time
- `head` Show first few lines of a file
- `tail` Show last few lines of a file
  - For both head & tail:
    - `-n #` Show # lines instead of default (10) num.
      e.g. `head -n 20 file.txt`
  - For just tail:
    - `-f` Show last 10 lines, then wait for file to grow, and show new lines as they appear
- `cat` "Concatenate" files
  - Useful for combining multiple files
  - E.g. `cat file1.txt file2.txt file3.txt`

Output Redirection

- Each program in unix has three modes of input/ output
  - Standard input
  - Standard output
  - Standard error (for error messages)
- Shell can redirect input or output
  - | Connect standard output to input (pipe) of another program
  - <   Standard input **from** file
  - >   Standard output **to** file
  - >& Standard output *and* standard error **to** file

Output Redirection

- Pipe
  - Can have several pipes
    - E.g. `ls -l | tail -100 | more`
- Redirects to files

`>`	Standard output, **overwrite** file
`>>`	Standard output **append** to file
`>&`	Standard output *and* standard error, **overwrite** file

`>> file.out 2>&1`

Standard output *and* standard error, **append** to file.out. (don't ask)

Output Redirection

- Useful Examples
  - Save directory listing
    - `ls -l > FileList.txt`
  - Look at long listing page at a time
    - `ls -l | more`
  - Look at only the most recently modified files
    - `ls -lt | head`
  - Concatenate a bunch of files into a new one
    - `cat file1.txt file2.txt > allfiles.txt`

Shell Variables

- **Assign** variables with equal sign =

```
NextSim=TestProg
```

- **Dereference** with dollar sign $

```
echo $NextSim
```

  … just shows "TestProg"

The simplest shell command: `echo`
Just 'echoes' the command line

- Protect dereferencing with brackets

```
echo $NextSim_1
```

  …no output, variable `NextSim_1` undefined

```
echo ${NextSim}_1
```

  … shows "TestProg_1"

Shell Variables: Local vs Global

- **Local variables** do not get passed on to child processes

```
NextSim=TestProg
```

bash

Start a new shell! Yes, you can do that any time.

```
echo $NextSim
```

… no output

- **Global variables** passed to 'child' processes
  - Mark global variable with "export"

```
export NextSim=TestProg
bash
echo $NextSim
```

… shows "TextProg"

  - By convention (only) global variables are capitalised

Important Shell Variables

- `USER`
  - Your user name
- `HOME`
  - Your home directory, same as ~
- `PS1`
  - Prompt string. Try…

```
PS1="Your wish is my command> "
```

Most Important Shell Variable

- `PATH`
  - Colon-separated list of directories

```
echo $PATH
```

  … might show something like

```
/usr/local/bin:/usr/bin:/bin
```

  - These are the directories searched when you type a command.
  - If you type "Ls", the shell will first look in /usr/local/bin for a program named "ls" and then run it; if it isn' there, it will look in "/usr/bin", and then "/bin", etc.
  - Finally, if it doesn't find it, you get "bash: Ls: command not found"

Setting Shell Variables Permanently

- Configuration Files

  `~/.profile`
  Run each time you **log in**

  `~/.bashrc`
  Run each time you start a new **interactive shell**
- Login-only?
  - E.g. when SGE runs programs on your behalf
- Interactive shell?
  - E.g. whenever you ssh, or start a new shell with "bash"
- Change your `PATH` in `.profile`
- Change command-line goodies in `.bashrc`
  - e.g. `PS1`, aliases

Editing Configuration Files SAFELY!

- Editing .profile and .bashrc is **dangerous**!
  - If you introduce an error to .profile, you might not be able to log in!!
  - Be careful! Always use two terminal windows!
- Terminal Window 1
  - Make a backup-copy
    - `cp .bashrc .bashrc_safe`
    - `cp .profile .profile_safe`
  - Open a text editor; make edit to .profile/.bashrc
- Terminal Window 2
  - After making edit, try running a new shell
    - `bash`
  - **ALSO**, log out, and try logging back in
    - `exit`
    - `ssh buster`
- If you *can't* login or get errors
  - Fix them *before* closing the editor and Terminal 1!!!
  - Worst case, restore safe version
    - `cp .bashrc_safe .bashrc`

    … and re-confirm that you can run bash and login!!!

Terminal Text Editors

emacs – Hard to learn, but incredibly powerful. Can be entirely driven by control-key combinations, making you incredibly fast.

vim - Emacs' arch enemy. Don't use.

