BMO Training

Getting started with Linux

Peter Muia

Day 1: Modules

1. Linux overview
2. Command Line Interface or the “CLI”
3. Permissions
4. Editors
5. Ubuntu Linux and more commands

Module 1: Linux Overview

Unix vs. Linux

Are they the same?

- Yes, at least in terms of operating system interfaces
- Linux was developed independently from Unix
- Unix is much older (1969 vs. 1991)

Scalability and reliability

- Both scale very well and work well under heavy load

Flexibility

- Both emphasize small, interchangeable components

Manageability

- Remote logins rather than GUI
- Scripting is integral

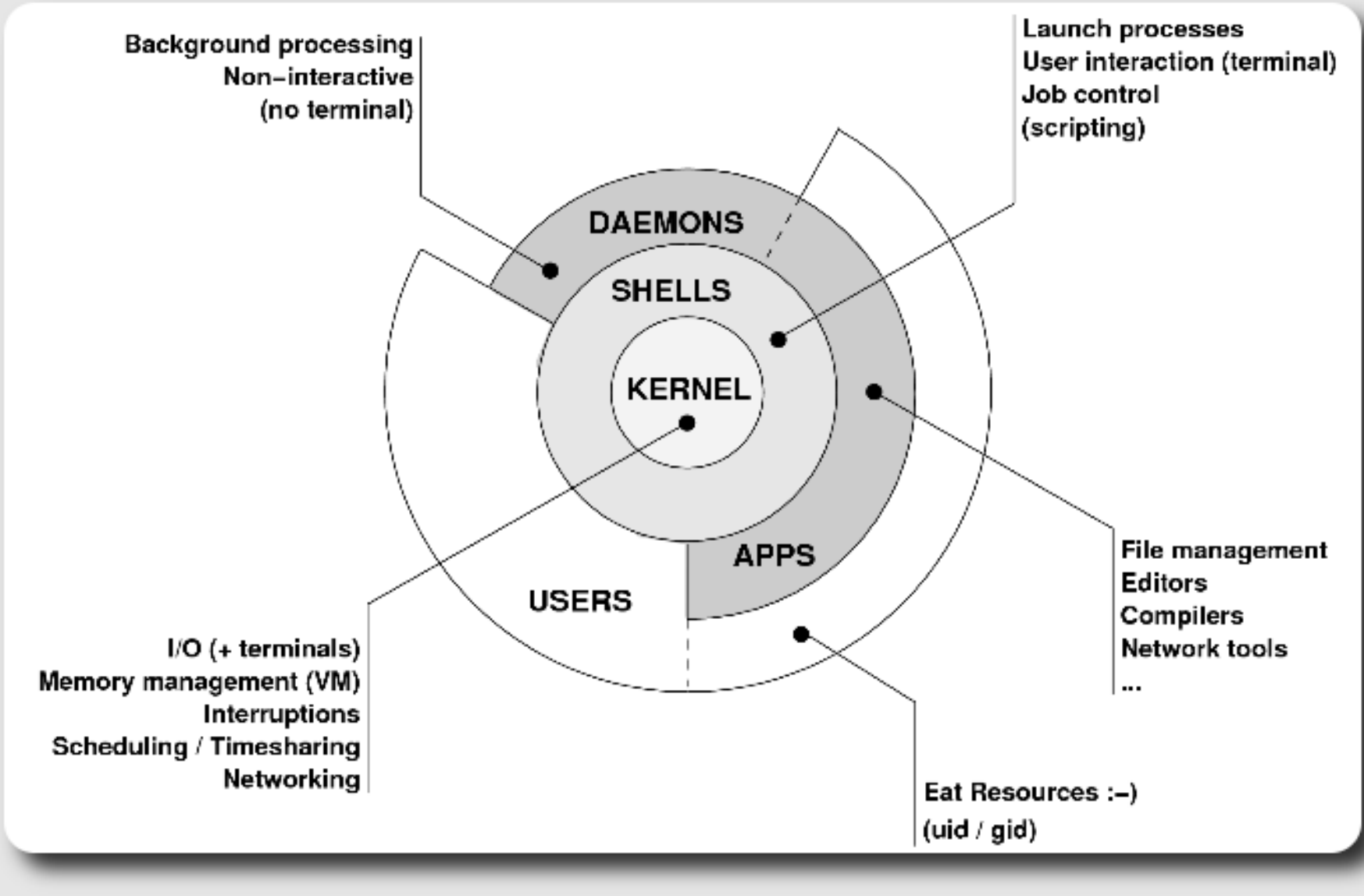
Security

- Due to modular design has a reasonable security model

What's running Linux?

