What is Open Source?

- Open source: software and source code available to all
  - The freedom to distribute software and source code
  - The ability to modify and create derived works
  - Integrity of author's code
- The Free Software Foundation and the Four Freedoms

Linux Origins

- 1984: The GNU Project and the Free Software Foundation
  - Creates open source version of UNIX utilities
  - Creates the General Public License (GPL)
    - Software license enforcing open source principles
- 1991: Linus Torvalds
  - Creates open source, UNIX-like kernel, released under the GPL
  - Ports some GNU utilities, solicits assistance online
- Today:
  - Linux kernel + GNU utilities = complete, open source, UNIX-like operating system
    - Packaged for targeted audiences as *distributions*

Red Hat Distributions

- Linux *distributions* are OSes based on the Linux kernel
- Red Hat Enterprise Linux
  - Stable, thoroughly tested software
  - Professional support services
  - Centralized management tools for large networks
- The Fedora Project
  - More, newer applications
  - Community supported (no official Red Hat support)
  - For personal systems

Linux principles

- Everything is a file (including hardware)
- Small, single-purpose programs
- Ability to chain programs together to perform complex tasks
- Avoid captive user interfaces
- Configuration data stored in text

Logging in to a Linux System

- Two types of login screens: virtual consoles (text-based) and graphical logins (called display managers)
- Login using login name and password
- Each user has a home directory for personal file storage

Switching between virtual consoles and the graphical environment

- A typical Linux system will run six virtual consoles and one graphical console
  - Server systems often have only virtual consoles
  - Desktops and workstations typically have both
- Switch among virtual consoles by typing: *Ctrl-Alt-F[1-6]*
- Access the graphical console by typing *Ctrl-Alt-F7*

Elements of the X Window System

- The X Window System is Linux's graphical subsystem
- Xorg is the particular version of the X Window System used by Red Hat
  - Open source implementation of X
- Look and behavior largely controlled by the desktop environment
- Two desktop environments provided by Red Hat:
  - GNOME: the default desktop environment
  - KDE: an alternate desktop environment

Starting the X server

- On some systems, the X server starts automatically at boot time
- Otherwise, if systems come up in virtual consoles, users must start the X server manually
  - The X server must be pre-configured by the system administrator
  - Log into a virtual console and run **startx**
  - The X server appears on *Ctrl-Alt-F7*

Changing Your Password

- Passwords control access to the system
  - Change the password the first time you log in
  - Change it regularly thereafter
  - Select a password that is hard to guess
- To change your password using GNOME, navigate to System->Preferences->About Me and then click **Password**.
- To change your password from a terminal: **passwd**

The *root* user

- The *root* user: a special administrative account
  - Also called the *superuser*
  - `root` has near complete control over the system
    - ...and a nearly unlimited capacity to damage it!
- Do not login as `root` unless necessary
  - Normal (*unprivileged*) users' potential to do damage is more limited

Changing Identities

- **su -** creates new shell as root
- **sudo** ***command*** runs ***command*** as root
  - Requires prior configuration by a system-administrator
- **id** shows information on the current user

Editing text files

- The **nano** editor
  - Easy to learn, easy to use
  - Not as feature-packed as some advanced editors
- Other editors:
  - **gedit**, a simple graphical editor
  - **vim**, an advanced, full feature editor
  - **gvim**, a graphical version of the vim editor

Some Simple Commands

- **date** - display date and time
- **cal** - display calendar

Getting Help

- Don't try to memorize everything!
- Many levels of help
  - **whatis**
  - ***command*** **--help**
  - **man** and **info**
  - `/usr/share/doc/`
  - Red Hat documentation

The whatis Command

- Displays short descriptions of commands
- Uses a database that is updated nightly
- Often not available immediately after install

```
$ whatis cal
cal             (1)   - displays a calendar
```

The --help Option

- Displays usage summary and argument list
- Used by most, but not all, commands

```
$ date --help
Usage: date [OPTION]... [+FORMAT] or:
date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]
Display the current time in the given FORMAT,
or set the system date.
...argument list omitted...
```

Reading Usage Summaries

- Printed by **--help**, **man** and others
- Used to describe the syntax of a command
  - Arguments in `[]` are optional
  - Arguments in CAPS or <> are variables
  - Text followed by `...` represents a list
  - `x|y|z` means "`x` or `y` or `z`"
  - `-abc` means "any mix of `-a`, `-b` or `-c`"

The man Command

- Provides documentation for commands
- Almost every command has a man "page"
- Pages are grouped into "chapters"
- Collectively referred to as the Linux Manual
- **man [<chapter>] <command>**

3-9

Almost every command (as well as most configuration files and several developer's libraries) on a Red Hat Enterprise Linux system has an associated man page, which provides more thorough documentation than the --help option. Man pages normally contain sections discussing the following aspects of a command's usage:

- Its NAME and a short description of what it does
- A SYNOPSIS of its usage, including available switches
- A longer DESCRIPTION of the command's functionality
- A switch-by-switch listing of its OPTIONS
- Any FILES associated with this command
- Any known BUGS in the command
- EXAMPLES, showing how to use the command
- A SEE ALSO section for further reference

The collection of all man pages on a system is called the Linux Manual. The Linux Manual is divided into sections, each of which covers a particular topic, and every man page is associated with exactly one of these sections. The sections are:

Manual sections

1	User commands	4	Special files	7	Miscellaneous
2	System calls	5	File formats	8	Administrative commands
3	Library calls	6	Games		

Often, Linux commands, calls, and files are referenced by a name followed by manual section number in parentheses. For example, `passwd(1)`, which is accessed by running **man 1 passwd**, refers to the user command **passwd**, whereas `passwd(5)`, accessed by running **man 5 passwd**, refers to the file format for `/etc/passwd`.

Navigating man Pages

- While viewing a man page
  - Navigate with arrows, *PgUp*, *PgDn*
  - **`/text`** searches for text
  - **`n`**/**`N`** goes to next/previous match
  - **`q`** quits
- Searching the Manual
  - **man -k *keyword*** lists all matching pages
  - Uses **whatis** database

The info Command

- Similar to **man**, but often more in-depth
- Run **info** without args to list all page
- **info** pages are structured like a web site
  - Each page is divided into "nodes"
  - Links to nodes are preceded by *
  - **info** [***command***]

Navigating info Pages

- While viewing an **info** page
  - Navigate with arrows, *PgUp*, *PgDn*
  - ***Tab*** moves to next link
  - `Enter` follows the selected link
  - `n`/`p` /`u` goes to the next/previous/up-one node
  - `s` ***`text`*** searches for text (default: last search)
  - `q` quits **info**

Extended Documentation

- The `/usr/share/doc` directory
  - Subdirectories for most installed packages
  - Location of docs that do not fit elsewhere
    - Example configuration files
    - HTML/PDF/PS documentation
    - License details

Linux File Hierarchy Concepts

- Files and directories are organized into a single-rooted inverted tree structure
- Filesystem begins at the *root* directory, represented by a lone / (forward slash) character.
- Names are case-sensitive
- Paths are delimited by /

Filesystem Basics

These Linux file hierarchy concepts will be expanded upon in the pages that follow.

- Files and directories are organized into a single-rooted inverted-tree structure, including distinct physical volumes such as floppy disks, CD-ROMs and multiple hard drives.
- The base of the inverted-tree hierarchy is known as *root* or / - the top of the file structure.
- A forward slash separates elements of a pathname, for example `/usr/bin/X11/X`
- Names in the Linux file hierarchy are case-sensitive.
- Each shell and process on the system has a designated *current* or *working* directory.
- `..` refers to the parent directory of any particular directory - one level up in the file hierarchy.
- `.` refers to the current directory.
- Files and directories whose names begin with a `.` are *hidden* -- that is, they are not displayed by default in filename listings.
- A user's *path* is a list of directories that are searched for commands typed at the command line.

Some Important Directories

- Home Directories: `/root`,`/home/`*username*
- User Executables: `/bin`, `/usr/bin`, `/usr/local/bin`
- System Executables: `/sbin`, `/usr/sbin`, `/usr/local/sbin`
- Other Mountpoints: `/media`, `/mnt`
- Configuration: `/etc`
- Temporary Files: `/tmp`
- Kernels and Bootloader: `/boot`
- Server Data: `/var`, `/srv`
- System Information: `/proc`, `/sys`
- Shared Libraries: `/lib`, `/usr/lib`, `/usr/local/lib`

Current Working Directory

- Each shell and system process has a *current working directory*(cwd)
- **pwd**
  - Displays the absolute path to the shell's cwd

File and Directory Names

- Names may be up to 255 characters
- All characters are valid, except the forward-slash
  - It may be unwise to use certain special characters in file or directory names
  - Some characters should be protected with quotes when referencing them
- Names are case-sensitive
  - Example: `MAIL`, `Mail`, `mail`, and `mAiL`
  - Again, possible, but may not be wise

redhat. 4-6

Filenames

Using the default filesystem, file names may be up to 255 characters (different restrictions may apply, depending on the particular configuration of your system) .

File names generally consist of letters of the alphabet, numbers, and certain punctuation marks. All other characters, except the / character, are valid, but it is often unwise to use certain special characters in file names. Among the characters to avoid are: `> < ? * "` and quotation marks, as well as spaces, tabs and other non-printable characters

To access a file whose name contains special characters, enclose the filename in quotes. For example:

```
[student@stationX ~]$ ls -l "file name with spaces.txt"
-rw-rw-r--  1 student student 0 Dec 14 21:48 file name with spaces.txt
```

Absent the quotes, you would be asking the system to list four different files.

File names are case-sensitive. This means that `FILE` is different from `file` and `File`. Again, although it is possible to create these files, it may be unwise to do so, as it may confuse you later.

Absolute and Relative Pathnames

- Absolute pathnames
  - Begin with a forward slash
  - Complete "road map" to file location
  - Can be used anytime you wish to specify a file name
- Relative pathnames
  - Do not begin with a slash
  - Specify location relative to your current working directory
  - Can be used as a shorter way to specify a file name

The location of a directory or file can be specified by either of two methods: by its *absolute* pathname or its *relative* pathname.

An absolute pathname begins with a slash (/). It contains the name of each directory that must be traversed from the root file system, in order, to reach the object being named, for example: `/usr/share/doc/HTML/index.html`

The file we are referencing, `index.html`, is contained within a directory named `HTML`, which is in turn contained in a directory named doc, which is contained in the directory share, which is contained in directory usr, which is contained in the root (/) directory.

The absolute pathname specifies a 'road map' from the root of the file tree to its location in the file system. This 'road map' is valid regardless of the current directory.

A relative pathname does not begin with a slash. It contains the name of each directory that must be traversed from the current directory to reach the object being named. The first component of the pathname *must* exist in the current directory for the pathname to the object to be valid. A filename by itself is a relative pathname; that is, the file must be in the current directory for its reference to be valid.

The special directory name `..` refers to the parent of the current directory, and can be used as part of a pathname.

Some examples of relative pathnames, relative to particular directories, follow. In each case, the file being referenced is `/usr/share/doc/HTML/index.html`.

Current Directory	Relative Path to `index.html`
/usr/share/doc/HTML/	index.html
/usr/share/doc/	HTML/index.html
/usr/share/	doc/HTML/index.html
/usr/	share/doc/HTML/index.html
/	usr/share/doc/HTML/index.html
/usr/share/doc/HTML/en/	../index.html
/usr/share/doc/nautilus-2.1.91/	../HTML/index.html

Changing Directories

- **cd** changes directories
  - To an absolute or relative path:
    - **cd** `/home/joshua/work`
    - **cd** `project/docs`
  - To a directory one level up:
    - **cd** . .
  - To your home directory:
    - **cd**
  - To your previous working directory:
    - **cd** -

Listing Directory Contents

- Lists the contents of the current directory or a specified directory
- Usage:
  - **ls [options]** [***files_or_dirs***]
- Example:
  - **ls -a** (include hidden files)
  - **ls -l** (display extra information)
  - **ls -R** (recurse through directories)
  - **ls -ld** (directory and symlink information)

Copying Files and Directories

- **cp** - copy files and directories
- Usage:
  - **cp [options]** ***file destination***
- More than one file may be copied at a time if the destination is a directory:
  - **cp [options]** ***file1 file2 dest***

Copying Files and Directories: The Destination

- If the destination is a directory, the copy is placed there
- If the destination is a file, the copy overwrites the destination
- If the destination does not exist, the copy is renamed

Moving and Renaming Files and Directories

- **mv** - move and/or rename files and directories
- Usage:
  - **mv [options]** ***file destination***
- More than one file may be moved at a time if the destination is a directory:
  - **mv [options]** ***file1 file2 destination***
- Destination works like **cp**

Creating and Removing Files

- **touch** - create empty files or update file timestamps
- **rm** - remove files
- Usage:
  - **rm [options] <*file*>...**
- Example:
  - **rm -i *file*** (interactive)
  - **rm -r *directory*** (recursive)
  - **rm -f *file*** (force)

Creating and Removing Directories

- **mkdir** creates directories
- **rmdir** removes empty directories
- **rm -r** recursively removes directory trees

Determining File Content

- Files can contain many types of data
- Check file type with file before opening to determine appropriate command or application to use
- **file [options] <filename>...**

Users

- Every user is assigned a unique User ID number (*UID*)
  - UID 0 identifies root
- Users' names and UIDs are stored in `/etc/passwd`
- Users are assigned a home directory and a program that is run when they log in (usually a shell)
- Users cannot read, write or execute each others' files without permission

Groups

- Users are assigned to groups
- Each group is assigned a unique Group ID number (*gid*)
- GIDs are stored in `/etc/group`
- Each user is given their own private group
  - Can be added to other groups for additional access
- All users in a group can share files that belong to the group

Linux File Security

- Every file is owned by a UID and a GID
- Every process runs as a UID and one or more GIDs
  - Usually determined by who runs the process
- Three access categories:
  - Processes running with the same UID as the file (*user*)
  - Processes running with the same GID as the file (*group*)
  - All other processes (*other*)

Permission Precedence

- If UID matches, *user* permissions apply
- Otherwise, if GID matches, *group* permissions apply
- If neither match, *other* permissions apply

Permission Types

- Four symbols are used when displaying permissions:
  - `r`: permission to read a file or list a directory's contents
  - `w`: permission to write to a file or create and remove files from a directory
  - `x`: permission to execute a program or change into a directory and do a long listing of the directory
  - `-`: no permission (in place of the `r`, `w`, or `x`)

Examining Permissions

- File permissions may be viewed using **ls -l**

```
$ ls -l /bin/login
-rwxr-xr-x 1 root root 19080 Apr 1 18:26 /bin/login
```

- File type and permissions represented by a 10-character string

Interpreting Permissions

```
-rwxr-x--- 1 andersen trusted 2948 Oct 11 14:07 myscript
```

- Read, Write and Execute for the owner, `andersen`
- Read and Execute for members of the `trusted` group
- No access for all others

Changing File Ownership

- Only root can change a file's owner
- Only root or the owner can change a file's group
- Ownership is changed with **chown**:
  - **chown [-R]** ***user_name file|directory***
- Group-Ownership is changed with **chgrp**:
  - **chgrp [-R]** ***group_name file|directory***

Changing Permissions - Symbolic Method

- To change access modes:
  - **chmod [-R]** ***mode file***
- Where *mode* is:
  - **u**,**g** or **o** for user, group and other
  - **+** or **-** for grant or deny
  - **r**, **w** or **x** for read, write and execute
- Examples:
  - **ugo+r**: Grant read access to all
  - **o-wx**: Deny write and execute to others

chmod Syntax

The **chmod** command changes access mode for files and directories. The **chmod** command takes a permission instruction followed by a list of files or directories to change. The permission instruction can be issued either symbolically (the symbolic method) or numerically (the numeric method).

Using the symbolic method, the permission expression contains three fields: an indicator of *who* has access to the file, an *operator* for selecting how the permissions should be changed, and a *permission*. If no who value is given, then the permission is added or removed for user, group and other. Multiple, comma separated operations can be given in a single command.

who may be	*operator* may be	*permissions* may be
`u` User who owns the file	+ Add a permission	`r` Read
`g` Users in the file's Group	- Remove a permission	`w` Write
`o` Other users	= Assign a permission	`x` Execute or cd
`a` All three categories		`s` Set user ID bit or group
		`t` Sticky bit (for directories)

Examples:

chmod u+w,go-w .bashrc grants write access to owner but denies it to group and other.

chmod u=rw .bashrc sets user permissions to read and write, with execute turned off, regardless of the current permissions

chmod +r .bashrc makes the file world-readable

A useful option to **chmod** is **-R** (recursive). This option tells **chmod** to traverse an entire directory tree to change the permissions of all its files and subdirectories. The **s** and **t** permissions will be discussed in a later unit.

Changing Permissions - Numeric Method

- Uses a three-digit mode number
  - first digit specifies owner's permissions
  - second digit specifies group permissions
  - third digit represents others' permissions
- Permissions are calculated by adding:
  - `4` (for read)
  - `2` (for write)
  - `1` (for execute)
- Example:
  - **chmod 640 myfile**

Command Line Shortcuts
File Globbing

- Globbing is wildcard expansion:
  - `*` - matches zero or more characters
  - `?` - matches any single character
  - `[0-9]` - matches a range of numbers
  - `[abc]` - matches any of the character in the list
  - `[^abc]` - matches all except the characters in the list
  - Predefined character classes can be used

Command Line Shortcuts
The *Tab* Key

- Type *Tab* to complete command lines:
  - For the command name, it will complete a command name
  - For an argument, it will complete a file name
- Examples:

```
$ xte<Tab>
$ xterm
$ ls myf<Tab>
$ ls myfile.txt
```

Command Line Shortcuts
History

- **bash** stores a history of commands you've entered, which can be used to repeat commands
- Use **history** command to see list of "remembered" commands

```
$ history
14   cd /tmp
15   ls -l
16   cd
17   cp /etc/passwd .
18   vi passwd
... output truncated ...
```

History

In addition to the basic command recall with the arrow keys, the bash history mechanism supports a variety of advanced ways of retrieving commands from the history list.

!!	repeats last command
!*char*	repeats last command that started with *char*
!*num*	repeats a command by its number in history output

To view past commands with the history numbers, use the **history** command.

