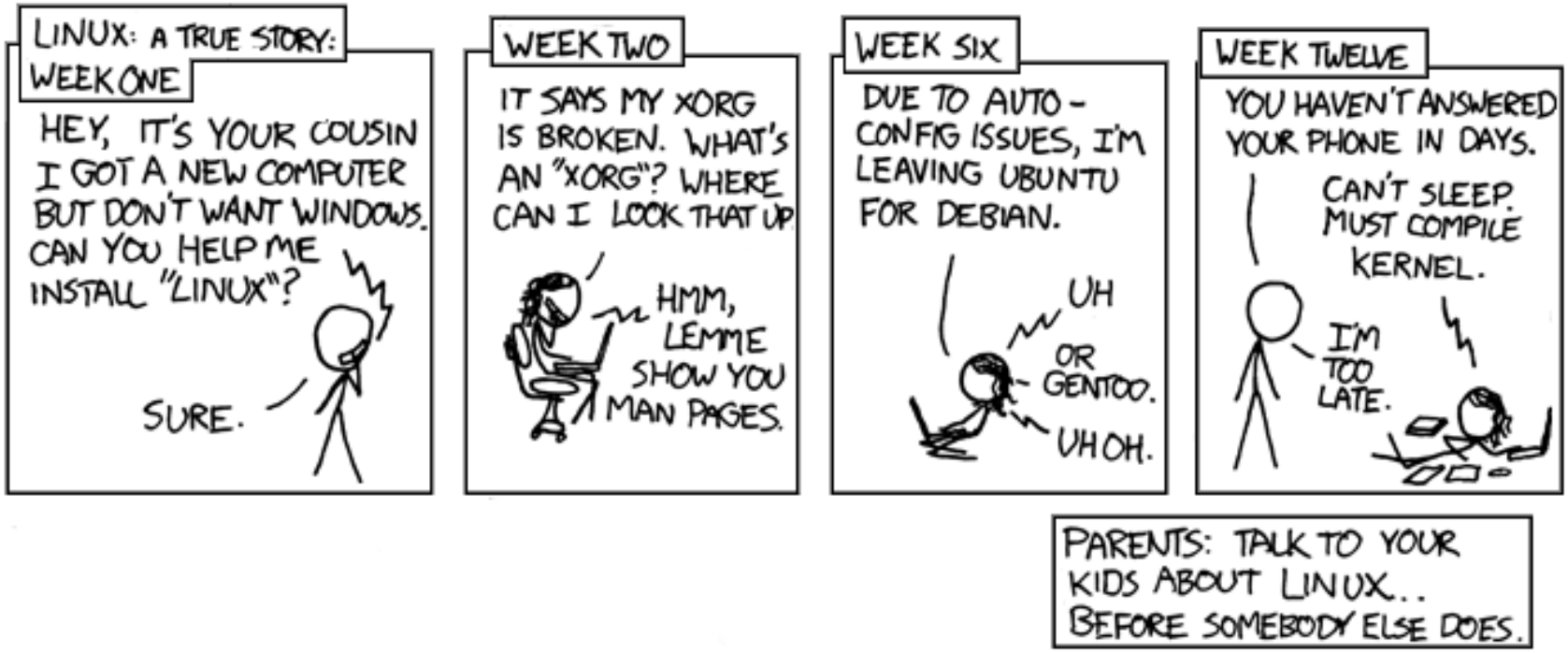
Lecture 1

Introduction to Linux/Unix environment

slides created by Marty Stepp, modified by Jessica Miller (University of Washington)

On to Linux



Courtesy XKCD.com

Linux

- **Linux**: A kernel for a Unix-like operating system.
  - commonly seen/used today in servers, mobile/embedded devices, ...
- **GNU**: A "free software" implementation of many Unix-like tools
  - many GNU tools are distributed with the Linux kernel
- **distribution**: A pre-packaged set of Linux software.
  - examples: Ubuntu, Fedora

- key features of Linux:
  - **open source software**: source can be downloaded
  - free to use
  - constantly being improved/updated by the community

Shell

- **shell**: An interactive program that uses user input to manage the execution of other programs.
  - `bash` : the default shell program on most Linux/Unix systems

- Why should I learn to use a shell when GUIs exist?