- 90% of the supercomputer TOP500, including all TOP 10
- Half of the world's 10 most reliable hosting companies
- The Internet of Things (to some extent)
- Maybe your smart phone?
  Android is *based on* Linux

The Unix/Linux System



Kernel

The "core" of the operating system

Device drivers

- communicate with your hardware
- block devices, character devices, network devices, pseudo devices (/dev/null)

Filesystems

- organize block devices into files and directories

Memory management

Timeslicing (multitasking)

Networking stacks - esp. TCP/IP

Enforces security model

Shells

Command line interface for executing programs

- Windows equivalent: command.com or command.exe

Choice of similar but slightly different shells

- **sh:** the "Bourne Shell"
- **csh:** the "C Shell". Not standard, but includes command history
- **bash:** the "Bourne-Again Shell"
- Others: **ksh**, **tcsh**, **zsh**

User processes

- The programs that you choose to run
- Frequently-used programs tend to have short cryptic names
  - "**`ls`**" = list files
  - "**`cp`**" = copy file
  - "**`rm`**" = remove (delete) file
- Lots of stuff included in most base systems
  - editors, compilers, system admin tools
- Lots more stuff available to install too
  - Using the Debian/Ubuntu repositories

System processes

Programs that run in the background; also known as "daemons" ==>  *

Examples:

- **cron**: executes programs at certain times of day
- **syslogd**: takes log messages and writes them to files
- **inetd**: accepts incoming TCP/IP connections and starts programs for each one
- **sshd**: accepts incoming logins
- **sendmail** (or other MTA daemon like Postfix): accepts incoming mail

* ”Sparky” from the FreeBSD world

Security model

Numeric IDs

- user id (uid 0 = "*root*", the superuser)
- group id
- supplementary groups

Mapped to names

- /etc/passwd, /etc/group (plain text files)

Suitable security rules enforced

- e.g. you cannot kill a process running as a different user, unless you are "*root*"

Filesystem security

Each file and directory has three sets of permissions

- For the file's uid (user)
- For the file's gid (group)
- For everyone else (other)

Each set of permissions has three bits: **rwx**

- File: **r**=read, **w**=write, **x**=execute
- Directory: **r**=list directory contents, **w**=create/delete files within this directory, **x**=enter directory (e**x**ecutable)

Filesystem security

The permission flags are read as follows left to right:

`-rw-r--r--`	for regular files,
`drwxr-xr-x`	for directories

Any questions?

?

Standard filesystem layout

/bin	**essential binaries**
/boot	**kernel and boot support**
/dev	**device access nodes**
/proc	**pseudo-filesystem with config/system info**
/etc	**configuration data**
/etc/default	**package startup defaults**
/etc/init.d	**startup scripts**
/home/*username*	**user's “home” directory**
/lib	**essential libraries**
/sbin	**essential sysadmin tools**
/tmp	**temporary files**
/usr	**programs & appl. data**
/var	**changing files (logs, E-mail messages, queues, …)**

Don't confuse the the “root account” (/root) with the “root” (“/”) partition.

More filesystem details

/usr	
/usr/bin	binaries
/usr/lib	libraries
/usr/sbin	sysadmin binaries
/usr/share	misc application data
/usr/src	kernel source code
/usr/local/...	3rd party applications not installed with apt
/var	
/var/log	log files
/var/mail	mailboxes
/var/run	process status
/var/spool	queue data files
/var/tmp	temporary files

Log files (a few examples)

```
/var
   /var/log
   /var/log/apache2
   /var/log/apache2/access.log
   /var/log/apache3/error.log
   /var/log/auth.log
   /var/log/boot.log
   /var/log/dmesg
   /var/log/kern.log
   /var/log/mail.info
   /var/log/mail.err
   /var/log/mail.log
   /var/log/messages
   /var/log/mysql
   /var/log/syslog
```

Log file: who & what's doing what

The most critical place to solve problems

- System messages, including:
  - Problems
  - Security issues
  - Configuration errors
  - Access issues
- Service messages, including:
  - Same as above

When something does not work…

…Look in your log files first!

