

Linux Productivity Tools

Ketan M. (km0@ornl.gov)

Oak Ridge National Laboratory

Table of Contents

- Part 1: [Overview and Logistics](#) *[3 slides]*
- Part 2: [The Basics](#) *[12 slides]*
- Part 3: [Streams, pipe and redirection](#) *[11 slides]*
- Part 4: [Classic Tools](#) *[21 slides]*
- Part 5: [Session management](#) *[4 slides]*
- Part 6: [Safe and secure use of facilities](#) *[9 slides]*
- Part 7: [Debugging](#) *[4 slides]*
- Part 8: [Scripting and programming tools](#) *[13 slides]*
- Part 9: [Miscellaneous Utilities](#) *[12 slides]*
- [Summary](#)

Part 1: Overview and Logistics *[3 slides]*

orientation and practical stuff

[back to toc](#)

Slides and practice data for download

- Two text files, a pdf and a pptx file for slides
- states.txt
 - Tabular data
 - Five columns
- prose.txt
 - Prose with sentences and paragraphs
- www.olcf.ornl.gov/calendar/linux-command-line-productivity-tools/
- uncompress: **unzip lpt.zip**

About You and Me

- Basic knowledge of Linux is assumed but feel free to interrupt and ask questions
 - common commands, basic understanding of Linux files and directories, editing of simple files etc.
- Access to a basic Linux terminal is assumed
 - A linux laptop, Macbook will do
 - login node to some cluster
- About Me
 - Linux Engineer with CADES
 - Command line enthusiast

Overview

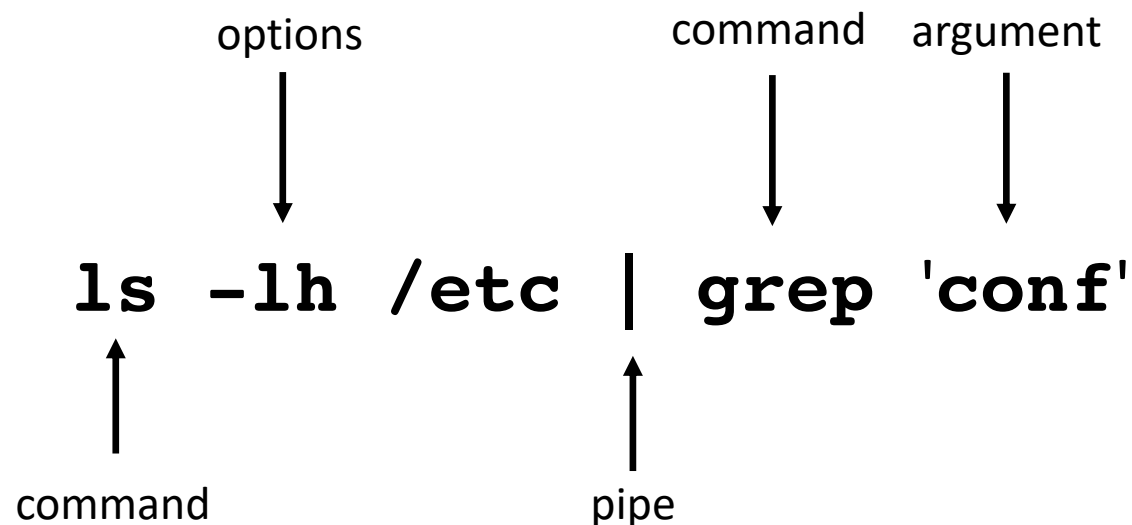
- Build powerful **command-lines and scripts**
 - **We will use Bash with default key-bindings**
 - **We will assume GNU/Linux and call it Linux**
- Linux utilities that are available on most installations
- Goal is to be efficient and effective rather than to be an "expert"
- Benefits: save time, efficient for system, good longterm payback
- **We cover: productive tools and techniques for an HPC Linux user**
- **We do not cover:** System administration, security, networking

part 2: The Basics *[12 slides]*

welcome to the school of command line wizardry!

[back to toc](#)

Anatomy of a Linux Command



append & at the end to run the command in background

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 information about the cpu architecture
- **free**: free and used memory (try **free -g**)
- **lsb_release**: distribution information (try **lsb_release -a**)

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

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

Wildcards

- *** any number of characters**
 - `ls -lh /etc/*.conf`
- **? expands to one character**
 - `ls -ld ? ?? ???`
- **Negation (!) eg. `ls -ld [!0-9]*`**
- **Escaping and quoting**
 - `\` for escaping a wildcard
 - `'` for quoting a wildcard
 - both will prevent expansion

Many ways to get help

- **man nano**
 - Manual pages
 - Organized sectionwise--one page for each section (if exists)
- **wget --help**
 - Handy for quick syntax reference
- **info curl**
 - Modern
- Browse **/usr/share/doc/git-1.***
 - Usually a README file has info

Work with Files

- **cat** for relatively short files
 - `cat prose.txt`
- **less** is more than **more** for long files
 - `less /etc/ntp.conf`
- **tail -f**: 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
- Compare files with **cmp** (byte by byte), **comm** (line by line), **diff** (differences line by line)

Work with Processes

- List the processes: **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**
- Be **nice** and fly under the radar
 - Append "nice -n <niceeness_value>" to any command to lower its prio, eg:
 - **nice -n 19 tar cvzf archive.tgz large_dir**
- Kill a process: **kill <pid>**

Work with Web: curl, wget, links

- **curl**: a tool to transfer data to/from the web
- **curl** is commonly used as a quick command to download files from the web
 - `curl -O http://www.gutenberg.org/files/4300/4300-0.txt`
- *libcurl* is a curl based library that could be used in program development
- **wget** is a similar utility; it has no library though
 - `wget http://www.gutenberg.org/files/4300/4300-0.txt`
- **links** is a useful text-based browser:
 - over remote connections and when **curl**/**wget** won't work
 - avoid pesky ads on the web
 - when internet is slow and only care about text