Shell

- **shell**: An interactive program that uses user input to manage the execution of other programs.
  - `bash` : the default shell program on most Linux/Unix systems

- Why should I learn to use a shell when GUIs exist?
  - faster
  - work remotely
  - programmable
  - customizable
  - repeatable

- input, output, and errors
- directories: working/current directory, home directory

Shell commands

command	description
`exit`	logs out of the shell
`ls`	lists files in a directory
`pwd`	outputs the current working directory
`cd`	changes the working directory
`man`	brings up the manual for a command

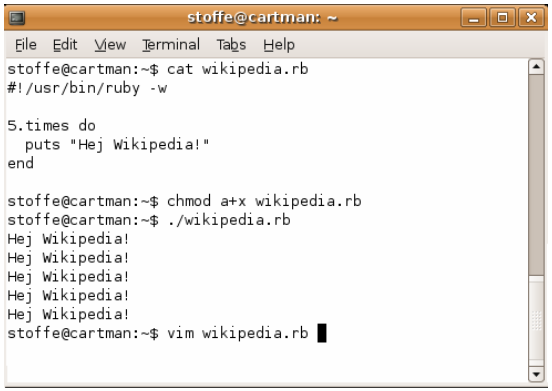
```
$ pwd
/homes/iws/dravir
$ cd CSE390
$ ls
file1.txt file2.txt
$ ls -l
-rw-r--r-- 1 dravir vgrad_cs 0 2010-03-29 17:45 file1.txt
-rw-r--r-- 1 dravir vgrad_cs 0 2010-03-29 17:45 file2.txt
$ cd ..
$ man ls
$ exit
```

Relative directories

directory	description
`.`	the directory you are in ("working directory")
`..`	the parent of the working directory (`../..` is grandparent, etc.)
`~`	your home directory (on many systems, this is `/home/`***`username`***)
`~`***`username`***	***`username`***'s home directory
`~/Desktop`	your desktop

Shell commands

- many accept **arguments** or **parameters**
  - example: cp (copy) accepts a source and destination file path



- a program uses 3 streams of information:
  - stdin, stdout, stderr (standard in, out, error)

- **input**: comes from user's keyboard
- **output**: goes to console
- **errors** can also be printed (by default, sent to console like output)

- parameters vs. input
  - *parameters*: before Enter is pressed; sent in by shell
  - *input*: after Enter is pressed; sent in by user

Directory commands

command	description
`ls`	list files in a directory
`pwd`	output the current working directory
`cd`	change the working directory
`mkdir`	create a new directory
`rmdir`	delete a directory (must be empty)

- some commands (`cd`, `exit`) are part of the shell ("builtins")
- others (`ls`, `mkdir`) are separate programs the shell runs

Command-line arguments

- most options are a `-` followed by a letter such as `-c`
  - some are longer words preceded by two `-` signs, such as `--count`
- parameters can be combined: `ls -l -a -r` can be `ls -lar`
- many programs accept a `--help` or `-help` parameter to give more information about that command (in addition to `man` pages)
  - or if you run the program with no arguments, it may print help info
- for many commands that accept a file name parameter, if you omit the parameter, it will read from standard input (your keyboard)
  - note that this can conflict with the previous tip

Shell/system commands

command	description
`man or info`	get help on a command
`clear`	clears out the output from the console
`exit`	exits and logs out of the shell

command	description
`date`	output the system date
`cal`	output a text calendar
`uname`	print information about the current system

- "man pages" are a very important way to learn new commands

```
man ls
man man
```

File commands

command	description
`cp`	copy a file
`mv`	move or rename a file
`rm`	delete a file
`touch`	create a new empty file, or update its last-modified time stamp

- caution: the above commands do not prompt for confirmation
  - easy to overwrite/delete a file; this setting can be overridden (how?)

- *Exercise* : Modify a `.java file` to make it seem as though you finished writing it on March 15 at 4:56am.