Partitioning considerations

- Single large partition or multiple?
- A single partition is flexible, but a rogue program can fill it up…
- Multiple partitions provides a more “protected” approach, but you may need to resize later, on older filesystems, or without a “Volume Manager”
  - Is ***/var*** big enough? /tmp?
  - How much *swap* should you define?

Note...

- Partitioning is just a logical division
- If your hard drive dies, most likely *everything* will be lost.
- If you want data security, then you need to set up mirroring or RAID with a separate drive.

Remember, "`rm –rf /`" on a mirror will erase everything on both disks

Data Security <==> Backup

/dev

Virtual files pointing to hardware or other

/dev/sda = the first harddisk
(SCSI/SATA/SAS or IDE)

Dynamically created /dev entries

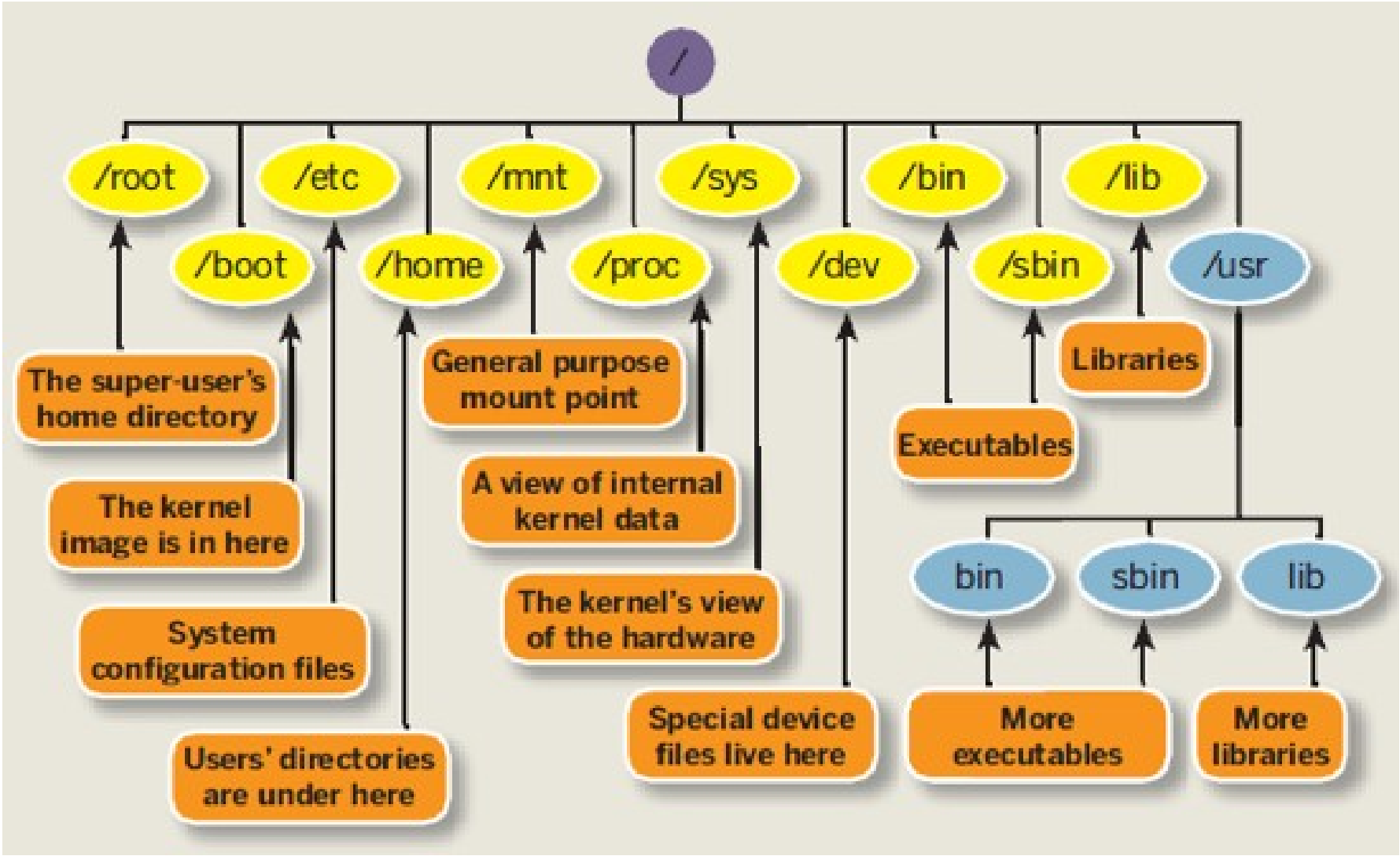
e.g. when you plug in a new USB device

pseudo-devices:

/dev/null

/dev/random

Sample Linux File System



http://www.tuxradar.com/content/take-linux-filesystem-tour

How Does Linux boot?

