
CSE 391

Lecture 4

Persistent shell settings; users/groups; permissions

slides created by Marty Stepp, modified by Jessica Miller and Ruth Anderson

<http://www.cs.washington.edu/391/>

Lecture summary

- Persistent settings for your bash shell
- User accounts and groups
- File permissions
- The Super User
- Useful Tidbit: tar files

.bash_profile and .bashrc

- Every time you log in to bash (e.g. ssh attu), the commands in `~/.bash_profile` are run
 - you can put any common startup commands you want into this file
 - useful for setting up aliases and other settings for *remote login*
- Every time you launch a non-login bash terminal (e.g. bash), the commands in `~/.bashrc` are run
 - useful for setting up persistent commands for *local shell usage*, or when *launching multiple shells*
 - *Do not put things that would produce output in .bashrc (e.g. echo)*
 - often, `.bash_profile` is configured to also run `.bashrc`, but not always

Note: a dot (.) in front of a filename indicates a normally hidden file, use `ls -a` to see

Exercise: Edit your .bashrc

- *For 16au: You will need to change the permissions on these two files on the CSE VM using this command:*

```
sudo chown username:username .bashrc .bash_profile
```

- *Use whatever your username is on your VM*

- *Exercise : Make it so that our attu alias from earlier becomes persistent, so that it will work every time we run a shell.*
- *Exercise : Make it so that whenever you try to delete or overwrite a file during a move/copy, you will be prompted for confirmation first.*

Making Changes Visible

- After editing your `.bashrc` or `.bash_profile`, how do you make the aliases etc. in the file take effect?
 - `.bash_profile`
 - log on again (e.g `ssh attu`), or
 - `bash -l` (el not one) will start a login shell, or
 - `source .bash_profile`
 - `.bashrc`
 - start another bash shell (type: `bash`), or
 - `source .bashrc`

.plan

- Another fun settings file
- Stored in your home directory
- Contains information you'd like others to be able to see
 - is displayed when the **finger** protocol is run
- *Exercise:* create a quick .plan file, and make sure it works with **finger**

Users

Unix/Linux is a multi-user operating system.

- Every program/process is run by a user.
- Every file is owned by a user.
- Every user has a unique integer ID number (UID).
- Different users have different access permissions, allowing user to:
 - read or write a given file
 - browse the contents of a directory
 - execute a particular program
 - install new software on the system
 - change global system settings
 - ...

People & Permissions

- **People:** each user fits into only one of three permission sets:
 - owner (u) – if you create the file you are the owner, the owner can also be changed (using `chown`)
 - group (g) – by default a group (e.g. `ugrad_cs`, `fac_cs`) is associated with each file
 - others (o) – everyone other than the owner and people who are in the particular group associated with the file

You are in the most restrictive set that applies to you – e.g. if you are the owner, those permissions apply to you.

- **Permissions:** For regular files, permissions work as follows:
 - read (r) – allows file to be open and read
 - write (w) – allows contents of file to be modified or truncated
 - execute (x) – allows the file to be executed (use for executables or scripts)

* Directories also have permissions (covered later). Permission to delete or rename a file is controlled by the permission of its parent directory.

Groups

| command | description |
|---------|---|
| groups | list the groups to which a user belongs |
| chgrp | change the group associated with a file |

- **group:** A collection of users, used as a target of permissions.
 - a group can be given access to a file or resource
 - a user can belong to many groups
 - see who's in a group using `grep <groupname> /etc/group`
- Every file has an associated group.
 - the owner of a file can grant permissions to the group
- Every group has a unique integer ID number (GID).
- *Exercise:* create a file, see its default group, and change it

File permissions

| command | description |
|---------|---------------------------------------|
| chmod | change permissions for a file |
| umask | set default permissions for new files |

- *types* : read (r), write (w), execute (x)
- *people* : owner (u), group (g), others (o)
- on Windows, .exe files are executable programs;
on Linux, any file with x permission can be executed
- permissions are shown when you type `ls -l`

is it a directory?

owner (u)
group (g)
others (o)

↓ ↓ ↓ ↓

drwxrwxrwx

File permissions Examples

Permissions are shown when you type `ls -l`:

```
-rw-r--r-- 1 rea fac_cs      55 Oct 25 12:02 temp1.txt
-rw--w---- 1 rea orca       235 Oct 25 11:06 temp2.txt
```

temp1.txt:

- **owner** of the file (rea) has read & write permission
- **group** (fac_cs) members have read permission
- **others** have read permission

temp2.txt:

- **owner** of the file (rea) has read & write permission
- **group** (orca) members have write permission (but no read permission – can add things to the file but cannot cat it)
- **others** have no permissions (cannot read or write)

Changing permissions

- letter codes: `chmod who(+-)what filename`

`chmod u+rw myfile.txt` (allow owner to read/write)

`chmod +x banner` (allow everyone to execute)

`chmod ug+rw,o-rwx grades.xls` (owner/group can read and

note: -R for recursive write; others nothing)

Note, no space after the comma!