Be a command line ninja: Navigation

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

The diagram illustrates the command `ls -lh | grep 'May 10' > result.txt` with arrows indicating navigation shortcuts. A horizontal line is drawn below the command, with vertical arrows pointing up to specific characters. The shortcuts are: `ctrl-a` for the start of the command, `ctrl-xx` for the pipe character, `alt-b` for the start of the search term, `cursor` (in red) for the opening quote, `alt-f` for the end of the search term, and `ctrl-e` for the end of the command.

```
ls -lh | grep 'May 10' > result.txt
```

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

Be a command line ninja: Deletion

ls -lh | grep 'May 10' > result.txt

← ctrl-u ctrl-w **cursor** alt-d ctrl-k →

use ctrl-y to paste back the last deleted item

Useful Shortcuts

- **!!** 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 Useful Tricks

- **>x.txt** #quickly create an empty file
- **fc** # (bash builtin) edit to **f**ix last **c**ommand
- **ctrl-l** #clear terminal
- **cd -** #change to previous dir
- **cd** #change to homedir
- **ctrl-r** #recall from history
- **ctrl-d** #logout from terminal

Practice and Exercises [5-7 mins]

- 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:]]*`

Part 3: Streams, pipe and redirection

[11 slides]

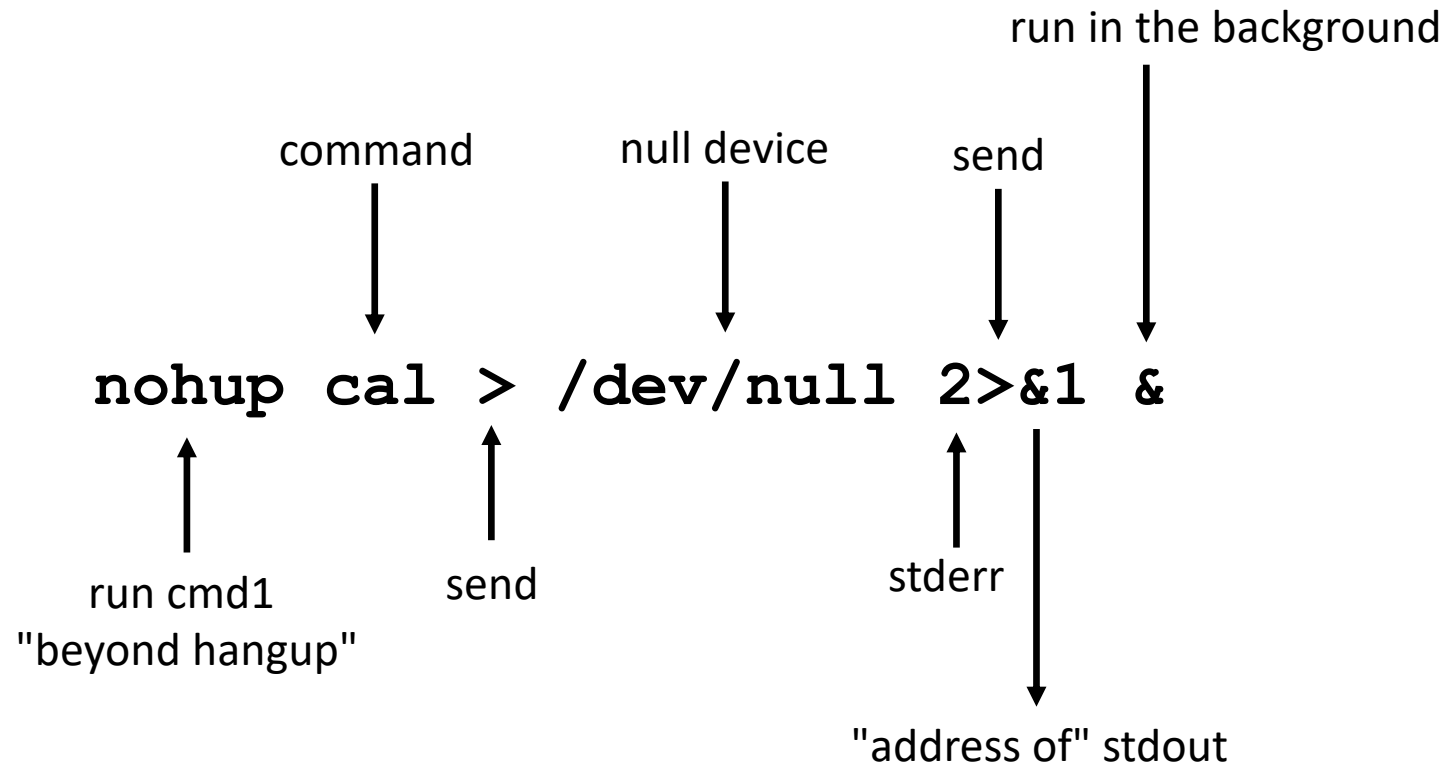
I am sure a gardener designed them!