- The *BIOS* loads and runs the *MBR*:
  - The ***M**aster **B**oot **R**ecord* points to a default partition, or lets you select the boot partition
- MBR code then loads the boot loader, such as GRUB
- Boot loader reads configuration parameters (/boot) presents the user with options on how to boot system
- kernel is loaded and started, filesystems are mounted, modules are loaded
- init(8) process is started
- system daemons are started

http://en.wikipedia.org/wiki/Linux_startup_process

Packages & Exercises

We'll reinforce some of these concepts using exercises...

Right now please connect to your virtual Linux machine using SSH:

- ssh nsrc@197.136.7.1
  - Pass: *password*
- # virsh console hostxx.ws.nsrc.org
- You will be given a username and password

Packages & Exercises

We'll run a few commands to get started:

- `ls` (list files / directories)
- `pwd` (current working directory)
- …
- …
- …

Module 2: Command Line Interface

The format of a command

command [options] parameters

“Traditionally, UNIX command-line options consist of a dash, followed by one or more lowercase letters. The GNU utilities added a double-dash, followed by a complete word or compound word.”

Two very typical examples are:

`-h`

`--help`

and

`-v`

`--version`

Command parameters

The ***parameter*** is what the command ***acts on***.

Often there are multiple parameters.

In Unix UPPERCASE and lowercase for both options and parameters matter.

Spaces ___ are ___ critical ___ .

“-- help” is wrong.

“--help” is right.

Some command examples

Let's start simple:

Display a list of files:

```
ls
```

Display a list of files in a long listing format:

```
ls -l
```

Display a list of all files in a long listing format with human-readable file sizes:

```
ls -alh
```

Some command examples cont.

Some equivalent ways to do "`ls -alh`":

```
ls -lah
ls -l -a -h
ls -l –all --human-readable
```

Note that there is no double-dash option for "`-l`". You can figure this out by typing:

```
man ls
```

Or by typing:

```
ls --help
```

Spot the deliberate mistake. Why does it work?

Where's the parameter?

We typed the “`ls`” command with several options, but no parameter. Do you think “`ls`” uses a parameter?

What is the parameter for “`ls -l`”?

It is “.” -- our current directory.

“`ls -l`” and “`ls -l .`” are the same.

A disconcerting Linux feature

If a command executes successfully there is no output returned from the command execution. *this is normal*.

That is, if you type:

```
cp file1 file2
```

The result is that you get your command prompt back. *Nothing means success*.

Let's give this a try...

A disconcerting Linux feature

Try doing the following on your machine:

```
$ cd                          [cd = change dir]
$ touch file1          [touch = create/update]
$ cp file1 file2              [cp = copy]
```

- The “$” indicates the command prompt for a normal user.
- A “#” usually means you are the *root* user.

Using pipes

In Unix it is very easy to use the result of one command as the input for another.

To do this we use the pipe symbol “|”. For example:

```
ls -l /sbin | sort
```

What will this command do?

Stopping command output

Stopping commands with continuous output:

- Terminate foreground program: CTRL+C

```
$ ping yahoo.com
PING yahoo.com (67.195.160.76): 56 data bytes
64 bytes from 67.195.160.76: icmp_seq=0 ttl=45 time=221.053 ms
64 bytes from 67.195.160.76: icmp_seq=1 ttl=45 time=224.145 ms
^C      here press CTRL + C
```

- Terminate paging like “less <filename>”

```
$ less /etc/passwd
sysadm:x:1000:1000:System Administrator,,,:/home/sysadm:/bin/bash
postfix:x:104:113::/var/spool/postfix:/bin/false
mysql:x:105:115:MySQL Server,,,:/var/lib/mysql:/bin/false
(END)     press the “q” key
```

Proper command line use

The command line in Unix is *much more powerful* than what you may be used to in Windows. ***You can...***

...easily edit long commands

...find and recover past commands

...quickly copy and paste commands.

