

# Linux Terminal Tools

Ketan M. (km0@ornl.gov)

Oak Ridge National Laboratory

# Table of Contents

- Part 1: [Overview and Logistics](#)
- Part 2: [Basics](#)
- Part 3: [Streams, pipe and redirection](#)
- Part 4: [Classic Tools: find, grep, awk, sed](#)
- Part 5: [Session Management: tmux](#)
- Part 6: [ssh: config and tunneling](#)
- Part 7: [Secure Communication with GnuPG](#)
- Part 8: [Bash Tools](#)
- Part 9: [Program Development Tools](#)
- Part 10: [Miscellaneous Utilities](#)
- [Summary](#)
- Practice and Exercises (if time permits else Offline)

# Part 1: Overview and Logistics

*orientation and practical stuff*

[back to toc](#)

# Overview: What shall we learn

- Build powerful **command-lines**
  - We will use **Bash shell with default key-bindings**
  - We will **assume GNU/Linux** and call it Linux
- Tools that are available (or easily installable) on most installations
- Goal is to be efficient and effective rather than to be an "expert"
- Benefits: save time, efficient for system, long-term payback
- **We do not cover:** Sysadmin, Networking

# Slides and practice data for download

- Slides and two text files available for practice

<https://github.com/ketancmaheshwari/lisa19>

- states.txt

Tabular data with five columns

- prose.txt

Prose with sentences and paragraphs

- c\_example.tgz

Code and Makefile example files

# About You and Me

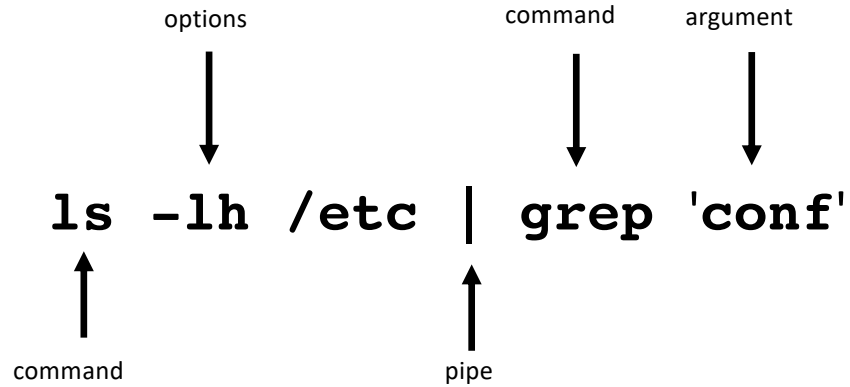
- Basic exposure to Linux is assumed but feel free to interrupt and ask questions
  - common commands, basic understanding of files and directories, editing.  
eg. `cd`, `ls`, `pwd`, `cat`
- About Me
  - Linux Engineer at Oak Ridge National Laboratory
  - Command line enthusiast

## part 2: Basics

*welcome to the school of command line wizardry!*

[back to toc](#)

# Anatomy of a Typical Command





# Know the System

- **id**: know yourself
- **w**: who is logged in (**-f** to find where they are logging in from)
- **lsblk**: list block storage devices
- **lscpu**: display info about the CPUs
- **lstopo**: display hardware topology (need **hwloc**, **hwloc-gui** packages)
- **free**: free and used memory (try **free -g**)
- **lsb\_release -a** : distribution info (sometimes not available)

PS0: Use **ctrl-c** to kill stuck commands or long running ones

PS1: Some commands may not be available: **which <cmdname>** to verify

# Know the Processes

- List the processes by name, pid etc: **ps** (commonly used flags: **aux**)
- **ps** implementations: POSIX, GNU and BSD!
  - implementations differ in behavior
  - determined by style of options: POSIX (-), GNU (--), BSD (no dash) before options
- Display processes: **top**, **htop**, **atop**
- Lower process priority by being **nice** and fly under the radar, eg.:
  - **nice -n 19 tar cvzf archive.tgz large\_dir**
- Kill a process: **kill <pid>**
  - to kill non-responsive processes
  - hung sessions

# Many ways to get help

- **man nano**

- Manual pages organized section-wise (see **man man** for more on sections)
- One page for each section (if exists) eg. **man 5 passwd** #5th section

- **wget --help**

- Handy for quick syntax reference

- **info curl**

- Modern

- Browse **/usr/share/doc**

- Usually a README file has info and examples
- Browse with a web-browser

# Working with Files

- **cat** for relatively short files  
`cat states.txt`
- **less** is more than **more** for long files  
`less /etc/ntp.conf`
- **tail -f** to watch a file growing live
- What can you do about binary files? (not much)
  - **strings** will print the printable strings of file
  - **od** will print file in octal format
  - **cmp** will compare them byte by byte
- Compare text files with
  - **comm** *sorted* files line by line
  - **diff** differences line by line -- used most frequently, rich options set, see man

# Internet on command line

- **curl** is commonly used to download from the web:

```
curl -O http://www.gutenberg.org/files/4300/4300-0.txt
```

```
curl ifconfig.me #quickly find my IP
```

- **wget** is similar:

```
wget http://www.gutenberg.org/files/4300/4300-0.txt
```

```
wget https://kubernetespodcast.com/episodes/KPfGep{001..062}.mp3
```

- **lynx** can be a useful text-based browser:

- avoid pesky ads on the web
- when internet is slow / only care about text eg. **lynx text.npr.org**
- read local html pages, eg. those found in **/usr/share/doc**
- **w3m** and **links** are other text-based browsers: **w3m lite.cnn.com**

# Be a command line ninja: Navigation

MAC users: terminal pref > profile > keyboard settings > Use option as meta key

**kubect**l** set subject rolebinding admin --user=ldf --group=nsed**

ctrl-a / ctrl-xx      alt-b      **cursor**      alt-f      ctrl-e

**ctrl-] <char>** moves cursor to 1<sup>st</sup> occurrence of <char> to right

**ctrl-alt-] <char>** moves cursor to 1<sup>st</sup> occurrence of <char> to left

# Be a command line ninja: Deletion

`kubectl get -o template pod/web-pod-13je7 --template={{.status.phase}}`

ctrl-u      ctrl-w      alt-d      ctrl-k

cursor

use ctrl-y to paste back the deleted

# Wildcards: characters that expand at runtime

- **\* expands to any number of characters:**

```
ls -lh /etc/*.conf #all items with .conf extension
```

- **? expands to one character:**

```
ls -ld ? ?? ??? #list items 1,2 or 3 chars long
```