Other slightly more advanced history tricks include:

!?abc	repeats last command that contains (as opposed to ?started with?) abc
!-n	repeats a command entered n commands back

Use **^old^new** to repeat the last command with old changed to new, for example:

```
[student@stationX ~]$ cp filter.c /usr/local/src/project
[student@stationX ~]$ ^filter^frontend
cp frontend.c /usr/local/src/project
```

More History Tricks

- Use the *up* and *down* keys to scroll through previous commands
- Type *Ctrl-r* to search for a command in command history.
  - *(reverse-i-search)`':*
- To recall last argument from previous command:
  - *Esc,.* (the escape key followed by a period)
  - *Alt-.* (hold down the alt key while pressing the period)

6-6

History

Using your history is a great productivity-enhancing tool. Linux users who develop a habit of using their history can streamline and speed their use of the shell. Try playing with the keystrokes listed above.

You can ignore repeated duplicate commands and repeated lines that only differ in prepended spaces by adding the following to your `.bashrc`.

```
[student@stationX ~] export HISTCONTROL=ignoreboth
```

Command Line Expansion
The tilde

- Tilde (~)
- May refer to your home directory

```
$ cat ~/.bash_profile
```

- May refer to another user's home directory

```
$ ls ~julie/public_html
```

Command Line Expansion
Commands and Braced Sets

- Command Expansion: `$()` or ``` ``
  - Prints output of one command as an argument to another

```
$ echo "This system's name is $(hostname)"
This system's name is server1.example.com
```

- Brace Expansion: `{ }`
  - Shorthand for printing repetitive strings

```
$ echo file{1,3,5}
file1 file3 file5
$ rm -f file{1,3,5}
```

The bash shell has some special features relating to expansion which significantly improve the power of working at the command line.

Curly braces are useful for generating patterned strings. For example without the curly braces, the **mkdir** command below would take almost two hundred keystrokes to execute.

```
[student@stationX ~]$ mkdir -p ↙
work/{inbox,outbox,pending}/{normal,urgent,important}
```

Use of the backquotes is called command substitution. In command substitution, the command in backquotes is executed and the output of the command is placed on the command line as if the user had typed it in. An alternative syntax for the backquotes is to place the command in parentheses preceded by a dollar sign $( ).

Command Editing Tricks

- *Ctrl-a* moves to beginning of line
- *Ctrl-e* moves to end of line
- *Ctrl-u* deletes to beginning of line
- *Ctrl-k* deletes to end of line
- *Ctrl-*`arrow` moves left or right by word

gnome-terminal

- Applications->Accessories->Terminal
- Graphical terminal emulator that supports multiple "tabbed" shells
  - *Ctrl-Shift-t* creates a new tab
  - *Ctrl-PgUp/PgDn* switches to next/prev tab
  - *Ctrl-Shift-c* copies selected text
  - *Ctrl-Shift-v* pastes text to the prompt

Standard Input and Output

- Linux provides three I/O channels to Programs
  - Standard input (STDIN) - keyboard by default
  - Standard output (STDOUT) - terminal window by default
  - Standard error (STDERR) - terminal window by default

Redirecting Output to a File

- STDOUT and STDERR can be redirected to files:
  - *command operator filename*
- Supported operators include:
  - > Redirect STDOUT to file
  - 2> Redirect STDERR to file
  - &> Redirect all output to file
- File contents are overwritten by default. >> appends.

Redirecting Output to a File
Examples

- This command generates output and errors when run as non-root:

```
$ find /etc -name passwd
```

- Operators can be used to store output and errors:

```
$ find /etc -name passwd > find.out

$ find /etc -name passwd 2> /dev/null

$ find /etc -name passwd > find.out 2> find.err
```

Redirecting STDOUT to a Program (Piping)

- Pipes (the | character) can connect commands:

  command1 | *command2*

  - Sends STDOUT of command1 to STDIN of command2 instead of the screen.
  - STDERR is *not* forwarded across pipes
- Used to combine the functionality of multiple tools
  - *command1* | *command2* | *command3*... etc

Redirecting STDOUT to a Program
Examples

- **less**: View input one page at a time:

```
$ ls -l /etc | less
```

  - Input can be searched with /

- **mail**: Send input via email:

```
$ echo "test email" | mail -s "test" user@example.com
```

- **lpr** : Send input to a printer

```
$ echo "test print" | lpr
```

```
$ echo "test print" | lpr -P printer_name
```

Combining Output and Errors

- Some operators affect both STDOUT and STDERR
  - &>: Redirects all output:

```
$ find /etc -name passwd &> find.all
```

  - 2>&1: Redirects STDERR to STDOUT
    - Useful for sending all output through a pipe

```
$ find /etc -name passwd 2>&1 | less
```

  - (): Combines STDOUTs of multiple programs

```
$ ( cal 2007 ; cal 2008 ) | less
```

Redirecting all output

Suppose you wanted to redirect both STDOUT and STDERR, to the same file. You might first try running a command like **find /etc -name passwd > find.all 2> find.all**, but this will not have quite the results that you expect. Depending on the circumstances, you may only find STDOUT messages in the file or you may find all the STDOUT messages together and all the STDERR messages together, even if **find** printed them in a different order. The proper way to redirect all output from a command is using the **&>** operator:

```
[student@stationX ~]$ find /etc -name passwd &> find.all
```

Redirecting I/O Channels to Each Other

Try running a command like **find /etc -name passwd | less** as a non-root user. You will find that while STDOUT is displayed through **less**, STDERR is not. This is because a pipe only redirects STDOUT. If you wanted to send all output to **less**, you would need to redirect STDERR to STDOUT first, then forward STDOUT over the pipe.

You can redirect one I/O channel to another using **>** and the channels' file descriptor numbers. But, for example, consider what would happen if you simply redirected STDERR to "1". This would redirect to a file called "1" rather than to file descriptor 1 (STDOUT). To tell your shell that you are referring to a file descriptor, prepend the **&** character to the destination number:

```
[student@stationX ~]$ find /etc -name passwd &2>&1 | less
```

This will cause **find**'s STDERR (descriptor 2) to be redirected to its STDOUT (descriptor 1) before STDOUT is forwarded to **less** via the pipe.

Combining Outputs

Suppose you wanted to run two commands back-to-back and send their output through a pipe. For example, perhaps you would like to generate a calendar for the years 2007 and 2008. This can be done by running the commands **cal**

2007 and **cal 2008**. However, if you tried to run **cal 2007 ; cal 2008 | lpr** to print the output of these commands, you would find that only the calendar for 2008 was printed, while the calendar for 2007 went to the screen. This is because a pipe only affects the command immediately to its left. This can be overcome by running the **cal** commands in a *subshell*. This causes your shell to create a separate instance of its self, run the commands and return their output as if they were a single program. To indicate that you want commands to run in a subshell, simply place them in parentheses:

```
[student@stationX ~]$ (cal 2007 ; cal 2008) | lpr
```

Redirecting to Multiple Targets (tee)

- `$ command1 | tee filename | command2`
- Stores STDOUT of *command1* in *filename*, then pipes to *command2*
- Uses:
  - Troubleshooting complex pipelines
  - Simultaneous viewing and logging of output

redhat 7-9

Uses of tee

tee is useful for saving output at various stages in a long sequence of pipes. For example:

```
[student@stationX ~]$ ls -lR /etc | tee stage1.out | sort | tee stage2.out | ↙
 uniq -c | tee stage3.out | sort -r | tee stage4.out | less
```

It is also useful for commands that take a long time to run or print a large amount of text. An administrator might want to be able to check in on the operation from time to time and watch the command's output, but would also like to have a record of the output stored in a file. Using normal output redirection, the administrator could dump to a file, but not view output as the program ran. Hence, **tee** can be useful even when it is the last item in a pipeline:

```
[student@stationX ~]$ generate_report.sh | tee report.txt
```

Redirecting STDIN from a File

- Redirect standard input with <
- Some commands can accept data redirected to STDIN from a file:

```
$ tr 'A-Z' 'a-z' < .bash_profile
```

  - This command will translate the uppercase characters in `.bash_profile` to lowercase
- Equivalent to:

```
$ cat .bash_profile | tr 'A-Z' 'a-z'
```

Sending Multiple Lines to STDIN

- Redirect multiple lines from keyboard to STDIN with ***<<WORD***
  - All text until ***WORD*** is sent to STDIN
  - Sometimes called a *heretext*

```
$ mail -s "Please Call" jane@example.com <<END
> Hi Jane,
>
> Please give me a call when you get in. We may need
> to do some maintenance on server1.
>
> Details when you're on-site,
> Boris
> END
```

redhat 7-11

Using Heretexts

Normally when a program reads data from STDIN it only takes a single line and then assumes that input is over. A *heretext* is a simple technique that causes input to terminate only when a given word is encountered. This is useful for when multiple lines of input need to be sent to a program, as in the above slide's example, which allows the system administrator to send a quick email without even opening a client directly.

Tools for Extracting Text

- File Contents: **less** and **cat**
- File Excerpts: **head** and **tail**
- Extract by Column: **cut**
- Extract by Keyword: **grep**

Viewing File Contents
less and cat

- **cat**: dump one or more files to STDOUT
  - Multiple files are con*cat*enated together
- **less**: view file or STDIN one page at a time
  - Useful commands while viewing:
    - **/text** searches for *text*
    - **n/N** jumps to the next/previous match
    - **v** opens the file in a text editor
  - **less** is the pager used by **man**


redhat 8-4

Dumping files with ***cat***

cat opens the files given as its arguments and displays their contents to the terminal.

To see how **cat** works, try running **cat /etc/profile**

Unless you can read very fast, you probably noticed a problem with the output scrolling off the top of the page too quickly. **cat** is most useful for viewing short files; other commands, such as less, are more suited to viewing larger files.

If you dump the content of a binary file with **cat** to a terminal, you will make it unusable. You can use **reset** to clean up your garbled terminal and go on with it. When you type **reset**, it won't be correctly echo-ed. It may also take a couple of seconds before your terminal is reset.

Some useful options to use with **cat**:

-A	Show all characters, including control characters and non-printing characters
-s	"Squeeze" multiple adjacent blank lines into a single blank line
-b	Number each (non-blank) line of output

Navigating text with ***less***

Space	moves ahead one full screen
b	moves back one full screen
Enter	moves ahead one line
k	moves back one line
g	moves to the top of the file
G	moves to the bottom of the file
/text	searches for text
n	repeats the last search
N	repeats last search, but in the opposite direction
q	quits
v	opens the file in (**vi** by default)

Viewing File Excerpts
head and tail

- **head**: Display the first 10 lines of a file
  - Use **-n** to change number of lines displayed
- **tail**: Display the last 10 lines of a file
  - Use **-n** to change number of lines displayed
  - Use **-f** to "follow" subsequent additions to the file
    - Very useful for monitoring log files!

The head command is used to display just the first few lines of a file. The default is 10 lines.

```
[student@stationX ~]$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/adm:
lp:x:4:7:lp:/var/spool/lpd:
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:
news:x:9:13:news:/var/spool/news:
```

-n specifies the number of lines to display:

```
[student@stationX ~]$ head -n 3 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
```

tail is used to display the last few lines of a file. The default is 10. **tail** is often used by the system administrator to read the most recent entries in log files.

```
[root@stationX ~]# tail -n 3 /var/log/cron
root (10/13-18:01:00-658) CMD (run-parts /etc/cron.hourly)
CRON (10/15-13:45:56-422) STARTUP (fork ok)
root (10/15-13:50:00-781) CMD (/sbin/rmmod-as)
```

Using **-f** causes **tail** to continue to display the file in "real time", showing additions to the end of the file as they occur. This is very useful for watching growing files, such as the output of the make command. System administrators use this feature to keep an eye on the system log using the following command:

```
[root@stationX ~]# tail -f /var/log/messages
```

tail -f will continue to show updates to the file until *Ctrl-c* is pressed.

Extracting Text by Keyword
grep

- Prints lines of files or STDIN where a pattern is matched

```
$ grep 'john' /etc/passwd

$ date --help | grep year
```

- Use **-i** to search case-insensitively
- Use **-n** to print line numbers of matches
- Use **-v** to print lines *not* containing pattern
- Use **-A*x*** to include the *x* lines after each match
- Use **-B*x*** to include the *x* lines before each match

redhat 8-6

grep displays the lines in a file that match a pattern. It can also process standard input. The pattern may contain regular expression metacharacters and so it is considered good practice to always quote your regular expressions. Examples:

To list lines containing either "Cat" or "cat" from the `pets` file

```
[student@stationX ~]$ grep '[Cc]at' pets
```

To list only lines of output from **ps**, which lists running processes, that contain the string "init": from the ps command

```
[student@stationX ~]$ ps ax | grep 'init'
```

Common **grep** options include:

-v	return lines that do *not* contain the pattern
-n	precede returned lines with line numbers
-c	only return a count of lines with the matching pattern
-l	only return the names of files that have at least one line containing the pattern
-r	perform a recursive search of files, starting with the named directory
-i	perform a case-insensitive search
-B, -A	When followed by a number, these options print that many lines before or after each match. This is useful for seeing the context in which a match appears within a file.

Extracting Text by Column
cut

- Display specific columns of file or STDIN data

```
$ cut -d: -f1 /etc/passwd
```

```
$ grep root /etc/passwd | cut -d: -f7
```

- Use **-d** to specify the column delimiter (default is TAB)
- Use **-f** to specify the column to print
- Use **-c** to cut by characters

```
$ cut -c2-5 /usr/share/dict/words
```

8-7

cut is used to "cut" fields or columns of text from a file and display it to standard output.

For example:

```
[student@stationX ~]$ cut -f3 -d: /etc/passwd
```

will display a list of UIDs from `/etc/passwd`, because UIDs are stored in the third colon-delimited field.

A slightly more complicated example shows how **cut** can be used with other tools to extract a single piece of information, in this case the system's IP addresses, from a larger block of output. After examining the command and output below, try running **ifconfig** by its self and see if you can piece together what each step of the pipeline does:

```
$  ifconfig | grep 'inet addr' | cut -d: -f2 | cut -d' ' -f1
192.168.0.254
127.0.0.1
```

Tools for Analyzing Text

- Text Stats: **wc**
- Sorting Text: **sort**
- Comparing Files: **diff** and **patch**
- Spell Check: **aspell**

Gathering Text Statistics
wc (word count)

- Counts words, lines, bytes and characters
- Can act upon a file or STDIN

```
$ wc story.txt
39      237     1901 story.txt
```

- Use **-l** for only line count
- Use **-w** for only word count
- Use **-c** for only byte count
- Use **-m** for character count (not displayed)

Sorting Text
sort

- Sorts text to STDOUT - original file unchanged

```
$ sort [options] file(s)
```

- Common options
  - **-r** performs a reverse (descending) sort
  - **-n** performs a numeric sort
  - **-f** ignores (folds) case of characters in strings
  - **-u** (unique) removes duplicate lines in output
  - **-t *c*** uses *c* as a field separator
  - **-k *x*** sorts by *c*-delimited field *X*
    - Can be used multiple times

sort is used to sort text data. This data can be in a file or the output of another command. **sort** is often used with pipes as in the example below which displays an alphabetical list of users whose login shell is set to **bash**:

```
[student@stationX ~]$ grep bash /etc/passwd | sort
alex:x:503:504::/home/alex:/bin/bash
gdm:x:42:42::/home/gdm:/bin/bash
joshua:x:500:500:Joshua M. Hoffman:/home/joshua:/bin/bash
root:x:0:0:root:/root:/bin/bash
star:x:504:505::/home/star:/bin/bash
```

The **-k** option sets the sort field. The following command will sort the `/etc/passwd` file by the UID number:

```
[student@stationX ~]$ sort -t : -k 3 -n /etc/passwd
```

The argument to the **-k** option can be two numbers separated by a dot. In this case, the number before the dot is the field number and the number after the dot is the character within that field with which to begin the sort.

The **-n** option sorts numerically, instead of by character. Without the **-n** option, the numbers 71, 12, and 9 would sort as 12, 71, and 9, whereas with the **-n** option, they will sort as 9, 12, and 71.

Eliminating Duplicate Lines
sort and uniq

- **sort -u**: removes duplicate lines from input
- **uniq**: removes duplicate *adjacent* lines from input
  - Use **-c** to count number of occurrences
  - Use with **sort** for best effect:

```
$ sort userlist.txt | uniq -c
```

redhat 8-11

uniq "removes" duplicate adjacent lines from a file. To print only unique line occurrences in a file ("removing" all duplicate lines), input to uniq must first be sorted. Since uniq can be given fields or columns on which to base its decisions, these are the fields or columns upon which its input must be sorted.

Used without switches, uniq removes duplicate lines in its input, using the entire record as a decision key.

Use **-u** to output only the lines that are truly unique - only occurring once in the input.

Use **-d** to output only print one copy of the lines that are repeated in the input.

Use **-c** to produce a frequency listing. Each line will be prepended with a number indicating how many times it appears in the input.

Use **-fn** or **-sn** to avoid comparing the first n fields or characters in each line, respectively.

The following example uses sort and uniq to list the shells used in `/etc/passwd`:

```
[student@stationX ~]$ cut -d: -f7 /etc/passwd | sort | uniq
/bin/bash
/bin/false
/bin/sync
/dev/null
/sbin/halt
/sbin/nologin
/sbin/shutdown
```

Comparing Files
diff

- Compares two files for differences

```
$ diff foo.conf-broken foo.conf-works
5c5
<    use_widgets = no
---
>    use_widgets = yes
```

  - Denotes a difference (change) on line 5
- Use **gvimdiff** for graphical **diff**
  - Provided by `vim-X11` package

diff is used to compare the contents of two files for differences. Suppose that a service on station1 is malfunctioning but the same service works on station20. Thanks to **diff** and the use of simple, text-based configuration files, we can easily compare the working and non-working configurations:

```
$ diff file.conf-station1 file.conf-station20
1c1
< Hostname = station1
---
> Hostname = station20
2c2
< Setting1 = a
---
> Setting1 = A
3a4
> Setting2 = B
5d5
< Setting4 = D
```

diff shows us that there are four places where the configurations differ:

- On line 1, the `Hostname` variable is different in each file. Given that the files are from different systems, this makes sense and is probably not the problem.
- On line 2, the `Setting1` variable is set to `a` on station1 and `A` on station20. This could cause problems if the service is case-sensitive.
- Setting2 is set on station20, but not on station1. In other words, if you were to convert the station1 version of the file to the station20 version of the file you would have to *a*dd Setting2 at line 2, which would become line 3 of the file.
- Setting4 is set on station1, but not on station20. In other words, you would need to *d*elete line 4 in the station1 file to make it like the station20 file.

