Introduction to Linux Shell

15-213/15-513/14-513:

Introduction to Computer Systems

Linux Shell

- The shell is a program that takes commands from the keyboard and gives them to the operating system to perform
- Computers “think” in text commands
  - Write commands using a “Command Line Interface” (CLI), often called a “terminal”
  - Say the basics of being in a directory, include slide

What is linux?

- On most Linux systems a program called bash acts as the shell
  - Other shell programs which include: sh, ksh, tcsh and zsh.

The Basics: Directories

Two commands commonly used to work with the current working directory:

- **`pwd`** – print working directory
  - This tells you what directory you are currently in
- **`cd`** - change directory
  - This lets you change into a different directory

Important Directory Names:

- `~` – the home directory
- **`~andrewid`** – the home directory of user “andrewid”
- `.` – the current directory
- `..` – the parent directory (the directory right above the current one)
- `/` – the root directory (the main directory that has no parent)

Manual pages (man pages)

- If you are ever unsure about a command one helpful resource is to utilize man pages
- **`$ man <command>`**
  - Gives information on what a command does and what options you can give it.
- You can search through a man page by typing: **`/thing_i_want_to_find`**
  - Advance from one match to the next by pressing **`n`**
- Most commands have a **`--help`** or **`-h`** option that will print out a help message

For more information about the man command, enter:

```
$ man man
```

Transferring files between machines

scp: a secure way to copy files between 2 machines

```
$ scp user@alpha.com:/somedir/somefile.txt user@beta.com:/anotherdir
```

Remote to Local `$ scp username@from_host:file.txt /local/directory/`

Local to Remote `$ scp file.txt username@to_host:/remote/directory/`

Remote to Remote `$ scp username@from_host:/remote/file.txt username@to_host:/remote/directory/`

Flags

- **-r**: recursive [useful for copying directories]
- **-v**: verbose mode [useful for debugging]
- **-q**: quiet [useful for when updated messages are not needed]

NOTE: If you are copying a file to a current directory, use **.** as the file path. If you are recursively copying a directory from your local machine, use **.** as the file path. **See slide about DIRECTORIES for more.**

Managing your files

Managing files: Moving, creating & deleting files

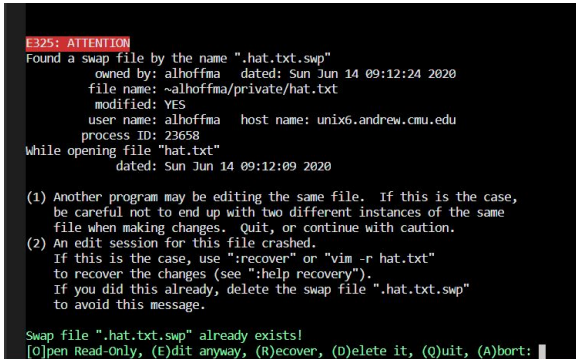
- `cp <source> <destination>` – copy files
- `mv <source> <destination>` - move and rename files
- `rm <filename>` - PERMANENTLY delete files
- `rmdir <filename>` - PERMANENTLY delete empty directory
- `mkdir <directory>` - make directories
- `touch <file>` - create an empty file
- **List the files in the current directory:**
  - `ls [path]` – listing files
  - `tree [path]` – recursively listing files

WHAT NOT TO DO

```
$ rm -rf /
$ rm -rf *
$ rm -rf .
$ mv /home/user/* /dev/null
```

- What Not to Do Part 1
- What Not to Do Part 2

Hidden & Temporary Files



- Hidden files begin with a . and are hidden unless you specify a command for -a (all)

Swap Files **applies to vim**

`malloc.c` **`.malloc.c.swp`**

- A copy of an old version of a file that was not properly saved
- Solution:
  - Delete swap file from command line

Open Read Only
Useful for when you only want to view contents

Edit anyway
Be careful! If the file is being edited in another vim session, you will have 2 versions

Recover
Useful for when you know the swap file contains the changes you want to recover

Delete it
Useful for when you no longer need the file

Quit
Useful to not edit the current file but want to keep other vim sessions open

Abort
Useful to close any open vim sessions

Tar

A way to archive files in 1 bundle (and compress them)

Flags
-c: create a tarball
-x: open a tarball
-z: zipped using gzip
-v: verbose mode [displays progress]
-f: specify file name

`tar -cf name-of-archive.tar /path/to/dir/` compress directory

`tar -cf name-of-archive.tar /path/to/filename` compress file

`tar -cf name-of-archive.tar dir1 dir2 dir3` compress multiple directories

`tar -xf name-of-archive.tar` open a tar file in current directory

This might be helpful for bootcamp labs!