...auto-complete commands using the tab key (in *bash* shell).

Find and recover past commands

As noted on the previous slide. Use:

```
$ history | grep “command string”
```

Find command number in resulting list.

Execute the command by typing:

```
$ !number
```

So, to find any command you typed “many” commands ago you can do:

```
$ history | grep command
```

Find and recover past commands

For last few commands use the up-arrow.

Don't re-type a long command if you just typed it.

Instead use the up arrow and adjust the command.

Copy and paste commands

In Unix/Linux once you highlight something it is *already* in your copy buffer.

To copy/paste in Linux/Unix do:

- Highlight text with left mouse cursor. It is now copied (like *ctrl-c* in Windows).
- Move mouse/cursor where you want (any window), and press the *middle* mouse button. This is paste (like *ctrl-v*).

In Windows / Mac use the traditional ctrl-c / ctrl-v

Auto-complete commands with tab

Very, very, very powerful

“The tab key is good”, “the tab key is my friend”, “press the tab key”, “press it again” - This is your mantra.

Tab works in the *bash* shell. Note, the *root* user might not use the *bash* shell by default.

Auto-complete commands with tab

Core concept:

Once you type something unique, press TAB. If nothing happens, press TAB twice.

If text was unique text will auto-complete. A command will complete, directory name, file name, command parameters will all complete.

If not unique, press TAB twice. All possibilities will be displayed.

Works with file types based on command!

Auto-completion

We’ll do this now:

```
$ cat /etc       (TAB twice quickly)
$ cat /etc/netw             (TAB)
$ cat /etc/network/in       (TAB)
```

Viewing Files (part I)

Several ways to view a file:

1. `cat <filename>`
2. `more <filename`
3. `less <filename>`

- `cat` is short for *conCATenate*
- “`less` is `more`”

Obtaining “help”

To get help explaining commands you can do:

- `man <command>`
- `<command> --help`

man stands for “man”ual.

More on “man”

- `man man`

Your mission

Pay close attention to options and parameters.

Use “`man command`” or “`command --help`” to figure out how each command works.

Use command line magic to save lots and lots and lots and lots of time.

A command acts upon its parameters based on the options you give to the command...

Module 3: Permissions

Goal

Understand the following:

- The Linux / Unix security model
- How a program is allowed to run
- Where user and group information is stored
- Details of file permissions

Users and Groups

Linux understands Users and Groups

A user can belong to several groups

A file can belong to only one user and one group at a time

A particular user, the superuser *“root”* has extra privileges (uid = “0” in /etc/passwd)

Only root can change the ownership of a file

Users and Groups cont.

User information in `/etc/passwd`

Password info is in `/etc/shadow`

Group information is in `/etc/group`

`/etc/passwd` and `/etc/group` divide data fields using ":"

/etc/passwd:

```
joeuser:x:1000:1000:Joe User,,,:/home/joeuser:/bin/bash
```

/etc/group:

```
joeuser:x:1000:
```

A program runs...

A program may be run by a user, when the system starts or by another process.

Before the program can execute the kernel inspects several things:

- Is the file containing the program accessible to the user or group of the process that wants to run it?
- Does the file containing the program permit execution by that user or group (or anybody)?
- In most cases, while executing, a program inherits the privileges of the user/process who started it.