[back to toc](#)

Streams and Redirection

- Three streams: standard input (stdin), standard output (stdout) and standard error (stderr)
- Represented by **"file descriptors"**:
 - 0 for stdin
 - 1 for stdout
 - 2 for stderr
- **&** is used to **"write into"** a stream, eg. **&1** to write into stdout
- Angle brackets are used for redirection:
 - > to send
 - < to receive
 - >> to append
 - << to in-place append (used in "heredoc")
 - <<< is used in "herestring" (not covered today)

Anatomy of a redirection using streams



Redirection Quickref

- Send stdout to a file: `ls > stdout.txt`
- Send stderr to a file: `bad_command 2> stderr.txt`
- Send stdout and stderr to same file:
`cmd > stdouterr.txt 2>&1`
`cmd &> stdouterr.txt` #bash v4 and later
- Send stdout and stderr to a file and background the command:
`long_running_cmd > stdouterr.txt 2>&1 &`
- Disregard stdout and stderr: `cmd > /dev/null 2>&1`
- Disregard stdout and stderr and let it run after you log out:
 - `nohup overnight_running_cmd > /dev/null 2>&1 &`

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
```

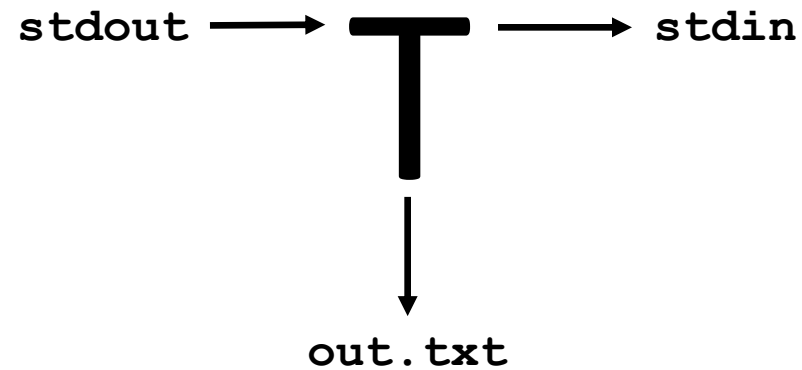
```
history | tail #last 10 commands
```

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

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


tee: send stdout to file and pipe/console

```
find . | tee out.txt | less
```



Exceptions

- Most commands receive input from stdin (so, pipe) **and** file, eg.
 - `wc states.txt #ok`
 - `wc < states.txt #ok`
- Produce error messages (**if any**) on stderr
- 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`
 - `echo states.txt #NOT OK` (assuming you 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
 - Appends **standard input** to commands as **arguments**
 - Partitions the list of arguments to a limited number and runs the command over them repeatedly as needed
- For instance create files with names on the somelist.txt file:
cat somelist.txt | xargs touch

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

GNU Parallel Examples

- Find all html files and move them to a dir

```
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>/dev/null"
```

Practice and Exercises-1 [5 mins]

- Try the commands discussed in this section
- List all conf files in /etc that are permitted to access, redirect stderr to /dev/null
- Build a software and collect errors and output in separate files, replace underlines with right values
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-2 [5 mins]

- Create a file titled the words that start with letter 'C' (**fill the __**):
 - `grep '^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`

Part 4: Classic Tools *[21 slides]*

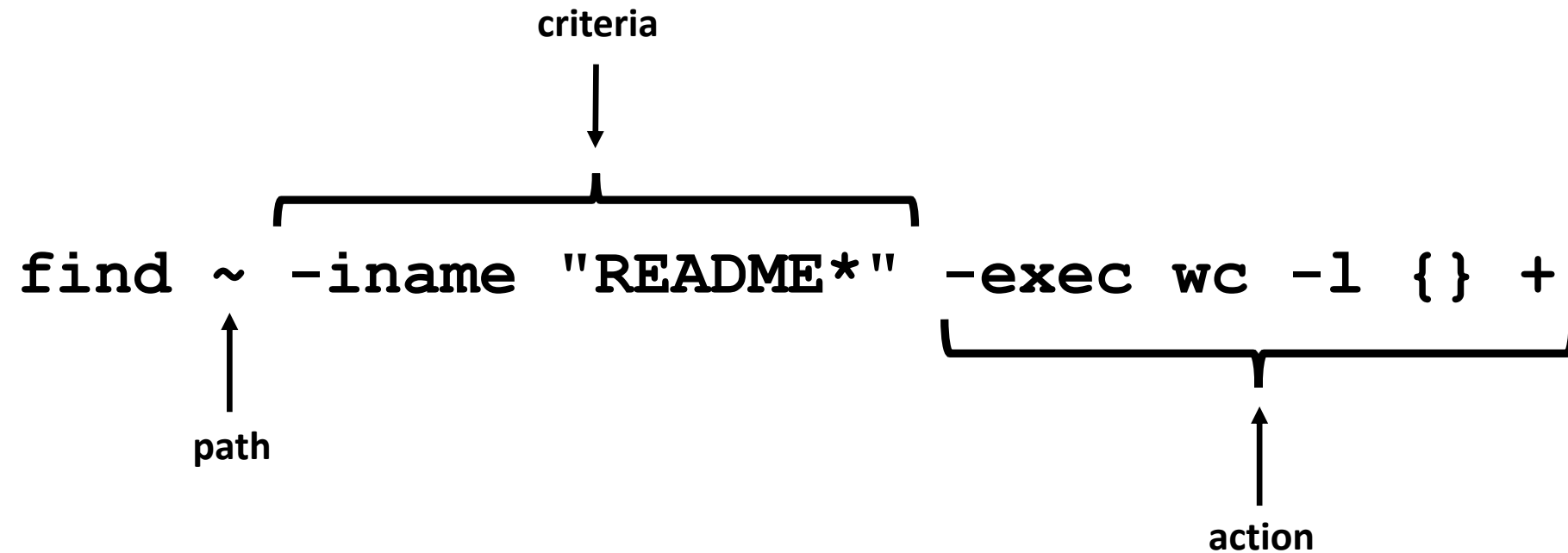
the evergreens

[back to toc](#)

The Versatile find

- *Recursively* examines a directory tree to look for files matching criteria and optionally takes *action* on found files
- Flexible: multiple criteria may be combined to build a very specific description of the file being searched
- Efficient: simple **find** is often faster than **ls**—very handy on slower filesystems with a large number of files
 - **find . -maxdepth 1** #equivalent to **ls**

Anatomy of find



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)`
 - `-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
 - `-exec cmd` -- execute command *cmd*
 - `-ls` -- run **ls -lids** command on each resulting file
 - `-ok cmd` like `exec` except that command executed after user confirmation

find Examples

- **find . -type f -name "*.txt"** #all text files in current dir
- **find ./somedir -type f -size +512M -print** #all files larger than 512M in ./somedir
- **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
↓
grep **-i -n** 'col' states.txt
↑
regular expression
input file
↓