Text Processing

`$ sort` sort lines of text file

`-d`: dictionary order `-f`: ignore case `-n`: numeric sort `-r`: reverse

`$ cat` concatenates inputs and prints on the screen

`$ uniq` reports or omits repeated lines

`$ head/tail` prints first (or last) lines of a file

`-bx`: print out first (or last) *x* bytes `-nx`: print out first (or last) *x* lines

Other helpful aspects of a shell

File Redirection

Syntax	Meaning
`command < file.txt`	Read the `stdin` of “command” from “file.txt”
`command > file.txt`	Send the `stdout` of “command” to “file.txt”, overwriting its contents
`command >> file.txt`	Append the `stdout` of “command” to the end of “file.txt”
`command 2> file.txt`	Send the `stderr` of “command” to “file.txt”, overwriting its contents
`command 2>> file.txt`	Append the `stderr` of “command” to the end of “file.txt”

Example*:

```
# 'hello.txt' doesn't exist, so it will be created
$ echo "Hello" > hello.txt
$ cat hello.txt
Hello
```

*more on echo at the end :)

Grep (Global Regular Expression Print)

grep searches for patterns in a file [if no file is provided, all files are recursively searched]

```
$ grep [OPTION...] PATTERNS [FILE...]
```

Standard Flags

-c: prints count of matching lines
-h: display matches without filenames
-i: ignores case for matching
-l: displays list of only filenames
-n: display matches and line numbers
-e exp: specificies expression with this option
-f file: takes pattern from file
-o: print only matching parts of lines
-r: read all files under each directory, recursively

Examples

```
$ grep “test” *
$ grep -r “test”
$ grep -rc “test”
```

Pipes (|)

- Pipes are a way to chain together the output from one command with the input to another.
  - To create a pipe, we use the Unix pipe character: `|`
- Lets use `grep` to find words in the computer's dictionary that contain the string "compute"
- ```
  $ grep compute /usr/share/dict/words
  >> compute
     computer
       ...
     Uncomputed
  # Pipe output of grep (on stdout) to the input of wc (on stdin)
  $ grep "compute" /usr/share/dict/words | wc -l
    >> 34  # Thus, 34 words have the word 'compute' in them
  ```
- Using pipes effectively can reduce some incredibly hard problems down to one line of code
  ```

Lab Time!

http://tinyurl.com/r76fb5nj

What is git?

- git ≠ GitHub
- Version control system
  - Better than:
    - copy pasting code
    - emailing the code to yourself
    - taking a picture of your code and texting it to yourself
    - zipping the code and messaging it to yourself on facebook
- **using git this semester will (with high probability) be mandatory!!!**

  style point deductions if you don't use it

Configuring git

```
$ git config --global user.name "<Your Name>"
```

```
$ git config --global user.email "<Your Email>"
```

```
$ git config --global push.default simple
```

(Make sure the email is your Andrew ID, and make sure to add that email to your GitHub account!)

To see the current set of configs: `$ git config --list`

Creating a Repository

- `$ git init` turn the current folder into a new repository.

OR

- `$ git clone` initialize a repository locally from a remote server

For example, in ~/private/15213:

```
$ mkdir datalab

$ cd datalab

$ git init
```

Core Gameplay Loop

1. `$ git add` Stages files to be committed. Flags: --a (all files), -u (only previously added files) Can also add specific files by listing them after ‘add’.
1. `$ git commit -m “<MESSAGE>”` Commit the changes in the staged files. Write descriptive, meaningful messages for future you!

If repository is connected to remote server (like GitHub):

1. `$ git push` Push changes to a remote server.

If working with others on remote server:

0. `$ git pull` Pull changes from a server

Other Important Commands

- `$ git status` shows key information such as current branch, “add”ed files, “commit”ed files not yet pushed.
- `$ git log` show commit history. Can use --decorate --graph --all to make it pretty.

- `$ git diff` shows the changes you’ve made
- `$ git rm` stages files to be removed.
- `$ git reset HEAD <FILE_NAME>` unstages “FILE_NAME” from the commit

Documentation for all commands: Git - Documentation (git-scm.com)

Chronomancy (The Greatest Magic of All!)

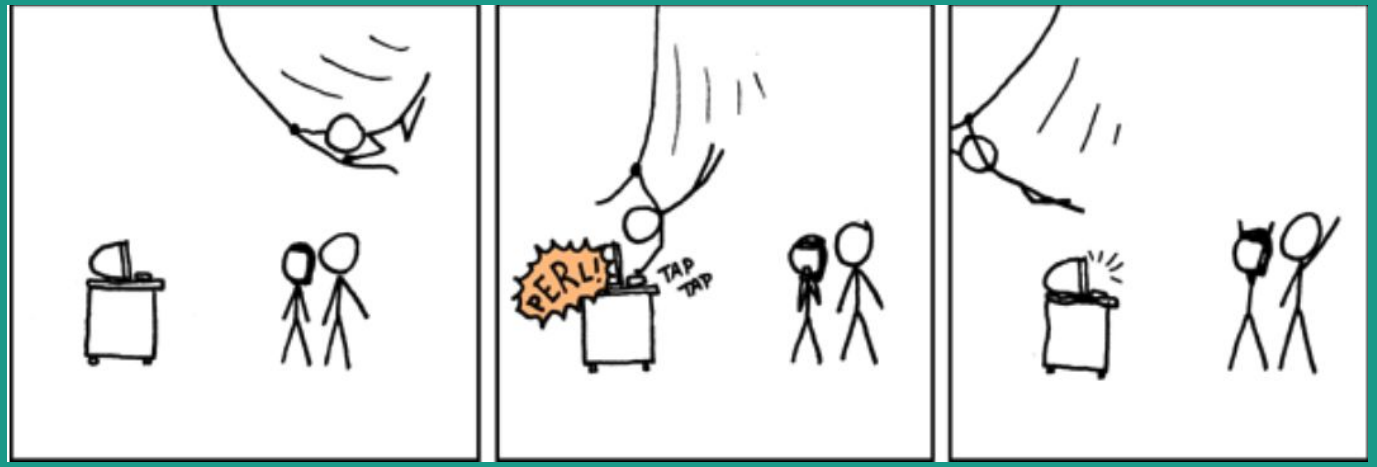
A Time Lord’s toolkit:

- **`$ git revert <COMMIT_HASH>`**
  - Creates a new commit where everything is the same as the commit with COMMIT_HASH
- **`$ git checkout <FILE_NAME>`**
  - Used to reset any changes made to a file to previous commit.
- **`$ git reset --hard <COMMIT_HASH>`**
  - Sets you back to COMMIT_HASH.
  - Commits between COMMIT_HASH and present time will disappear.
- Many more ways to do similar and different things, all in Git documentation.

Time Magic is Dangerous!

- Uncommitted changes will be destroyed.
- Often cannot be undone
- `$ git reset --hard`
  - Going back multiple commits will rewrite history and make crew members time-sick.
  - Last resort that should only be used in private repos. Revert is safer.

Helpful Tips and Tricks



The elusive *power programmer*

Permissions

- `fs la`: to understand permissions for a directory
- Permissions:
  - **r** read
  - **l** list files and see basic information
  - **i** create new files
  - **d** delete files
  - **w** edit existing files
  - **k** “lock” files so no one can edit them at the same time
  - **a** admin
- `fs sa <directory> <user> <permission>`: how to set permissions on a file or directory

