03 – UNIX Permissions and find

CS 2043: Unix Tools and Scripting, Spring 2019 [2]  

Matthew Milano  
January 28, 2019  
Cornell University
1. As always: Everybody! ssh to wash.cs.cornell.edu

2. Quiz time! Everybody! run quiz-01-28-19

3. So, if a special user owns printing stuff and serial ports…
   How to I print?

4. Types of Files and Usages

5. Flags & Command Clarification
As always: Everybody! ssh to wash.cs.cornell.edu
Quiz time! Everybody! run quiz-01-28-19
You and 188 of your closest friends

- Welcome back to WASH!
- Wash is a *multi-user* machine
  - there are multiple users on here
  - how to solve conflicts?
- You’re going to be working on wash..
- Security is *kinda* important
• There are 217 users on wash.
• Even on a personal Linux computer, there are usually at least 30 different user accounts.
• Why so many users?
  • Security
  • Separation of concerns
  • Principle of least privilege
• **Example:** The `lp` user owns printing-related files.
So, the security model is...

- *NIX security model is access control-based
- Define who is *allowed* to use what resources
- What do users control?
  - file ownership and permissions
  - processes
- Most things are [represented by] a file
- **EXAMPLE**: The file `/dev/ttySO` represents the serial port
  - early USB predecessor
  - ...what, I’m not *that* old.
So, if a special user owns printing stuff and serial ports... How to I print?
Groups

- Users can belong to [lots of] groups

**List groups to which a user belongs**

```
groups [user name]
```

- Lists groups to which [argument] belongs.
- With no argument, lists your groups

- All files are owned by both a **user** and a **group**
- Groups grant **permissions** on certain files and actions
  
  - **Example**: the `lp` group allows printing
  - **Example**: the `uucp` group allows serial port access
Let’s see what groups we’re in!

#### Groups with groups

```
$ groups
mpm288 cs2043student student_only
```

- the **netID** group is just for you (you’re the only one in it)
- the **cs2043student** group is for the entire class — everyone is in it!
- the **student_only** group is for security; defines “student” as your *maximum privilege*
  - not getting into what that means this lecture ;)


File Ownership

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

<table>
<thead>
<tr>
<th>Permissions with <code>ls</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$ ls -l README</code></td>
</tr>
<tr>
<td><code>-rwxrw---- 1 milano cs2043tas 20 Jan 26 15:48 README</code></td>
</tr>
<tr>
<td># milano &lt;-- the user</td>
</tr>
<tr>
<td># cs2043tas &lt;-- the group</td>
</tr>
</tbody>
</table>

- Third column is the *user*, fourth column is the *group*.
- Other columns are the *link count* and *size*.
  - we’ll talk about like count in .... 5 lectures?
What is this RWX Nonsense?

- \( r = \) read, \( w = \) write, \( x = \) execute.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>User permissions</td>
<td>User permissions</td>
<td>User permissions</td>
</tr>
<tr>
<td>Group permissions</td>
<td>Group permissions</td>
<td>Group permissions</td>
</tr>
<tr>
<td>Other permissions</td>
<td>Other permissions</td>
<td>Other permissions</td>
</tr>
</tbody>
</table>

- Directory permissions begin with a \( d \) instead of a -
- Other: “neither the owner, nor a member of the group”.
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 can read the file
  - Group members **cannot** write to or execute the file.
  - Other cannot do *anything* with it.
Hold on... *execute* the file?

- Programs are just files!
- Most programs contain a special format of *binary data*, called ELF
- Some programs are *scripts*, which means they’re just text!
- try to **cat** out the **handin** command, for example
  - (the path to it is: `/course/cs2043/bin/handin`)
Ok but... execute the directory?