Duplicating File Changes
patch

- **diff** output stored in a file is called a "patchfile"
  - Use **-u** for "unified" diff, best in patchfiles
- **patch** duplicates changes in other files (use with care!)
  - Use **-b** to automatically back up changed files

```
$ diff -u foo.conf-broken foo.conf-works > foo.patch
$ patch -b foo.conf-broken foo.patch
```

While the default output of **diff** can be easier to read, an alternate output format called a "unified diff" is also available. This format includes information that helps the **patch** utility more accurately apply the changes described. The result of this is that if you have several similar files that all need the same changes made, you can make the change once, store a unified diff comparing the changed file to an unchanged file and then use **patch** to apply your changes to as many files as you need.

```
$ diff -u file.conf-station1 file.conf-station20
1c1
--- file.conf-station1        2007-01-03 18:36:36.000000000 -0800
+++ ffile.conf-station20      2007-01-02 21:10:30.000000000 -0800
@@ -1,4 +1,4 @@
-Hostname = station1
-Setting1 = a
+Hostname = station20
+Setting1 = A
+Setting2 = B
 Setting3 = C
-Setting4 = D
```

The details of the differences between these two files has already been discussed. The unified diff is just an alternate way of displaying the same information. The first three lines describe details of the files the patch was taken from and where the changes start. Lines that begin with + exist in station1's configuration, but not in station20's. Lines that begin with - exist on station20, but not station1. Lines with no special character exist in both and are just there to provide "context" so programs **patch** can more accurately decide where the changes go.

To use **patch**, simply store the output of a **diff -u** in a file and run a command like the following, which would make `file.conf-station1` look like `file.conf-station20`:

```
$ patch -b file.conf-station1 file.conf.patch
```

But *beware*! Do you actually want all of the changes above to be made? It would be advisable to first edit `file.conf.patch` and remove the two lines describing the `Hostname` variable, since those should remain different between systems. If anything terrible happens, **patch -b** automatically creates a backup of each file it changes. Backups are given the `.orig` extension.

Spell Checking with aspell

- Interactively spell-check files:

```
$ aspell check letter.txt
```

- Non-interactively list mis-spelled words in STDIN

```
$ aspell list < letter.txt

$ aspell list < letter.txt | wc -l
```

redhat 8-14

aspell is an interactive spell checker. It offers suggestions for corrections via a simple menu-driven interface.

```
[student@stationX ~]$ aspell check file.txt
Some times *peple* type stuff wrong.
1) people                              6) peel
2) Pele                                7) Pelee
3) Peale                               8) peopled
4) purple                              9) peoples
5) Peel                                0) pep
i) Ignore                              I) Ignore all
r) Replace                             R) Replace all
a) Add                                 x) Exit
?
```

aspell list will non-interactively list the misspelled words in a file read from standard input.

```
[student@stationX ~]$ aspell list < standfast.txt
Carcrashes
Morningcharm
Braincheck
```

More information on aspell can be found at http://aspell.sourceforge.net.

A quick spelling dictionary look-up can be performed with the **look** command. It comes in handy when you need the spelling of a word of which you know the first few letters.

```
[student@stationX ~]$ look exer
exercise
exercised
exerciser
exercisers
exercises
exercising
exert
... output truncated ...
```

Tools for Manipulating Text
tr and sed

- Alter (translate) Characters: **tr**
  - Converts characters in one set to corresponding characters in another set
  - Only reads data from STDIN

```
$ tr 'a-z' 'A-Z' < lowercase.txt
```

- Alter Strings: **sed**
  - **s**tream **ed**itor
  - Performs search/replace operations on a stream of text
  - Normally does not alter source file
  - Use **-i.bak** to back-up and alter source file

redhat 8-15

tr is used to translate characters; that is, given two ranges of characters, any time a character in range 1 is found, it is translated into the equivalent character in range 2. This command is commonly used in shell scripts to ensure that data is in an expected case:

```
echo -n "Enter yes or no: "
read answer
answer="$(echo $answer | tr 'A-Z' 'a-z')"
```

In this example, the user is queried for data. If the user responds in lower case, the **tr** command will do nothing, but if the user responds in upper case, the characters will be changed to lower case.

The **sed** command is the stream editor, used to perform edits on a stream of data. Given a file name to process, sed will perform a search and replace on all lines in the file, sending the modified data to standard output; that is, it does not actually modify the existing file. As with grep, sed is often used in pipelines. A basic **sed** command may look like this:

```
[student@stationX ~]$ sed 's/cat/dog/' pets
```

In this example, the pets file will be sent to standard output with the string `cat` being replaced by the string `dog`. By default, **sed** makes a maximum of one change per line. To instruct sed to make multiple changes per line, the `g` command, standing for global, should be appended to the end of the search and replace pattern. It should also be noted that **sed** searches are case-sensitive by default. To change this, append an `i` to the pattern.

```
[student@stationX ~]$ sed 's/cat/dog/gi' pets
```

sed
Examples

- Quote search and replace instructions!
- **sed** addresses
  - **sed 's/dog/cat/g' pets**
  - **sed '1,50s/dog/cat/g' pets**
  - **sed '/digby/,/duncan/s/dog/cat/g' pets**
- Multiple **sed** instructions
  - **sed -e 's/dog/cat/' -e 's/hi/lo/' pets**
  - **sed -f myedits pets**

As with **grep**, it is good practice to quote **sed**'s search and replace string, as demonstrated in the examples below.

By default, **sed** operates on all lines in a file. It is possible to provide sed with addresses limiting replacements to just those lines. For example:

```
[student@stationX ~]$ sed '10,35s/cat/dog/' pets
```

In this example, the entire pets file will be sent to standard output, but the replacement of "cat" for "dog" will only be performed on lines 10 through 35, inclusive.

A fancy but seldom used feature of sed is that addressing can be done by string searches. For example:

```
[student@stationX ~]$ sed '/digby/,/duncan/s/cat/dog/' pets
```

In this example, starting on the line that contains the string "digby" and continuing through the line that contains "duncan", the string substitution of "dog" for "cat" will be performed on the pets file. As before, the entire file will be sent to standard output, but changes will be performed only on the specified lines.

It is common to make several changes on a file. Two different methods are provided by **sed** to perform multiple edits. For a small number of edits, the **-e** option indicates that a search and replace pattern is forthcoming, and can be used several times on a command line:

```
[student@stationX ~]$ sed -e 's/cat/dog/g' -e 's/cow/goat/g' pets
```

For a larger number of edits, or to save edits for the future, place them in a file and use **-f** to invoke them:

```
[student@stationX ~]$ sed -f myedits pets
```

Special Characters for Complex Searches
Regular Expressions

- ^ represents beginning of line
- $ represents end of line
- Character classes as in **bash**:
  - `[abc]`, `[^abc]`
  - `[[:upper:]]`, `[^[:upper:]]`
- Used by:
  - **grep**, **sed**, **less**, others

Regular Expressions

The term "regular expression" (often shortened to "regex") refers to a search pattern that uses special characters, called "metacharacters", to represent something other than the literal meaning of those characters. For example, the character ^, when used in a regex-aware program's search pattern means, "the current line starts with", rather than an actual "^" symbol. As in **bash**, you can tell the program that you wish to use a metacharacter literally by pre-pending a \, so `\^` would match a literal "^".

Metacharacter	Meaning
`^`	Line Begins
`$`	Line Ends
`[xyz]`	A character that is x, y or z
`[^xyz]`	A character that is not x, y or z

So a regular expression like `^[Cc]hapter [Oo]ne$` is like saying "The line begins with 'C' or 'c', which is followed by 'hapter ', then 'O' or 'o' and ends with 'ne'". This is very useful for situations where you need to specify not just what you are looking for, but where you are looking for it. To provide a practical example, suppose you wanted to print the `/etc/passwd` line for user `brad` using **grep**. A command like:

```
$ grep brad /etc/passwd
```

would match the line for `brad`, but also lines for `bradley`, `notbrad` and any other username that contained `brad`. A regular expression can be used for more precise results:

```
$ grep '^brad:' /etc/passwd
```

This course covers only the subset of regular expressions most often used by system administrators. The full set of regular expressions is more commonly used by developers writing applications that interpret text data. For more complete coverage of regular expressions, see **man regex** for "standard" regular expressions, supported by most regex-aware utilities, and **perldoc perlre** for "perl-style" regular expressions, supported by most programming languages.

Introducing vim

- Newer version of **vi**, the standard Unix text editor
  - Executing **vi** runs **vim** by default
- **gvim**: Graphical version of **vim**
  - Applications + Programming -> Vi iMproved
  - Provided by `vim-X11` package
- Advantages:
  - Speed: Do more with fewer keystrokes
  - Simplicity: No dependence on mouse/GUI
  - Availability: Included with most Unix-like OSes
- Disadvantages
  - Difficulty: Steeper learning curve than simpler editors
    - Key bindings emphasize speed over intuitiveness

vim: A Modal Editor

- Keystroke behavior is dependent upon **vim**'s "mode"
- Three main modes:
  - Command Mode (default): Move cursor, cut/paste text, change mode
  - Insert Mode: Modify text
  - Ex Mode: Save, quit, etc
- `Esc` exits current mode
- `EscEsc` always returns to command mode

vim Basics

- To use vim, you must at least be able to
  - Open a file
  - Modify a file (insert mode)
  - Save a file (ex mode)

Opening a file in vim

- To start **vi**:
  - **vim *filename***
  - If the file exists, the file is opened and the contents are displayed
  - If the file does not exist, **vi** creates it when the edits are saved for the first time

Modifying a File
Insert Mode

- `i` begins insert mode at the cursor
- Many other options exist
  - `A` append to end of line
  - `I` insert at beginning of line
  - `o` insert new a line (below)
  - `O` insert new line (above)

Editing text in Insert mode

Many commands will take you into insert mode. The slide above lists six common ways to do so.

The `i` command will *insert* your data *before* the cursor. The letter `i` will not appear in your document, but all other characters that you type will appear, until you exit insert mode. To exit insert mode, hit the *Esc* key.

The `a` command will place you in insert mode, allowing you to *append after* the cursor. Again, exit insert mode by hitting the *Esc* key.

The `o` command *opens* a line *below* the current line, placing you in insert mode.

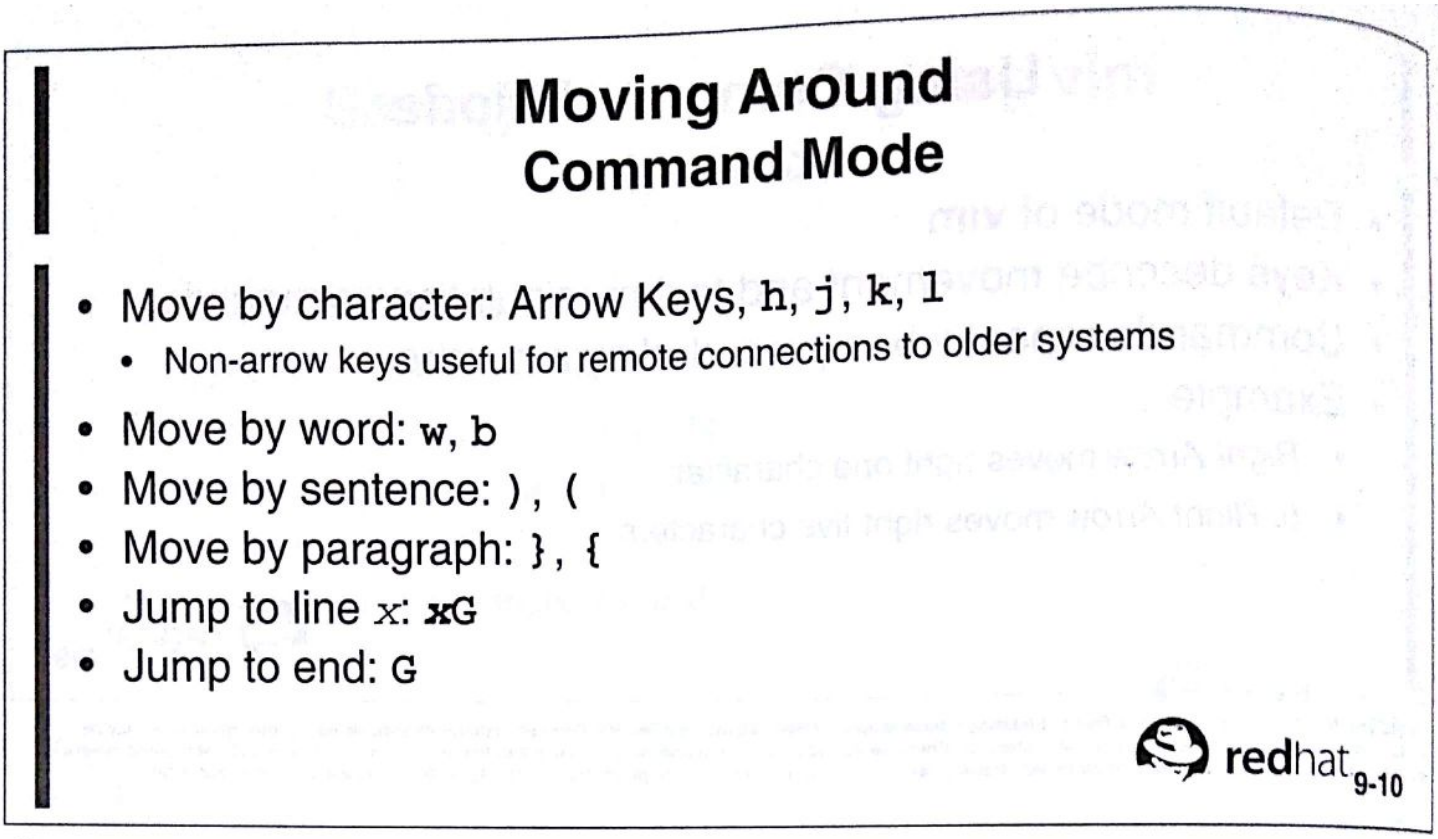
Note the relationship between the lower case `a`, `i`, and `o`, and the upper case `A`, `I`, and `O`. Whereas the `a` appends after the cursor, the `A` appends at the end of the line. The `i` inserts before the cursor; the `I` inserts at the beginning of the line. The `o` opens a line below the current line; the `O` opens a line above the current line. Patterns such as these permeate the **vi** and **vim** commands.

Saving a File and Exiting vim
Ex Mode

- Enter Ex Mode with `:`
  - Creates a command prompt at bottom-left of screen
- Common write/quit commands:
  - `:w` writes (saves) the file to disk
  - `:wq` writes and quits
  - `:q!` quits, even if changes are lost

Using Command Mode

- Default mode of **vim**
- Keys describe movement and text manipulation commands
- Commands repeat when preceded by a number
- Example
  - *Right Arrow* moves right one character
  - *5*, *Right Arrow* moves right five characters



Moving by character

In command mode, one of the most important actions that you will want to take is to move the cursor. Not only are cursor movements important in themselves, but they are also the basis for much of the rest of the**vi** and **vim** command vocabulary.

The commands above will move the cursor about the file. There are many more cursor movements, but these are the most important and useful ones.

The `h`, `j`, `k`, and `l` movements move the cursor left, down, up, and right, respectively. The arrow keys also work, but note that if you become adept at using the `h`, `j`, `k`, and `l` keys, you can move the cursor without moving your hands from the home row of the keyboard.

Moving by larger chunks

Often, it is useful to jump around in a file. The `G` command takes you to the bottom of the file. Precede the `G` command with a number and it will take you to that line number. A common `G` command is `1G`, go to the first line. This is also useful when an error message tells you that an error exists on the particular line of a file. You can use the `G` command preceded by that number to jump right to the offending line.

Search and Replace
Command Mode

- Search as in **less**
  - /, n, N
- Search/Replace as in **sed**
  - Affects current line by default
  - Use *x,y* ranges or % for whole file
    - :1,5s/cat/dog/
    - :%s/cat/dog/gi

Searching a document

To search for a string of text, use the / or ? commands as in the *less* command:

/dog search downwards in the file for the string "dog"

?mouse search upwards in the file for the string "mouse"

Once the first match is found, you can find the *next* match with the n command. To reverse the direction of the search, use the N command.

Search/Replace operations

vi and **vim** can perform search and replace operations, much like the sed command. The primary difference between the sed and the other commands is that absent an address, sed works on the entire file, whereas with **vi** and **vim**, absent an address, they work only on the current line, the line on which the cursor resides. To make a change on the entire file, you must specify the lines:

:%s/Ohio/Iowa/g

In this command, the colon signifies that this is an ex command. 1,$ is the address, starting from line 1 and continuing through the last line. ;s/Ohio/Iowa/; indicates that for the string of characters Ohio;, replace the characters Iowa;. By default (and as with sed), only the first instance of Ohio; will be changed on any particular line. If you would like all instances of Ohio; on a line changed to Iowa;, then you put the trailing g character, as above.

The default substitution delimiter is the / character as seen above. However, vi treats whatever character follows the "s" command as the delimiter, so you can use other characters if necessary. This is especially useful in instances where the / character appears in your search or substitution strings. For example, to replace all instances of /dev/hda with /dev/sda you could do:

:%s\/dev\/hda\/\/dev\/sda/g

But note that the slashes in /dev/hda and /dev/sda had to be escaped to prevent vi from thinking they were delimiters. A much simpler option in this case would be to use a different delimiter: **:%s'/dev/hda'/dev/sda'g**

Manipulating Text
Command Mode

	Change (replace)	Delete (cut)	Yank (copy)
Line	`cc`	`dd`	`yy`
Letter	`cl`	`dl`	`yl`
Word	`cw`	`dw`	`yw`
Sentence ahead	`c)`	`d)`	`y)`
Sentence behind	`c(`	`d(`	`y(`
Paragraph above	`c{`	`d{`	`y{`
Paragraph below	`c}`	`d}`	`y}`

redhat 9-12

Change, Delete and Yank (Replace, Cut and Paste)

At first glance, the chart above may seem intimidating, but note the pattern. The characters `w`, `{`, `}`, `(`, and `)` mean the same thing here as in cursor movement. The commands `c`, `d`, and `y`, are combined with these to perform an action.

The `cc` command will change a line: it will delete the current line and place you into insert mode to enter the replacement, which could be another line, several lines, or just a few characters. As before, press *Esc* to return to command mode.

The `cl` command will delete the current character and place you in insert mode; the `cw` command will delete the current word and place you into insert mode. The paragraph and sentence commands will operate similarly.

`dd` deletes the current line, leaving you in command mode. The remaining deletion commands operate in a similar fashion.

