All About UNIX

Table of Contents

Introduction

1. What is UNIX? What is a CLI?
    a. CLI - command line interface (as opposed to GUI - graphical user interface)
    b. uses only text-based input to signal interaction, as opposed to point-and-click or top-down databases

2. History of UNIX
    a. UNIX was a pet project to help “time-share” computers with more than one user that was written up as a paper and distributed in the 70s.
    b. People tried to make special versions in the 80s, but at this point most people have settled into the same underlying structure for just about every modern OS (beside Windows)

3. Strengths of UNIX
    a. Simplicity (just typing and reading text)
    b. Power (fast and can get at everything on your computer)
    c. Versatility (works for all kinds of programs and files)
    d. Accessibility (it’s always available on your computer even when other applications fail)
    e. Universality (if you run anything besides Windows, you have it)
    f. Badassery (you look like a serious pro when you’re using it)

Moving around

4. ls
    a. `ls` - list the contents of the director you’re in
    b. Can be run on other directories, like `ls Documents/`

5. man
    a. `man [command]` - look up the usage and meaning of *command*
    b. Useful for built-in commands you know the name of but don’t remember how to use perfectly
    c. If there’s no man page, try running the command with `[command] --help`

6. Structure of a command
    a. Example command: `ls -lh Documents/`
    b. Command: the thing you run (ls)

c. Flags: optional settings you can enable or disable (-l meaning show more information, -h meaning list file sizes in human-friendly formats, and -lh meaning both -l and -h)
d. Operands: the "arguments" of your command, or the specific targets of it (Documents/)

7. cd and paths

a. `cd [path]`: change directory to the *path* specified
b. absolute paths: start with the root directory (/) and work your way down

e.g. /Users/xanda/Documents/

c. based on homedir: where user-specific files go, and specified by '~'

e.g. ~/Documents/

d. relative paths with . or ..

e.g. if you start at Documents, ., or ../Downloads/

8. tab completion

a. if you've started typing a part of a path to a directory, hit tab once to avoid typing the rest.
b. Hitting tab once after typing ~/Doc should fill in ~/Documents.
c. If you hit tab once and nothing happens, hit tab twice in a row - you may have entered the prefix for multiple paths (e.g. ~/Do could be ~/Documents or ~/Downloads)

9. Regular expressions

a. if you want Unix to match all paths of a given format, you can use regular expressions
b. e.g. `ls ~/Do*` should run ls for both Documents and Downloads (and any other directory starting with D)
c. other regexps work, but stackoverflow is preferred to spending much time on them - they're written differently than python ones

10. mv and cp

a. `mv [path1] [path2]`: move the file at *path1* to *path2*. If *path2* is an existing directory, moves the file specified by *path1* into *path2*. Also used to rename files.
b. `cp [path1] [path2]`: make a copy of the thing from *path1* at new location path2. Again, if *path2* is an existing directory, moves *path1* to *path2*.
c. Both of these can work to move/copy directories, but to do so with `cp`, you should use the flag -R (recursive) to make sure everything in the directory copies over

11. rm

a. `rm [path]`: permanently, irrecoverably deletes the thing at *path*.

b. has several flags to watch out for:
    i. -r: recursively goes through specified directory deleting (-r is a common flag)
    ii. -f: deletes anything it has permission to delete, even if the system would rather you didn't
    iii. -i: prompts before deleting each file
    iv. -v: prints the list of files you delete (v is a common flag for being "verbose")
c. if anyone tells you to `rm -rf /` or `rm -rf *`, they are trying to make you do bad things

12. Interrupts

a. Sometimes, you run something bad and need to make it stop
b. `Ctrl-d` is computer-interrupt-speak for "Please stop"
c. `Ctrl-c` is computer-interrupt-speak for "STOP NOW"
d. If neither of these work, close the terminal window
e. If that still doesn't work, it will be worthwhile to learn about killing processes, but this is out of scope

Stream basics

13. the stream

a. UNIX uses "streams" to refer to the movement of textual or byte-wise information from one place to another. Streams include
    i. the output stream (also called "standard out" or stdout) printed to the terminal window
    ii. the input stream (also called "standard in" or stdin)
    iii. the error stream (also called "standard error" or stderr)
b. [command] > [file] : takes the text printed to the command line output stream by *command* and writes it to new file *file* (typically, whatever would print directly on your screen). This may overwrite an existing file.
c. [command] < [file]: takes the text from *file* and feeds it into *command* as input
d. [command1] | [command2] : takes the output from *command1* and feeds it in as input to *command2*
e. [command] >> [file]: like [command] > [file], but the output of *command* is appended to *file* instead of overwriting

14. cat

a. cat [file1] : prints out the entirety of *file1* to stdout
b. cat [file1] [file2] …: prints out files *file1*, *file2*, etc. in order to the screen, effectively con*cat*enating them

c. Used to get files into streams so you can do other fun stuff to them

15. echo

a. `echo [string]` repeats *string* back to you
b. Like the “print” statement of the UNIX world, except it doesn’t require quotes around the text
c. Gets known strings into streams so you can do fun stuff to them

Stream-friendly commands

(These can be called with either a file as an argument or as part of a set of streamed commands)

16. head and tail

a. `head` grabs the beginning lines of your input stream and prints it to the output stream
b. `head -n [number]` prints the first *number* lines of the file to the output stream
c. `tail` works the same way, but refers to the last lines of the file
d. `tail -f` will start by printing the end of the file and then wait for further lines to be written to the file. This won’t stop without an interrupt.