- octal (base-8) codes: `chmod NNN filename`

- three numbers between 0-7, for owner (u), group (g), and others (o)

- each gets +4 to allow read, +2 for write, and +1 for execute

`chmod 600 myfile.txt` (owner can read/write (rw))

`chmod 664 grades.dat` (owner rw; group rw; other r)

`chmod 751 banner` (owner rwx; group rx; other x)

chmod and umask

`chmod u+rw myfile.txt` (allow owner to read/write)

Note: leaves “group” and “other” permissions as they were.

`chmod 664 grades.dat` (owner rw; group rw; other r)

Note: sets permissions for “owner”, “group” and “other” all at once.

`umask` – returns the “mask” in use, determines the default permissions set on files and directories I create. Can also be used to set that mask.

```
% umask
```

```
0022 ←
```

```
% touch silly.txt
```

```
% ls -l silly.txt
```

```
-rw-r--r-- 1 rea fac_cs 0 Oct 25 12:04 silly.txt
```

0022 means that files I create will have group and other “write bits” turned off:

- 1) Take the bitwise complement of $022_8 \rightarrow 755_8$
- 2) AND with 666_8 for files (777_8 for directories) : $755_8 = 111\ 101\ 101$
 $666_8 = \underline{110\ 110\ 110}$
 $\quad\quad 110\ 100\ 100 = 644_8$
(owner rw, group r, other r)

Exercises

- Change the permissions on `myfile.txt` so that:
 - Others cannot read it.
 - Group members can execute it.
 - Others cannot read or write it.
 - Group members & Others can read and write it.
 - Everyone has full access.

- Now try this:
 - Deny all access from everyone.
 - !!! is it dead?

Exercises (Solutions)

- Change the permissions on `myfile.txt` so that:
 - Others cannot read it. `chmod o-r myfile.txt`
 - Group members can execute it. `chmod g+x myfile.txt`
 - Others cannot read or write it. `chmod o-rw myfile.txt`
 - Group members & Others can read and write it. `chmod go+rw myfile.txt`
 - Everyone has full access. `chmod ugo+rwx myfile.txt`

- Now try this:
 - Deny all access from everyone. `chmod ugo-rwx myfile.txt`
 - !!! is it dead?
 - I own this file. Can I change the Owner's (u) permissions?

Directory Permissions

- Read, write, execute a directory?
 - **Read** - permitted to read the contents of directory (view files and sub-directories in that directory, run `ls` on the directory)
 - **Write** - permitted to write in to the directory (add, delete, or rename & create files and sub-directories in that directory)
 - **Execute** - permitted to enter into that directory (`cd` into that directory)
- It is possible to have any combination of these permissions:

Try these:

- Have **read** permission for a directory, but NOT **execute** permission
 - ????
- Have **execute** permission for a directory, but NOT **read** permission
 - ???

***Note:** permissions assigned to a directory **are not inherited** by the files within that directory

Directory Permissions

- Read, write, execute a directory?
 - **Read** - permitted to read the contents of directory (view files and sub-directories in that directory, run `ls` on the directory)
 - **Write** - permitted to write in to the directory (add, delete, or rename & create files and sub-directories in that directory)
 - **Execute** - permitted to enter into that directory (`cd` into that directory)
- It is possible to have any combination of these permissions:
 - Have **read** permission for a directory, but NOT **execute** permission
 - Can do an `ls` from outside of the directory but cannot `cd` into it, cannot access files in the directory
 - Have **execute** permission for a directory, but NOT **read** permission
 - Can `cd` into the directory, can access files in that directory if you already know their name, but cannot do an `ls` of the directory

***Note:** permissions assigned to a directory **are not inherited** by the files within that directory

Permissions don't travel

- Note in the previous examples that permissions are separate from the file
 - If I disable read access to a file, I can still look at its permissions
 - If I upload a file to a directory, its permissions will be the same as if I created a new file locally
- Takeaway: permissions, users, and groups reside on the particular machine you're working on. If you email a file or throw it on a thumbdrive, no permissions information is attached.
 - Why? Is this a gaping security hole?

Careful with -R

- Say I have a directory structure, with lots of .txt files scattered
 - I want to remove all permissions for Others on all of the text files
 - First attempt:
 - `chmod -R o-rwx *.txt`
 - What happened?

This command will:

- change permissions on all the files that end with .txt in the current directory, AND
- it will recursively change the permissions on any files that end with .txt in **directories whose name end in .txt**
(you probably do not have any directories whose names end that way!)
- This is not really recursive in the way you meant it to be! (see next slide...)