The `yy` command may take some explanation. These days, the common name for this action is "copy": that is, place some text in a buffer without modifying the original data. However, in the original Unix **vi**, and still today in the modern **vim**, the action taken is not called "copy", but rather "yank". A line is yanked into a buffer, presumably to be put (or pasted) back into the document at another location. As before, `yl` yanks a letter, `yw` yanks a word, and the remaining commands yank sentences and paragraphs.

In the next slide, we will learn to put (or paste) yanked data back into the document.

Undoing Changes
Command Mode

- `u` undo most recent change
- `U` undo all changes to the current line since the cursor landed on the line
- ***Ctrl*-`r`** redo last "undone" change

Help! I made mistake!

You will not work with **vi** or **vim** long before needing to undo a change that you made, either incorrectly or unintentionally. To undo the most recent change, use the `u` command. Several `u` commands in a row will undo several previous changes. The `u` will not undo a previous `u`; that is, it will not toggle a change, but rather undo several previous changes.

To undo all successive changes to the current line, use the `U` command.

To redo a change undone by a `u` command, use the ***Ctrl*-`r`** command.

Configuring vi and vim

- Configuring on the fly
  - `:set` or `:set all`
- Configuring permanently
  - `~/.vimrc` or `~/.exrc`
- A few common configuration items
  - `:set number`
  - `:set autoindent`
  - `:set textwidth=65 (vim only)`
  - `:set wrapmargin=15`
  - `:set ignorecase`
- Run `:help option-list` for a complete list

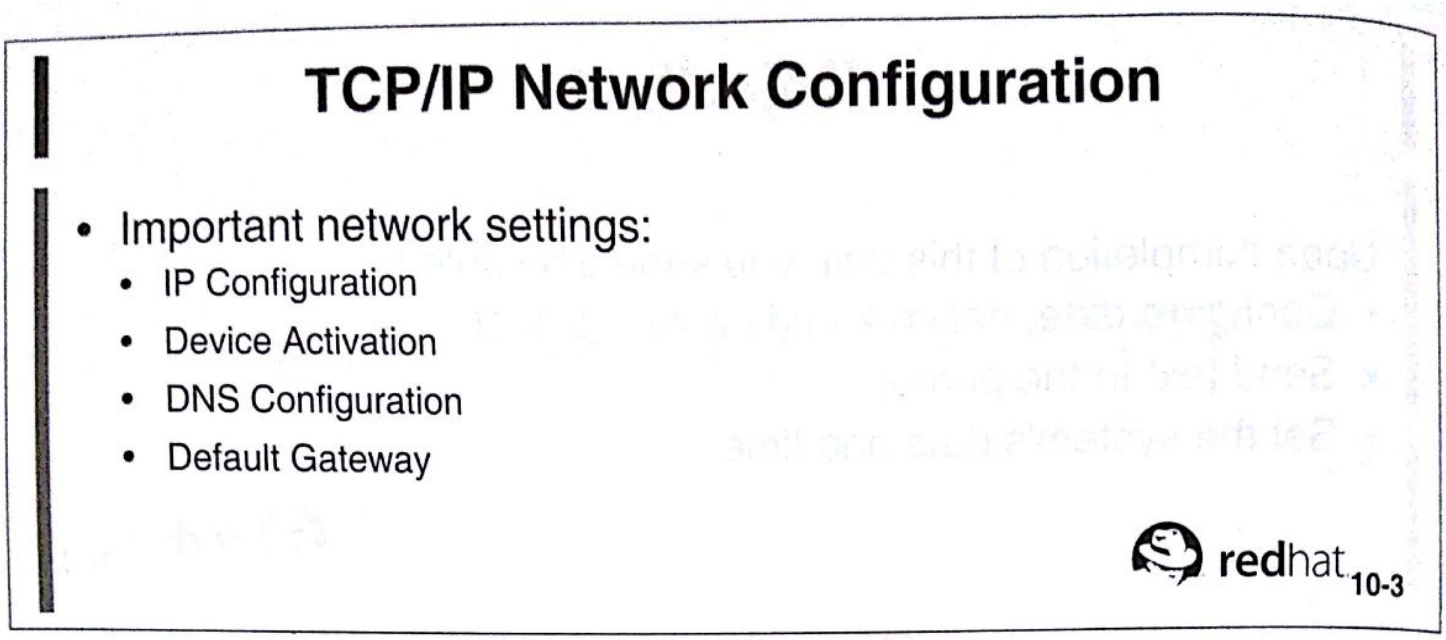
redhat 9-16

Dozens of configuration items exist for vi and vim. To examine your current configuration, run:

- `:set` lists a small number of important configuration items
- `:set all` lists the vast panoply of configuration items

To change a configuration item, use the `:set` command. Some common items to set include:

- `:set showmatch` causes the cursor to momentarily jump to the matching left curly brace or left parenthesis when a right curly brace or a right parenthesis is typed. `:se sm` is an abbreviation of this. `:set noshowmatch` (`:se nosm`) turns off this behavior.
- `:set autoindent` (`:se ai`) causes new lines to inherit the indentation level of the previous line. This is very useful for programmers. `:set noautoindent` (`:se noai`) turns this off.
- `:set textwidth=65` causes text to wrap (by inserting a hard return) when the text exceeds 65 characters (of course, any number can be given). `:set textwidth=0` turns this off. This option only works in **vim**.
- `:set wrapmargin=15` (`:se wm=0`) causes text to wrap when it reaches 15 characters from the right margin. `:set wrapmargin=0` turns this off.
- `:set number` (`se nu`) causes line numbers to be displayed in the left margin (visual only; line numbers are not actually stored in the document). `:set nonumber` (`se nonu`) turns this off.
- `:set ignorecase` (`se ic`) causes searches to be case-insensitive (by default, they are case sensitive). `:set noignorecase` (`se noic`) turns this off.
- To save these settings so that they are run at every invocation of the editors, place the commands above in `~/.vimrc`. The older `~/.exrc` will be read by both **vi** and **vim** if `~/.vimrc` does not exist.



Basic Network Configuration

Most networks use TCP/IP as the basic protocols for network communication. Regardless of the operating system you use, certain TCP/IP settings always need to be configured for your system to communicate with other systems on such a network.

The most basic of these settings is your network interface card's *IP address*, *subnet mask* and `network number`. In general these settings and more will be provided automatically by your network's dynamic host configuration protocol (DHCP) server. Even when manual configuration is required, often the IP address, which is used to identify your host, is the only one of these that needs to be set explicitly.

If you are not using DHCP, you will also need to specify a *domain name service (DNS) server* and *default gateway*.

The DNS server is used to translate *domain names* like `www.redhat.com` to IP addresses like `209.132.177.50`.

The default gateway is the IP address of the device or system to which communications destined for hosts on another network should be sent. It is the job of the gateway to see to it that such messages reach their intended destination.

Subsequent slides will discuss tools that can be used for manipulating these settings.

Managing Ethernet Connections

- Network interfaces are named sequentially: `eth0`, `eth1`, etc
  - Multiple addresses can be assigned to a device with *aliases*
  - Aliases are labeled `eth0:1`, `eth0:2`, etc.
  - Aliases are treated like separate interfaces
- View interface configuration with **ifconfig [eth*x*]**
- Enable interface with **ifup eth*x***
- Disable interface with **ifdown eth*x***

Network connection names consist of a prefix, based on the device type, and a number to distinguish a particular device from others of its type. For example, all ethernet devices have the prefix `eth`. The first detected ethernet card is assigned the name `eth0`, the second `eth1` and so forth. Every system also has a special network device called the `lo`, which represents the "localhost" or "loopback" device with address 127.0.0.1. You can view the basic settings of a network device by running the **ifconfig** command. By default, **ifconfig** will print information on all active devices. If given a device name as an argument, it will print information about that device only:

```
[student@stationX ~]$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:09:6B:CD:2B:87
          inet addr:192.168.0.254  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::209:6bff:fecd:2b87/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:851525 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1132322 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:211140434 (201.3 MiB)  TX bytes:1113058956 (1.0 GiB)
```

While at first glance this output may be a bit overwhelming, it is helpful to know that the most important information is in the top three lines. The first line tells us that `eth0` is an ethernet device with the hardware (or MAC) address `00:09:6B:CD:2B:87`. This is a unique address built into every ethernet card by the manufacturer and can be useful for identifying specific devices on the network.

The second line lists the three fundamental IP configuration parameters: the IP address (`192.168.0.254`), broadcast address (`192.168.0.255`) and network mask (`255.255.255.0`). These settings are used to identify your system to other machines on the network and define how your system interacts with others. A detailed explanation of how each setting works is beyond the scope of this class, but we will be seeing how each can be set, both automatically and manually, in subsequent slides.

An interface can be configured but not running. Such an interface is said to be *disabled* or *down*. If an interface is down it will not be shown in **ifconfig**'s output unless the name of the device is passed as an argument. Devices can be brought up and down by an administrator using the **ifup** and **ifdown** commands.

Network Configuration Files
Ethernet Devices

- Device configuration is stored in text files
  - `/etc/sysconfig/network-scripts/ifcfg-ethX`
  - Complete list of options in `/usr/share/doc/initscripts-*/sysconfig.txt`

Dynamic Configuration	Static Configuration

`DEVICE=ethX`
`HWADDR=0:02:8A:A6:30:45`
`BOOTPROTO=dhcp`
`ONBOOT=yes`
`Type=Ethernet`
	`DEVICE=ethX`
	`HWADDR=0:02:8A:A6:30:45`
	`IPADDR=192.168.0.254`
	`NETMASK=255.255.255.0`
	`GATEWAY=192.168.2.254`
	`ONBOOT=yes`
	`Type=Ethernet`

Whether or not you use **system-config-network** to configure your network interfaces, all network interface settings are stored in files in the `/etc/sysconfig/network-scripts/` directory. These files generally have the name `ifcfg-ethX`, although any name that begins with `ifcfg-` can be used. These *interface configuration files* are read by **system-config-network**, **ifup**, **ifdown** and other tools that bring network interfaces up and down.

If you examine one of these files closely, you may find that the syntax seems familiar. An interface configuration file is really just a collection of **bash** variables that define the device's settings. Remember that the syntax for setting a **bash** variable is ***VARIABLE=VALUE***, with no spaces on either side of the **=**.

While the Red Hat Enterprise Linux installer and **system-config-network** tend to add a large number of configuration options to these files, only a few strictly necessary. The slide above describes two simple configurations, one for an interface that gets it IP settings from a DHCP server and one for an interface that is manually, or *statically* assigned an address. The following is a list of useful interface configuration options. See `/usr/share/doc/initscripts-*/sysconfig.txt` for a more complete list.

Setting	Meaning
DEVICE	Specifies the device alias (eg `eth0`) that this file describes.
HWADDR	Associates the configuration with a specific device, identified by a MAC address. If the alias $DEVICE is currently applied to a different device, the system will attempt to re-label it. If it fails, an error is generated. This setting is optional and can cause problems when an ethernet card is replaced.
BOOTPROTO	Where IP settings should be retrieved from. Set to **`dhcp`** to use DHCP. Leave the variable unset or set it to **`static`** to define IP settings yourself.

IPADDR and NETMASK	Basic IP settings. Only necessary when not using DHCP.
GATEWAY	The IP Address of the system or device to send messages destined for hosts on another network. It is the responsibility of the gateway to determine how to contact the destination host. Specifying this is only necessary when not using DHCP. This can also be set in the `/etc/sysconfig/network` file. If a gateway is defined there and in an `ifcfg` file, the gateway defined in the most recently activated `ifcfg` file is used.
ONBOOT	Whether to bring the device up automatically when the system boots. Can be set to **`yet`** or **`no`**. The default value is **`no`**.
USERCTL	Whether to allow non-root users to bring this device up and down. Can be set to **`yet`** or **`no`**. The default value is **`no`**.
TYPE	Specifies the type of network interface being used. Can be omitted when using ordinary ethernet. For other connection types, such as wireless ethernet, a setting like **`TYPE=Wireless`** would cause supplementary configuration scripts to run and look for wifi-specific settings such as **`ESSID`**.

Network Configuration Files
Other Global Network Settings

- Global Settings in `/etc/sysconfig/network`
  - Many may be provided by DHCP
  - GATEWAY can be overridden in ifcfg file

```
NETWORKING=yes
HOSTNAME=server1.example.com
GATEWAY=192.168.2.254
```

Some network settings are defined globally, rather than on a per-interface basis. These global network settings are defined in the `/etc/sysconfig/network` file. Some of the more important settings that can be defined in this file include:

Setting	Meaning
NETWORKING	Whether to enable networking at all. Should normally be set to `yes`.
GATEWAY	The IP Address of the system or device to send messages destined for hosts on another network. It is the responsibility of the gateway to determine how to contact the destination host. Specifying this is only necessary when not using DHCP. This can also be set in an `ifcfg` file. If a gateway is defined here and in an `ifcfg` file, the gateway defined in the most recently activated `ifcfg` file is used.
HOSTNAME	The system's hostname. This should be the DNS name that resolves to its primary IP address. If you are using DHCP, it is probably not necessary to define this. If you do not define this and your system is not using DHCP then it will ask DNS what name is associated with your IP address and use that. If DNS does not have a name associated with your IP, your system will be assigned the name `localhost.localdomain`.

Network Configuration Files
DNS Configuration

- Domain Name Service translates hostnames to network addresses
- Server address is specified by dhcp or in `/etc/resolv.conf`

```
search example.com cracker.org
nameserver 192.168.0.254
nameserver 192.168.1.254
```

The domain name service, or DNS, allows names to be assigned to and used in place of IP addresses. For example, suppose the IP address for example.com's web server is 192.168.0.254. A DNS server can be configured to allow me to refer to that system as www.example.com instead. When I type **`www.example.com`** into my web browser, it initiates a *DNS lookup*, in which it asks the local DNS server what IP is assigned to that name. If the local DNS server does not know, it will try to find out which other servers on the internet know about example.com and will forward my request to one of them.

DNS has the unique distinction of being at once one of the most and least important services on the internet. Because communication over IP networks is done using numeric addresses, there is technically no need for a service that uses words to identify a system instead. Why not just use numbers? The answer, of course, is that humans think in words much more easily than numbers. So while the internet could function just fine without DNS, it would be much more difficult for humans to use it.

Local DNS configuration is performed using the `/etc/resolv.conf` (note the strange spelling) file. The file gets its name from the fact that converting a DNS name to an IP address is commonly referred to as *resolving* the name.

There are only two settings one generally needs to remember when creating or modifying `resolv.conf`: **`search`** and **`nameserver`**. **`nameserver`** is the most important, as it specifies the IP address of a DNS server your system should use. Multiple **`nameserver`** lines may be added. Since servers are tried in order, be sure to put the one that is fastest and most likely to be available first.

The **`search`** directive specifies domains that should be tried when an incomplete DNS name is given to a command. For example, if my `resolv.conf` contains the line: **`search example.com cracker.org`** and I run the command **ping server1**, the system will first attempt to resolve the name `server.example.com`, trying `server1.cracker.org` only if that attempt fails.

Printing in Linux

- Printers may be local or networked
- Print requests are sent to queues
- Queued jobs are sent to the printer on a first come first served basis
- Jobs may be canceled before or during printing

redhat 10-9

Printing

Having created a file, you will no doubt want to print it. The printing system in Red Hat Enterprise Linux is very flexible. Printers may be parallel, serial, or networked. Support is included for printing to remote CUPS IPP, lpd (common Linux and Unix printing subsystem), Windows, Netware, and JetDirect printers.

Queues

One or more queues is associated with each printer. Print jobs are sent to a queue, not to a printer. Different queues for the same printer may have differing priority or output options. Setting up print queues is the responsibility of the system administrator; individual users do not create print queues.

Jobs

Once a file has been sent to a queue for printing, it is called a job. Jobs may be canceled while they are printing, or when they are in the queue waiting to be printed.

system-config-printer

- System->Administration->Printing
- Supported printer connections:
  - Local (parallel or usb)
  - Unix/Linux print server
  - Windows print server
  - Netware print server
  - HP JetDirect
- Configuration stored in `/etc/cups/printers.conf`

10-10

To run the printer configuration tool, select to System->Administration->Printing. To add a printer to the system, click **New** on the toolbar. The installation program will provide prompts during the process. The printer will have to be named and should have a description of its physical location.

To define a local printer, on the *Queue type* screen, select *Locally-connected* and */dev/lp0*. This will allow printing for a parallel printer. If the printer is a supported USB printer, plugging it in and clicking *Rescan devices* should detect the correct port.

Next, select the manufacturer from the pull down menu, and the model from the scroll menu. The generic print drivers might provide basic printing capacity for unsupported printers. Once finished, the program will ask to print a test page.

Rather than printing locally, Linux could print to a network device. On the *Queue type* screen, select *Networked Windows (SMB)*. The install program will search the workgroup for Windows systems with shared printers.

Click the triangle to the left of the system name to expand the system's shares. Highlight the printer, than click Forward. As with a local printer, select the manufacturer and model.

Printing Commands

- **lpr** sends a job to the queue to be printed
  - Accepts ASCII, PostScript, PDF, others
- **lpq** views the contents of the queue
- **lprm** removes a job from the queue
- System V printing commands such as **lp**, **lpstat** and **cancel** are also supported

Using the print utilities

The **lpr** command is used to send a job to the printer. The Linux printing system will print files in ASCII, PostScript, PDF, and other formats. Most applications under Linux output PostScript format.

The **-P** option is used to select a queue other than the default and **-#** is used to specify the number of copies. For example, to print 5 copies of the file `report.ps` on the accounting printer:

```
[student@stationX ~]$ lpr -P accounting -#5 report.ps
```

When entered without options, **lpq** lists the jobs in the default queue. As with **lpr**, *-P* is used to specify a queue other than the default. For example:

```
[student@stationX ~]$ lpq

Printer: ps@localhost
Queue: no printable jobs in queue
Server: no server active
Status: job 'jay@localhost+916' removed at 12:16:03.083
Rank   Owner/ID                  Class Job Files      Size Time
done   jay@localhost+185            A  185 results    2067 08:38:04
```

To remove a job from the print queue, use **lprm** followed by the job number, specifying a non-default print queue if necessary. For example:

```
[student@stationX ~]$ lprm 916
Printer ps@localhost:
```

In this example, **lprm** responds with the name of the queue from which the job was removed. Note that a user may only remove his own print jobs from the queue.