Lecture summary

- Unix file system structure
- Commands for file manipulation, examination, searching
- Java compilation: using parameters, input, and streams
- Redirection and Pipes

Unix file system

directory	description
`/`	root directory that contains all others (drives do not have letters in Unix)
`/bin`	programs
`/dev`	hardware devices
`/etc`	system configuration files ▪ `/etc/passwd` stores user info ▪ `/etc/shadow` stores passwords
`/home`	users' home directories
`/media`, `/mnt`, …	drives and removable disks that have been "mounted" for use on this computer
`/proc`	currently running processes (programs)
`/tmp, /var`	temporary files
`/usr`	user-installed programs

File examination

command	description
`cat`	output a file's contents on the console
`more or less`	output a file's contents, one page at a time
`head, tail`	output the first or last few lines of a file
`wc`	count words, characters, and lines in a file
`du`	report disk space used by a file(s)
`diff`	compare two files and report differences

- Let’s explore what we can do here…

Searching and sorting

command	description
`grep`	search a file for a given string
`sort`	convert an input into a sorted output by lines
`uniq`	strip duplicate (adjacent) lines
`find`	search for files within a given directory
`locate`	search for files on the entire system
`which`	shows the complete path of a command

- `grep` is actually a very powerful search tool

- *Exercise* : Given a text file `names.txt`, display the students arranged by the reverse alphabetical order of their names.

Keyboard shortcuts

^***KEY*** means hold `Ctrl` and press ***KEY***

key	description
`Up arrow`	repeat previous commands
`Home/End or ^A/^E`	move to start/end of current line
`"`	quotes surround multi-word arguments and arguments containing special characters
`*`	"wildcard" , matches any files; can be used as a prefix, suffix, or partial name
`Tab`	auto-completes a partially typed file/command name
`^C or ^\`	terminates the currently running process
`^D`	end of input; used when a program is reading input from your keyboard and you are finished typing
`^Z`	suspends (pauses) the currently running process
`^S`	don't use this; hides all output until ^Q is pressed

Programming

command	description
`javac` ***`ClassName`***`.java`	compile a Java program
`java` ***`ClassName`***	run a Java program
`python, perl, ruby, gcc, sml, ...`	compile or run programs in various other languages

- *Exercise* : Write/compile/run a program that prints "Hello, world!"

```
$ javac Hello.java
$ java Hello
Hello, world!
$
```

Streams in the Shell

- Stdin, stdout, stderr
  - These default to the console
  - Some commands that expect an input stream will thus read from the console if you don't tell it otherwise.
- *Example:* grep hi
  - What happens? Why?

We can change the default streams to something other than the console via redirection.

Output redirection

command* > *filename

- run ***command*** and write its output to ***filename*** instead of to console;
  - think of it like an arrow going from the command to the file...
  - if the file already exists, it will be overwritten (be careful)
  - `>>` appends rather than overwriting, if the file already exists
  - ***command*** `> /dev/null` suppresses the output of the command

- Example: `ls -l > myfiles.txt`
- Example: `java Foo >> Foo_output.txt`
- Example: `cat > somefile.txt`
  (writes console input to the file until you press ^D)

Input redirection

command* < *filename

- run ***command*** and read its input from ***filename*** instead of console
  - whenever the program prompts the user to enter input (such as reading from a `Scanner` in Java), it will instead read the input from a file
  - some commands don't use this; they accept a file name as an argument

- Example: `java Guess < input.txt`
- Exercise: run hello world with the input stream as a file instead of the console
- Exercise: Also change the output stream to write the results to file

- again note that this affects *user input*, not *parameters*
- useful with commands that can process standard input or files:
  - e.g. `grep`, `more`, `head`, `tail`, `wc`, `sort`, `uniq`, `write`

Combining commands

command1* | *command2

- run ***command1*** and send its console output as input to ***command2***
- very similar to the following sequence:

```
command1 > filename
command2 < filename
rm filename
```

- Examples:

```
diff students.txt names.txt | less
sort names.txt | uniq
```

- *Exercise* : `names.txt` contains CSE student first names, one per line. We are interested in students whose first names begin with "A", such as “Alisa".
  - Find out of how names beginning with "A" are in the file.
  - Then figure out how many characters long the name of the last student whose name starts with "A" is when looking at the names alphabetically.