- This one is a bit counter-intuitive
- **Reading** a directory means listing its contents
- **Writing** a directory means removing or creating
- **Executing** a directory means *interacting* with its contents
  - editing, printing, etc.
  - `drw------` means you can create, remove, and list contents, but *cannot* print, edit, or execute anything *inside* the directory
  - `d-wx------` means you can create, remove, and use contents, but *cannot* list them.
- Execute without read means you have to know the name of the contents in order to access them!
  - no other way to discover the contents.
  - kinda like a password...
Changing Permissions

**Change Mode**

`chmod <mode> <file>`

- Changes file or 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`.

  ```
  # Add read, write, & execute for user, group, & other
  $ chmod ugo+rwx <file> # or chmod a+rwx <file>
  # Remove read and write for other
  $ chmod o-rw <file>
  ```

- Can specify mode in octal: user, then group, then other.
  - E.g., **750** means `user=7`, `group=5`, `other=0` permissions.
The Octal Version of `chmod`

- For the formula hungry, you can represent `r`, `w`, and `x` as binary variables (where 0 is off, and 1 is on). Then the formula for the modes is

  \[ r \cdot 2^2 + w \cdot 2^1 + x \cdot 2^0 \]

- **Octal Ownership Permissions**

- **Examples**
  - `chmod 755`: `rwxr-xr-x`
  - `chmod 777`: `rwxrwxrwx`
  - `chmod 600`: `rw-----`

- If that makes less sense to you, feel free to ignore it.
  - Just use the `stat` command to help you convert :)

- The octal version can be confusing, but will save you time.

Excellent resource in [1].
Changing Ownership

- Changing the *group* of a file / directory

**Change Group**

*chgrp group <file>*

- Changes the group ownership of `<file>` to *group*.
- The `-R` flag will recursively change permissions of a directory.

- Changing the *owner* of a file / directory

**Change Owner and Group**

*chown user:group <file>*

- Changes the ownership of `<file>`.
- The *group* is optional (*chown user <file>*).
- The `-R` flag will recursively change permissions of a directory.
The root user

- The special user **root** is the ultimate administrator on the system
- Gets the permissions of *any* user on the system
  - *if anyone can read/write/exec it, root can too*
- can change permissions any way it wants
  - *can even set an owner/group combo where the owner’s not in the group!*
- can become root with the **su** or **sudo** commands
  - *we won’t be using these in this class...
  - you should never use **su** or **sudo** on wash!*
• You will likely forget which column is which in `ls -l` ...

### Status of a File or Filesystem

```
stat [opts] <filename>
```

- Gives you a wealth of useful information.
- **Uid** (%U) is the user, **Gid** (%G) is the group.
  - BSD/OSX: `stat -x <filename>` for “standard” behavior.
- Can be useful 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.
Types of Files and Usages
Plain Files

Plain text files are human-readable, used for things such as:

- Documentation,
- Application settings,
- Source code,
- Logs, and
- Anything you may want to read via the terminal
  - README
  - INSTALL
  - etc.
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.
Special Files

- Special Files represent things which ought not be files!
  - Sockets (connections)
  - Devices (hard disk, keyboard, etc)
  - Raw Memory (RAM)
  - The (software-emulated) terminal you’re using now!
  - A lot of really random other stuff

- The UNIX philosophy: represent *everything you possibly can* as a file
Default Permissions on Creation

**User Mask**

`umask <mode>`
- Remove `mode` from the file’s permissions.
- Similar syntax to `chmod`:
  - `umask 077`: `+rwx` for owner, `-` for all others.
  - `umask g+w`: enables group write permissions.
- `umask -S`: display the current mask.
- Just a bit mask with `0o777` and your `mode`.

<table>
<thead>
<tr>
<th>Full permissions</th>
<th>0o777</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample User Mask</td>
<td>0o002</td>
</tr>
<tr>
<td>Logical &amp; Gives</td>
<td>0o002</td>
</tr>
</tbody>
</table>

- Changing `umask` only applies for the remainder of the session.
  - Permanent if put in your `~/.bashrc` or `~/.bash_profile`. 
• Using your terminal to examine a file is very convenient!

File Perusal Filter for (crt) Viewing

more <filename>
- Scroll through one page at a time.
- Program exits when end is reached.

As the saying goes...

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.

<table>
<thead>
<tr>
<th>Print the Beginning (head) or End (tail) of a File</th>
</tr>
</thead>
<tbody>
<tr>
<td>head -[numlines] &lt;filename&gt;</td>
</tr>
<tr>
<td>tail -[numlines] &lt;filename&gt;</td>
</tr>
</tbody>
</table>

- Prints the first / last `numlines` of the file.
- First 5 lines: `head -5 file.txt` or `head -n5 file.txt`
- Last 5 lines: `tail -5 file.txt` or `tail -n5 file.txt`
- Default is 10 lines.
You can talk to yourself in the terminal too!