```
graph TD; options --> flags["-i -n"]; flags --- grep["grep"]; flags --- regex["'col'"]; regex --- input_file["states.txt"]; input_file --> input_file_label["input file"];
```

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 number of occurrences
- **-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 is an **expression** that matches a **pattern**.
- example pattern:

^why waste time learning, when ignorance is instantaneous?\$

- regular expression:

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

 → no match
- regular expression:

e	a	r
---	---	---

 → **one** match → "learning"
- regular expression:

w	h
---	---

 → **two** matches → "why" and "when"
- regular expression:

^	w	h
---	---	---

 → one match → "why"
- regular expression:

u	s	?	\$
---	---	---	----

 → one match → "instantaneous?"

Regular Expressions-contd.

- Special characters:
 - `^<anything>` will match from beginning of a line
 - `<anything>$` will match up to end of line
 - `.` will match any character
- Character class: one of the items in the `[]` will match, sequences allowed
 - `'[Cc]at'` will match Cat and cat
 - `'[f-h]ate'` will match fate, gate, hate
 - `'b[^eo]at'` will match "brat" but **not** "boat" or "beat"
- **Extended** regular expressions (use with `egrep` or `grep -E`)
 - `'*'` matches zero or more, `'+'` matches one or more, `'?'` matches zero or one occurrence of the **previous character**
 - `'|'` is a delimiter for multiple patterns, `'(' and ') '` let you group patterns
 - `{ }` may be used to specify a repetition range in numbers

Who is this?



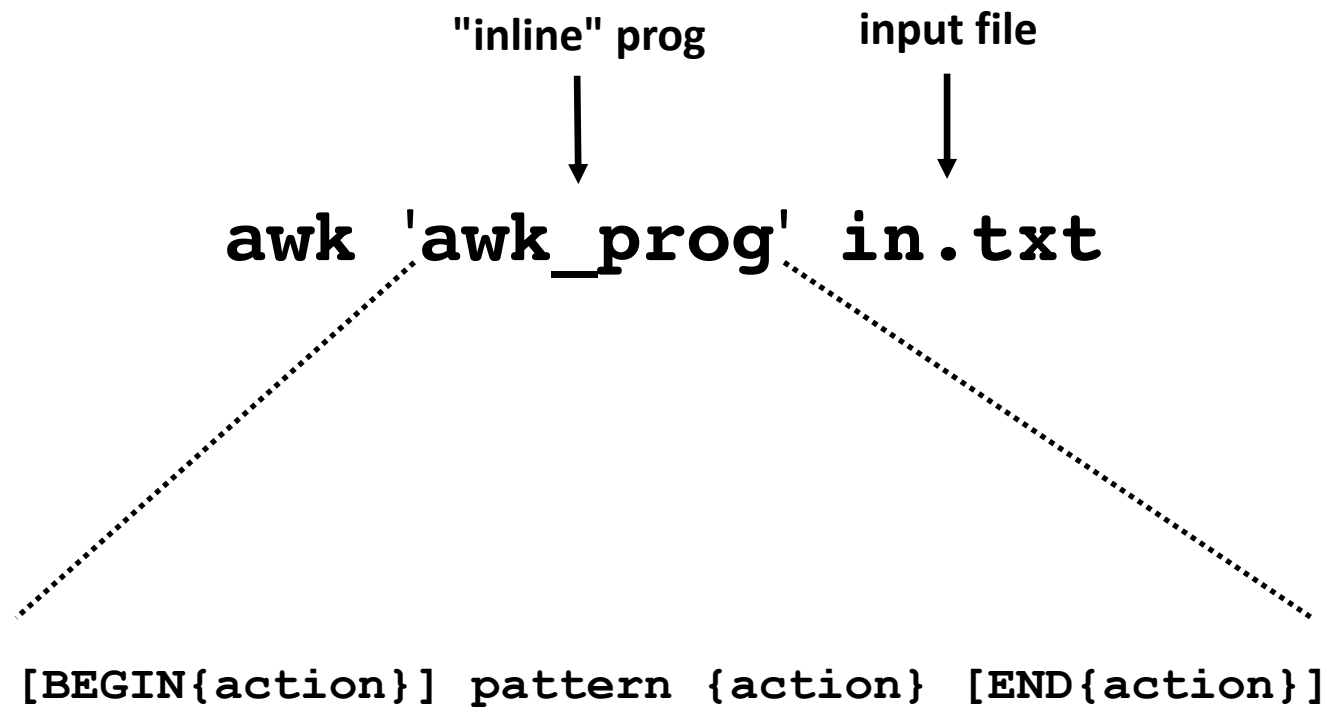
grep Examples

- Lines that end with two vowels: `grep '[aeiou][aeiou]$\ ' prose.txt`
- Count occurrence of term 'max': `grep -c 'max' prose.txt`
- Check 5 lines before and after the term 'little':
 - `grep -A5 -B5 'little' prose.txt`
 - `history | grep 'successful outputs' #comment commands and search with grep`
- A classic demo of the power of regular expressions
 - `M[ou] '?am+[ae]r ([AEae]1[-])?[GKQ]h?[aeu]+([dtz][dhz]?) {1,2}af[iy]`
- This regular expression matches the dictator's name used by various news agencies:
 - `Muammar al-Kaddafi` (BBC)
 - `Moammar Gadhafi` (Associated Press)
 - `Muammar al-Qadhafi` (Al-Jazeera)
 - `Mu' ammar Al-Qadhafi` (US Department of State)

awk: Extract and Manipulate Data

- A **programmable** filter that reads and processes input **line by line**
- Has variables, loops, conditionals, arrays and built-in functions
- Reads input from file as well as standard input (so, pipes are good)
- May be run as **command** as well as an independent **program**
- **We will only discuss the command form of awk**
- **Highly recommended book:** The awk programming language by Aho, Kernighan and Weinberger, PDF available to download here:
https://ia802309.us.archive.org/25/items/pdfy-MgN0H1joloDVoIC7/The_AWK_Programming_Language.pdf

Anatomy of awk



awk patterns and actions

- A **pattern** is a regex that matches (or not) to an input line, eg.
 - `/New/ {action}` #any line that contains 'New'
 - `/^[0-9]+ / {action}` #beginning with numbers
 - `/(POST|PUT|DELETE)/ {action}` #specific words
- An **action** is a sequence of ops performed on matching lines, eg.
 - `{print $1;}` #print first field
 - `{next;}` #skip to the next line of input
 - `{for (i=1;i<x;i++) {sum += $3;}}` #run a loop
- User defined functions may be defined in any action block

awk Feature Highlights

- Fields in text are addressed by: **\$1** , **\$2** , ... , **\$NF**
 - **\$0** means the whole line
- Pattern specified as **/regex/** or **\$n~/regex/**
- Special variables
 - may be modified by user: **FS (Field Sep)** , **RS (Record Sep)** , **OFS (Output FS)** , **ORS (Output RS)**
 - may **not** be modified by user: **NF (num fields)** , **NR (num records)**
- Built-in functions, eg.: **sin** , **cos** , **log** , **rand** , **substr**

Some Useful awk Examples

- `awk '{print $1}' states.txt`
- `awk '/New/{print $1}' states.txt`
- `awk NF>0 prose.txt #skip blank lines`
- `awk '{print NF, $0}' states.txt #num fields`
- `awk '{print length($0)}' states.txt #num chars`
- `awk 1 states.txt #just print the contents of file`
- `awk 'BEGIN{print substr("New York",5)}' #York`

sed: parse and transform text

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

Anatomy of sed

sed 's/New/Old/g' states.txt

Diagram illustrating the anatomy of the `sed` command:

- `sed`: The command itself.
- `'s'`: The command type (substitution).
- `New`: The regex (regular expression) to be replaced.
- `/`: The delimiter.
- `Old`: The replacement text.
- `g`: The modifier (global).
- `states.txt`: The input file.

sed Options

- address: may be a line number or a range, defaults to 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^{th} occurrence. **g** means apply globally
- Common **sed** flags: **-n** (no print), **-e** (multiple ops), **-f** (read sed commands from file), **-i** (in place)

Some Useful sed Examples

- **sed -n '5,9p' states.txt** #print lines 5 through 9
- **sed -n '\$p' states.txt** #print last line
- **sed 's/[Aa]/B/g' states.txt**
- **sed '1,3d' states.txt** #delete first 3 lines
- **sed '/^\$/d' states.txt** #delete all blank lines
- **sed '/^\$/, \$d' states.txt** #delete from the first blank line through last line
- **[negation]sed '/York/!s/New/Old/' states.txt**
#change New to Old except when York appears in the line

Practice and Exercises [7-10 mins]

- 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**)

Part 5:
Session Management *[4 slides]*
for when the network goes down on my world-saving project

[back to toc](#)

Workspace Management with tmux

- **tmux** 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 a remote copy operation will take a long time
 - for interactive multitasking
 - for exotic stuff such as pair programming

A Short tmux Tutorial

- Typical tmux workflow

