CSE 303
Lecture 4

users/groups; permissions; intro to shell scripting

read *Linux Pocket Guide* pp. 19-20, 25-27,
61-65, 118-119, 176

slides created by Marty Stepp
http://www.cs.washington.edu/303/

Lecture summary

- discuss ethics/society reading #1
- more I/O redirection, piping, combining commands
- user accounts, groups, and the super-user (root)
- file permissions
- introduction to shell scripting

Ethics/society reading #1

- What is the difference between "open source" and "free?
- Is it important that we can see the source code?
- Could Microsoft still make any money if they went open source?
- What is a "fork"? Are forks good or bad, and why?
- Is Marty allowed to sell you an Ubuntu CD for $1?

Aliases

command	description
`alias`	assigns a pseudonym to a command

`alias` ***name=command***

- must wrap the command in quotes if it contains spaces

- Example: When I type `q` , I want it to log me out of my shell.
- Example: When I type `ll` , I want it to list all files in long format.

```
alias q=exit
alias ll="ls -la"
```

- *Exercise* : Make it so that typing `q` quits out of a shell.
- *Exercise* : Make it so that typing `woman` runs `man`.
- *Exercise* : Make it so that typing `attu` connects me to `attu`.
- *Exercise* : Make it so that typing `banner` on `attu` runs banner.

Recall: combined commands

command1* `>` *filename

- run ***command1*** and write its output to ***filename*** instead of to console;

  `>>` appends rather than overwriting if the file already exists

command1* `<` *filename

- run ***command1*** and read its input from ***filename*** instead of console

command1* `|` *command2

- run ***command1*** and send its console output as input to ***command2***
- note that console input is not the same thing as parameters!

- Example: Find unique lines containing "secret" in all text files.

```
grep secret *.txt | uniq
```

Commands in sequence

command1* ; *command2

- run ***command1*** and then ***command2*** afterward (they are not linked)

command1 && ***command2***

- run ***command1***, and if it succeeds, runs ***command2*** afterward
- will not run ***command2*** if any error occurs during the running of 1

- Example: Make directory `songs` and move my files into it.

```
mkdir songs && mv *.mp3 songs
```

More combining commands

***command1* \`*command2*\`**

- run ***command2*** and pass its console output to ***command1*** as a parameter; \` is a back-tick, on the ~ key; not an apostrophe
- best used when ***command2***'s output is short (one line)

- Example: Create directory "stepp" (when logged in as stepp).

```
mkdir `whoami`
```

  - Why not `whoami | mkdir ?`

- Example: Display all files that were last modified during this year.

```
ls -l | grep `date +%G`
```

xargs

command	description
`xargs`	runs each line of its input as a command

- `xargs` allows you to repeatedly run a command over a set of lines
  - often used in conjunction with `find` to process each of a set of files

- Example: Remove all evidence of my BitTorrent transfers.

```
find ~ -name *.torrent | xargs rm
```

- *Exercise* : List in long format all `.txt` files that contain the text "303", sorted in reverse alphabetical order.

```
-rw-------  1 stepp None 30300 Apr  6 10:07 todo.txt
-rw-------  1 stepp None  5434 Apr  6 10:07 ideas.txt
```

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
  - ...

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

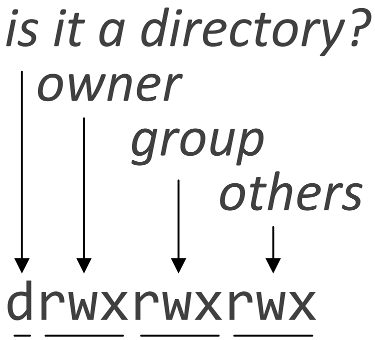
- Every file has an associated group.
  - the owner of a file can grant permissions to the group
- Every group has a unique integer ID number (GID).

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`



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 write; others nothing)

- 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)

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`

http://xkcd.com/149/

- Most users have more limited permissions than `root`
  - protects system from viruses, rogue users, etc.

- Example: Install the `sun-java6-jdk` package on Ubuntu.

```
sudo apt-get install sun-java6-jdk
```

Shell scripts

- **script**: A short program whose purpose is to run other programs.
  - a series of commands combined into one executable file

- **shell script**: A script that is executed by a command-line shell.
  - `bash` (like most shells) has syntax for writing script programs

- To write a `bash` script (in brief):
  - type one or more commands into a file; save it
  - type a special header in the file to identify it as a script (next slide)
  - enable execute permission on the file
  - run it!

Basic script syntax

#!***interpreter***

- written as the first line of an executable script; causes a file to be treated as a script to be run by the given interpreter
  - (we will use `/bin/bash` as our interpreter)

- Example: A script that removes some files and then lists all files:

```
#!/bin/bash
rm output*.txt
ls -l
```

Running a shell script

- by making it executable (most common):

```
chmod u+x myscript.sh
./myscript.sh
```

- by launching a new shell:

```
bash myscript.sh
```

- by running it within the current shell:

```
source myscript.sh
```

  - advantage: any variables defined by the script remain in this shell (seen later)

`.bash_profile`

- every time you log in to `bash`, it runs the file `~/.bash_profile`
  - you can put any common startup commands you want into this file
  - useful for setting up aliases and other settings

- *Exercise* : Make it so that our `q` and `L` aliases from earlier become persistent, so that they 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.

echo

command	description
echo	produces its parameter(s) as output (the `println` of shell scripting)

- Example: A script that prints the time and your home directory.

```
#!/bin/bash
echo "This is my amazing script!"
echo "Your home dir is: `pwd`"
```

- *Exercise* : Make it so that whenever I log in to `attu`, it:
  - clears the screen
  - displays the current date: `The time is: 04/06 10:40`
  - shows me an ASCII cow welcoming my user name