- **Negation (!)**

```
ls -ld [!0-9]* #items that don't start with a number
```

- **Escaping and quoting**

- \ for escaping a wildcard
  - ' for quoting a wildcard
- } prevent expansion



# Quick and Useful Tricks

- **!!** repeats the last command
- **!\$** change command, keep last argument:
  - **cat states.txt** # file too long to fit screen
  - **less !\$** #reopen it with less
- **!\*** change command, keep all arguments:
  - **head states.txt | grep '^A1'** #should be tail
  - **tail !\*** #no need to type the rest of the command
- **alt-.** #paste last argument of previous command
- **alt-<n>-alt-.** #paste **nth** argument of previous command

# More Tricks

- **>x.txt** #create an empty file / "zero" a large file
- **lsof -P -i -n #appsusinginternet**  
tag & later search hard-to-remember command from history
- **ctrl-l** #clear terminal
- **cd -** #change to previous dir
- **cd** #change to homedir
- **ctrl-r** #recall from history
- **ctrl-d** #**logout** from terminal

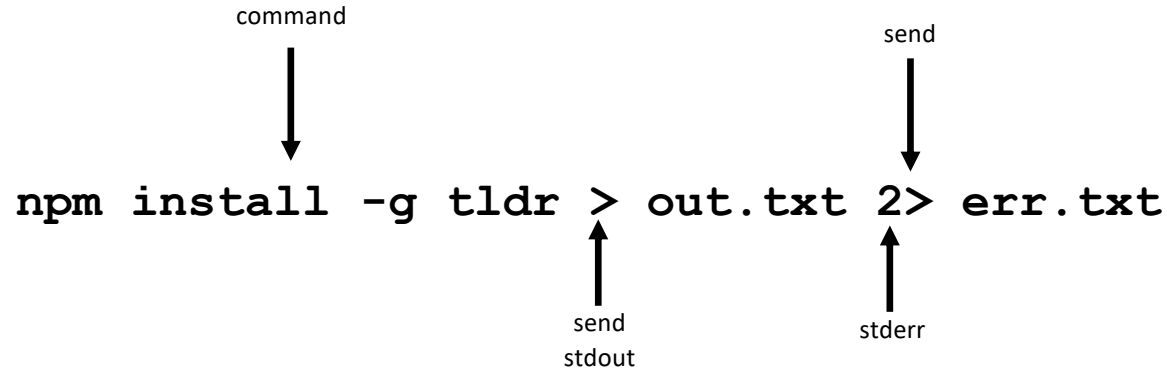
## Part 3: Streams, pipe and redirection

*I am sure a gardener designed them!*

# Terminal I/O Streams and Redirection

- Three I/O streams on terminal:  
standard input (**stdin**), standard output (**stdout**) and standard error (**stderr**)
- Represented by "**file descriptors**" (think of them as ids):  
0 for stdin, 1 for stdout, 2 for stderr
- Angle bracket notation used for redirect to/from commands/files:
  - > send stream to a file (eg. `ls > list.txt`)
  - < receive stream from a file (eg. `wc < list.txt`)
  - >> to append (eg. `cat list.txt >> masterlist.txt`)
  - << to in-place append (used in "heredoc")
  - <<< is used in "herestring" (not covering today)
- & is used to "**write into**" a stream, eg. **&1** to write into stdout

# Anatomy of a redirection using streams



# Redirection Examples

- Send stdout and stderr to same file:

```
pip3 install rtv > stdouterr.txt 2>&1  
ac -pd &> stdouterr.txt #short form (bash v4+)
```

- Disregard both stdout and stderr:

```
wget imgs.xkcd.com/comics/command_line_fu.png &> /dev/null  
#/dev/null is a "null" file to discard streams
```

- Read from stdin as output of a command

```
diff <(ls dirA) <(ls dirB)
```

- Append stdout to a log file:

```
sudo yum -y update >> yum_update.log
```

# The pipe: run second command using output of first

- A pipe is a Linux concept that automates redirecting the output of one command as input to a next command.
- Use of pipe leads to powerful combinations of independent commands. eg.:

```
find . | less #read long list of files page wise
```

```
head prose.txt | grep -i 'little'
```

```
echo $PATH | tr ':' '\n' #translate : to newline
```

```
history | tail #last 10 commands
```

```
free -m | grep Mem: | awk '{print $4}' #available memory
```

```
du -s * | sort -n | tail #10 biggest files/dirs in pwd
```

# Demystifying and debugging piped commands

```
free -m|grep Mem:|awk '{print $4}'
```

is equivalent to running the following 4 commands:

```
free -m > tmp1.txt
```

```
grep Mem: tmp1.txt > tmp2.txt
```

```
awk '{print $4}' tmp2.txt
```

```
rm tmp1.txt tmp2.txt
```

Reducing the piped stages is often efficient and easier to debug. For instance, the above pipeline may be reduced like so:

```
free -m|awk '/Mem:/{print $4}' #more on awk later
```



# pipe examples

#get pdf of a man page

```
man -t diff | ps2pdf - diffhelp.pdf
```

#get today's files

```
ls -al --time-style=+%D | grep `date +%D`
```

#top 10 most frequently used commands

```
history | awk '{a[$2]++}END{for(i in a){print  
a[i] " " i}}' | sort -rn | head
```

# Commands that only accept literal args

- Most commands receive input from stdin (so, pipe) **and** file, eg.

```
wc < states.txt #ok
```

```
wc states.txt #ok
```

- There are some exceptions though
- Some receive input **only from stdin** and not from file, eg.
  - `tr 'N' 'n' states.txt`  **#(strangely) NOT OK**
  - `tr 'N' 'n' < states.txt` #ok
- Some receive input **neither from stdin nor from file**, eg.
  - `echo < states.txt`  **#NOT OK** (assuming want to print file contents)
  - `echo states.txt`  **#NOT OK** (assuming want to print file contents)
  - `echo "Hello miss, howdy?"` #ok, takes literal args
  - `cp, touch, rm, chmod` are other examples

# **xargs**: When pipe is not enough!

- Some commands do not read from standard input, pipe or file; they need arguments
- Additionally, some systems limit on number of arguments on command line
  - for example: `rm tmpdir/*.log` will fail if there are too many `.log` files
- **xargs** fixes both problems
  - Converts **standard input** to commands into **literal args**
  - Partitions the args to a permitted number and runs the command over them repeatedly
- For instance, create files with names on the **somelist.txt** file:  
**xargs touch < somelist.txt**

# GNU Parallel

- Run tasks in parallel from command-line
- Similar to **xargs** in syntax
- Treats parameters as independent arguments to command and runs command on them in parallel
- Synchronized output -- as if commands were run sequentially
- Configurable number of parallel jobs
- Well suited to run simple commands or scripts on compute nodes to leverage multicore architectures
- May need to install as not available by default :  
[www.gnu.org/software/parallel](http://www.gnu.org/software/parallel)

# GNU Parallel Examples\*

- Find all html files and move them to a directory

```
find . -name '*.html' | parallel mv {} web/
```

- Delete pict0000.jpg to pict9999.jpg files (16 parallel jobs)

```
seq -w 0 9999 | parallel -j 16 rm pict{}.jpg
```