```
tmux new -s mysession #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 mysession #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 mysession #kill a session
```


Live collaboration with tmux

#user1#

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

#user2#

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

Practice and Exercises [5 mins]

- Try the commands discussed in this section
- 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

Part 6: Safe and secure use of facilities

[9 slides]

build secure tunnels

[back to toc](#)

ssh

- ssh (secure shell) used most commonly when connecting remotely
- Offers commands and options to efficiently and securely work with remote systems
- Uses the versatile and universal key-based exchange
- A rich set of practical and useful configuration features:
 - keep connection alive during inactivity
 - graphical content forwarding from remote to local
 - allows compressed data motion
 - lets you mount remote dirs using sftp

Basic usage of ssh

- Connect to a remote host

ssh [id@]remotehost

- Run a command on remote host and return

ssh id@remotehost uptime

- Connect with X11 forwarding

ssh -X id@remotehost #-Y for secure X11 fwd

- Copy your identity to remote host for keybased auth

ssh-copy-id -i id_file id@remotehost

- Connect with verbose output for troubleshooting

ssh -v id@remotehost

ssh config (~/.ssh/config)

```
Host summit
  Port 22
  hostname summit.olcf.ornl.gov
  User ketan2
  ServerAliveCountMax=3 #max num of alive messages sent without ack
  ServerAliveInterval=15 #send a null message every 15 sec
```

```
Host cades
  Port 22
  hostname or-condo-login.ornl.gov
  User km0
  ServerAliveCountMax=3
  ServerAliveInterval=15
```

```
# 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
- For 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
```

Secure Copy using scp