Setting the System's Date and Time

- GUI: **system-config-date**
  - System->Administration->Date & Time
  - Can set date/time manually or use NTP
  - Additional NTP servers can be added
  - Can use local time or UTC
- CLI: **date [MMDDhhmm[[CC]YY][.ss]]**
  - `# date 01011330`

    `# date 010113302007.05`

The most basic way to alter your system's date is to use the **date** command. **date --help** reports that the following syntax should be used:

```
[MMDDhhmm[[CC]YY][.ss]]
```

This means that you must at least provide two-digit values for the month, day, hour and minute you wish to set, with an optional two or four digit year and an optional number of seconds, preceded by a period. Some examples:

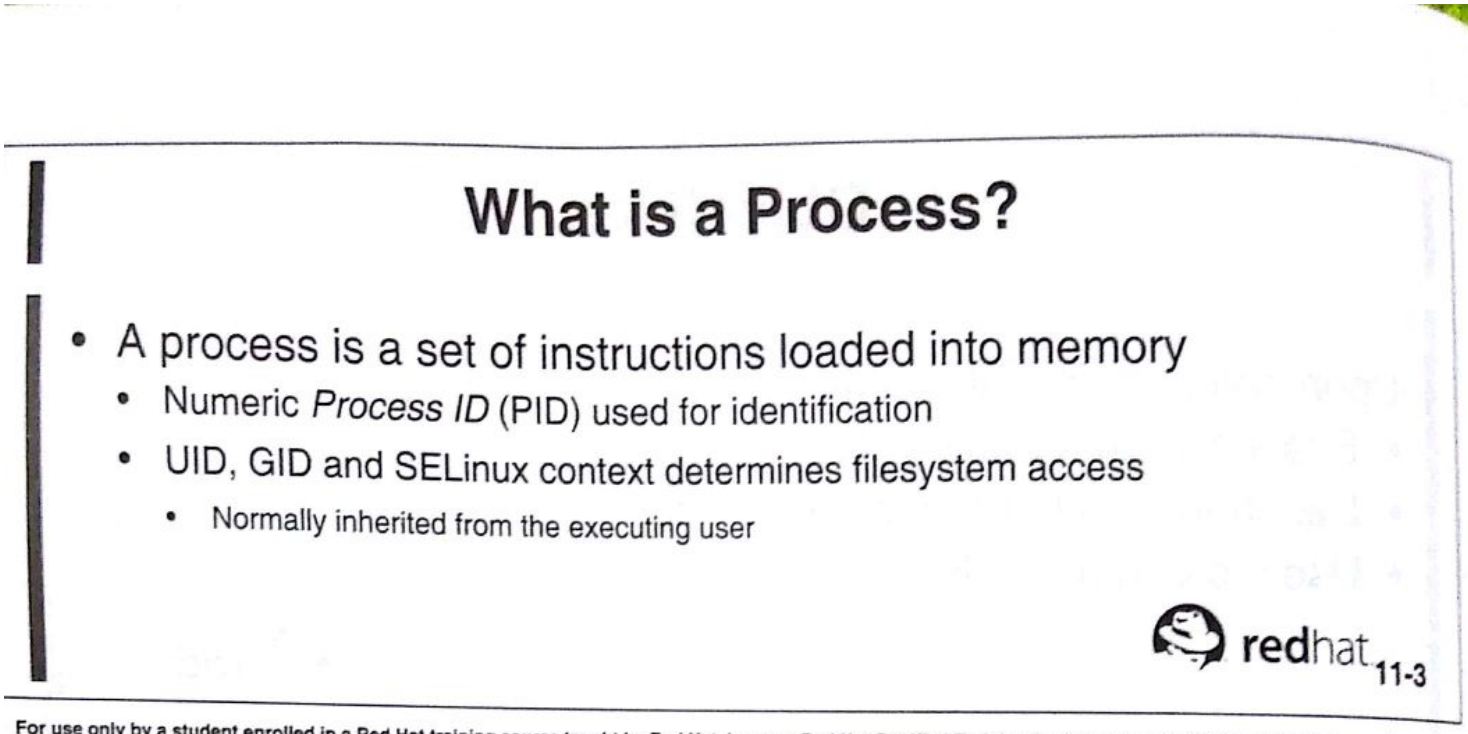
```
# date 12312359          # 31st of Dec, at 11:59pm. No change to the year.

# date 123123592007 # As above, but also sets the year to 2007

# date 01010101.01  # 1st of Jan, at 01:01:01am. No change to the year.
```

While less intuitive than graphical tools, **date** has the advantages of not relying on a graphical environment and being easy to use in a shell script.

system-config-date is used to graphically configure your system's date and time settings. It can be accessed by selecting System->Administration->Date & Time.

Occasionally, all the controls on the first tab are unelectable (grayed-out). This means that the system's time is being set from another server via the Network Time Protocol (NTP) and cannot be set manually. To regain manual control, click the second tab, **Network Time Protocol**, and deselect the **Enable Network Time Protocol** checkbox.

On **Time Zone** tab the system clock can be set for local time or for UTC (Greenwich Mean Time). This is controlled by selecting the **System clock uses UTC** checkbox.



Processes

A process is an executing program with several components and properties, including a memory context, priority, and environment. The Linux kernel tracks every aspect of a process by its PID under `/proc/PID`.

Listing Processes

- View Process information with **ps**
  - Shows processes from the current terminal by default
  - **-a** includes processes on all terminals
  - **-x** includes processes not attached to terminals
  - **-u** prints process owner information
  - **-f** prints process parentage
  - **-o PROPERTY,...** prints custom information:
    - **pid, comm, %cpu, %mem, state, tty, euser, ruser**

Viewing All Processes

The **ps** command provides an easy way to view these properties and has several styles of output depending on the options used. Without options, it displays information about processes specific to the active terminal. The options described below are based on output conforming to the UNIX98 standard. Consult the online documentation for more options.

-a	displays all processes, not including processes not controlled by a terminal
-x	includes processes not controlled by a terminal, such as daemon processes (? as the tty in output).

Process States

Every process has a *state* property, which describes whether the process is actively using the cpu (*Running*), in memory but not doing anything (*Sleeping*), waiting for a resource to become available (*Uninterruptable Sleep*) or terminated, but not flushed from the process list (*Zombie*). There are other process states listed in **man ps**, but the four described above are the most common. Of these, Running and Sleeping are normal, but the presence of Uninterruptable Sleep or Zombie processes may indicate problems lurking on your system.

- *Uninterruptable sleep*: Process is sleeping and can not be woken up until an event occurs. It can not be woken up by a signal. Typically, the result of an I/O operation, such as a failed network connection (for NFS hard mounts).
- *Zombie*: Just before a process dies, it sends a signal to its parent and waits for an acknowledgment before terminating. Even if the parent process does not immediately acknowledge this signal, all resources except for the process identity number (PID) are released. Zombie processes are cleared from the system during the next system reboot and do not adversely affect system performance.

Finding Processes

- Most flexible: **ps** ***options*** | ***other commands***

```
ps axo comm,tty | grep ttyS0
```

- By predefined patterns: **pgrep**

```
$ pgrep -U root
$ pgrep -G student
```

- By exact program name: **pidof**

```
$ pidof bash
```

Viewing Specific Process Information

Since there may be hundreds of processes on a system, a common technique to locate a specific process is to send output from **ps** to **grep**:

```
[student@stationX ~]$ ps axo pid,comm | grep 'cups'
 2734 cupsd
 3502 eggcups
```

Compare the above output with that from **pgrep**:

```
[student@stationX ~]$ pgrep cups
2734
3502
```

A more exact method of obtaining PID's for processes is the **pidof**, which matches exactly on the program name specified and therefore requires that you know the specific program name:

```
[student@stationX ~]$ pidof cupsd
2734
```

Signals

- Most fundamental inter-process communication
  - Sent directly to processes, no user-interface required
  - Programs associate actions with each signal
  - Signals are specified by name or number when sent:
    - Signal 15, TERM (default) - Terminate cleanly
    - Signal 9, KILL - Terminate immediately
    - Signal 1, HUP - Re-read configuration files
    - **man 7 signal** shows complete list

redhat 11-6

Signals

Signals are simple messages that can be communicated to processes with commands like **kill**, which will be discussed on the next page. The advantage of signals is that they can be sent to a process even if it is not attached to a terminal and displaying an interface. So a web server or a graphical program whose interface has "frozen" may still be shut down by sending it the appropriate signal.

Though in most cases software developers can associate any action with the reception of a particular signal, almost all signals have standard meanings. Signals can be specified by their name, such as KILL, or by their number, such as 9. A detailed list of signals including names, numbers and meanings can be displayed with **man 7 signal**.

Sending Signals to Processes

- By PID: **kill** [***signal***] ***pid ...***
- By Name: **killall** [***signal***] ***comm ...***
- By pattern: **pkill** [**-*signal***] ***pattern***

redhat 11-7

Sending Signals

kill can send many signals, but processes only respond to the signals they have been programmed to recognize. For example, most services are programmed to reload their configuration when they receive a HUP.

Processes ordinarily terminate on their own when they have completed their task. Interactive applications may need the user to issue a `quit` command. In other cases, processes may need to be terminated with *Ctrl-C*, which sends an interrupt (INT) signal to the process. The process is shutdown "cleanly": that is, child processes are terminated first and any pending I/O operations are completed. The same is true if a process is sent a terminate (TERM) signal via the **kill** command.

If a process will not respond to a TERM signal, the KILL signal can be used. However, the process may not be ended cleanly. The KILL signal should be used only if a process will not respond to a *Ctrl-C* or a TERM signal. Using KILL signals on a routine basis may cause zombie processes and lost data.

The following are all identical and will send the default TERM signal to the process with PID number 3428:

```
[student@stationX ~]$ kill 3428
[student@stationX ~]$ kill -15 3428
[student@stationX ~]$ kill -SIGTERM 3428
[student@stationX ~]$ kill -TERM 3428
```

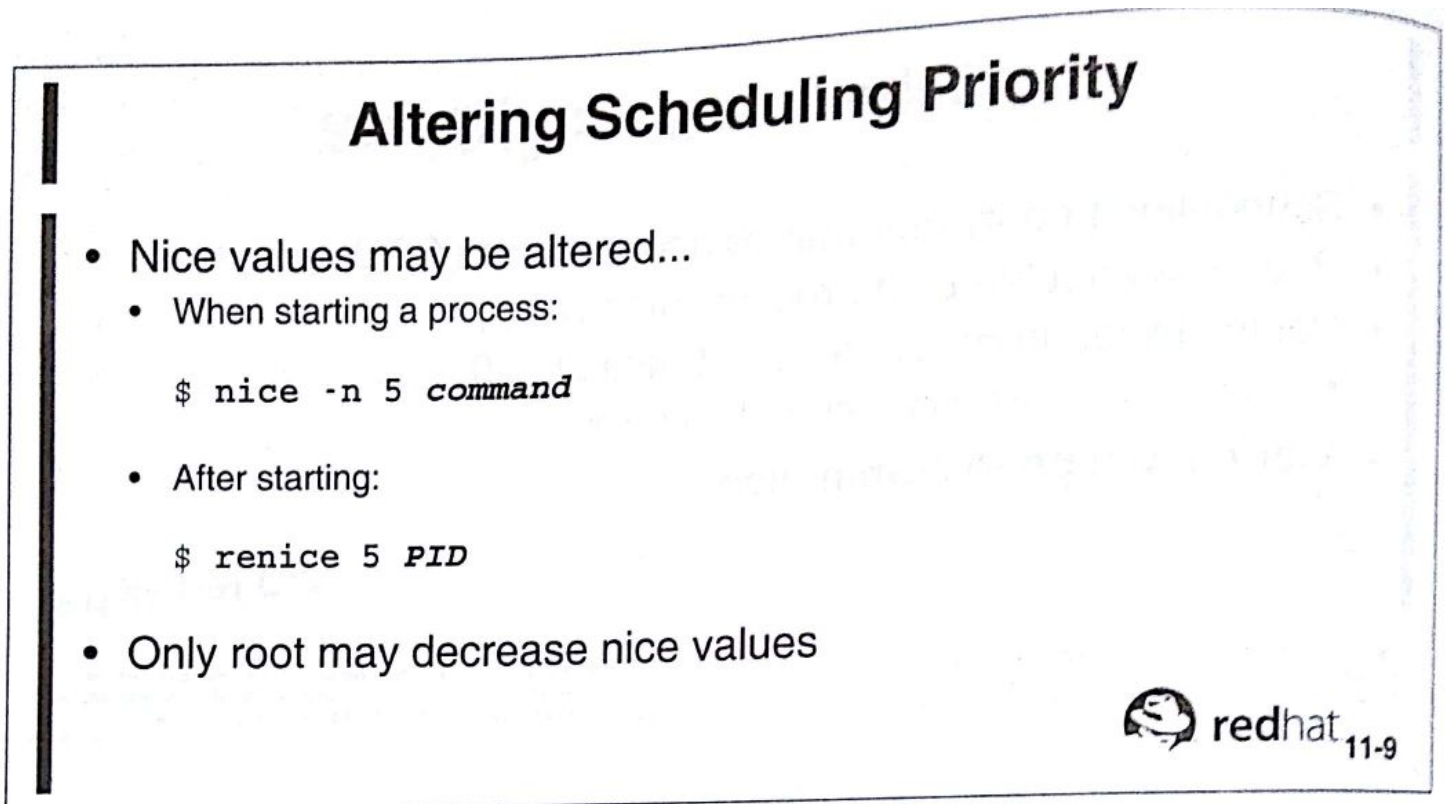
Scheduling Priority

- Scheduling priority determines access to the CPU
- Priority is affected by a process' *nice value*
- Values range from -20 to 19 but default to 0
  - Lower nice value means higher CPU priority
- Viewed with **ps -o comm,nice**

redhat 11-8

Schedule priorities

Every running process has a scheduling priority: a ranking among running processes determining which should get the attention of the processor. The formula for calculating this priority is complex, but users can affect the priority by setting the "niceness" value, one element of this complex formula. The niceness value defaults to zero but can be set from -20 (least nice, highest priority) to 19 (most nice, lowest priority).



Adjusting priorities for programs

To set the niceness value to a specific value when starting a process, use **nice -n**:

```
[student@stationX ~]$ nice -n 15 myprog
```

Non-privileged users may not set niceness value to less than zero: that is, they may not request a higher than normal priority for their processes. Only root may allocate additional CPU clock cycles to a process.

Altering priorities of running programs

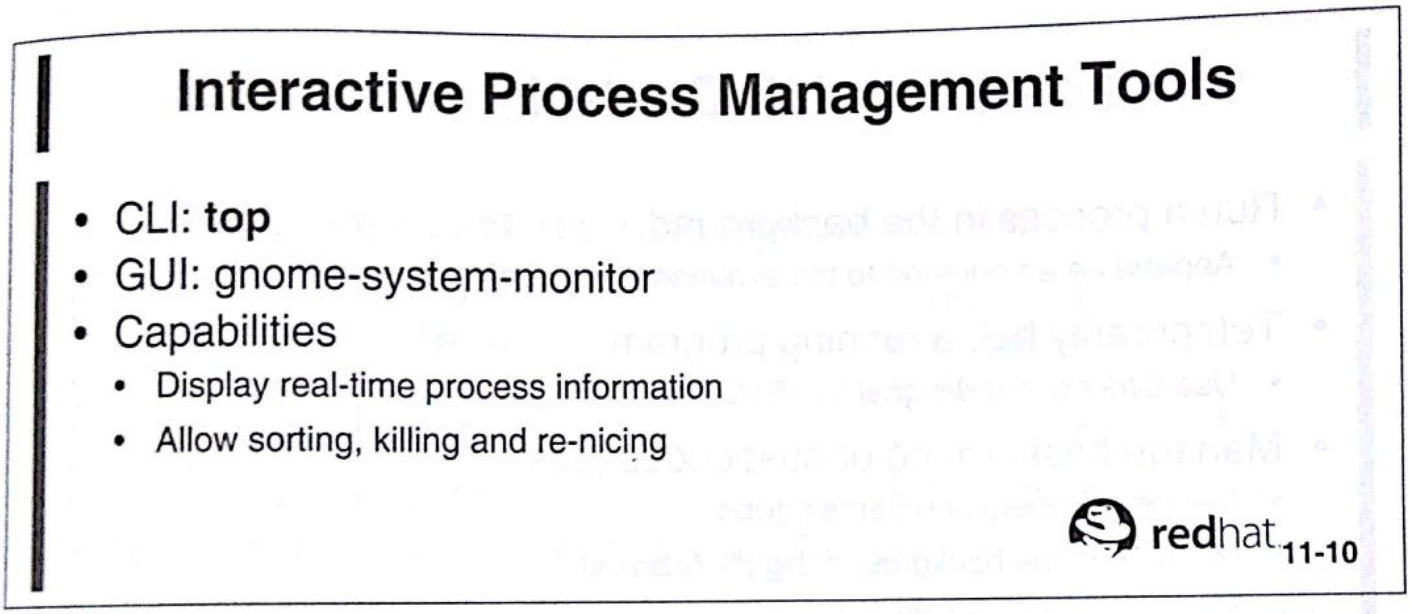
All users may raise the niceness (reduce the priority) of their own jobs using the renice command:

```
[student@stationX ~]$ renice 15 -p PID
```

Note that non-privileged users may start a process at any positive nice value but cannot lower it once raised.

root is permitted to reduce the niceness (raise the priority) of currently running processes:

```
[root@stationX ~]# renice -15 -p PID
```



*The **top** command*

Running **top** presents a list of the processes running on your system, updated every 5 seconds. You can use keystrokes to kill, renice and change the sorting order of processes. Processes that are in the Running state are highlighted and you can colorize processes and create multiple windows to view more than one sorted list of processes at a time. Press the *?* key while in top to view the complete list of hotkeys. You can exit top by pressing the *q* key.

The gnome-system-monitor command

The gnome-system-monitor, which can be run from the console or by selecting ApplicationsSystem ToolsSystem Monitor from the menu system, is a graphical tool that also offers the ability to view sorted lists of processes and to kill or renice those processes.

By default the tool only displays processes that are in the "Running" state. However, this can be changed by selecting All Processes or My Processes from the dropdown menu in the upper-right. The preferences dialog (EditPreferences) allows you to customize which fields are displayed and you can sort by a particular field by clicking on its heading. You can send a process the TERM signal by right-clicking on it and selecting End Process or the KILL signal (equivalent to a **kill -9**) by selecting Kill Process. A process can be re-niced by right-clicking on it and selecting Change Priority.