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
```

Lecture summary

- A bit more on combining commands
- Processes and basic process management
- Text editors

Review: Redirection and Pipes

- ***command*** > ***filename***
  - Write the output of ***command*** to ***filename*** (>> to append instead)
- ***command*** < ***filename***
  - Use ***filename*** as the input stream to ***command***
- ***command1*** | ***command2***
  - Use the console output of ***command1*** as the input to ***command2***
- ***command1*** ; ***command2***
  - Run ***command1*** and then run ***command2***
- ***command1*** && ***command2***
  - Run ***command1***, if completed without errors then run ***command2***

Tricky Examples

- The wc command can take multiple files: wc names.txt student.txt
  - Can we use the following to wc on every txt file in the directory?
    - `ls *.txt | wc`

- Amongst the top 250 movies in movies.txt, display the third to last movie that contains "The" in the title when movies titles are sorted.

- Find the disk space usage of the man program
  - Hints: use `which` and du…
  - Does `which man | du` work?

The back-tick

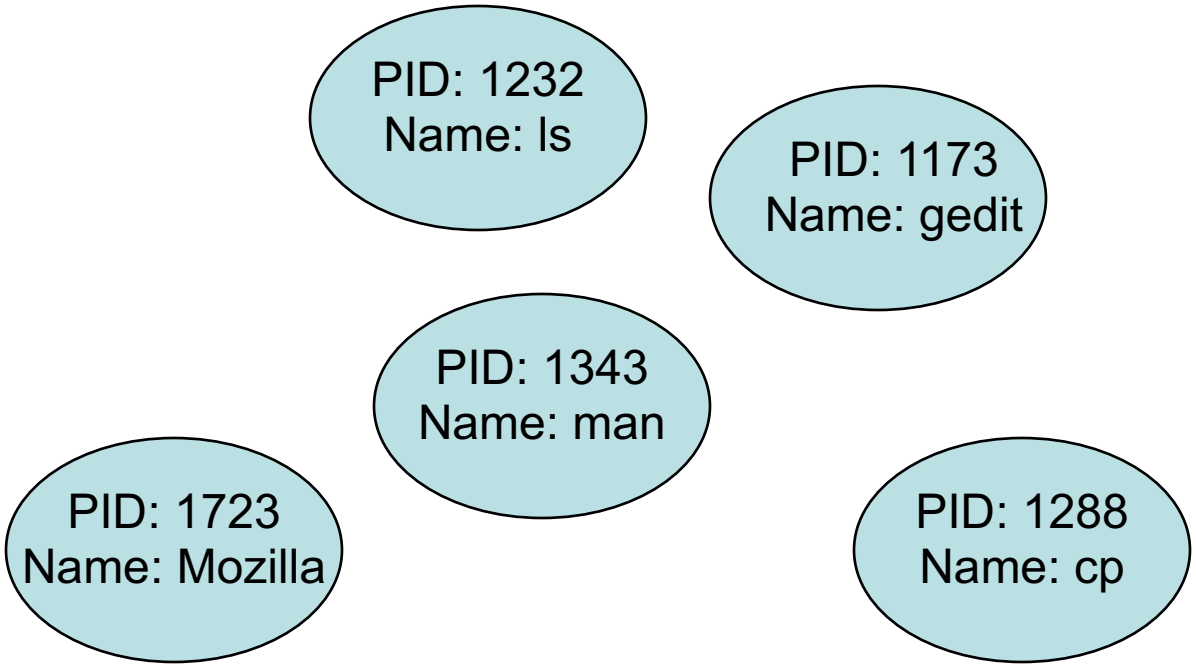
command1 `command2`

- 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)

- Finish the example!
  - `du `which man``

Processes

- **process**: a program that is running (essentially)
  - when you run commands in a shell, it launches processes for each
  - Process management is one of the major purposes of an OS



Process commands

command	description
`ps or jobs`	list processes being run by a user; each process has a unique integer id (PID)
`top`	show which processes are using CPU/memory; also shows stats about the computer
`kill`	terminate a process by PID
`killall`	terminate several processes by name

- use `kill` or `killall` to stop a runaway process (infinite loop)
  - similar to ^C hotkey, but doesn't require keyboard intervention

Background processes

command	description
`&`	(special character) when placed at the end of a command, runs that command in the background
`^Z`	(hotkey) suspends the currently running process
`fg`, `bg`	resumes the currently suspended process in either the foreground or background

- If you run a graphical program like `gedit` from the shell, the shell will lock up waiting for the graphical program to finish
  - instead, run the program in the background, so the shell won't wait:

    ```
    $ gedit resume.txt &
    ```
  - if you forget to use `&`, suspend `gedit` with `^Z`, then run bg
  - lets play around with an infinite process…

Connecting with ssh

command	description
ssh	open a shell on a remote server

- Linux/Unix are built to be used in multi-user environments where several users are logged in to the same machine at the same time
  - users can be logged in either locally or via the network
- You can connect to other Linux/Unix servers with ssh
  - once connected, you can run commands on the remote server
