03 - Manipulating Files and Using Git

CS 2043: Unix Tools and Scripting, Spring 2017 [1]

Stephen McDowell

January 30th, 2017

Cornell University

Table of contents

Some Logistics

- Last day to add is Wednesday 2/1.
- On moving forward independently, and using **sudo**.
    - I strongly advise taking a *snapshot* of your VM.

Working with Files

Users and Groups

Like most OS's, Unix allows multiple people to use the same machine at once. The question: who has access to what?

- Access to files depends on the users' account.
- All accounts are presided over by the Superuser, or `root` account.
- Each user has absolute control over any files they own, which can only be superseded by `root`.
- Files can also be owned by a `group`, allowing more users to have access.

File Ownership

- You can discern who owns a file many ways, the most immediate being `ls -l`

Permissions with `ls`

```
$ ls -l Makefile
-rw-rw-r--. 1 sven users 4.9K Jan 31 04:42 Makefile
              sven       # the user
                   users # the group
```

- The third column is the *user*, and the fourth column is the *group*.

What is this RWX Nonsense?

- R = read, W = write, X = execute.
- rwxrwxrwx
  - User permissions.
  - Group permissions.
  - Other permissions (a.k.a. neither the owner, nor a member of the group).
- Directory permissions begin with a **d** instead of a **-**.

An example

What would the permissions `-rwxr-----` mean?

- It is a file.
- User can read and write to the file, as well as execute it.
- Group members are allowed to read the file, but cannot write to or execute.
- Other cannot do *anything* with it.

Changing Permissions

Change Mode

`chmod <mode> <file>`

- Changes file / directory permissions to **<mode>**.
- The format of **<mode>** is a combination of three fields:
  - Who is affected: a combination of **u**, **g**, **o**, or **a** (all).
  - Use a **+** to add permissions, and a **-** to remove.
  - Specify type of permission: any combination of **r**, **w**, **x**.
- Or you can specify mode in octal: user, then group, then other.
  - e.g. **777** means user=7, group=7, other=7 permissions.

The octal version can be confusing, but will save you time.
Excellent resource in [2].

Changing Ownership

Changing the group

Change Group

`chgrp group <file>`

- Changes the group ownership of `<file>` to `group`.

As the super user, you can change who owns a file:

Change Ownership

`chown user:group <file>`

- Changes the ownership of `<file>`.
- The `group` is optional.
- The `-R` flag is useful for recursively modifying everything in a directory.

File Ownership, Alternate

If you are like me, you often forget which column is which in `ls -l`...

Status of a file or filesystem

`stat [opts] <filename>`

- Gives you a wealth of information, generally more than you will every actually need.
- **`Uid`** is the user, **`Gid`** is the group.
  - BSD/OSX: use **`stat -x`** for standard display of this command.
- Can be useful if you want to mimic file permissions you don't know.
  - Human readable: **`--format=%A`**, e.g. **`-rw-rw-r--`**
    - BSD/OSX: **`-f %Sp`** is used instead.
  - Octal: **`--format=%a`** (great for **`chmod`**), e.g. **`664`**
    - BSD/OSX: **`-f %A`** is used instead.

Platform Notes

- Convenience flag for `chown` and `chmod` on non-BSD Unix:

```
$ chmod --reference=<src> <dest>
```

- Set the permissions of `dest` to the permissions of `src`!
- BSD/OSX users: `--reference` does not exist, you will have to execute two commands.

```
$ chmod $(stat -f %A <src>) <dest>
```

- The command inside of `$(...)` gets evaluated *before* `chmod`.
  - You may see backticks: `` `stat -f %A <src>` ``, this is the `old` way, and is no longer supported.
- The `stat` command performs a little differently on BSD/OSX by default. Read the `man` page.

Types of Files and Usages

Plain Files

Plain text files are human-readable, and are usually used for things like:

- Documentation,
- Application settings,
- Source code,
- Logs, and
- Anything you may want to read via the terminal (e.g. README.txt).

Binary Files

Binary files are not human-readable. They are written in the language your computer prefers.

- Executables,
- Libraries,
- Media files,
- Archives (.zip, etc), and many more.

Reading Files Without Opening

Concatenate

`cat <filename>`

- Prints the contents of the file to the terminal window

`cat <file1> <file2>`

- Prints `file1` first, then `file2`.

more

`more <filename>`

- Scroll through one page at a time.
- Program exits when end is reached.

less

`less <filename>`

- Scroll pages or lines (mouse wheel, space bar, and arrows).
- Program does not exit when end is reached.

Beginning and End

Long files can be a pain with the previous tools.

Head and Tail of Input

```
head -[numlines] <filename>
tail -[numlines] <filename>
```

- Prints the first / last numlines of the file.
- Default is 10 lines.

Not Really a File...YET

You can talk to yourself in the terminal too!

Echo

`echo <text>`

- Prints the input string to the standard output (the terminal).
- We will soon learn how to use **echo** to put things into files, append to files, etc.

Let’s Git Started

What is `git`?

- `git` is a *decentralized* version control system.
- Ever used ”track changes” for a word document? It’s basically the same thing.
- Except for exceptionally more advanced, and you don’t have to pay for it.
- Basically, it enables you to save changes as you go to your code.
  - As you make these changes, if at any point in time you discover your code is ”broken”, you can *revert* back in time!
  - Of course, if you haven’t been ”saving” frequently, you have less to work with.
  - Mantra: *commit* early and often.

`git` Terminology

- The ”document” is called a *repository* (*repo*).
  - The initial download is called *clone*.
- The location where files are being stored on the server is the *remote*.
- We’ll refer to the copies on your computer as the *local*, or sometimes *client*.
- The act of ”saving” is *commit*.
  - Just because you saved it *locally* doesn’t mean anything for the *remote*.
  - To publish changes to the *remote*, you *push*.
- When the version you have is different than what is online, this can produce a *conflict* - if `git` cannot figure out what to do, it will tell you.
- To acquire updates from the *remote*, you need to *pull*.

On the Board

What does it actually look like?

Teaser: Example Scenario

- Suppose you (A), and your best friend B are working in the same repo.
- You both `clone` the repository at the same time, and both make different changes to the same file.
- B hacks your internet and takes you offline, and `push`es their changes to the `remote`.
- You get internet back, and go to `push`. What happens?
  - The `remote` will reject your `push`, and force you to merge in the changes from B first.
- Basically, `git` can get complicated quickly.
- HOWEVER! You **must** work independently in this class, so you won't have nearly as many problems ;)

Demo Time!

Our first in class demo

- Ok, lets not get too carried away with `git`.
- The first thing you'll want to do is learn how to download a `repo`.

```
$ git clone https://github.com/cs2043-sp17/lecture-demos.git
```

- ... lets walk through the demo ...
- Hey a solution! To get it now:

```
$ git pull
```

References I

[1] B. Abrahao, H. Abu-Libdeh, N. Savva, D. Slater, and others over the years.
Previous cornell cs 2043 course slides.

[2] C. Hope.
Linux and unix chmod command help and examples.
`http://www.computerhope.com/unix/uchmod.htm`, 2016.