```
scp -r srcdir id@remotehost:$HOME #recursively copy dir
```

```
scp -C largefile id@remotehost:~/ #compress and copy
```

```
scp srcfile id@remotehost:$HOME #local to remote
```

```
scp id@remotehost:/etc/host.conf . #remote to local
```


SSH Tunneling

Why:

To access firewall'd ports on a host which is otherwise accessible by ssh

Terminology:

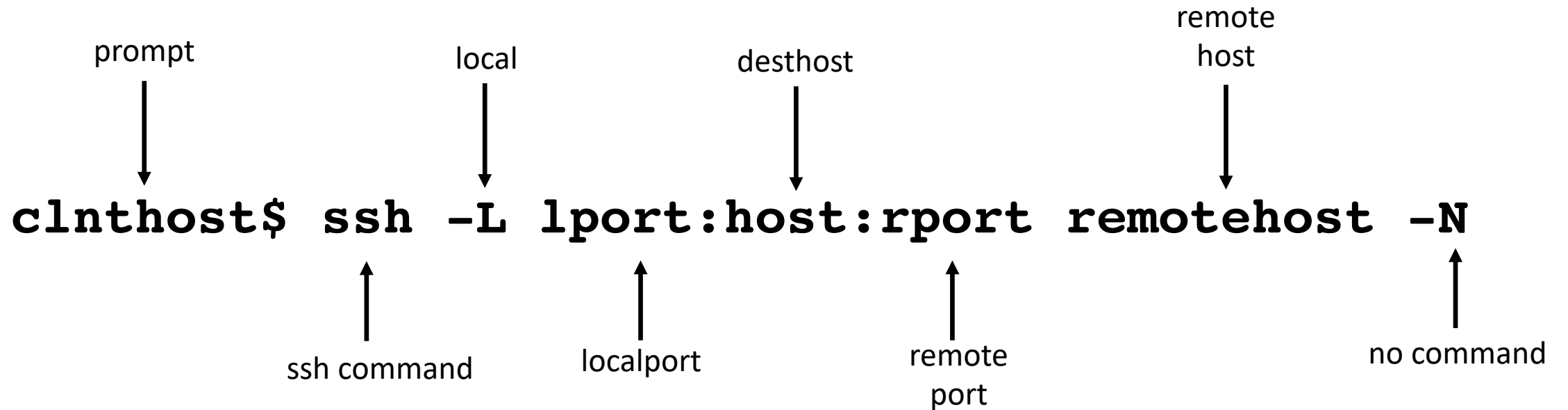
Local host: From where a connection goes out, usually runs ssh client

Remote host: One that gets connected to, runs ssh service

Port forwarding: When contents of a port are forwarded to another port on another host

Application: The application that is serving contents on given ports

Anatomy of SSH Tunneling Operation*



* simplest form

SSH Tunneling Example

- Run an HTTP server on remote node and browse through local web browser:
 - step 1. `remote$ python -m SimpleHTTPServer 25000`
 - step 2. `local$ ssh -L 8000:localhost:25000 id@remote -N`
 - Open browser on `local` and navigate to `localhost:8000`

Practice and Exercises [5-7 mins]

- Try the commands discussed in this section
- 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, add the contents presented in previous slides to it. How will you test if it works?

Part 7: Debugging *[4 slides]*

Am I (or my guests) firewalled?

[back to toc](#)

strace

- Strace will print every system call your program uses:
strace <options> <program>
strace -o lstrace.txt ls
- Useful options
 - -f : trace child processes created by current program
 - -p <pid> : trace a running process by its pid
 - -s <len> : limit the string length of the output to len characters
 - -o <outfile> : save output in a file for later analysis
 - -e <syscall> : trace only a selected system call

netcat : 'cat' over network

- Send data over network

```
cat somefile | nc <remote_ip> <rport>
```

will send contents of somefile to remote_ip on port rport

- Useful to test if a remote socket (ip + port) is accepting data or even connected!
- Can be used to test client server connections:

```
server (ip: 172.23.213.81)  
nc -l 21000 > incoming.txt
```

```
client  
cat outgoing.txt|nc  
172.23.213.81 21000
```

netstat

- Every network program either sends data to a socket (host+port) or listens for data on a socket (host+port)
- netstat will give you information about what programs are running currently and listening on what ports:

```
sudo netstat -tunapl #tuna please!
```

- **-t:** tcp
- **-u:** udp
- **-n:** do not resolve names
- **-a:** display all sockets
- **-p:** display PID/Program name for sockets
- **-l:** display listening server sockets

Practice and Exercises [5 mins]

- Try the commands discussed in this section
- Create a **netcat** client and server on the localhost and transfer a file over port 45000
- strace the **date** command, what can you say about the command from trace?
- List only the tcp connections using **netstat**
- How will you find port statistics using **netstat** (hint: man netstat)

part 8: Scripting and Programming Tools

[13 slides]

For when that 'hello world' becomes an NFS/DOE project

[back to toc](#)

Shell Scripting Basics

- Set of shell commands in a file that constitute an executable
- Arithmetic operations may be performed
- Variables and constants may be defined
- Conditionals, loops and functions may be defined
- Commands and utilities such as **grep**, **sed**, **awk** may be invoked
- A simple shell script:

```
#!/bin/bash
#prints hello
echo "Hello World"
```
- Save in a file **my.sh** and run as
bash -x my.sh # -x is a handy debugging tool

Digression: heredoc

- Handy when you need to create "inplace" files from within a script
- example:
- **sh << END**
echo "Hello World"
END *<press enter>*
- Uses of heredoc
 - Multiline message using cat
 - Use variables to plug into created files