Job Control

- Run a process in the background
  - Append an ampersand to the command line: **firefox &**
- Temporarily halt a running program
  - Use *Ctrl-z* or send signal 17 (STOP)
- Manage background or suspended jobs
  - List job numbers and names: **jobs**
  - Resume in the background: **bg [%*jobnum*]**
  - Resume in the foreground: **fg [%*jobnum*]**
  - Send a signal: **kill [-*SIGNAL*] [%*jobnum*]**

A background process is still the child of the process that spawned it. The parent process, however, does not wait for the child process to terminate before continuing. Instead, a new bash "subshell" is created. The **bash** program is then replaced by the command being executed. You should typically redirect STDOUT and STDERR for background jobs to avoid confusion.

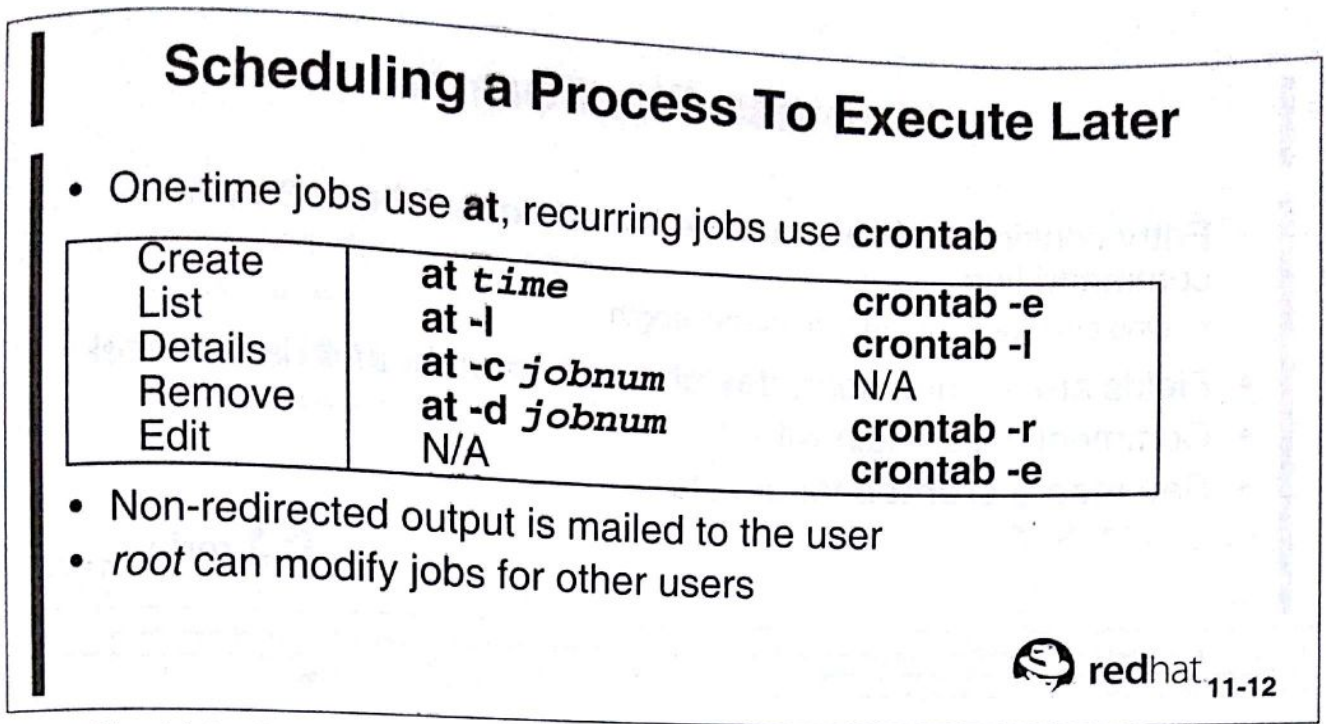
Jobs running in the foreground can be suspended: temporarily stopped, without being killed. To suspend a job, use the keys *Ctrl-z*. Once a process is suspended, it can be resumed in the background using the **bg** command or resumed in the foreground using the **fg** command. When the job resumes, it will continue executing from the point at which it was suspended. It will not have to start over from the beginning.

```
[student@stationX ~]$ find / -name '*.conf' -print &>/tmp/conf.list
[student@stationX ~]$ Ctrl-z
[1]+  Stopped                 find / -name '*.conf' -print &>/tmp/conf.list
[student@stationX ~]$ bg
[1]+ find / -name '*.conf' -print &>/tmp/conf.list &
[student@stationX ~]$ firefox &
[2] 6159
[student@stationX ~]$ jobs
[1]-  Running                 find / -name '*.conf' -print &>/tmp/conf.list &
[2]+  Running                 firefox &
```

The number next to [2] after backgrounding **firefox** is the PID. The plus or minus sign next to the job number tells which job is the 'default', i.e., the job that is referenced if no job number is given.

Typically, a job will be resumed in the background because the user intended to start the job in the background in the first place, but forgot to provide the ampersand background symbol. Alternately, the user may not have realized how long the job would take to run.

Also, typically, a job will be suspended and then resumed in the foreground because the user needs to temporarily jump out of the job for some reason. Perhaps the user is editing a file using **vi** and then suspends **vi** while looking up some other information on the system. The user would then resume **vi** using the **fg** command.



*Schedule a one-time job with **at***

Commands are entered, one per line, and terminated with a *Ctrl-d* on a line by itself. In the following example, you would like to compare network services on your machine to a baseline that is stored on non-writable media:

```
[student@stationX ~]$ at 0200
> netstat -tulpn | diff - /media/cdrom/netstat_baseline
> Ctrl-d
job 1 at 2007-01-08 22:45
[student@stationX ~]$ at -l
1       2007-01-09 02:00 a student
```

at -l (an alias to **atq**) lists the jobs currently pending. **at -c *jobnum*** cats the full environment for the specified job number, and **at -d *jobnum*** deletes the job.

root can modify other users' jobs by first getting a login shell as that user (**su - *username***).

The *time* argument has many formats which are illustrated by the following examples.

at 8:00pm December 7	**at 7 am Thursday**
at midnight + 23 minutes	**at now + 5 minutes**

See **man at** for more information.

*Schedule recurring jobs with **crontab***

The cron mechanism is controlled by a process named **crond**. This process runs every minute and determines if an entry in users' cron tables need to be executed. If the time has passed for a entry to be started, it is started. A cron job can be scheduled as often as once a minute or as infrequently as once a year.

root can modify recurring jobs for any user with **crontab -u *username*** and any of the other options, such as **-e**.

See **man crontab** for more information.

Crontab File Format

- Entry consists of five space-delimited fields followed by a command line
  - One entry per line, no limit to line length
- Fields are minute, hour, day of month, month, and day of week
- Comment lines begin with #
- See **man 5 crontab** for details

redhat 11-13

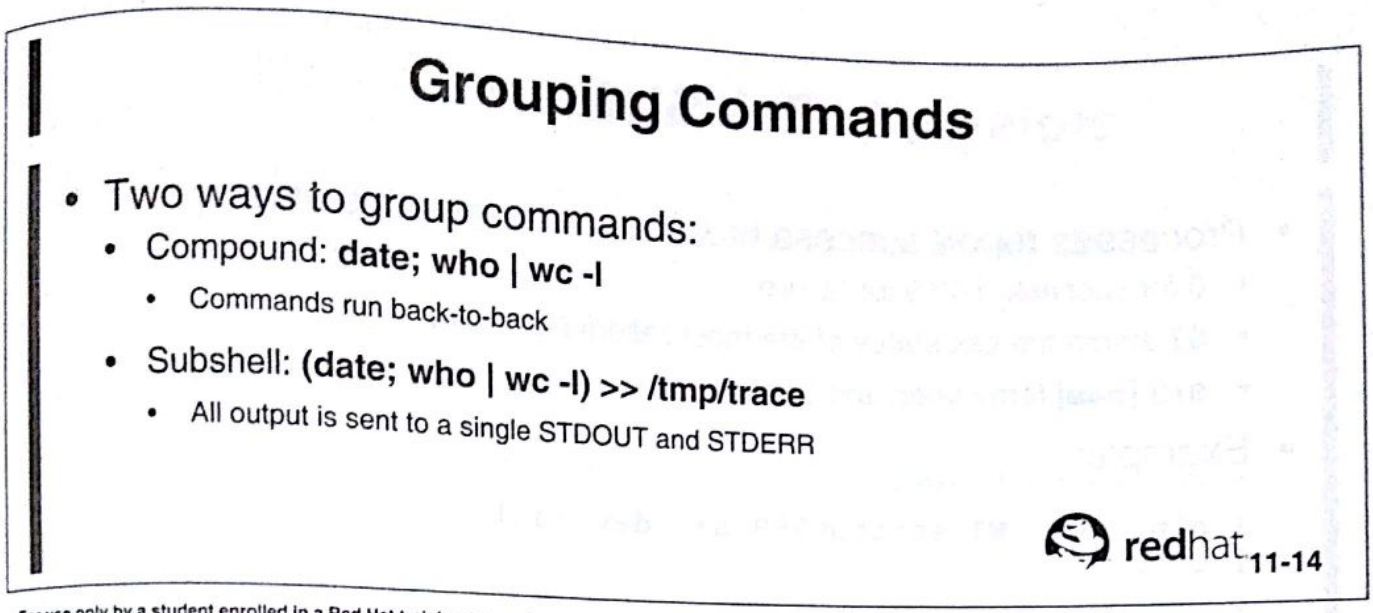
Crontab examples

Entry fields can be separated by any number of tabs or spaces. Valid field values are as follows:

Minute	0-59
Hour	0-23
Day of Month	0-31
Month	0-12
Day of Week	0-6 (0=Sunday)

Multiple values may be separated by commas. An asterisk in a field represents all valid values. A user's crontab may look like the following:

```
#Min   Hour   DoM   Month   DoW     Command
0      4      *     *       1,3,5   find ~ -name core | xargs rm -f {}
0      0      31    10      *       mail -s "boo" $LOGNAME < boo.txt
0      2      *     *       *       netstat -tulpn | diff - /media/cdrom/baseline
```



Aggregating Output Redirection

Suppose you want to maintain a count of the number of users logged on, along with a time/date stamp, in a log file. This could be accomplished with two commands:

```
[student@stationX ~]$ date >> logfile
[student@stationX ~]$ who | wc -l >> logfile
```

This command sequence requires that you enter two lines of commands, append to logfile twice, and in general, type much more than is necessary. When writing to the terminal, this task can be simplified by combining the commands on one line separated by semicolons:

```
[student@stationX ~]$ date; who | wc -l
```

But if your intent is to redirect standard output, this will not work as expected:

```
[student@stationX ~]$ date; who | wc -l >> logfile
```

Both commands will run, but only the second one will redirect its output to logfile.

Subshells

Commands inside parentheses are run in their own instance of **bash**, called a *subshell*. The output of all commands run inside a subshell are sent to the subshell's STDOUT and STDERR, making it possible to send multiple programs through the same pipe:

```
[student@stationX ~]$ (date; who | wc -l) >> logfile
```

Exit Status

- Processes report success or failure with an exit status
  - 0 for success, 1-255 for failure
  - **$?** stores the exit status of the most recent command
  - **exit [*num*]** terminates and sets status to *num*
- Example:

```
$ ping -c1 -W1 station999 &> /dev/null
$ echo $?
2
```

Upon completion, every command returns an exit status. The exit status will be a number in the range of 0 to 255, and it indicates whether or not the command ran successfully. As a general rule, an exit status of 0 indicates success and an exit value in the range of 1 to 255 indicates failure. To see the exit status of the most recently executed command, echo the `$?` variable:

```
[student@stationX ~]$ ping -c1 -W1 localhost &> /dev/null
[student@stationX ~]$ echo $?
0
[student@stationX ~]$ ping -c1 -W1 station999 &> /dev/null
[student@stationX ~]$ echo $?
2
```

It is possible for a shell script to deliberately set the exit value. Usually, a shell script will exit with the exit status of the last command run in the script. But an exit status can be forced by giving a numeric argument to the **exit** command. For example, this command will exit with an exit status of 255:

```
exit 255
```

The exit status is often referred to as an error code in documentation. As a challenge, review **man ping** to determine which error codes **ping** may produce.

Conditional Execution Operators

- Commands can be run conditionally based on exit status
  - && represents conditional AND THEN
  - || represents conditional OR ELSE
- Examples:

```
$ grep -q no_such_user /etc/passwd || echo 'No such user'
No such user

$ ping -c1 -W2 station1 &> /dev/null  \
>    && echo "station1 is up"         \
>    || $(echo 'station1 is unreachable'; exit 1)
station1 is up
```

It is possible to run a command conditionally based on the exit status of the previous command. When executing two commands separated by **&&**, the second command will only run if the first command exits successfully (exits with a status of zero). When executing two commands separated by ||, the second command runs if the first command fails (it exits with an exit status in the range of 1 to 255).

When using the conditional operators, we are not usually interested in the actual output of the first command. Therefore it is typical to suppress output or redirect it to `/dev/null`. This avoids cluttering the real output from scripts.

The following example combines several techniques presented in the course to this point:

```
for x in $(seq 1 10); do
echo adding test$x
(
  echo -ne "test$x\t"
  useradd test$x 2>&1 > /dev/null && mkpasswd test$x
) >> /tmp/userlog
done
echo 'cat /tmp/userlog to see new passwords'
```

The test Command

- Evaluates boolean statements for use in conditional execution
  - Returns 0 for true
  - Returns 1 for false
- Examples in long form:

```
$ test "$A"  =  "$B" && echo "Strings are equal"
$ test "$A" -eq "$B" && echo "Integers are equal"
```

- Examples in shorthand notation:

```
$ [ "$A"  =  "$B" ] && echo "Strings are equal"
$ [ "$A" -eq "$B" ] && echo "Integers are equal"
```

redhat 11-17

The **test** command evaluates true-or-false scenarios to simplify conditional execution. See **man test** for a complete list of arguments.

Some points should be made to illustrate subtleties of the examples above:

First, it may seem counter-intuitive that strings should be compared using a mathematical operator while numbers are compared using an abbreviation (`-eq`). This is a reflection of the second-generation programming languages that were in use at the time that shells were developed.

Second, string comparisons should always be quoted to protect embedded spaces.

File Tests

- File tests:
  - **-f** tests to see if a file exists and is a regular file
  - **-d** tests to see if a file exists and is a directory
  - **-x** tests to see if a file exists and is executable

```
[ -f ~/lib/functions ] && source ~/lib/functions
```

File tests can be used to test a variety of conditions that relate to files on the system. This provides an easy mechanism to test for the existence of files, conditions or permissions. Some of the supported file tests are:

-d *FILE*	True if the file is a directory.
-e *FILE*	True if the file exists.
-f *FILE*	True if the file exists and is a regular file.
-h *FILE*	True if the file is a symbolic link.
-L *FILE*	True if the file is a symbolic link.
-r *FILE*	True if the file exists and is readable by you.
-s *FILE*	True if the file exists and is not empty.
-w *FILE*	True if the file exists and is writable by you.
-x *FILE*	True if the file exists and is executable by you.
-O *FILE*	True if the file is effectively owned by you.
-G *FILE*	True if the file is effectively owned by your group.

Scripting: if Statements

- Execute instructions based on the exit status of a command

```
if ping -c1 -w2 station1 &> /dev/null; then
    echo 'Station1 is UP'
elif grep "station1" ~/maintenance.txt &> /dev/null; then
    echo 'Station1 is undergoing maintenance'
else
    echo 'Station1 is unexpectedly DOWN!'
    exit 1
fi
```

redhat. 11-19

Every process reports an exit status on termination, whether successful (exit=0) or error (1-255). This exit status can be checked within the body of the **if** as shown in the example, or you can explicitly assign the exit status to a variable using a subshell, as in:

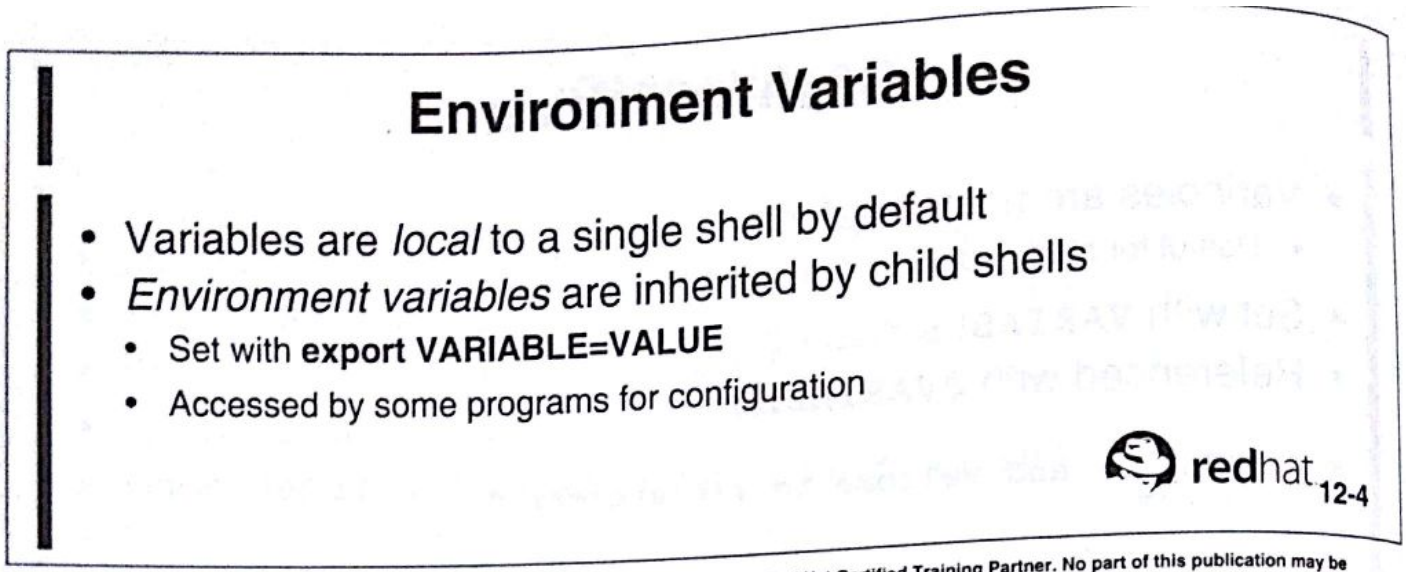
```
STATUS=$(test -x /bin/ping6)
```

The **if** structure can be combined with conditional operators and output redirection (to `/dev/null`) to simplify your scripting logic and avoid unwanted output. For example:

```
if test -x /bin/ping6; then
    ping6 -c1 ::1 &> /dev/null && echo "IPv6 stack is up"
elif test -x /bin/ping; then
    ping -c1 127.0.0.1 &> /dev/null && echo "No IPv6, but IPv4 stack is up"
else
    echo "Oops! This should not happen."
    exit 255
fi
```

This script tests to see if the IPv6 version of the ping command exists. If it does, it uses **ping6** to send a test packet to the system's IPv6 loopback interface. If **ping6** does not exist, the script checks for the IPv4 version, **ping**. If this exists, it is used to send a packet to the IPv4 loopback. If neither exists, something is probably wrong and a non-zero return code is issued along with a warning message.