- Create thumbnails for all picture files (imagemagick software needed)

```
ls *.jpg | parallel convert -geometry 120 {} thumb_{} 
```

- Download from a list of urls and report failed downloads

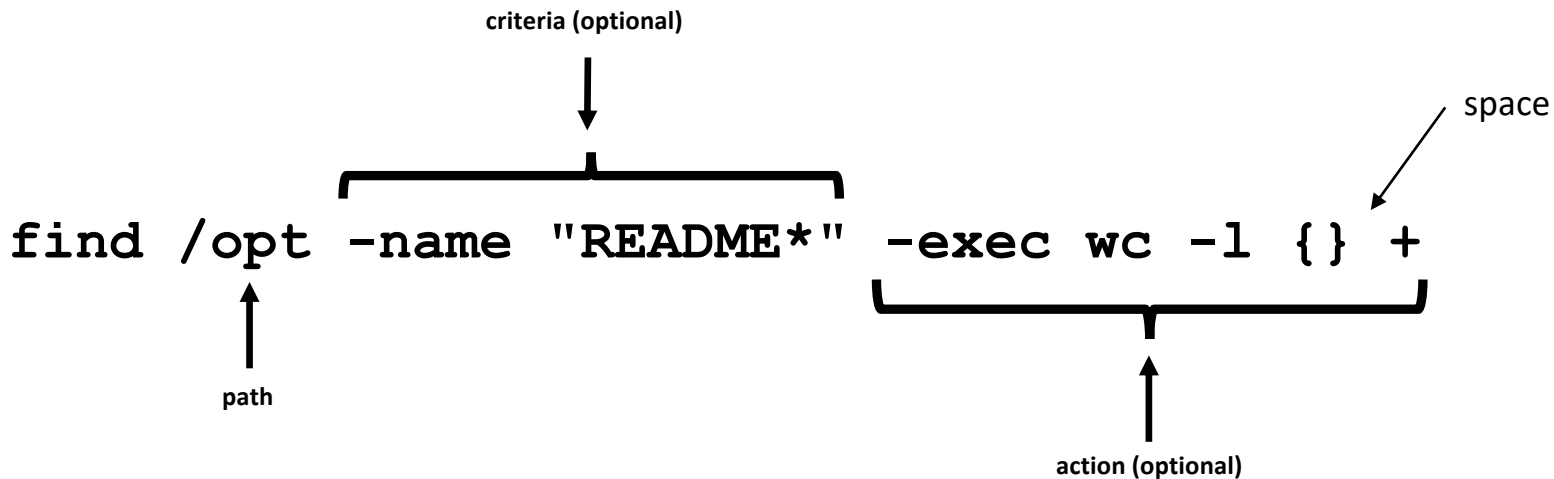
```
cat urlfile | parallel "wget {} 2>errors.txt"
```

# Part 4: Classic Tools: find, grep, awk, sed

*the evergreens*

[back to toc](#)

# find: search files based on criteria



# Features of find

- **path:** may have multiple paths, eg. `find /usr /opt -iname "*.so"`
- **criteria**
  - `-name`, `-iname`, `-type (f,d,l)`, `-inum <n>`
  - `-user <uname>`, `-group <gname>`, `-perm (ugo+/-rwx)`
  - `-size +x[c]`, `-empty`, `-newer <fname>`
  - `-atime +x`, `-amin +x`, `-mmin -x`, `-mtime -x`
  - criteria may be combined with logical **and** (`-a`) and **or** (`-o`)
- **action**
  - `-print`: default action, display
  - `-ls`: run `ls -lids` command on each resulting file
  - `-exec cmd`: execute command
  - `-ok cmd`: like exec except that command executed after user confirmation



# find Examples

- **find . -type f -iname "\*.txt"** #txt files in curdir
- **find . -maxdepth 1** #equivalent to ls
- **find ./somedir -type f -size +512M -print** #all files larger than 512M in ./somedir
- **find /usr/bin ! -type l** #not symlinks in /usr/bin
- **find \$HOME -type f -atime +365 -exec rm {} +**  
#delete all files that were not accessed in a year
- **find . \( -name "\*.c" -o -name "\*.h" \)** #all files that have either .c or .h extension

# **grep**: Search for patterns in text

- **grep** originally was a command "global regular expression print" or 'g/re/p' in the **ed** text editor
- It was so useful that a separate utility called **grep** was developed
- **grep** will fetch **lines** from a text that has a match for a specific pattern
- Useful to find lines with a specific pattern in a large body of text, eg.:
  - look for a process in a list of processes
  - spot check a large number of files for occurrence of a pattern
  - exclude some text from a large body of text

# Anatomy of grep

options

input file

grep **-i -n 'c.l'** states.txt

regular expression

# Useful grep Options

- **-i**: ignore case
- **-n**: display line numbers along with lines
- **-v**: print inverse ie. lines that do not match the regular expression
- **-c**: print a count of lines of matches
- **-A<n>**: include n lines after the match
- **-B<n>**: include n lines before the match
- **-o**: print only the matched expression (not the whole line)
- **-E**: allows "extended" regular expressions that includes (more later)

# Regular Expressions

- A regular expression (regex) is an **expression** that matches a **pattern**.

- Example pattern .....

```
^Linux is fun.$  
^So is music.$  
^Traffic not so much.$
```

- regex: 

b	a	r
---	---	---

 → no match
- regex: 

f	u	n
---	---	---

 → **one** match → "Linux is fun."
- regex: 

i	s
---	---

 → **two** matches → "Linux is fun." and "So is music."
- regex: 

^	s	o
---	---	---

 → one match → "So is music."
- regex: 

i	c	.	\$
---	---	---	----

 → one match → "So is music."

# Regular Expressions-contd.

- `.` is a Special character; will match **any** character (except newline)  
eg. `b.t` will match bat, bbt, b%t, and so on but **not** bt, xbt etc.
- Character class: one of the items in the `[]` will match, sequences allowed:
  - '`[Cc]at`' will match **C**at and **c**at
  - '`[f-h]ate`' will match fate, gate, hate
- `^` within a character class means negation  
eg. '`b[^eo]at`' will match brat but **not** boat or beat

# Extended Regular Expressions

- Enable by using **egrep** or **grep -E**
- '\*' matches zero or more, '+' matches one or more, '?' matches zero or one occurrence of the **previous character**  
eg. **[hc]<sup>+</sup>at** will match hat, cat, hhat, chat, cchhat, etc.
- '|' is a delimiter for multiple patterns, '(' and ')' let you group patterns  
eg. **([cC]at) | ([dD]og)** will match cat, Cat, dog and Dog
- {} may be used to specify a repetition range  
eg. **ba{2,4}t** will match baat, baaat and baaaat but **not** bat