A program in detail

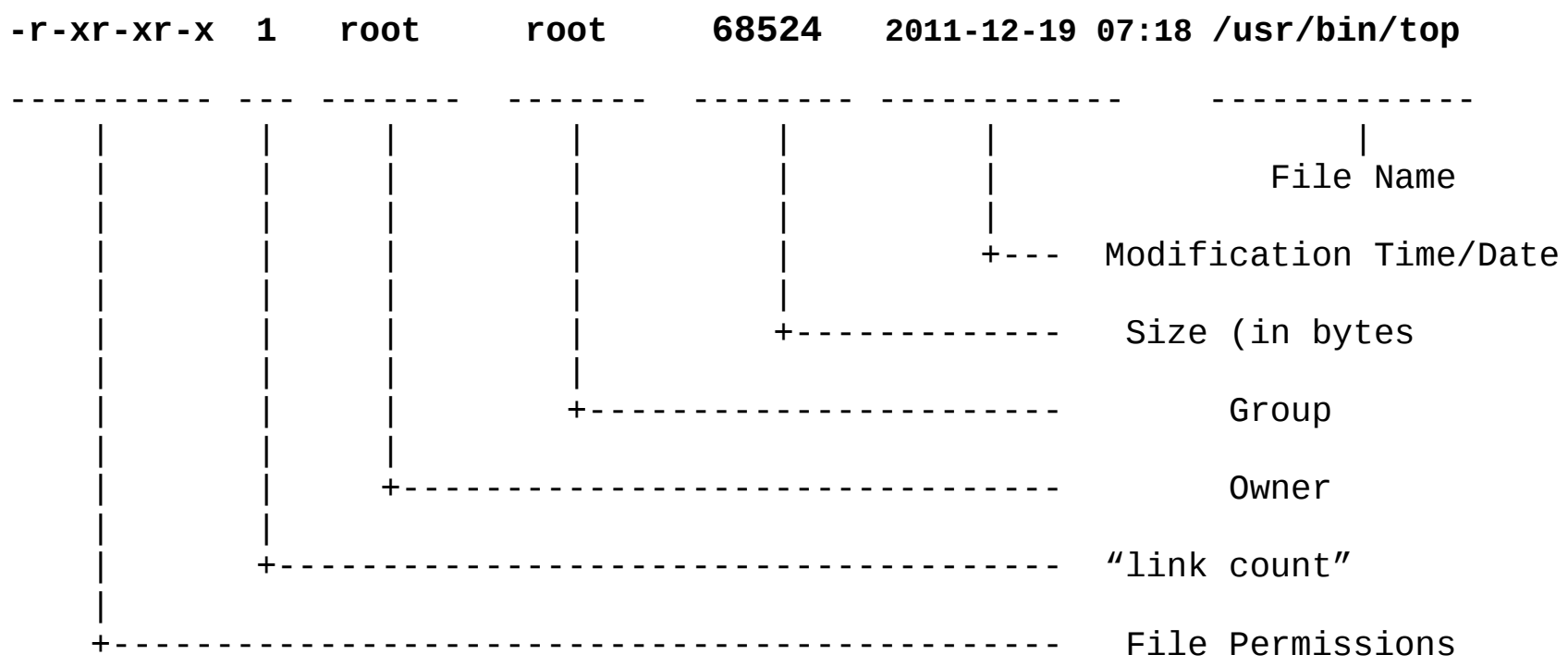
When we type:

```
ls -l /usr/bin/top
```

We'll see:

```
-rwxr-xr-x 1 root root 68524 2011-12-19 07:18 /usr/bin/top
```

What does all this mean?



Group

The name of the group that has permissions in addition to the file's owner.

Owner

The name of the user who owns the file.

File Permissions

The first character is the type of file. A "-" indicates a regular (ordinary) file. A "d” indicate a directory. Second set of 3 characters represent the read, write, and execution rights of the file's owner. Next 3 represent the rights of the file's group, and the final 3 represent the rights granted to everybody else.

(Example modified from **http://www.linuxcommand.org/lts0030.php**)

Access rights

Files are owned by a *user* and a *group* (ownership)

Files have permissions for the user, the group, and *other*

"*other*" permission is often referred to as "world"

The permissions are *Read, Write* and *Execute* (r, w, x)

The user who owns a file is always allowed to change its permissions

Some special cases

When looking at the output from “`ls -l`” in the first column you might see:

```
d = directory
- = regular file
l = symbolic link
s = Unix domain socket
p = named pipe
c = character device file
b = block device file
```

Some special cases cont

In the Owner, Group and other columns you might see:

`s = setuid`	[when in Owner column]
`s = setgid`	[when in Group column]
`t = sticky bit`	[when at end]

Some References

http://www.tuxfiles.org/linuxhelp/filepermissions.html

http://www.cs.uregina.ca/Links/class-info/330/Linux/linux.html

http://www.onlamp.com/pub/a/bsd/2000/09/06/FreeBSD_Basics.html

File permissions

There are two ways to set permissions when using the chmod command:

Symbolic mode:

testfile has permissions of `-r--r--r--`

```
                                    u   g   o*
$ chmod g+x testfile          ==>   -r--r-xr--
$ chmod u+wx testfile         ==>   -rwxr-xr--
$ chmod ug-x testfile         ==>   -rw--r--r—
u=user, g=group, o=other (world)
```

File permissions cont.