17. cut

a. `cut -f [number]` splits each line by tabs, takes the *number*th field after splitting, and prints that to the output stream.
b. Fields start at 1, so if you wanted just the text from the csvs we made in class with text in the third column, that would be done with `cut -f 3`
c. if your file has delimiters besides tabs, you can use -d ‘[other delimiter]’ to specify the correct delimiter
d. has some other uses and ways to specify ranges of fields - check out the man page

18. sort

a. `sort` puts the lines in lexicographic order
b. Other orders can be used, check out the man page

19. uniq

a. `uniq` filters lines to only those that are not direct copies of the line preceding them
b. This doesn’t filter out lines that appeared somewhere earlier in the file and is thus often paired with `sort`

c. `uniq -c` gives you tab-separated counts preceding each line of how many times it appeared in a row in that location

20. wc

a. `wc` count things across all lines of a file
b. `wc -l` counts lines, `wc -w` counts words
c. check out the man page for fun usage

Heavy-duty text tools

21. grep

a. Comes from an old and common ed command formatted `g/re/p` (**g**lobal / **r**egular **e**xpression / **p**rint)
b. `grep [text]` will return all the lines in the specified path
c. Useful flags:
    i. -v: return any lines that don't contain *text*
    ii. -c: just return the count of lines that match
    iii. -i: ignore case when searching
    iv. --color: highlights the matched string with a color, default red
d. If you're doing something complicated, you can use a regular expression with -G or -E; go look on StackOverflow for examples or particular types of searches
e. Alternately, you can chain together grep commands

grep 'cat' *.txt | grep -v 'dog' returns all lines in text files in your current directory with 'cat' and not 'dog' in them

22. sed

a. Name comes from **s**tream **ed**itor
b. In effect, a little programming language for describing transformations on a string
c. Most common one is replacement: `sed s/[something]/[something else]/` will replace the first instance of *something* in every line with *something else*.
d. More tips on sophisticated use can be found at Sed: An Introduction and Tutorial or on StackOverflow for particular operations

23. Chained commands

a. All these tools together can get you some crazy commands. Suppose I have a file like the State of the Union Mallet-friendly with the year as the second column. To find out how many years talk about "militia" from the 19th century, I can run
`grep militia sotu.txt | cut -f 2 | grep '^18' | sort | uniq | wc -l > count.txt`
which looks for lines with militia, takes out the year field, checks that it begins

with 18, sorts them lexicographically, deduplicates repeating years, and then counts the number of lines, saving that count as text in the file count.txt

Bonus material

24. nano

a. Basic text editor for use in the command line
b. Lists commands on the bottom of the screen, with ^ meaning the Control key
c. Most people using the command line a lot will eventually switch to vim or emacs, but they take a bit of time to learn

25. micro

a. Slightly nicer basic editor on the command line with similar commands to nano (e.g. ^Q = ctrl-Q = quit)
b. Alt-G summarizes editor tools, Ctrl-G opens more detailed help
c. Still definitely way friendlier than vim/emacs for learning curve
d. (thanks Prof. Bang for suggesting this!)

26. find

a. Searches for files in a directory of a certain name pattern
b. Useful if you're lost in a complicated directory structure
c. Check out man for how to specify search depth and patterns

27. which

a. `which [command]` tells you where the code for that command is currently
b. Useful if you're using multiple versions of Python in different places and want to figure out which installation your computer thinks you're using

28. alias

a. `alias [command]="[long command]"` tells your shell that while it's open, it should treat typing *command* as if you're typing *long command*.
b. Useful if you have lots of flags or long paths you need to use
    i. e.g. if you have a special version of python you installed that's not the default, you could do `alias mypython="/path/to/python"` and then run `mypython something.py` to use it
    ii. Alternately, if you have some weird directory you navigate to often but takes a while you get to, you can do `alias proj="cd /really/nasty/long/path"`
c. Aliases are only remembered by that shell - look up .bashrc and .profile files for information on how to make these always saved

29. ssh
    a. Used to access one machine from another securely - **s**ecure **sh**ell
    b. If you're on Windows, you can do this with PuTTy to get to UNIX machines
    c. Has corresponding commands for transmitting between machines (e.g. `scp` copies files securely between machines)