```
$ fs sa foo acarnegie rldwik
```

Recovering Lost Files

Oh no, I've deleted all of my files!

- Plan A: Git version control (revert old commit) or Github
- Plan B: CMU keeps a nightly snapshot of files in ~/OldFiles
- Plan C: Run the following command to create ~/OldFiles from backup:

```
$ cd ~
$ fs mkmount OldFiles user.ANDREW_ID_HERE.backup
```

Wildcards

- *: Matches any characters
- ?: Matches any single character
- [characters]*: Matches any character that is a member of the set *characters*.
- [!characters]: Matches any character that is NOT a member of the set *characters*.
- More info: Wildcards (tldp.org)

`rm g*`	remove all files starting with “g”
`ls b*.txt`	list all files that begin with “b” and end with “.txt”
`cat Data???`	cat any file beginning with Data and has exactly 3 more characters
`tree [abc]*`	tree any files that begin with “a” “b” or “c”
`[[:upper:]]*`	references any file beginning with an upper case character
`*[![:lower:]]`	references any file that does not end with a lowercase letter

*this also works with any POSIX character classes!

Processes

- A process is an instance of a running program.
  - Not the same as “program” or “processor”
- A process is a currently executing command (or program), sometimes referred to as a job.
- At any given time there may be a couple hundred or less processes running.
- If you’re running Linux or a Unix based machine you can run a number of different commands:
  - `ps aux` // this will display a list of processes
  - `top` // detailed information about all processes, threads, and more

Foreground vs Background Jobs

A job is a process that is currently running or has been stopped or terminated.

- Unique job ID (JID) to each job, and can be run in the **foreground** or the **background**
  - Foreground job: a job that occupies the terminal until it is completed
  - Background job: a job that executes in the background and does not occupy the terminal
    - A background job can be run by writing a & at the end of the line
- The shell can only handle 1 foreground job and many background jobs at the same time

Commands related to jobs

- `jobs`: lists the state of all jobs
- `fg %n`: brings current or specified job in foreground; n is JID
- `bg %n`: places current or specified job in background; n is JID
- `CTRL + z`: stops foreground job and places it in the background as a stopped job [this job can be restarted later]
- `CTRL + c`: sends SIGINT to a foreground job and usually causes it to exit [it can never be restarted]

Echo & sed commands

- **`echo`** is used to display line of text/string that are passed as an argument
- `$ echo ls -l | sh`
  - #Passes the output of "`echo ls -l`" to the shell, with the same result as a simple **`ls -l`**

- **`sed`** can do insertion, deletion, search and replace (substitution)
- `$ sed 's/old_word/new_word/' file.txt`
  - `s` → substitution
  - Substitutes '`old_word`' with '`new_word`' in file.txt

Sources

https://www.tldp.org/LDP/abs/html/textproc.html

http://linuxcommand.org/lc3_lts0010.php

https://www.cs.cmu.edu/~15131/

https://www.cs.cmu.edu/~213

https://haydenjames.io/linux-securely-copy-files-using-scp/

Feedback:

https://forms.gle/w5yKdJ6BgN3nk42TA