Absolute mode:

We use octal (base eight) values represented like this:

Letter	Permission	Value
r	read	4
w	write	2
x	execute	1
-	none	0

For each column, User, Group or Other you can set values from 0 to 7. Here is what each means:

0= **---** 1= **--x** 2= **-w-** 3= **-wx**

4= **r--** 5= **r-x** 6= **rw-** 7= **rwx**

File permissions cont.

Numeric mode cont:

Example index.html file with typical permission values:

```
$ chmod 755 index.html
$ ls -l index.html
-rwxr-xr-x  1 root  wheel  0 May 24 06:20 index.html

$ chmod 644 index.html
$ ls -l index.html
-rw-r--r--  1 root  wheel  0 May 24 06:20 index.html
```

Inherited permissions

Two critical points:

1. The permissions of a directory affect whether someone can see its contents or add or remove files in it.
2. The permissions on a file determine what a user can do to the data in the file.

Example:

If you don't have write permission for a directory, then you can't delete a file in the directory. If you have write access to the file you can update the data in the file.

Module 4: Editors

Goals

- Be able to edit a file using vi
- Use some of vi's more advanced features
- Begin to understand the “language” of configuration files
- Use alternate editors: ee, joe, pico, nano, emacs, xemacs, gedit, etc.

vi Philosophy

- It's available!
- Wait, what was that? Oh yeah, it's available!
- It's has some very powerful features.
- It's ubiquitous in UNIX and Linux (`visudo, vipw, vigr,` etc.)
- Not that hard to learn after initial learning curve.

Why is vi “so hard to use”?

Like all things it's not really – once you are used to how it works.

The ***critical*** vi concept:

1. vi has two modes
2. These modes are ***insert*** and ***command***

Let's see how we use these...

vi command and insert modes

Swapping modes

- When you open a file in vi you are in *command mode* by default.
- If you wish to edit the file *you need to switch to insert mode first*.
- To exit *insert mode* press the ESCape key.
- If you get used to this concept you are halfway done to becoming a competent vi user.

vi insert mode

Two common ways to enter insert mode upon opening a file include:

- Press the “i” key to start entering text directly after your cursor.
- Press the “o” key to add a new line *below* you cursor and to start adding text on the new line.
- Remember, to exit *insert mode* press the ESCape key at any time.

vi command mode

Many, many commands in vi, but some of the most common and useful are:

- Press “**x**” to delete a character at a time.
- Press “**dd**” quickly to press the line you are on.
- Press “/”, and text to search for and press <ENTER>.
  - Press “n” to find the next occurrence of text.
  - Press “N” to find previous occurrences of text.

Saving a file or “How to exit vi”

1. In vi press the *ESC*ape key to verify you are in command mode.
2. Depending on what you want to do press:

:w → write the file to disk

:wq → write the file to disk, then quit

:q → quit the file (only works if no changes)

:q! → quit and lose any changes made

:w! → override r/o file permission if you are owner or *root* and write the file to disk.

:wq! → override r/o file permission if you are owner or *root* and write the file to disk and quit.

For all commands which start with colon, you need to hit Enter at end

Speed-Up your config file editing!

1. In vi press the *ESC*ape key to verify you are in command mode.
2. To search for the first occurrence of something:
    - **`/string`** → press <ENTER>
    - **“n”** → press “n” for each following occurrence
    - **“N”** → press “N” for each previous occurrence
3. To replace *all* occurrences of a string in a file:
    - **`:%s/old_string/new_string/g`**

- To replace *all* occurrences of a string in a file (confirming each one):
    1. **`:%s/old_string/new_string/gc`**

Speed things up some more!

1. In vi press the *ESC*ape key to verify you are in command mode.
2. Go directly to a specific line number
   - **:*num*** → press <ENTER>. If *num*=99, go to line 99
3. Go to start/end of a line
   1. press *Home* or press *End* on your keyboard
4. Go to bottom of a file (in command mode):
   1. **press "G"**

- Undo the last change you made (in command mode)
  - **press “u”**

Configuration file patterns

There are patterns to how configuration files work:

- The most common comment character is “#”.
- In some files you'll see “/* .... */” or “//”.
- There are a few others, but they are less common.