For good examples of real-world scripts, look at the scripts in `/etc/init.d/*`.



The use of variables is central to the process of configuration, whether configuring the shell or some other program.

A variable is a label that equates to some value. The value can change over time, across systems, or across accounts, but the label remains constant. For example, a shell script may place a file in `$HOME`, a reference to the value of the variable `HOME`. This value may differ, depending on who is running the shell script.

Some variables, like `$HOME`, are intended to be referenced but not set by the user. However, users can also create their own variables:

```
[student@stationX ~]$ HI="Hello, pleased to meet you."
[student@stationX ~]$ echo $HI
Hello, pleased to meet you.
```

When setting variables at the command line, do not include spaces between the variable, the equals sign and the value. Entering something like `HI = "Hello, pleased to meet you"` would probably generate a syntax error because **bash** will try to execute a command called **HI**, which is unlikely to exist.

Two types of variables exist: local variables, also called shell variables, and environment variables. The difference between the two is that the shell will pass the environment variables on to commands that it calls, but it will not pass on local variables. As a result, local variables are used to configure the shell itself, while environment variables are used to configure other commands.

The **set**, **env**, and **echo** commands can be used to display all variables, environment variables, and a single variable value, respectively. Examples:

```
[student@stationX ~]$ set | less
[student@stationX ~]$ env | less
[student@stationX ~]$ echo $HOME
/home/student
```

Some Common Variables

- Configuration variables
  - PS1: Appearance of the **bash** prompt
  - PATH: Directories to look for executables in
  - EDITOR: Default text editor
  - HISTFILESIZE: Number of commands in **bash** history
- Information variables
  - HOME: User's home directory
  - EUID: User's *effective UID*

The PS1 variable sets the prompt. The prompt can vary each time it is displayed by using special escaped sequences. For a complete list see the PROMPTING section of the **bash** man page. Some common escape sequences are:

\h	short hostname (not the fully qualified domain name)
\u	user name (useful if you have multiple accounts)
\w	the current working directory
\!	the history number of the current command
\$	shows $ if you are a non-privileged user and a # if privileged user, useful if you sometimes become superuser.

Consider the following prompt setting where the quotes are important as they ensure a space between the prompt and command to be typed:

```
[student@stationX ~]$ PS1="\u@\h:\w <\!>\$ "
digby@kennel:/tmp <1067>$
```

The PATH variable determines the location and order of directories to be searched by the shell when a user wishes to execute a command. eg. a when a user issues the command `ls` the shell iterates through the directories specified by PATH until the executable `ls` is found.

The EDITOR variable is often used by applications that can invoke an external text editor. A user can set this variable to their preferred editor so that, for example, the `crontab -e` command allows the user to manipulate their crontab file with their chosen application.

The HISTFILESIZE variable determines the maximum number of lines saved in a user's .bash_history file.

LS_COLORS is a highly configurables variable which determines how file extensions and permissions can change the way that the `ls` command presents filenames. eg. executable files and files ending in .sh are usually represented in green type.

The information variables HOME and EUID are set according to the user's entry in the /etc/passwd file to their home directory and user ID respectively.

Aliases

- Aliases let you create shortcuts to commands

```
$ alias dir='ls -laF'
```

- Use **alias** by itself to see all set aliases
- Use **alias** followed by an alias name to see alias value

```
$ alias dir
alias dir='ls -laF'
```

redhat. 12-6

Aliases

Aliases are shortcut names for longer commands. If you have commands that you run often, but take a considerable amount of typing, you can reduce these to an alias:

```
[student@stationX ~]$ alias lf="ls -FCa"
```

The alias value must be a single word and so you will almost always want to quote the value as shown.

Even a relatively short command, if executed often, can save considerable typing if reduced to an alias:

```
[student@stationX ~]$ alias c=clear
```

Aliases can also be used for security purposes to force you to use certain flags:

```
[student@stationX ~]$ alias rm="rm -i"
```

In this case, if you ever want to use the **rm** command itself, instead of your alias, you can precede the command with a backslash:

```
[student@stationX ~]$ \rm -r Junk
```

How bash Expands a Command Line

1. Split the line into words
2. Expand aliases
3. Expand curly-brace statements ({})
4. Expand tilde statements (~)
5. Expand variables ($)
6. Command-substituation ($() and ``)
7. Split the line into words again
8. Expand file globs (*, ?, [abc], etc)
9. Prepare I/O redirections (<, >)
10. Run the command!

redhat 12-7

The process by which the shell interprets elements of the command line is extraordinarily complex. Here, however, is a relatively simple version of the overall process:

1. The line is split into shell words, delimited by spaces, tabs, new lines, and a few other characters, and possibly overridden by single quotes, double quotes, and backslashes. Tokens: space tab newline " ' | & ; ( ) < >
2. Function and alias expansion is performed. Whether a function or an alias takes precedence is a fairly complex matter.
3. Curly brace expansion, for such sequences as "cmd.{o,c}", is performed. This may cause a change in word count, of course. Tokens: { , }
4. Tilde expansion is performed: ~, ~/, and ~username are all substituted for the appropriate strings. Token: ~
5. Parameter and variable substitution is performed. This includes arithmetic and command substitution. In other words, all substitutions beginning with a $, plus the backquote characters. If there are multiple substitutions, then changes will be made on the line from left to right. Common tokens: $ ${ } $(()) $[] $() `
6. The line is split into shell words again.
7. File glob expansion is performed. Tokens: * ? [ ]
8. File redirection is performed. Common tokens: > >> < << 2> 2>>
9. The command is executed!

Preventing Expansion

- Backslash (\ ) makes the next character literal

```
$ echo Your cost: \$5.00
Your cost: $5.00
```

- Quoting prevents expansion
  - Single quotes (') inhibit all expansion
  - Double quotes (") inhibit all expansion, except:
    - $ (dollar sign) - variable expansion
    - ` (backquotes) - command substitution
    - \ (backslash) - single character inhibition
    - ! (exclamation point) - history substitution

redhat 12-8

Escaping characters

Protecting characters from expansion with the backslash (\) is very important for some operations, such as when using find to print filenames that match a pattern:

```
[student@stationX ~]$ find / -name foo*
```

In this case the * is expanded by the shell. If the pattern matches a single filename in the current directory (such as foobar or foofram), the expanded name will be passed to find as a parameter. Thus, the find command will return only files with that specific name, instead of files with the pattern `foo*`.

To find all file and directory names that begin with foo, "escape" the wildcard character to pass it intact to **find**.

```
[student@stationX ~]$ find / -name foo\*
```

Inhibiting expansion with quotes

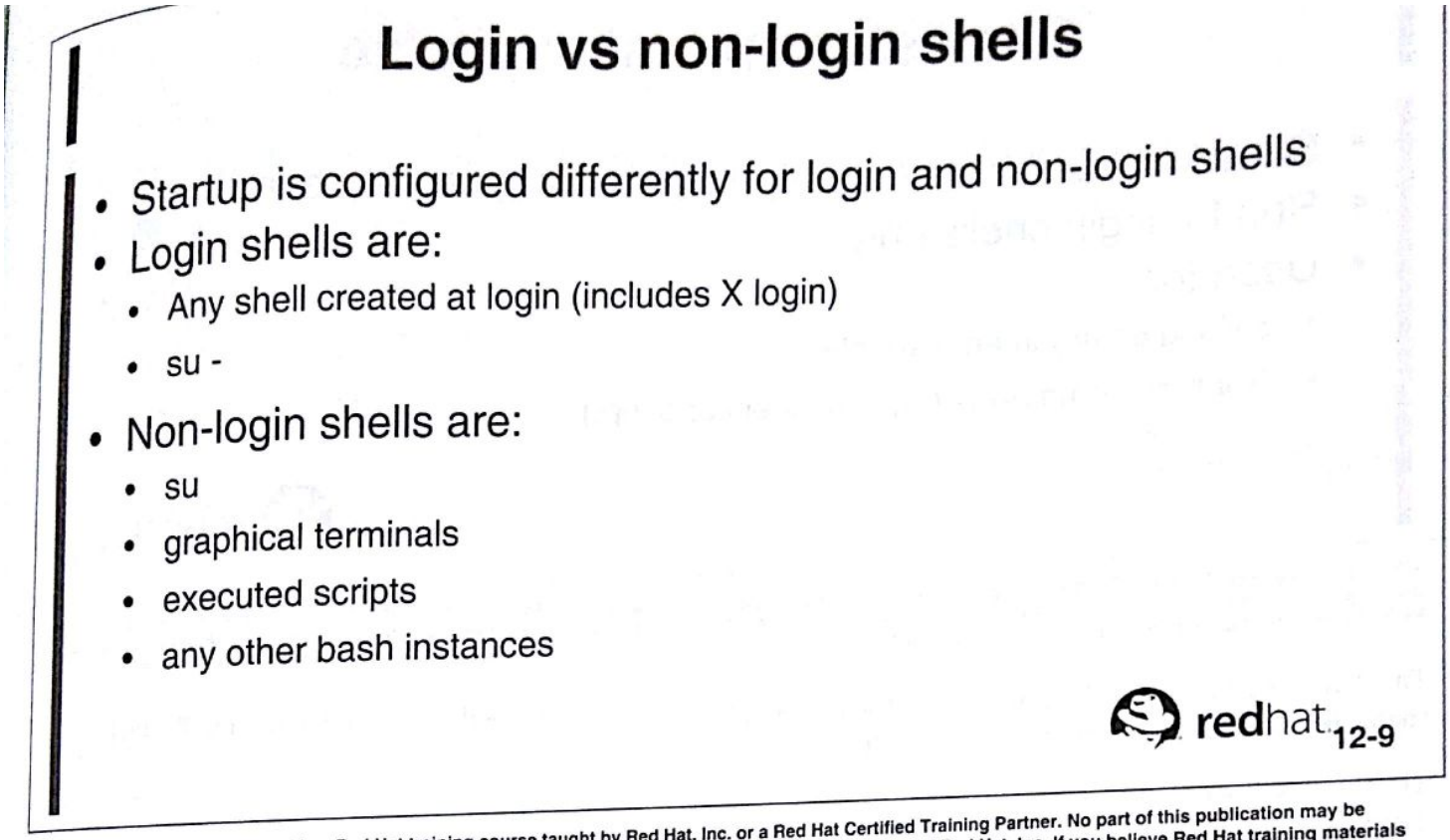
The use of the backslash to inhibit shell expansion of the command line can be burdensome if many characters need to be escaped. For example:

```
[student@stationX ~]$ echo \*\*\* SALE \*\*\*
```

Not only is this displeasing to the eye, but it also invites typing errors. To inhibit multiple characters from being expanded it is useful to use either single or double quotes:

```
[student@stationX ~]$ echo '*** SALE ***'
```

The difference between using single and double quotes is that single quotes will inhibit just about all command line expansion, whereas double quotes will inhibit some expansion, but allow others. As a rule of thumb, double quotes inhibit file name generation expansion, but not other types.



Most of the configuration items that we have seen in this unit are only valid in the given shell. But typically, we will want settings to be established every time we start a shell, rather than having to type in all of our variables, aliases, and other commands on a per shell basis.

To accomplish this, we use startup scripts, scripts that run when a shell is created. The system administrator sets up some startup scripts, but individual users can control startup by editing scripts in their home directories.

A critical concept to understand when it comes to startup scripts is the idea of the *login shell*. A login shell is a shell that someone started by logging into the system. A non-login shell is a shell started up in some other way, perhaps by a user or a program issuing the **bash** command.

The login shell is an important concept because different startup scripts run, depending on whether a shell is a login shell or a non-login shell.

Bash startup tasks: profile

- Stored in /etc/profile (global) and ~/.bash_profile (user)
- Run for login shells only
- Used for
  - Setting environment variables
  - Running commands (eg mail-checker script)

redhat 12-10

The `/etc/profile` shell script is the first startup script run when a login shell is started. It only runs for login shells; non-login shells do not invoke this script.

The script will set a series of variables including `PATH`, `USER`, `LOGNAME`, `MAIL`, `HOSTNAME`, `HISTSIZE`, and `INPUTRC`.

It will also run the scripts found in the `/etc/profile.d` directory.

Login shells first call `/etc/profile`, which calls `/etc/profile.d`. Then, the file `~/.bash_profile` is called. This file, in turn, calls `~/.bashrc`, which calls `/etc/bashrc`. Each script in turn can undo changes made in previously called scripts. For example, the **PATH** variable is set in the `/etc/profile` script, but is then modified in the `~/.bash_profile` script.

Bash startup tasks: bashrc

- Stored in /etc/bashrc (global) and ~/.bashrc (user)
- Run for all shells
- Used for
  - Setting local variables
  - Defining aliases

redhat 12-11

Non-login shells reference some of the same files, but in a different order. Non-login shells first call `~/.bashrc`. This calls `/etc/bashrc`, which calls `/etc/profile.d`. Note that the `/etc/bashrc` file only calls `/etc/profile.d` for non-login shells; for login shells, the previously called `/etc/profile` calls the `/etc/profile.d` scripts.

Typical sorts of commands placed in startup scripts include:

local variable settings, particularly `PS1`

environment variable settings, such as `PATH` or `LESS`

aliases, or perhaps unaliases to remove undesired aliases, set globally in earlier scripts

a umask, to be discussed in a later unit

Bash exit tasks

- Stored in ~/.bash_logout (user)
- Run when a login shell exits
- Used for
  - Creating automatic backups
  - Cleaning out temporary files

redhat 12-12

There are certain tasks that a user may wish to be performed when they log out of a system. Tasks of this type could include deleting temporary files as well as creating backups of files which the user is working on.

To accomplish this all commands placed in **.bash_logout**, which can be found in a user's home directory, will be executed when a login shell exits. For example a user may login through a display managed graphical login, work on some files and then log out. This **.bash_logout** script is then automatically called when the X session is ended.

Scripting: Taking input with positional Parameters

- Positional parameters are special variables that hold the command-line arguments to the script.
- The positional parameters available are `$1`, `$2`, `$3`, etc. . These are normally assigned to more meaningful variable names to improve clarity.
- `$*` holds all command-line arguments
- `$#` holds the number of command-line arguments

redhat 12-13

Many commands and scripts can perform different tasks depending on the arguments supplied to the program. These values are assigned to positional parameters that may be accessed in the script. The variable `$0` is reserved and specifies the program name as it was executed on the command line. Variables above `$9` require special handling and so they must be enclosed in curly braces, e.g. `${11}`.

All positional parameters are read only variables. In most cases they are reassigned to other more meaningful names which improve the readability of the script. In addition, reassigning the variables allows read - write access to the information contained within the positional parameter.

Example:

```
#!/bin/bash
# positionaltester
# Demonstrates the use of positional parameters
################################################################################
echo "The program name is $0"
printf "The first argument is %s and the second is %s\n" $1 $2
echo -e "\nAll command line parameters are $*\n"
```

Output:

```
[student@stationX ~]$ ./positionaltester Red Hat Enterprise Linux
The program name is ./positionaltester
The first argument is Red and the second is Hat
All command line parameters are Red Hat Enterprise Linux
```

Scripting: Taking input with the read command

- Use **read** to assign input values to one or more shell variables:
  - **-p** designates prompt to display
  - **read** reads from standard input and assigns one word to each variable
  - Any leftover words are assigned to the last variable
  - `read -p "Enter a filename: " FILE`

redhat 12-14

read takes a line from standard input and breaks it down into individual words. (Usually a word is defined as a character string surrounded by white space such as spaces and tabs). The way the shell interprets words may be changed by setting the `IFS` variable (e.g. `IFS=':'` will tell the shell that words are separated by colons instead of white space). The first word is assigned to the first variable and the second word is assigned to second variable, and so on. If there are more words than variables, the last variable is assigned all the remaining words.

Example:

```
#!/bin/bash
read -p "Enter name ( first last ): " FIRST LAST
echo "Your first name is $FIRST and your last name is $LAST"
```

The **-p** option is used to display a prompt string. Place quotes around the string if you need to prompt the user with a multiple-word command.

Example:

```
#!/bin/bash
read -p "Enter several values:" value1 value2
value3
echo "value1 is $value1"
echo "value2 is $value2"
echo "value3 is $value3"
```

find

- **find [directory...] [criteria...]**
- Searches directory trees in real-time
  - Slower but more accurate than **locate**
  - CWD is used if no starting directory given
  - All files are matched if no criteria given
- Can execute commands on found files
- May only search directories where the user has read and execute permission

redhat 13-5

Unlike **locate**, **find** will do a real time search of the machines file system to find files that match the criteria of the command line arguments. But realize that since **find** is looking at files in the file system as your user account, you must have read and execute permission on a directory to examine its content.

find requires an argument of what directory it should start finding files in. So if you only wanted to find in a user student's home directory you would give **find** a starting directory of `/home/student`. If you would like **find** to look over the entire system, you would provide a starting directory of `/`.

find has a huge amount of options that can be provided to describe exactly what kind of file should be found. You can search based on file name, file size, last modified time stamp, inode number, and many, many more.

The ability to locate files based on almost any combination of criteria is powerful, but it is only part of what makes **find** such a useful tool. **find** can also execute arbitrary commands on any of the files that it matches, greatly simplifying some otherwise mind-numingly repetitive tasks.