```
cat << feed >afile.txt
message line1
message line2
feed
```

```
#!/bin/sh
now=$(date)
cat <<END>timestamped.txt
The script $0 was last executed at $now.
other stuff
END
```

Shell Variable and Assignment

- 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 -1 | wc -l) items in the current dir"**

Command line Parameters

- Parameters may be provided to a script at command line
- Accessible as **\$1**, **\$2**, ...
- Special parameters:
 - **\$0**: program name
 - **\$#**: number of parameters
 - **\$***: all parameters concatenated
 - **"\$@"**: each quoted string treated as a separate argument

```
#!/bin/sh
echo "Program: $0" # $0 contains the program name
echo "number of parameters specified is $#"
```

echo "they are \$*" #all parameters stored in \$*

```
grep "$1" $2
```

Conditionals

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

```
#!/bin/sh
if test $USER = 'km0'
then
    echo "I know you"
else
    echo "who are you"
fi
```

```
#!/bin/sh
if [ -f /etc/yum.conf ]
then
    echo "yum conf exists"
    if [ $(wc -l < /etc/yum.conf) -gt 10 ]
    then
        echo "and is more than 10 lines"
    else
        echo "and is less than 10 lines"
    fi
else
    echo "file does not exist"
fi
```

Conditional test summary

- string based tests
 - -z string: length of string 0
 - -n string: length of string not 0
 - string1 = string2: strings are identical
- numeric tests
 - int1 -eq int2: first int equal to second
 - -ne, -gt, -ge, -lt, -le: not-equal, greater-than, -greater-or-equal, -less-than
- file tests
 - -r file: file exists and is readable
 - -w file: 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:

```
#!/bin/sh
for var in list
do
    command
done
```

```
#!/bin/sh
for i in $(seq 0 9)
do
    echo $i
done
```

- Typically used with positional params or a list of files:

```
#!/bin/sh
sum=0
for var in "$@"
do
    sum=$(expr $sum + $var)
done
echo The sum is $sum
```

```
#often used as one-liners on command-line
for file in $(\ls -1 *.txt) ; do echo we have "$file"; done
```

Should I write a script or a program

script

- (often) Quick and dirty
- Limited functionality
- Slow
- Interpreted
- Poor fit for algorithms
- example: bash, R

program

- (often) Structured and "systemic"
- Broader functionality
- Fast
- Compiled
- Good fit for algorithms
- example: C, C++

Elements of program development

- Four phases:
 1. development (aka program writing)
 2. compile
 3. execution (aka runtime)
 4. debug (optional but almost inevitable)
- Tangible elements:
 - Source code file(s)
 - Object code file(s)
 - Config file(s)
 - Shared and static lib file(s)
 - Executable file(s)

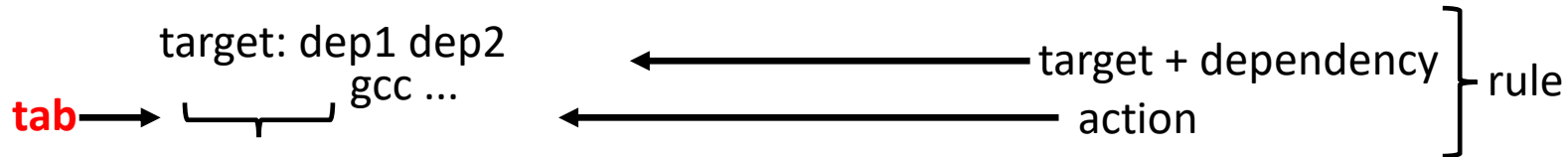
Program development tools

- Most systems have all you need to develop simple programs
 - Editors: `nano`, `vim`, `emacs`
 - Compilers: `gcc`, `g++`, `gfortran`, etc.
 - Program library tool: `ar`
 - Build system: `make`
- C code
 - Compile: `gcc -c example.c -o example.o`
 - Link: `gcc main.o example.o -o example`
 - Build a shared library: `gcc example.o -shared -o example.so`
 - Build a static library: `ar -rv example.a main.o example.o`
 - To Build using Makefile: `make -f <makefile>` #reads Makefile by default

Anatomy of a Makefile

action is taken to achieve **target**; **dependencies** are **targets** that are resolved first

target: dep1 dep2
gcc ...

tab → 

target + dependency

action

rule

To run:

\$ **make target** # optionally -f for a non-default makefile

example (use comments to interpret lines):

```
sum: main.o sum.o          # main.o and sum.o are needed to produce sum
    gcc -o sum main.o sum.o # run this when main.o and sum.o are available
main.o: main.c              # main.c is needed to produce main.o
    gcc -c main.c           # run this to produce main.o
sum.o: sum.c                # sum.c is needed to produce sum.o
    gcc -c sum.c            # run this to produce sum.o
```

An Example Makefile