Careful with `-R` (fix)

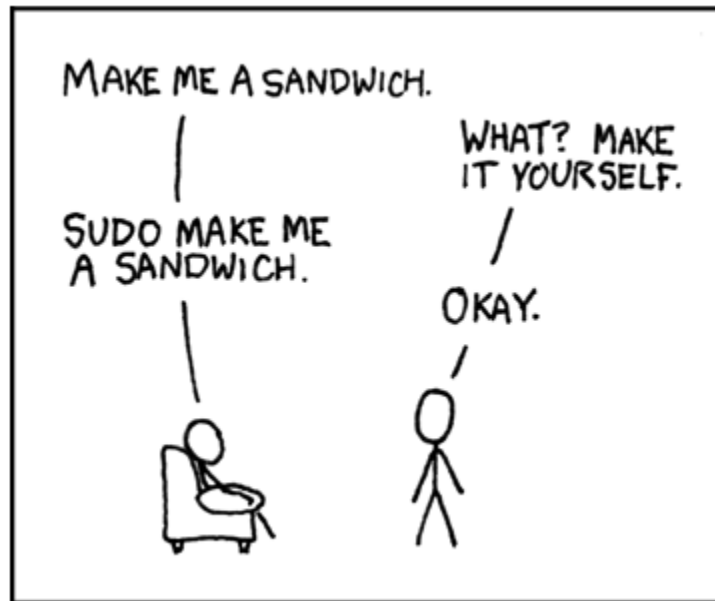
- Say I have a directory structure, with lots of `.txt` files scattered
 - I want to remove all permissions for Others on all of the text files
 - First attempt:
 - `chmod -R o-rwx *.txt`
 - What happened?
 - Try and fix this using `find` and `xargs`!
 - `find -name "*.txt"`
 - `find -name "*.txt" | xargs chmod o-rwx`

Super-user (root)

| command | description |
|---------|--|
| sudo | run a single command with root privileges (prompts for password) |
| su | start a shell with root privileges (so multiple commands can be run) |

- **super-user:** An account used for system administration.
 - has full privileges on the system
 - usually represented as a user named root
- Most users have more limited permissions than root
 - protects system from viruses, rogue users, etc.
 - if on your own box, why ever run as a non-root user?
- Example: Install the valgrind tool on the CSE VM.
`sudo yum install valgrind`

Playing around with power...



Courtesy XKCD.com

Playing around with power...

- Create a file, remove all permissions
 - Now, login as root and change the owner and group to root
 - Bwahaha, is it a brick in a user's directory?
- Different distributions have different approaches
 - Compare Fedora to Ubuntu in regards to sudo and su...
- Power can have dangerous consequences
 - `rm *` might be just what you want to get rid of everything in a local directory
 - but what if you happened to be in `/bin...` and you were running as root...

tar files

| | description |
|-----|--|
| tar | create or extract .tar archives (combines multiple files into one .tar file) |

- Originally used to create “tape archive” files
- Combines multiple files into a single .tar file
- You probably always want to use `-f` option and **IT SHOULD COME LAST**
- To **create** a single file from multiple files:

```
$ tar -cf filename.tar stuff_to_archive
```

 - `-c` **creates** an archive
 - `-f` read to/from a file
 - `stuff_to_archive` - can be a list of filenames or a directory
- To **extract** files from an archive:

```
$ tar -xf filename.tar
```

 - `-x` **extracts** files from an archive

Compressed files

| command | description |
|----------------|---|
| zip, unzip | create or extract .zip compressed archives |
| gzip, gunzip | GNU free compression programs (single-file) |
| bzip2, bunzip2 | slower, optimized compression program (single-file) |

- To **compress** a file:

\$ `gzip filename` produces: `filename.gz`

- To **uncompress** a file:

\$ `gunzip filename.gz` produces: `filename`

Similar for zip, bzip2. See man pages for more details.

.tar.gz archives

- Many Linux programs are distributed as .tar.gz archives (sometimes called .tgz)
- You could unpack this in two steps:
 1. `gzip foo.tar.gz` produces: `foo.tar`
 2. `tar -xf foo.tar` extracts individual files
- You can also use the tar command to create/extract compressed archive files **all in one step**:

```
$ tar -xzf filename.tar.gz
```

 - `-x` extracts files from an archive
 - `-z` filter the archive through gzip (compress/uncompress it)
 - `-f` read to/from a file

Handy tip: You can use the “file” command to see what type a file is, just changing the file extension on a file does **not** change its type.

tar examples

You can combine options (-v, -z, etc.) various ways:

Create a single .tar archive file from multiple files (without compression):

```
$ tar -cvf filename.tar stuff_to_archive
```

- -c **creates** an archive file called *filename*.tar
- -v verbosely list the files processed
- -f read to/from a file (as opposed to a tape archive)
- stuff_to_archive - can be filenames or a directory

Add -z option and use *filename*.tar.gz to use compression:

```
$ tar -cvzf filename.tar.gz stuff_to_archive
```

tar