# grep Examples

- Lines that end with two vowels:

```
grep '[aeiou][aeiou]$\n' prose.txt
```

- Check 5 lines before and after the line where term 'little' occurs:

```
grep -A5 -B5 'little' prose.txt
```

- Comment commands and search later from history

```
some -hard 'to' \remember --complex=command #success  
history | grep '#success'
```

- Confirm you got an ambiguous spelling right

```
grep -E '^ambig(uou|ou|ouo)s$\n' /usr/share/dict/linux.words
```

- find+grep is one very useful combination

```
find . -iname "*.py" -exec grep 'add[_-]item' {} +
```



# **awk**: Extract and Manipulate Data

- A **programmable** filter that reads and processes input **line by line**
- Rich built-in features:
  - explicit fields (\$1 ... \$NF) & records management
  - functions (math, string manipulation, etc.)
  - regular expressions parsing and filtering
- Features like variables, loops, conditionals, associative arrays, user-defined functions

**Highly recommended book:** The awk programming language by Aho, Kernighan and Weinberger, [ia802309.us.archive.org/25/items/pdfy-MgN0H1joloDVoIC7/The\\_AWK\\_Programming\\_Language.pdf](http://ia802309.us.archive.org/25/items/pdfy-MgN0H1joloDVoIC7/The_AWK_Programming_Language.pdf)

# Anatomy of an awk program

Often used as one-line **idiom** of the form:

```
awk 'awk_prog' file.txt
```

**OR**

```
command | awk 'awk_prog'
```

where **awk\_prog** is:

```
BEGIN {actions} #run one time before input data is read
```

```
/pattern/ or condition {actions} #run actions for each line of  
input files and/or stdin that satisfy /pattern or condition/
```

```
END {actions} #run one time after input processing section
```

At least one of the **BEGIN**, **/pattern/ or condition**, **{}**, **END** section needed

# /patterns/, conditions and actions

- A **pattern** is a regex that matches (or not) to an input line, eg.

```
/New/           # any line that contains 'New'  
/^[0-9]+ /      # beginning with numbers  
/(POST|PUT|DELETE)/ # has specific words
```

- A **condition** is a boolean expression that selects input lines, eg.

```
$3>1 # lines for which third field is greater than 1
```

- An **action** is a sequence of ops, eg.

```
{print $1, $NF}      #print first and last field/col  
{print log($2)}      #get log of second field/col  
{for (i=1;i<x;i++){sum += $3}} #get cumulative sum
```

- User defined functions may be defined in any action block

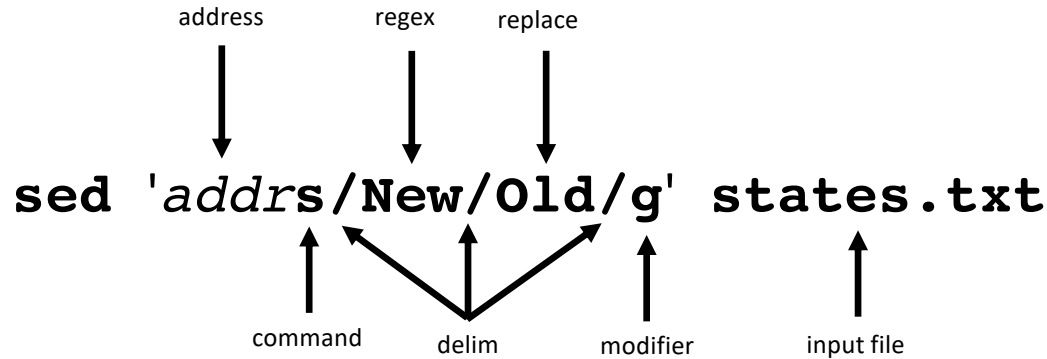
# Useful awk one-liners

- `awk '{print $1}' states.txt`
- `awk '/New/{print $1}' states.txt`
- `awk NF > 0 prose.txt # print lines that has at least one field (skip blank lines)`
- `awk '{print NF, $0}' states.txt #fields in each line and the line`
- `awk '{print length($0)}' states.txt #chars in each line`
- `awk 'BEGIN{print substr("New York",5)}' #York`

# **sed**: parse and transform text

- **sed** is a **stream editor**
- Looks for a pattern in text and applies changes (edits) to them
- A batch or non-interactive editor
- Reads from file or stdin (so, pipes are good) **one line at a time**
- The original input file is unchanged (sed is also a filter), results are sent to standard output
- Most frequently used idiom is for text substitution

# Anatomy of a typical sed command



# sed Options

- address: may be a line number, range, or a match; default: whole file
- command: **s**:substitute, **p**:print, **d**:delete, **a**:append, **i**:insert, **q**:quit
- regex: A regular expression
- delimiter: Does not have to be /, can be | or : or any other character
- modifier: may be a number **n** which means apply the command to n<sup>th</sup> occurrence, **g** means apply globally in the line
- Common **sed** flags: **-n** (no print), **-e** (multiple ops), **-f** (read sed from file), **-i** (in place edit **[careful]**)

# Useful sed Examples

- `sed -n '5,9p' states.txt` #print lines 5 through 9
- `sed '20,30s|New|Old|1' states.txt` #affects 1<sup>st</sup> occurrence in ln20-30
- `sed -n '$p' states.txt` #print last line
- `sed '1,3d' states.txt` #delete first 3 lines
- `sed '/^$/d' states.txt` #delete all blank lines
- `sed '/York/!s/New/Old/' states.txt` #substitute except York
- `kubectl -n kube-system get configmap/kube-dns -o yaml | sed 's/8.8.8.8/1.1.1.1/' | kubectl replace -f -`



# Part 5:

## Session Management: tmux

*for when the network goes down on my world-saving project*

# Workspace Management with tmux

- **tmux (v1.8)** is a terminal multiplexer that lets you create multiple, persistent terminals within one login
- In other words tmux is **a program which allows you to have persistent multiple "tabs" in a single terminal window.**
- Useful
  - when eg. a compilation or other operation will take a long time
  - for interactive multitasking
  - for exotic stuff such as pair programming
  - to preserve environment for multiple operations

# A Short tmux Tutorial

- Typical tmux workflow

```
tmux new -s s1 #start a new session  
# run any commands as normal  
ctrl-b :detach #detach the session, logout, go home  
#later, log in again  
tmux a -t s1 #get the same session back
```

- Other useful tmux commands

```
ctrl-b ( #switch to previous session  
ctrl-b ) #switch to next session  
tmux ls #list all sessions  
tmux kill-session -t s1 #kill a session
```

# Live collaboration with tmux

```
#user1#
```

```
tmux -S /tmp/collab  
chmod 777 /tmp/collab
```

```
#user2#
```

```
tmux -S /tmp/collab attach
```