```
all: exec
exec: main.o example.o
    gcc main.o example.o -o exec
main.o: main.c
    gcc -c main.c -o main.o
example.o: example.c
    gcc -c example.c -o example.o
clean:
    rm *.o exec
install: exec
    cp exec ${HOME}
```

Practice and Exercises [homework]

- Try the commands discussed in this section
- Add a new target **static** to the Makefile to build the archive **example.a**
- Develop a program in C that prints multiplication table of a number given as argument
- Develop a program in C that lists all prime numbers up to a given number
- Write a Makefile to compile the above programs using gcc

part 9: Miscellaneous Utilities *[12 slides]*

handy like a midnight snack

[back to toc](#)

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

- **at** will execute the desired command on a specific date and time
 - schedule a job for one-time execution
 - **at 17:00**
at> log_days_activities.sh #sometimes no at> prompt
[ctrl-d]
 - **at** offers keywords such as **now**, **noon**, **today**, **tomorrow**
 - also offers terms such as hours, days, weeks to be used with the + symbol

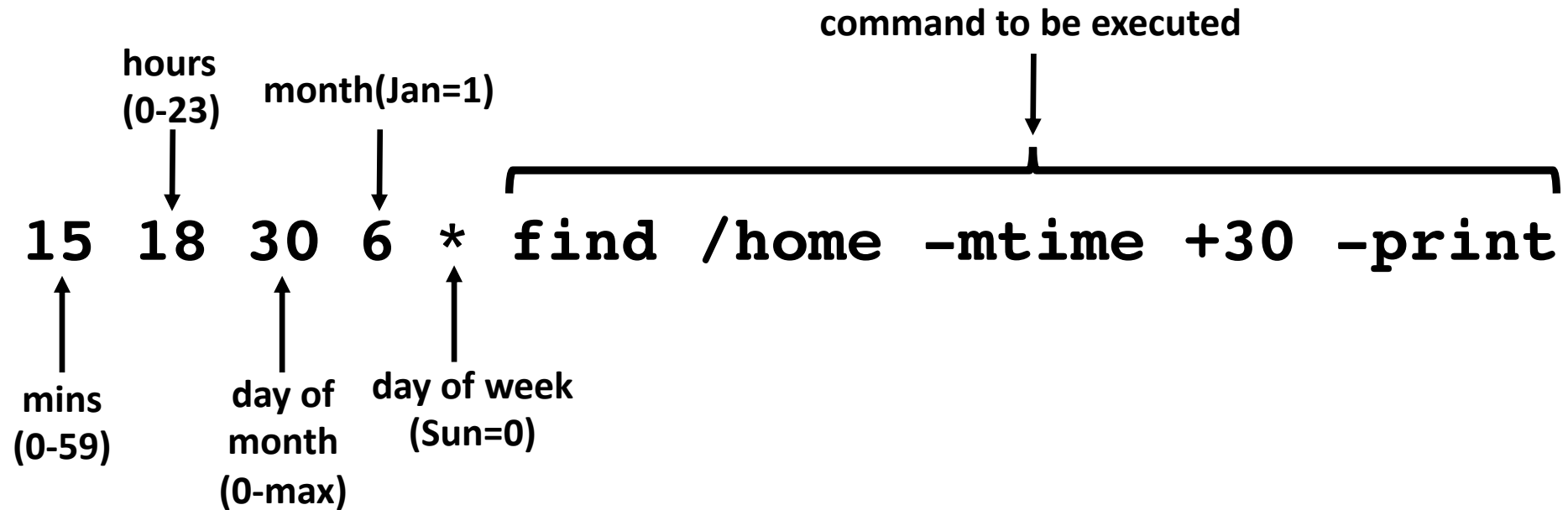
```
at noon
at now + 1 year
at 3:08pm + 1 day
at 15:01 December 19, 2018
```

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



Math

- Factorize numbers using **factor** (may need to install)
 - **factor 300**
- **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=6; 60/7.02' | bc** #arbitrary precision
- Bash for non-precise calculations
 - **echo \$((10/3))**
 - **echo \$((3**4))**

Python utilities

- Stand up a simple web server in under a minute with Python
 - `python -m SimpleHTTPServer`
 - `python3 -m http.server`
- Run small 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`

The powerful dd

- **dd**, the low-level data dumping utility is a dangerous but also one of the most useful commands when used lucidly.

For instance, burning an ISO image to USB stick is just one command away:

```
sudo dd if=ubuntu-18.04.1-desktop-amd64.iso of=/dev/sdb bs=1M
```

- **"hard-wipe"** your disk completely

```
sudo dd if=/dev/zero of=/dev/sda #dangerous
```

Aliases and Functions

- Aliases are useful to quickly type long command forms
- Aliases are usually defined in `.bashrc/.bash_profile` files or a separate `.aliases` file
- To temporarily bypass an alias (say we aliased `ls` to `ls -a`), use `\:`
`\ls`
- Functions are usually defined in `.bashrc/.bash_profile`
- Functions are useful to accomplish multiple steps in one command

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 texclean='rm -f *.toc *.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 }`
- See `/usr/share/doc/bash-*/examples/functions` for more function examples

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 | rm -r largedir #dangerous`
 - `yes ' ' | pdflatex report.tex`
- 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 group
 - `sg <grp-name> -c 'qsub myjob.pbs'`
- Format numbers with **numfmt**
 - `numfmt --to=si 1000`
1.0K
 - `numfmt --from=iec 1K`
1024
- Generate password
`head /dev/urandom|tr -dc a-z0-9|head -c8;echo`
-dc(delete complement) deletes all characters except a-z and 0-9
pwgen # may not be available by default

Practice and Exercises [5-7 mins]

- Practice the commands discussed in this section
- Run the command **yes** for 5 seconds using **timeout**
- 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**

Summary

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

Credits, references and resources

- The man, info and doc pages
- bash: www.gnu.org/software/bash/manual/bashref.html
- grep: www.gnu.org/software/grep/manual/grep.html
- sed: www.catonmat.net/blog/worlds-best-introduction-to-sed
- awk: ferd.ca/awk-in-20-minutes.html
- tmux: gist.github.com/MohamedAlaa/2961058
- wikipedia articles: unix, linux, Bash_(Unix_shell)

Where to go from here

- github.com/jlevy/the-art-of-command-line
- jeroenjanssens.com/2013/08/16/quickly-navigate-your-filesystem-from-the-command-line.html
- linux.byexamples.com/archives/42/command-line-calculator-bc
- catonmat.net/blog/bash-one-liners-explained-part-three
- wiki.bash-hackers.org
- <https://gist.github.com/MohamedAlaa/2961058#file-tmux-cheatsheet-markdown>
- wizardzines.com
- unix.stackexchange.com
- danyspin97.org/blog/makefiles-best-practices

Thank you for your attention