**Display a Line of Text**

**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.
- Show off to your friends how cool you are:

  $ echo 'I can have a conversation with my computer!'
  $ echo 'But it always copies me. RUDE.'
Flags & Command Clarification
Most commands take flags and optional arguments. These come in two general forms:

- Switches (no argument required), and
- Argument specifiers (for lack of a better name).

When specifying flags for a given command, keep in mind:

- Flags modify the behavior of the command / how it executes.
- Some flags take precedence over others, and some flags you specify can implicitly pass additional flags to the command.

There is no absolute rule here: research the command.
• A flag that is
  • One letter is specified with a single dash (-a).
  • More than one letter is specified with two dashes (--all).
  • The reason is because of how switches can be combined.

• We generally use “flag” and “switch” interchangeably:
  • “flag” the command, telling it that “action X” should occur
  • specify to the command to “switch on/off action X”
- **Switches** take no arguments, and can be specified in a couple of different ways.
- Switches are usually one letter, and multiple letter switches usually have a one letter alias.
- One option:
  - `ls -a`
  - `ls --all`
- Two options:
  - `ls -l -Q`
  - `ls -lQ`
- *Usually* applied from left to right in terms of operator precedence, but not always:
  - This is up to the developer of the tool.
  - Prompts: `rm -fi <file>`
  - Does **not** prompt: `rm -if <file>`
• The --argument="value" format, where the = and quotes are needed if value is more than one word.
  • Yes: ls --hide="Desktop" ~/
  • Yes: ls --hide=Desktop ~/
    • One word, no quotes necessary
  • No: ls --hide = "Desktop" ~/
    • Spaces by the = will be misinterpreted
    • It used = as the argument to hide

• The --argument value format (space after the argument).
  • Quote rules same as above.
  • ls --hide "Desktop" ~/
  • ls --hide Desktop ~/

• Usually, --argument value and --argument=value are interchangeable.
  • Not always!
• Generally, always specify the flags before the arguments.
• `ls -l ~/Desktop/` and `ls ~/Desktop/ -l` both work.
  • Sometimes flags after arguments get ignored.
  • Depends both on the command, and the flag(s).
• The special sequence `--` signals the end of the options.
  • Executes as expected: `ls -l -a ~/Desktop/`
  • Only uses `-l: ls -l -- -a ~/Desktop/`
    • "ls: cannot access -a: No such file or directory"
    • The `-a` was treated as an argument, and there is no `-a` directory (for me)
• In this example:
  • `-l` and `-a` are the flags.
  • `~/Desktop/` is the argument.
The special sequence -- that signals the end of the options is often most useful if you need to do something special.

Suppose I wanted to make the folder -a on my Desktop.

```
$ cd ~/Desktop # for demonstration purpose
$ mkdir -a       # fails: invalid option -- 'a'
$ mkdir -- -a    # success! (ls to confirm)
$ rmdir -a       # fails: invalid option -- 'a'
$ rmdir -- -a    # success! (ls to confirm)
```

This trick can be useful in many scenarios, and generally arises when you need to work with special characters of some sort.
Your new best friend

- How do I know what the flags / options for all of these commands are?

The Manual Command

```
man command_name
```

- Loads the manual (manpage) for the specified command.
- Unlike google, manpages are system-specific.
- Usually very comprehensive. Sometimes too comprehensive.
- Type `/keyword` to search for `keyword`, and hit `<enter>`.
- The `n` key jumps to the next search result.

- Search example on next page if that was confusing. Intended for side-by-side follow-along.
The **man** command is really useful!

$ man man # you now have the manual loaded
$ /useful # type /useful, then hit enter

#### [[[ first result highlighted ]]]

$ n # followed by enter

#### [[[ next result highlighted ]]]

# The default 'pager' is `less`, type `q`
# without backticks to exit.

- Subtle differences depending on distribution, e.g. **ls** `-B`
  - BSD/OSX: Force printing of non-printable characters in file names as \*xxx\*.
    - \*xxx\* is the numeric value of the character in **octal**.
  - GNU (Fedora, Ubuntu): don’t list implied entries ending with ~
    - Files ending with ~ are *temporary* backup files that certain programs generate (e.g. some text-editors, your OS).