# Create Panes and Synchronize with tmux

**tmux new -s s2** #start a tmux session

**ctrl-b "** #split horizontally

**ctrl-b %** #split vertically

**ctrl-b :setw synchronize-panes on**

#synchronized#

**ctrl-b :setw synchronize-panes off**

**ctrl-b o** #move through the panes

**ctrl-b x** #kill the active pane

# Part 6: ssh config and tunneling

*build secure tunnels*

[back to toc](#)

# ssh config (~/.ssh/config)

Host **login1**

hostname login1.ornl.gov

User km0

Host cades

Port 22

hostname or-slurm-login.ornl.gov

ProxyJump **login1**

User km0

**ServerAliveCountMax=3** #max num of alive messages sent without ack

**ServerAliveInterval=15** #send a null message every 15 sec

# now to ssh/scp to cades, just need "ssh/scp cades ..."

# Benefits of ssh config

- Makes ssh commands easier to remember in case of multiple hosts
- Customizes connection to individual hosts
- And much more, see **man 5 ssh\_config**
- For example: **ssh summit** is sufficient to connect to **summit.olcf.ornl.gov** with all the properties mentioned in the section:

```
Host summit
    Port 22
    hostname summit.olcf.ornl.gov
    User ketan2
    ServerAliveCountMax=3
    ServerAliveInterval=15
```



# Port forward over SSH Tunnel\*

**lclhost\$ ssh -L lport:host:hport remotehost -N**

The diagram illustrates the components of the SSH port forwarding command. Arrows point from descriptive labels to specific parts of the command:

- ssh**: ssh command (arrow pointing up)
- L**: local (arrow pointing down)
- lport**: localport (arrow pointing up)
- host**: "hostname" on remote host (arrow pointing down)
- hport**: port on remote host (arrow pointing up)
- remotehost**: remote host (arrow pointing down)
- N**: no command (arrow pointing up)

# SSH Tunneling Example

- Run an HTTP server on remote node and browse through local web browser:

```
step 1. remote$ python2 -m SimpleHTTPServer 25000  
OR
```

```
step1. remote$ python3 -m http.server 25000
```

```
step2. local$ ssh -L 8000:localhost:25000 id@remote -N
```

```
step3. Open browser on local and navigate to http://localhost:8000
```

# Incremental Remote Copy with **rsync**

- Synchronize data between local and remote storage
- Rich set of options (see man):
  - a and -v most commonly used
  - rsync -av localdir/ remotehost:~/remotedir**  
trailing **/** imp in localdir, else, the dir will be synced not contents
- A useful rsync hack: **fast deletion of a large directory**  
**mkdir empty && rsync -a --delete empty/ large\_dir/**

# part 7: Secure Communication with GnuPG

*Share top secrets securely over web*

[back to toc](#)

# GNU Privacy Guard Basics

- A tool for secure communication
- We cover
  - keypair creation
  - key exchange and verification
  - encrypting and decrypting documents
  - authenticating documents with digital signatures
- We do not cover
  - public-key cryptography concepts
  - sophisticated and advanced use-cases

# Create a new keypair

**gpg --gen-key** #answer the prompted questions

- Provide name and email as ID, choose hard-to-guess passphrase
- Keypair artefacts in **\$HOME/.gnupg** dir

- Create a **revocation certificate**

**gpg --output revoke.asc --gen-revoke <ID>**

- use the email as ID
- Useful to notify others the keypair may no longer be used -- eg. if you forgot your passphrase, lost keypair etc.

# Key Exchange and Verification

- Export a public key

```
gpg --output pub.gpg --export <ID>          #binary  
gpg --armor --export <ID> > pubtxt.gpg #ascii
```

- Import a public key

```
gpg --import billpub.gpg #import Bill's pubkey
```

- Verify and sign an imported key

```
gpg --edit-key b@ms.us #out key info & prompt  
...
```

```
command> fpr #fingerprint, verify over phone
```

```
command> sign #verify at prompt and done!
```

# Encrypting and Decrypting Documents

- Encrypt a document for Bill using Bill's public key

```
gpg --output doc_pdf.gpg --encrypt --recipient  
b@ms.us doc.pdf #must have Bill's public key
```

- Bill Decrypts the document (must have his private key & passphrase)

```
gpg --output doc.pdf --decrypt doc_pdf.gpg
```

- Documents may be encrypted without key, just with passphrase

```
gpg --output doc_pdf.gpg --symmetric doc.pdf  
Enter passphrase:
```



# Authenticate Docs with Digital Signatures

- Digitally signed document ensure they are authentic & untempered

```
gpg --output doc.signed --sign doc.pdf
```

Enter Passphrase:

Must have the private key to sign

- A signed document can be verified and decrypted like so:

```
gpg --ouput doc.pdf --decrypt doc.signed
```

Must have owner's public key

# part 8: Bash Tools

*For when that 'hello world' becomes a project*

[back to toc](#)

# Bash Shell Basics

- Commands and utilities such as **grep**, **sed**, **awk** may be invoked
- Variables, constants, conditionals, loops and functions may be defined
- Arithmetic operations available
- Logical operations **&&** (AND) and **||** (OR) available:
  - **wget ... || curl ...** : run curl *iff* wget **fails**
  - **make install && make test** : test *iff* install **succeeds**
- Shell "Startup" files set environment as you start your shell
  - **.bashrc** : a file that runs in each new shell that is spawned
  - **.bash\_profile** : a file that runs only in a "login shell" (and not all shells eg. it won't run if you invoke a shell script that creates a subshell)

# Aliases and Functions

- Aliases are short and convenient names for long commands
- They are usually defined in `.bashrc` or a separate `.aliases` file
- To temporarily bypass an alias (say we aliased `ls` to `ls -a`), use `\:`  
`\ls`
- Bash functions are usually defined in `.bashrc/.bash_profile`
- Functions are more expressive and preferred over aliases

# Examples of useful aliases

- `alias s=ssh`
- `alias c=clear`
- `alias cx='chmod +x'`
- `alias ls='ls -thor'`
- `alias more=less`
- `alias ps='ps auxf'`
- `alias psg='ps aux | grep -v grep | grep -i -e USER -e'`
- `alias ..='cd ..'`
- `alias myp='ps -fjH -u $USER'`
- `alias cleanup='rm -f *.tmp *.aux *.log'`

# Examples of useful Functions

- `mcd() { mkdir -p "$1"; cd "$1"; }`
- `cdl() { cd "$1"; ls; }`
- `backup() { cp "$1"{, .bak}; } #test first`
- `gfind() { find / -iname "$@" 2>/dev/null }`
- `lfind() { find . -iname "$@" 2>/dev/null }`
- `rtfm() { help "$@" || man "$@" || $BROWSER "http://www.google.com/search?q=$@"; }`
- See `/usr/share/doc/bash-*/examples/functions` for more function examples