Editing configuration files cont.

Some configuration files have lots of comments and few directives. Others are the opposite.

Blocks of configuration may be indicated in a programmatic manner, i.e.:

```
<VirtualHost *>
<SubSection>
directive
directive
</SubSection>
</VirtualHost>
```

Editing configuration files cont.

Another standard is to do the following:

```
## comment

## comment

# default setting=off
```

To change the default do:

```
default setting=on
```

Editing configuration files cont.

Things to watch out for:

- Spaces
- Quotes and single quotes: “directive” or 'directive'
- Caps or CamelCase syntax
  - Localhost=”myhost”
  - LocalHost=”myhost”
- Line end indicator, e.g. ":" or ";"
- New-line or continuation character “\”.

Other editors

ee

- ESC brings up the editor menu
- Cursors work as you expect

jed

- F10 brings up the editor menu
- Cursors work as you expect

joe

- Ctrl-k-h brings up the editor menu
- Ctrl-c aborts
- Cursors work as you expect

Conclusion

vi's most confusing feature is that it works in two modes and you must switch between them.

Questions?

Services

Startup scripts

In /etc/init.d/ (System V)
In /etc/init/ (Ubuntu 12.04 LTS and Upstart)

NOTE! Upon install services run!

Controlling services

- `update-rc.d` (default method)
- Stop/Start/Restart/Reload/Status Services

```
# service <Service> <Action>
```

or, “old school”

```
# /etc/init.d/<service> <action>
```

Runlevels

As Linux boots it executes service startup using links.

Based on your “runlevel” determines what services will start.

Traditional levels are used like this:

- runlevel 1: single user mode (emergency mode)
- runlevel 2: multi-user mode (No Desktop)
- runlevel 5: multi-user mode (Desktop)

With Ubuntu We *Actually* Do…

What happens at each runlevel?

- init 1 → Links in /etc/rc1.d are executed.
  Login as root user only.
  Minimal file system access.
- **init 2-5** → Links in /etc/rc5.d are executed.
  Gui is started if installed.
  Day-to-day working state.
  - Ubuntu runs at “runlevel 2”
  - Other Linux, with Desktop, run at “runlevel 5”
  - This is largely semantics

Special Runlevels

Runlevel 0: Halt the system

Runlevel 6: Reboot the system

Runlevel 1: Single user mode. No network. No services. System Recovery.

You must be at the machine console or have Out-of-Band (OoB) access to your machine to use Runlevel 1.

Packages vs. Source

Make and GCC

- Not installed by default. Why?
- 30,000'ish packages are available
- Install from source is “not clean” in the Ubuntu world.
- To install ability to compile C code:

```
# apt-get install build-essential
```

Root account Access

- Use of the *root* account is discouraged.
- *sudo* is used to access root privileges from general user account instead.
- You can get around this very easily.

Should you run as root?

Your decision.

Accessing *root* account

Set *root* user password:

- Login as general user
- *sudo -s* (Opens a root shell in bash)
- passwd (Set a root password)

Should you do this?

Security hole!

- Ubuntu allows *root* user access via SSH by default. Setting the *root* user password opens exposes this vulnerability.

See what's running

Check for a process by name

- `ps auxwww | grep apache`

```
sysadm@pc102:~$ ps auxwww | grep apache
root      1029  0.0 24.5 137524 125184 ?       Ss   01:29   0:02 /usr/sbin/apache2 -k start
www-data  1062  0.0 23.1 134788 117836 ?       S    01:29   0:00 /usr/sbin/apache2 -k start
www-data  1087  0.0 23.8 414236 121236 ?       Sl   01:29   0:00 /usr/sbin/apache2 -k start
www-data  1088  0.0 23.8 414236 121240 ?       Sl   01:29   0:00 /usr/sbin/apache2 -k start
sysadm    1426  0.0  0.1   3320   804 ttyS0    S+   02:40   0:00 grep --color=auto apache
```

Stop the process by PID (Process ID). From above listing:

- `sudo kill 1029` (why this one?)
- `Sudo kill -9 1029` (force stop if hung)

```
sysadm@pc102:~$ ps auxwww | grep apache
sysadm    1430  0.0  0.1   3320   808 ttyS0    S+   02:46   0:00 grep --color=auto apache
```

Questions

?