  - other users might also be connected; you can interact with them
  - can connect even from other operating systems

Multi-user environments

command	description
`whoami`	outputs your username
`passwd`	changes your password
`hostname`	outputs this computer's name/address
`w or finger`	see info about people logged in to this server
`write`	send a message to another logged in user

Network commands

command	description
`links or lynx`	text-only web browsers (really!)
`ssh`	connect to a remote server
`sftp or scp`	transfer files to/from a remote server (after starting sftp, use get and put commands)
`wget`	download from a URL to a file
`curl`	download from a URL and output to console
`pine, mail`	text-only email programs

Text editors

command	description
`pico or nano`	simple but limited text editors
`emacs`	complicated text editor (powerful)
`vi or vim`	complicated text editor (powerful)

- most advanced Unix/Linux users learn `emacs` or `vi`
  - these editors are powerful but complicated and hard to learn
  - `nano` is simpler (hotkeys are shown on screen)

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 `seed` connects me to `seed VM`.

Lecture summary

- Persistent settings for your bash shell
- User accounts and groups
- File permissions
- The Super User

.bash_profile and .bashrc

- Every time you log in to `bash`, the commands in `~/.bash_profile` are run
  - a . in front of a filename indicates a normally hidden file (ls –a to see)
  - 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, the commands in `~/.bashrc` are run
  - useful for setting up persistent commands for local shell usage, or when launching multiple shells
  - often, .bash_profile is configured to also run .bashrc, but not always

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

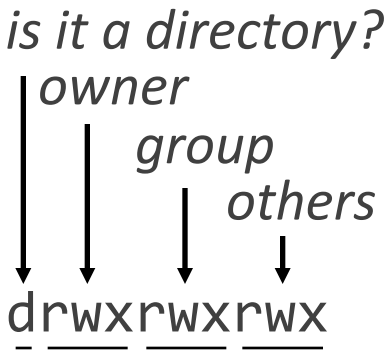
- **group**: A collection of users, used as a target of permissions.
  - a group can be given access to a file or resource
  - a user can belong to many groups
  - see who's in a group using grep <groupname> /etc/group

- 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`



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)

  note: -R for recursive

- 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)

Exercises

- Change a file to grant full access (rwx) to everyone
  - Now change it do deny all access (rwx) from everyone
    - !!! is it dead?
    - I own this file. Can I change the user?

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?

Lets combine things

- Say I have a directory structure, with lots of .txt files scattered
  - I want to remove all world permissions 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`

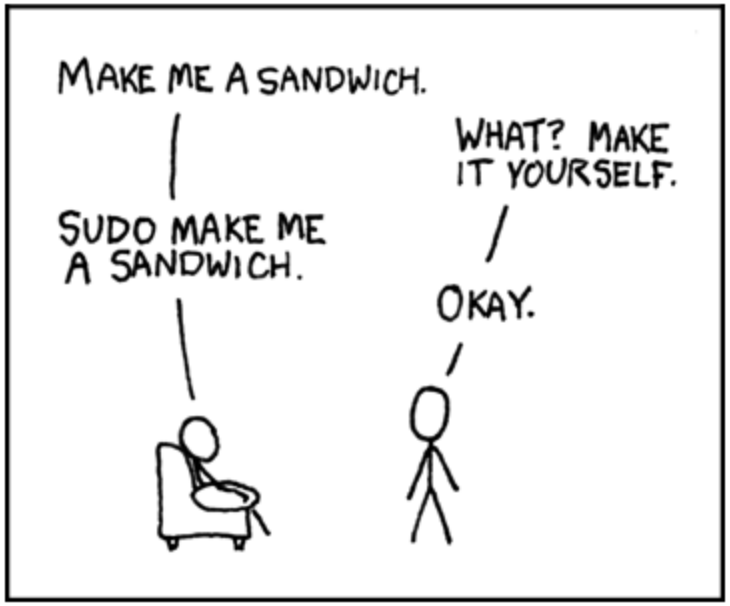
Super-user (root)

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 `sun-java6-jdk` package on Ubuntu.

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

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…