# Variables and Command Substitution

- Variables are implicitly typed
- May be a literal value or ***command substitute***
- **vname=value** #assign value to variable vname
- **\$vname** #read value of variable vname

```
#!/bin/sh  
msg="Hello World"  
echo $msg
```

- Command substitution:
  - **curdir=\$(pwd)**
  - **curdate=\$(date +%F)**
  - **echo "There are \$(ls -l | wc -l) items in the current dir"**

# Conditionals

- if-then-else construct to branch similar to programming languages
- Two forms of conditional evaluation mechanisms:
  - `test` and `[ ... ]`

```
$ if test $USER = 'km0'; then echo 'I know you';  
else echo 'Who are you'; fi
```

```
$ if [ -f /etc/yum.conf ]; then echo 'yum.conf  
exists'; else echo 'file do not exist'; fi
```



# Conditionals summary

- string
  - `-z string`: length of string 0
  - `-n string`: length of string not 0
  - `string1 = string2`: strings are identical (note a single =)
- numeric
  - `int1 -eq int2`: first int equal to second
  - `-ne`, `-gt`, `-ge`, `-lt`, `-le`: not-equal, greater-than, -greater-or-equal...
- file
  - `-r filename`: file exists and is readable
  - `-w filename`: file exists and is writable
  - `-f`, `-d`, `-s`: regular file, directory, exists and not empty
- logic
  - `!`, `-a`, `-o`: negate, logical and, logical or

# Loops

- Basic structure (three forms):

```
for i in {0..9}; do echo $i; done
```

```
for ((i=0;i<10;i++)){ echo $i;} #C-like
```

```
for var in list; do command; done #'python-like'
```

- often used with command substitution:

```
for i in $(\ls -l *.txt); do echo "$i"; done
```

```
for i in $(get_files.sh); do upload.sh "$i"; done
```

# The heredoc

- Create "inplace" files
- example:
- **sh << END**  
**echo "Hello World"**  
**END** *<press enter>*
- Uses of heredoc
  - Multiline message using cat
  - Use variables to plug into created files, eg test multiple configurations for a program

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx
EOF
```

```
#!/bin/bash
for i in local remote cluster all
do
  cat <<END>install.yml
  ---
  - hosts: $i
    <other stuff>
  END
  ansible-playbook install.yml --check > out"$i".txt
done
```

# part 9: Program Development Tools

*get-serious stuff*

[back to toc](#)

# Programming Language Platforms

- Interpreted programming platforms available on most systems
  - Python, Perl, awk, bash
  - We cover awk, some bash and a bit of python
- Compiled programming platforms available on most systems
  - C, Fortran
  - We cover C in this section
- Additionally, a build system called **Make** is available

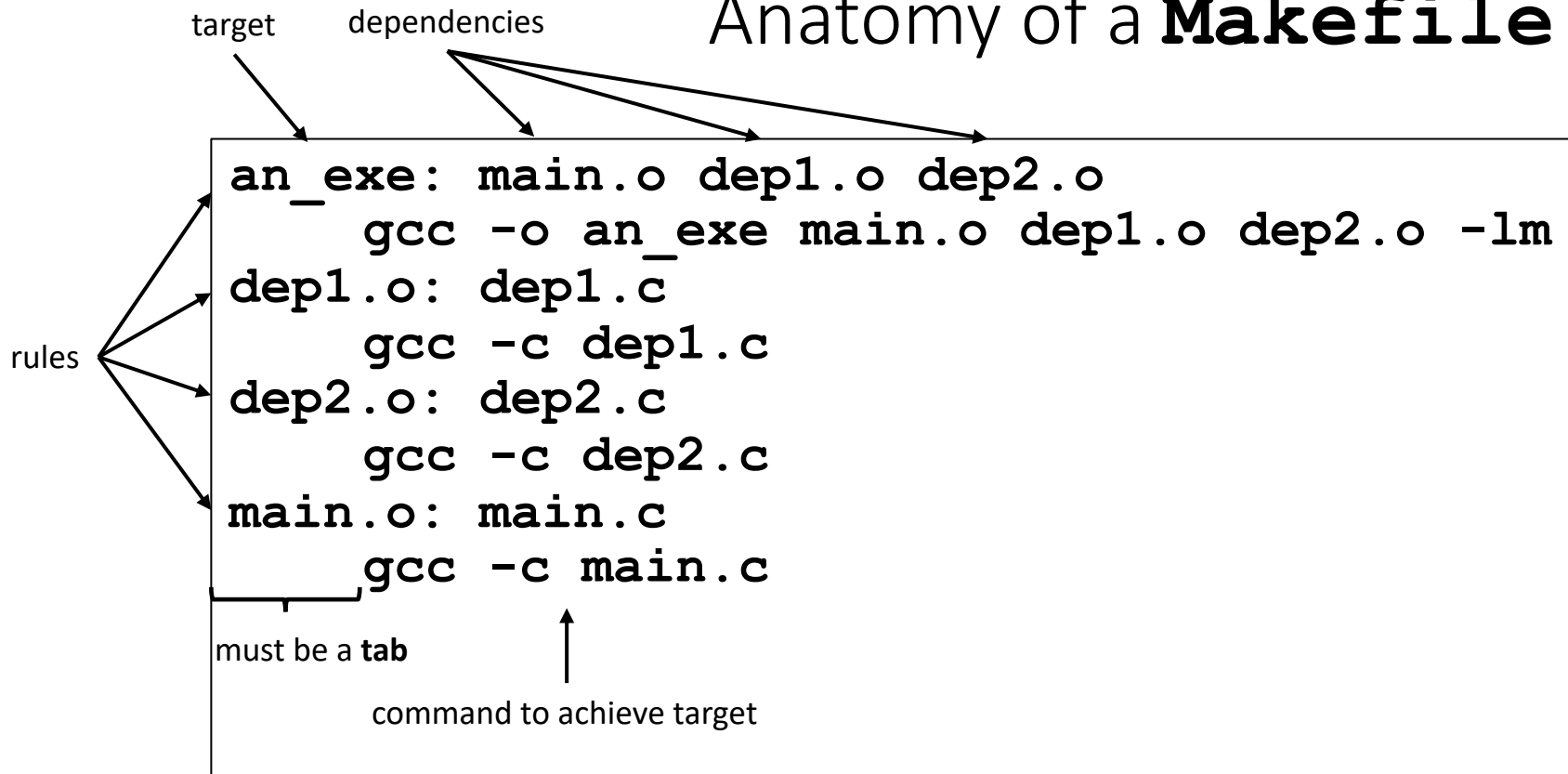
# Elements of **C** Program Development

- The **source code** that is written/edited by a programmer
  - Often split into header files ( **.h** ) and source code files ( **.c** )
- The compiler **gcc** does the following
  - compile ( **-S** ) convert the source code ( **.c** ) to **assembly code** ( **.s** )
  - assemble ( **-c** ) -- translate the assembly code to **object code** ( **.o** )
  - link ( **-l** ) -- link to the standard libraries to produce **executable**
- By default gcc combines the above stages producing the executable  
**gcc hello.c** #creates a.out; no .o or .s files

# The **make** build system

- Automates compilation of multiple source files in a complex project
- Streamlines dependent actions and performs them in order
- Reads configuration from a "build" file usually named as Makefile
- Makefile acts as an artefact of project build process

# Anatomy of a **Makefile**





# How the **make** command works

- The **make** command will read from the Makefile and run commands in order to build the ultimate target
- For instance, in the Makefile shown in previous slide, **make** will run commands for rule 2-4 followed by rule 1:

