Life in a Shell: Becoming a Power Linux User

Thomas Nichols, PhD
University of Warwick

http://warwick.ac.uk/tenichols/scripts/bash

10 January, 2016

Motivation

- Anything you do for research/paper/life, you’ll do more than once
- Hence, why not automate it

Plan

- Linux basics
- Basic shell scripts
- Some 'power' commands

Getting to Linux

- Mac OS, Linux
  - Congratulations! You’re already there!
- Windows
  - CASH - “The Cash Shell”
  - GOW – “Gnu on Windows”
  - putty - Connect to buster or other linux system

Shell Basics

- The Shell
  - Just another program
    - Accepts your keystrokes
    - Sends corresponding character to 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" *what we'll use!*
    - 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

File Paths

- Hierarchical directory tree

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

Forward slash /
not
Backslash \

- Special directories

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

Fundamental Commands

- `pwd` “Print working directory”
  - You are always *somewhere*
- `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)

Filenames

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

Command Parsing

- 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 the 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 ' "

Command Parsing: Escaping & Protecting

- How to deal with spaces?

1. Don’t use them in file or directory names!!

2. **Escape** them, with backslash (\\)

   E.g. `cd /to/My\ Programs/Desktop`

3. Protected them with quotes (' or ")

   E.g. `cd "/to/My Programs/Desktop"`

   E.g. `cd '/to/My Programs/Desktop'`

   - (more on single- vs double-quotes later)

Special Characters

These must be escaped or quoted to avoid their *special* meaning:

`!`	`#`	`$`	`&`
`'`	`"`	`(`	`)`
`{`	`}`	`*`	`+`
`-`	`.`	`|`	`\`
`;`	`&`	`~`	`?`
`<`	`=`	`>`	`@`
`[`	`]`	`^`	␣ *(space)*

There are more!

Command Parsing: Options

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

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

Shell Variables

- Behavior of the shell is modified by "shell variables"
- **Assign** variables with equal sign =

```
NextSim=Prog4
```

- **Dereference** with dollar sign $

```
echo $NextSim
```

  … just shows "`Prog4`"

- Protect dereferencing with **curly brackets**

```
echo $NextSim_1
```

  …no output, variable "NextSim_1" is undefined

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

  … shows "`Prog4_1`"

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

Vital Shell Variables

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

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

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

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't 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`"

Modifying your Shell: Setting 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

EXERCISE

- File editing practice
- Use `nano`, a simple text edit that works in a terminal (no graphics!)
  - `nano test.txt`
  - Write some text
  - Save with `^o` (specify name, press [return])
  - Exit with `^x`
- Other useful nano commands
  - ^K "cut line"
  - ^U "uncut line"

Convention for Describing Keyboard Shortcuts: "^X" means "Control+x"

Most keyboard shortcuts in Linux consist of holding the [control] key while pressing another key.

By convention this is denoted by a up-caret (^) and the character – in capitals (as it appears on the keyboard). ^X *does not mean*, [control]+[shift]+x

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 <AliasName>=<Command>

    E.g. `alias cdmyproj="cd /storage/myid/very/long/path/to/my/project"`
- Quiz!
  - Where should you add aliases, to `.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'
```

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.stats.warwick.ac.uk`
- 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`
    - `cp .profile_safe .profile` … and double check can run bash and login!!!

"Power User" Terminal Text Editors

emacs – Hard to learn, but incredibly powerful. Can be endlessly modified (using lisp-based configuration files)

vim - Emacs' arch enemy. Don't use. ☺

Simplest Shell Script

- Create a hello world script

```
nano ~/bin/hello.sh
```

- Enter

```
#!/bin/bash
echo "Hello world!"
```

- Now type hello.sh
  - Nothing happens! Need to tell shell it's a program!

```
chmod +x ~/bin/hello.sh
```

  - Now it'll work

More complex shell scripts

- Positional Parameters
  - Arguments to your script at the command line are accessible via special variables
  - E.g. if you ran

    ```
            hello.sh there my friend
    ```

    inside your script 3 variables would be defined,
    - $1 with value "there"
    - $2 with value "my" and
    - $3 with value "friend"

Now, create your own: Test1.sh

```
#!/bin/bash

echo "This is arg 1 $1"
echo This is arg 2 $2
```

Make it executable: chmod +x Test1.sh

Then try it:

```
> Test1.sh file1 file2 file3
```

Now, create your own: Test2.sh

```
#!/bin/bash

for f in $* ; do
  echo "Arg: $f"
done
```

Make it executable: chmod +x Test2.sh

Then try it:

```
> Test1.sh file1 "file 2" file3
```

Now, create your own: Test3.sh

```
#!/bin/bash

for f in "$@" ; do
  echo "Arg: $f"
done
```

Make it executable: chmod +x Test3.sh

Then try it:
> Test1.sh file1 "file 2" file3

Positional Parameters

- Arguments. Can be accessed an number of ways
- Number-named variable
  - $1 $2 $3
- Or with loop
  - $* or "$@"
  - "$@" always better, as is white-space aware

Loops

- for loops – loop through a list

```
for f in file1 file2 file3 ; do
   echo "$f"
done
```

  – Semicolon *must* be separated by spaces

Loops

- for loops, like in C

```
for ((i=0;i<10;i++)) ; do
    echo "$i"
done
```

  - Semicolon *must* be separated by spaces

- Magically, inside (( )), don't need "$"!!

```
Cnt=10
for ((i=0;i<Cnt;i++)) ; do
    echo "$i of Cnt"
done
```

Functions

- You can make 'mini shell scripts' within scripts, with function.
- Just define with "()" following a name and brackets

```
#!/bin/bash
Usage() {
    echo "Wrong usage, stupid"
    exit 1
}
```

Proto.sh

```
#!/bin/bash
#
# Script:
# Purpose:
# Author:
# Version: $Id: Proto.sh,v 1.2 2013/04/29 08:29:16 nichols Exp $
#


##########################################################################
#
# Environment set up
#
##########################################################################

shopt -s nullglob # No-match globbing expands to null
Tmp=/tmp/`basename $0`-${$}-
trap CleanUp INT

##########################################################################
#
# Functions
#
##########################################################################

Usage() {
cat <<EOF
Usage: `basename $0` arg1 [arg2]

How this works

_____________________________________________________________________
\$Id: Proto.sh,v 1.2 2013/04/29 08:29:16 nichols Exp $
EOF
exit
}

CleanUp () {
    /bin/rm -f /tmp/`basename $0`-${$}-*
    exit 0
}
```

```
##########################################################################
#
# Parse arguments
#
##########################################################################

while (( $# > 1 )) ; do
    case "$1" in
        "-help")
            Usage
            ;;
#           "-t")
#               shift
#               tval="$1"
#               shift
#               ;;
        -*)
            echo "ERROR: Unknown option '$1'"
            exit 1
            break
            ;;
        *)
            break
            ;;
    esac
done
Tmp=$TmpDir/f2r-${$}-

if (( $# < 1 )) ; then
    Usage
fi

##########################################################################
#
# Script Body
#
##########################################################################

for d in "$@" ; do
   echo "Argument: '$d'"
done

##########################################################################
#
# Exit & Clean up
#
##########################################################################

CleanUp
```

Key Take Homes For Scripting

- Document!!
  - What will *you* forget in 6 months?
  - Put enough so you can quickly remember what you did, not necessarily *any* user
- Always use quotes
  - `"$f"` instead of `$f` … avoids spaces headaches
- If in doubt, test test test with echos

EXERCISE

- Using the safe method for editing .bashrc, add the rm, mv and cp aliases.

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

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`

grep

- In Sandbox

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