Others???

Exercise

- Create a "bin" directory in your home; add to PATH in .profile

  ```
  mkdir ~/bin
  ```

  Now, in text editor, add this to `~/.profile`…

  ```
  export PATH="$HOME/bin:$PATH"
  ```

- Crucial details!!!
  - Must *add* to existing path
    - If you simply did

      ```
      export PATH=$HOME/bin
      ```

      … your shell would break; no ls! or any other command
  - Must use quotes, in case existing path has white space in it

bash aliases

- Best way to make shortcuts for frequently used commands
  - Instead of every day typing

    ```
    cd /storage/myid/very/long/path/to/my/project
    ```

  - You could type

    ```
    cdmyproj
    ```

  - Syntax

    ```
    alias cdmyproj="cd /storage/myid/very/long/path/to/my/project"
    ```

- Quiz!
  - Where should you add alises, .profile or .bashrc?

Essential Aliases

- IMHO, everyone should have these 3 aliases

```
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'
```

- Prevents you from accidently overwriting a file
- What if you *do* have lots of files to delete? Make a special "really delete" command

```
alias trash='rm –f'
```

Other Important Commands

- `man` Show “manual” pages
  - Gives (some) help on commands
- `sort`
  - Key options
    - `-r` Reverse the order of the sort
    - `-n` Try to sort numbers correctly (e.g. 2 < 10)
- `du` “Disk usage”
  - Key options
    - `-s` Silent, only report summary
- `df` Show file system usage

Very Basic Shell Scripting

- Making a script
  - Make sure you have a `~/bin` directory
  - Make sure `~/bin` directory is in your path
  - Create your script in `~/bin`

    ```
    emacs ~/bin/myscript.sh
    ```

    First line must be

    ```
    #!/bin/bash
    ```
  - Make it executable

    ```
    chmod +x emacs ~/bin/myscript.sh
    ```
- Magic!!!
  - Now anytime, anywhere that you type `myscript.sh` it will run!

.sh extension

There is no requirement to use .sh extension on shell scripts.

I like the convention, as it reminds me what is a script and what isn't. (e.g. vs. .R .m etc)

Special Variables in Scripts

- Command line "positional" arguments
  - `$0` Name of the program run
  - `$1` Frist argument, `$2` second argument, etc.
  - `$#` Number of arguments
  - `"$@"` All arguments
    - Later we'll see that the quotes important to deal with white space correctly

```
#!/bin/bash

echo "This is the program name: $0"
echo "There are $# args"
echo "This is the first arg: $1"
echo "All args: $@"
```

Looping

- For loops

  ```
  for <var> in <a list of stuff>  ;  do
        command1
        command2
  done
  ```

- Most typically over arguments…

```
#!/bin/bash

for f in "$@" ; do
  echo "This is an argument '$f'"
done
```

Integer Arithmetic

- Bash can natively handle integer variables and do simple arithmetic
- Double parenthesis mark "math mode"

  `((i=1+2))` … but if just assigning, no need for (( ))… `i=1`

  `((j=3))`

  `((k=i+j))`
- Special for loops available for math mode

```
#!/bin/bash

n=10
for ((i=n;i>0;i--)) ; do
  echo -n "$i "
done
echo "Lift off"
```

Bash Functions

- Essential for scripts and command line

```
functname() {
   Commands
}
```

- I have 2 shell functions I can't live without

```
lsh() {
      ls -lat "$@" | head
}
lsm() {
      ls -lat "$@" | less
}
```

  – What do these do?!
  – Are they in my .bashrc or .profile?

The Holy Trinity

- grep
  - Prints lines that match general regular expressions
- sed
  - Stream editor
- awk
  - A full programming language, brilliant for handling structured data files (e.g. tab or comma separated)

grep

- grep <pattern> <files>
  - Will print all lines in files that match the pattern
  - Key options
    - `-i` Ignores case
    - `-l` Only print file name when a match found
    - `-v` Print lines where match does *not* occur
    - `-n` Show line number where match occurs
    - `-r` Work recursively
- Ex: What aliases do I have?
  - `grep alias ~/.bashrc`

sed

- sed <command> <files>
- There is basically only kind of command you'll use, the "search" command
  - `sed 's/data/DATA/' file1.txt`
  - `sed 's/data/DATA/g' file1.txt` <- Use global option
  - `sed 's/data/DATA/g;s/img/hdr/' file1.txt` <- stack commands