```
gcc -c dep1.c #create dep1.o
```

```
gcc -c dep2.c #create dep2.o
```

```
gcc -c main.c #create main.o
```

```
gcc -o an_exe main.o dep1.o dep2.o -lm
```

## part 10: Miscellaneous Utilities

*handy like midnight snack*

[back to toc](#)

# Get things done at specific times with **at**

- **at** will execute the desired command on a specific day and time
  - **at 17:00 #press enter**  
**at> log\_days\_activities.sh #smtimes no at> prompt**  
**[ctrl-d]**
  - **at** offers keywords such as **now, noon, today, tomorrow**
  - offers terms such as **hours, days** to be used with the **+** symbol

**at noon**

**at now + 1 year**

**at 3:08pm + 1 day**

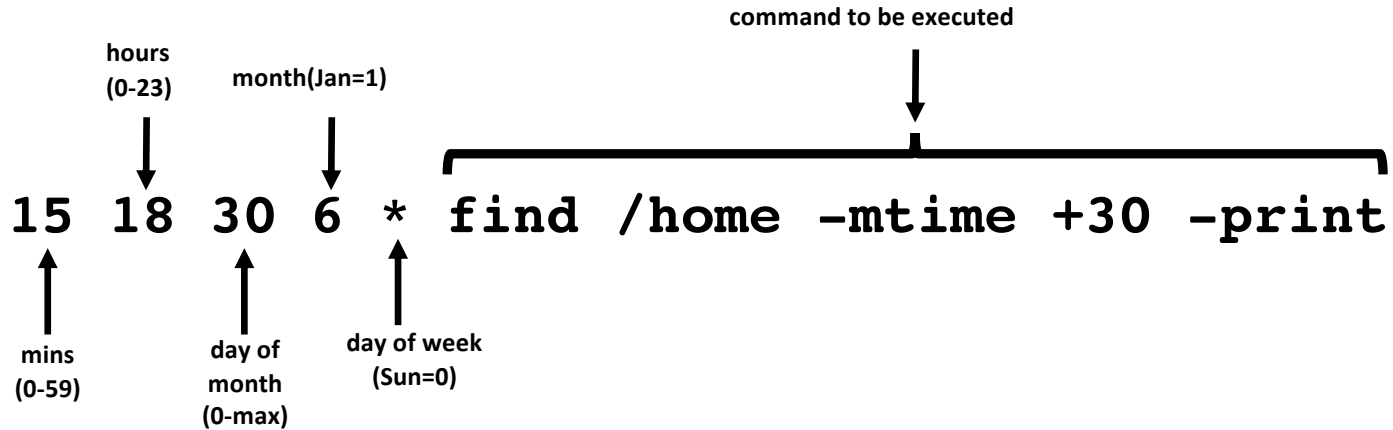
**at 15:01 December 19, 2018**

# Get things done periodically with **cron**

- **cron** will execute the desired command periodically
- A **crontab** file controls and specifies what to execute when
- An entry may be created in any file and added to system with the **crontab** command like so:  

```
echo '15 18 30 6 * find /home -mtime +30 -print' > f00  
crontab f00 #add above to system crontab
```
- **crontab -l** #list crontab entries  
**crontab -r** #remove crontab entries
- Output of the cron'd command will be in **mail** (alternatively it may be redirected to a file with '>')
- What does the entries in a crontab mean though? (see next slide)

# Anatomy of a crontab entry



Run the find command on June 30 of every year at 6:15 PM no matter what day of week it is.

# Math

- Generate random number using **shuf** (may need to install)
  - **shuf -i 1-100 -n 1**
- Format numbers with **numfmt**
  - **numfmt --to=si 1000**  
**1.0K**
  - **numfmt --from=iec 1K**  
**1024**
- **bc** is a versatile calculator
  - **bc <<< 48+36** #no space on either side of +
  - **echo 'obase=16; ibase=10; 56' | bc** #decimal to hex
  - **echo 'scale=8; 60/7.02' | bc** #arbitrary precision

# Python utilities

- Stand up a simple web server in under a minute with Python
  - `python3 -m http.server 35000`
- Pretty print a json file
  - `python3 -m json.tool afile.json`
- Run small python programs
  - `python -c "import math; print(str(math.pi)[:7])"`
- Do arithmetic
  - `python -c "print(6*6+20)"`
  - `python -c "fctrl=lambda x:0**x or x*fctrl(x-1); print(fctrl(6))" #compute factorial`

# Random stuff - 1

- Run a command for specified time using **timeout**:  
`timeout 2 ping google.com`
- **watch** a changing variable
  - `watch -n 5 free -m`
- Say **yes** and save time
  - `yes | pip install pkg --upgrade`
  - `yes "this is a test" | head -50 > testfile.txt` # create file with arbitrary no. of lines
- Create pdf from text using **vim**:  
`vim states.txt -c "hardcopy > states.ps | q" &&  
ps2pdf states.ps #convert ps to pdf`



## Random stuff - 2

- Run a command as a different Linux group
  - `sg grpgit -c 'git push'`
- Display a csv in columnar/tabular format
  - `column -t -s , filename.csv`
- Have difficulty sending binary executables over emails?
  - `xxd f.exe f.hex` #hexdump the exe, send over email
  - `xxd -r f.hex f.exe` #receiver convert back to exe
- Generate password
  - `head /dev/urandom | tr -dc A-Za-z0-9 | head -c 8`
  - `openssl rand 8 -base64 | cut -c1-8` #-base64 8 for some version

## Random stuff - 3

- pandoc to convert between md, tex, txt, html, docx, pdf, odt

```
pandoc manual.md -o manual.pdf
pandoc example.txt -o example.html
pandoc -s example.txt -o example.docx
```
- Parse and read **xml** files with **xmllint**
- Split a large file into small chunks (eg. to send as attachment in mail)
  - **split -b 20M large.tgz parts\_** #20MB chunks  
#send parts\_\* over mail
  - **cat parts\_a\* > large.tgz** #at receiving end

# Summary

- Linux command-line environment powerful if exploited well
  - Pipes and redirection key Linux contributions
  - Rewarding in the short-term as well as long-term
  - Classical and modern tools well suited for modern-style usage
  - Practice!
- 
- Send comments, feedback, questions: **km0@ornl.gov**

# Credits, references and resources

- The man, info and doc pages
- bash: [gnu.org/software/bash/manual/bashref.html](http://gnu.org/software/bash/manual/bashref.html)
- grep: [gnu.org/software/grep/manual/grep.html](http://gnu.org/software/grep/manual/grep.html)
- sed: [catonmat.net/blog/worlds-best-introduction-to-sed](http://catonmat.net/blog/worlds-best-introduction-to-sed)
- awk: [ferd.ca/awk-in-20-minutes.html](http://ferd.ca/awk-in-20-minutes.html)
- tmux: [gist.github.com/MohamedAlaa/2961058](https://gist.github.com/MohamedAlaa/2961058)
- wikipedia articles: unix, linux, Bash\_(Unix\_shell)
- [commandlinefu.com](http://commandlinefu.com)

# Where to go from here

- [github.com/jlevy/the-art-of-command-line](https://github.com/jlevy/the-art-of-command-line)
- [jeroenijanssens.com/2013/08/16/quickly-navigate-your-filesystem-from-the-command-line.html](https://jeroenijanssens.com/2013/08/16/quickly-navigate-your-filesystem-from-the-command-line.html)
- [linux.byexamples.com](https://linux.byexamples.com)
- [catonmat.net/blog/bash-one-liners-explained-part-three](https://catonmat.net/blog/bash-one-liners-explained-part-three)
- [wiki.bash-hackers.org](https://wiki.bash-hackers.org)
- [gist.github.com/MohamedAlaa/2961058#file-tmux-cheatsheet-markdown](https://gist.github.com/MohamedAlaa/2961058#file-tmux-cheatsheet-markdown)
- [wizardzines.com](https://wizardzines.com)
- [crontab.guru](https://crontab.guru)
- [leimao.github.io/blog/Tmux-Tutorial](https://leimao.github.io/blog/Tmux-Tutorial)
- [unix.stackexchange.com](https://unix.stackexchange.com)
- [danyspin97.org/blog/makefiles-best-practices](https://danyspin97.org/blog/makefiles-best-practices)

Thank you for your time and attention  
Questions?

# Practice and Exercises

- Create three tmux sessions: s1, s2 and s3; detach them
- List the active sessions with **tmux ls**
- Kill the active sessions with **tmux kill-session -t <name>**
- Can you kill them all with one command? hint: use xargs in a pipe
- Create a tmux session and split the screen into 4 panes vertically and horizontally
- Set it so that all panes are synchronized. Test with any command.

# Practice and Exercises

- Use your favorite editor to edit `.bashrc` and `.bash_profile` --
  - add a line: `echo 'I am bashrc'` to `.bashrc`
  - add a line: `echo 'I am bash_profile'` to `.bash_profile`
- Close and reopen terminal, what do you see? Within terminal type `/bin/bash`, what do you see?
- Create a copy of `prose.txt` using `cp prose.txt tmp.txt`; make small change to `tmp.txt` and compare `prose.txt` and `tmp.txt` with `cmp`, `comm` and `diff`
- Delete those lines from `.bashrc` and `.bash_profile` when done
- The character class `[[:class:]]` may be used as wild card: class may be `alpha`, `alnum`, `ascii`, `digit`, `upper`, `lower`, `punct`, `word`; try `ls /etc/[[:upper:]]*`



# Practice and Exercises

- List all conf files in **/etc** you have access to, redirect stderr to **/dev/null**
- Build a software and collect errors and output in separate files, fill in the \_\_\_\_  
**make all \_\_\_\_ std.out \_\_\_\_ >std.err**
- Run cmake command and gather all logs in a single file in background  
**cmake .. \_\_\_\_ cmake.log \_\_\_\_ #bash v4 and above**
- Same as above in long format  
**mpirun -np 8 ./a.out \_\_\_\_ outerr.txt 2>\_\_1**

# Practice and Exercises

Simplify the following command line:

```
TOKEN=$(kubectl describe secret -n kube-system  
$(kubectl get secrets -n kube-system | grep  
default | cut -f1 -d ' ') | grep -E '^token' |  
cut -f2 -d ':' | tr -d '\t' | tr -d " ")
```

Hints:

- Replace the **cut** commands with **awk** commands
- Accommodate the **grep** within **awk**
- Accommodate the two **tr** commands within **awk** commands (hint: use **awk**'s **gsub** built-in function)

# Practice and Exercises

- Create a file titled the words that start with letter 'C' ( **fill the \_\_** ):
  - `grep -i '^c' states.txt | awk '{print $4}' | __ touch`
- Remove temporary files:
  - `find . -iname '*.tmp' | __ rm #ok`
- Create a directory for all running processes
  - `ps | awk 'NR != 1 {print $4}' | mkdir #NOT OK`
  - `ps | awk 'NR != 1 {print $4}' | __ mkdir #ok`

# Practice and Exercises

- Use **sed** to print lines 11-15 of states.txt
- Fill up the `__` in the following find commands
  - `__ . -type d -perm 777 -exec chmod 755 {} +`
  - `find . -type __ -name "*.tmp" -exec rm -f {} +`
  - `find __ -atime +50 #files <50 days in /usr/local/lib`
  - `find . -mtime __ -mtime -100 #<50 & <100 days`
- Use **awk** to print only the state names and capitals columns from states.txt
- use **grep** to search for all lines of file states.txt containing a word of length four or more starting with the same two characters it is ending with. You may use extended regular expressions (**-E**)

# Practice and Exercises

Muammar Gaddafi was a Libyan politician. He was in the news a few years ago. News agencies spelled his name differently like so:

- **Muammar al-Kaddafi** (BBC)
- **Moammar Gadhafi** (Associated Press)
- **Muammar al-Qadhafi** (Al-Jazeera)
- **Mu' ammar Al-Qadhafi** (US Department of State)

Your task is to come up with a Regular expression that will match with all the above occurrences. (Hint: use extended regular expression)

- Test with both **grep** and **awk** by putting the above lines in a file as well as a *heredoc*

# Practice and Exercises

- Compare the time it takes with and without the -C switch of scp to send data remotely (hint: use the **time** command)
- Create a config file in your ~/.ssh directory, make appropriate changes and add the contents presented in previous slides to it. How will you test if it works?

# Practice and Exercises

- Run **yes** for 5 seconds using **timeout**, collect the output in file x.txt
- Create an alias **d** to print current **date**
- Run **style** and **diction** (if available) on prose.txt
- Interpret the following crontab entry:  
`30 21 * * * find /tmp /usr/tmp -atime +30 -exec rm -f {} +`
- Frame an **at** command to run the date command tomorrow at 8 p.m.
- write a shell script to find all the prime numbers between 1000 and 10000
  - hints: use **for**, **if**, **factor**, **wc**