Commands can be run on found files by using the **-ok** and **-exec** options. If **-ok** is used, you will be prompted for each file before the specified command is run. If **-exec** is used, the command will be run on all matching files with no prompts for confirmation. When specifying the command to run, the name of the found file can be represented with `{}` and the command must be terminated with a `Space` followed by `\;`. For example, the following command would fix any files that are writable by the `other` group, which is considered a security risk.

```
[root@stationX ~]# find / -perm -002 -exec chmod o-w {} \;
```

Basic *find* Examples

- **find -name snow.png**
  - Search for files named snow.png
- **find -iname snow.png**
  - Case-insensitive search for files named snow.png, Snow.png, SNOW.PNG, etc
- **find -user joe -group joe**
  - Search for files owned by the user *joe* and the group *joe*

Finding files based on their name will, unlike **locate**, look for files that are named the exact string passed to the **find** command. That is to say, if a **find** command was used like:

```
[student@stationX ~]$ find / -name .png
```

findwould only return the files that were named **.png**, not files that contained in their name the string **.png**. Fortunately, you can use shell wild cards with **find**, but they must be quoted. As an example:

```
[student@stationX ~]$ find / -name "*.png"
```

would find all files on the system that have **.png** as the end of their name. Related to **-name** is the **-iname** option which works the same way as **-name** but performs a case-insensitive search.

The **-regex** option in **find** does not work quite the way one would expect. **-regex** applies the regular expression to the name of the file, including the absolute path to the file. So in the above example, the regular expression **".*W.*\.png"** will match all files that have a capital **W** and **.png** in their names. If we had instead used **"W.*\.png"**, we would get no results because the full path to our file name will always begin with a / and our provided regular expression indicates we are only interested in files whose full path name begins with a **W**, something that can never be true.

find and Logical Operators

- Criteria are ANDed together by default.
- Can be OR'd or negated with **-o** and **-not**
- Parentheses can be used to determine logic order, but must be escaped in bash.
  - **find -user joe -not -group joe**
  - **find -user joe -o -user jane**
  - **find -not \(-user joe -o -user jane \)**

redhat 13-7

By default, if multiple criteria are given to **find**, they must all apply to a file for it to be considered a match. For example, the following will only match files whose names end in `.png` AND are owned by the user `student`:

```
[student@stationX ~]$find / -name "*.png" -user student
```

This behavior can be overridden with the `-o` option. The following command will match files whose names end in .png OR are owned by `student`:

```
[student@stationX ~]$find / -name "*.png" -o -user student
```

find also includes a logical "not" operator. Options preceded with a **!** or the **-not** option will cause find to look for the opposite of the given criterion. So the following command will match files whose names end in `.png` and are NOT owned by `student`.

```
[student@stationX ~]$find / -name "*.png" -not -user student
```

With the addition of logical operators, the question of operator precedence is raised. Logical ANDs have a higher priority than logical ORs, and logical NOTs have the highest priority of all. To force precedence of an expression, you can enclose options that should be grouped together in parentheses. Make sure to put spaces after the \(and before the \) or find will not work as expected. The find man page has more details about this in the OPERATORS section.

find and Permissions

- Can match ownership by name or id
  - **find / -user joe -o -uid 500**
- Can match octal or symbolic permissions
  - **find -perm 755** matches if mode is *exactly* 755
  - **find -perm +222** matches if *anyone* can write
  - **find -perm -222** matches if *everyone* can write
  - **find -perm -002** matches if other can write

find can search for files based on their ownership or permissions. Useful options when searching by owner are:

-user and **-group**: Search by name of user or group

-uid and **-gid** Search by user ID or group ID

The **-perm** option is used to look for files with a particular set of permissions. Permissions can be described as octal values (some combination of 4, 2 and 1 for read, write and execute, respectively) or using symbolic notation (eg `u+w` for "user has write access"). Permissions should be preceded by a + or - sign. The meanings of these operators are slightly different depending on whether you are using numeric or symbolic notation.

A numeric permission preceded by + will match files that have at least one bit (user, group or other) for that permission set. So, for example, a file with permissions `r--r--r--` would not match +222, but one with `rw-r--r--` would. A - sign before a permission means that all three instances of that bit must be on, so neither of the previous examples would match but something like `rw-rw-rw-` would.

So, to use a more complex example, the following command would match any file for which the user has read, write and execute permissions, the group has read permissions and others have read-only access:

```
[student@stationX ~]$ find -perm 764
```

To match files for which the user has *at least*read, write and execute permissions, the group has *at least* read permissions and others have *at least* read access:

```
[student@stationX ~]$ find -perm -764
```

And to match files for which the user has read, write and execute permissions, *or* the group has at least read permissions *or* others have at least read access:

```
[student@stationX ~]$ find -perm +764
```

When used with + or -, a value of 0 works like a wildcard, since it means "a permission of at least nothing. Thus, the following command would match any file for which others have at least read access:

```
[student@stationX ~]$ find -perm -004
```

find and Numeric Criteria

- Many **find** criteria take numeric values
- **find -size 1024k**
  - Files with a size of *exactly* 1 megabyte
- **find -size +1024k**
  - Files with a size *over* 1 megabyte
- **find -size -1024k**
  - Files with a size *less than* 1 megabyte

find can search your system for files that comply with certain numeric criteria such as the size of the file (**-size**), the number of links to the file (**-links**) the date of the last change to the file's data, (**-mtime**), date of the last change to the file's metadata (**-ctime**) or the date of the last time the file was read (**-atime**). All of these criteria accept a numeric value. When you provide numeric values to find you can look for an exact match, more than the number, or less than the number. For example:

```
[student@stationX ~]$ find / -atime 5
```

looks for files on the system whose last accessed time stamp (see the next slide for more information on **-atime**) is exactly five days ago, whereas:

```
[student@stationX ~]$ find / -atime +5
```

will find files whose last accessed time stamp is more than five days ago. Lastly:

```
[student@stationX ~]$ find / -atime -5
```

will print the list of files whose last accessed time stamp is less than five days ago.

find and Access Times

- **find** can match by inode timestamps
  - **-atime** when file was last read
  - **-mtime** when file data last changed
  - **-ctime** when file data or metadata last changed
- Value given is in days
  - **find -ctime -10**
    - Files modified less than 10 days ago

redhat 13-10

Examples of **find** timestamp-matching criteria are given in the notes of the previous slide. While the values passed to **-atime**, **-ctime** and **-mtime** are measured in days, there are also corresponding criteria that perform searches in minutes: **-amin**, **-cmin** and **-mmin**. You can even match access times relative to the timestamps of other files using **-anewer**, **-cnewer** and **-newer**, which tests mtime. For example:

```
[student@stationX ~]$ find -newer recent_file.txt
```

Would list all files with mtimes more recent than that of `recent_file.txt`. Note that there is no **-older** argument. To match files older than `recent_file.txt` you would simply negate the **-newer** criteria:

```
[student@stationX ~]$ find -not -newer recent_file.txt
```

Remember that the metadata, including all three timestamps, for a file can be manually examined using the **stat** command.

Executing Commands with find

- Commands can be executed on found files
  - Command must be preceded with **-exec** or **-ok**
    - **-ok** prompts before acting on each file
  - Command must end with ***Space*\ ;**
  - Can use {} as a filename placeholder
  - **find -size +102400k -ok gzip {} \;**

Using the **-exec** or **-ok** options with find will cause **find** to execute a command once for each file that matches the given criteria. This is commonly used for things like removing files or renaming files to have a certain extension. Be extremely careful when using **-exec** because it may perform the action on many files (remember that find recurses through subdirectories) and it does not ask for confirmation. Remember that running the search without **-exec** will list all matches, thus allowing you to preview which files will be acted upon. Alternately you can use the **-ok** option, which causes **find** to ask for each file.

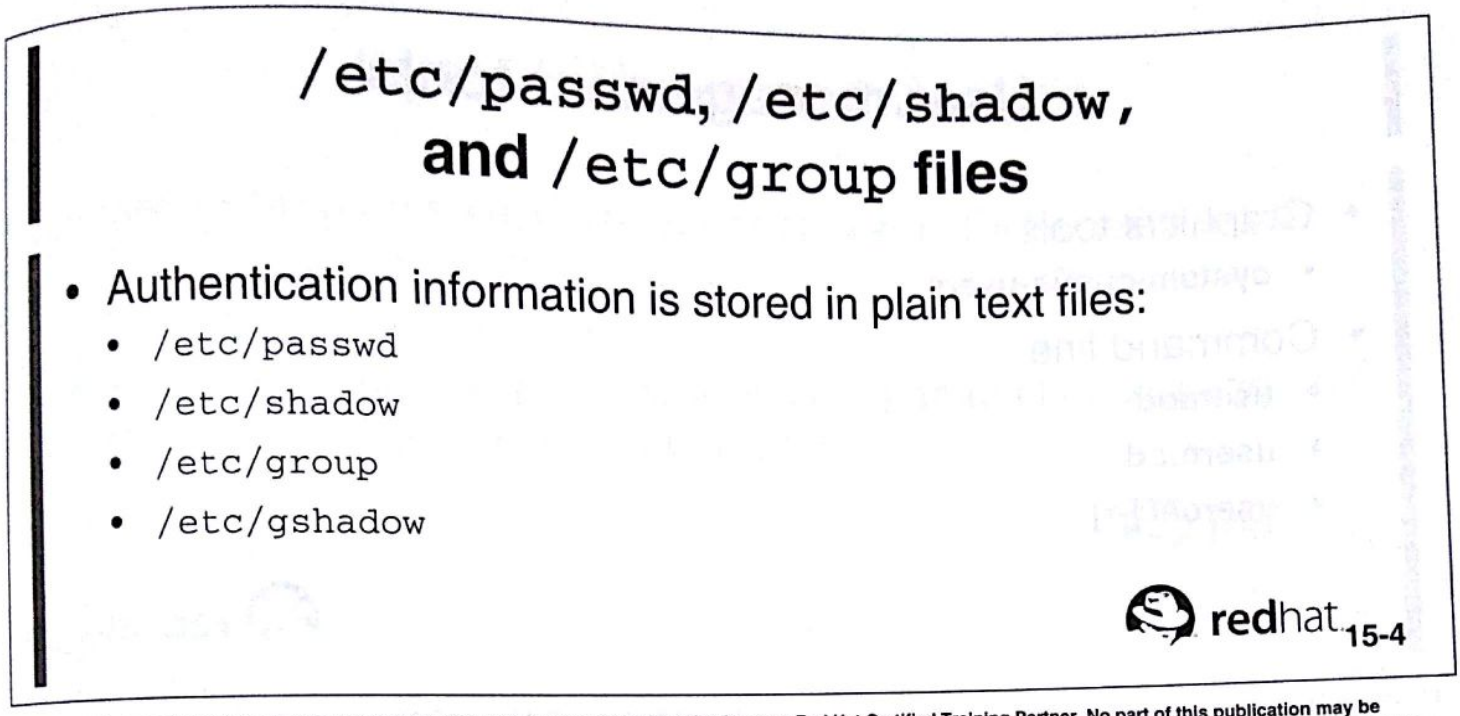
The reason that the commands given with **-exec** and **-ok** must end in a **\;** is because find uses **;** as the delimiting character. Unfortunately **;** is also a delimiting character for the shell so we must prevent bash from interpreting it. When a character is prepended with a backslash (\), **bash** is instructed to treat it literally, so typing **\;** at the **bash** command prompt will send **;** to **find** after **bash** has done its interpretation.

find Execution Examples

- **find -name "*.conf" -exec cp {} {}.orig \;**
  - Back up configuration files, adding a `.orig` extension
- **find /tmp -ctime +3 -user joe -ok rm {} \;**
  - Prompt to remove Joe's tmp files that are over 3 days old
- **find ~ -perm +o+w -exec chmod o-w {} \;**
  - Fix other-writable files in your home directory

redhat 13-12

You can use **find** to execute any command you would like, and if you use the {} as filename place holders, the **find** command will put the file name of a file it has found in place of {} and execute the command.



User and Group Information Files

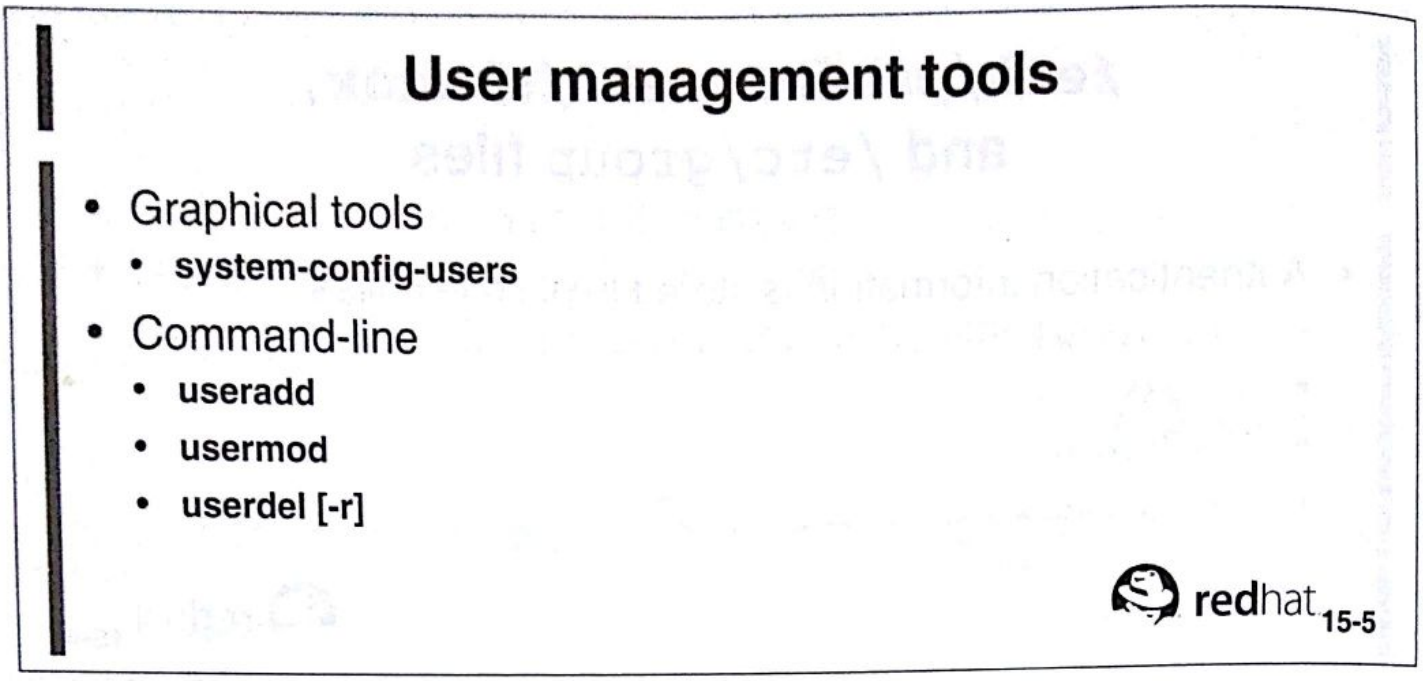
When a user runs a command such as **ls -l** that displays user and group information about files, the numeric information is translated into names; it is the names that are displayed. The mappings of numbers to names are in the files `/etc/passwd` and `/etc/group`. The `/etc/shadow` file maps user names to their encrypted passwords and password and account expiration information. All files are colon separated.

The `/etc/passwd` file contains seven fields: user name, password placeholder (for historical reasons), uid number, gid number of the user's primary group, GECOS field (typically containing the user's real name), home directory, and shell to be run when a user logs in.

The `/etc/group` file contains four fields: group name, group password placeholder, gid number, and a comma separated list of group members.

The `/etc/shadow` file is referenced when someone logs in: the file contains a mapping of a user name to a password. For a complete list of the fields, see the man page:

man 5 shadow



User management tools

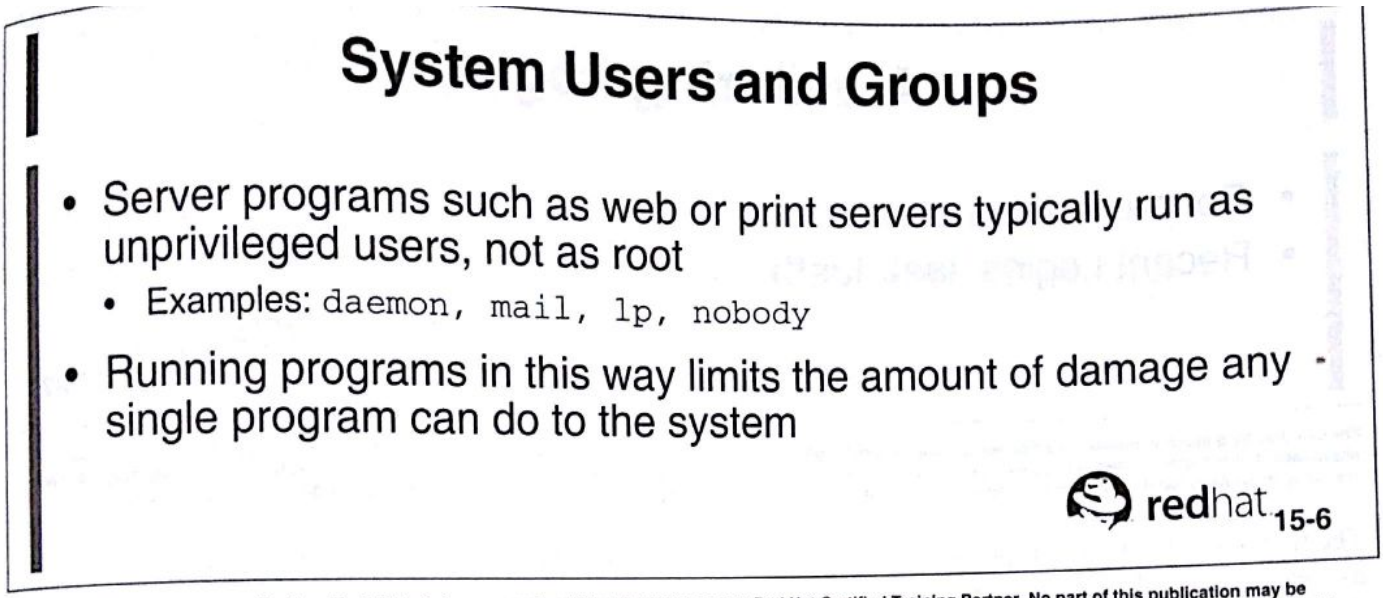
You have at least two ways to manage user accounts, either graphically or on the command-line. You could also directly alter the files with , but that can be more error-prone.

Using the graphical application, you would click on the Users and Groups entry of the System/Administration menu. At this point you would be able to add or remove users and groups. When you do so, the files seen in the previous slides are altered.

If you prefer the command-line, e.g. you have many users to create at once, the **useradd** and **userdel** commands are available.

When you create a user account, some files might automatically be copied from a skeleton.

When you remove a user account from your system, the files and directories of this user may still remain on the system. To also remove them during the account removal, you can use the **-r** switch to **userdel**.



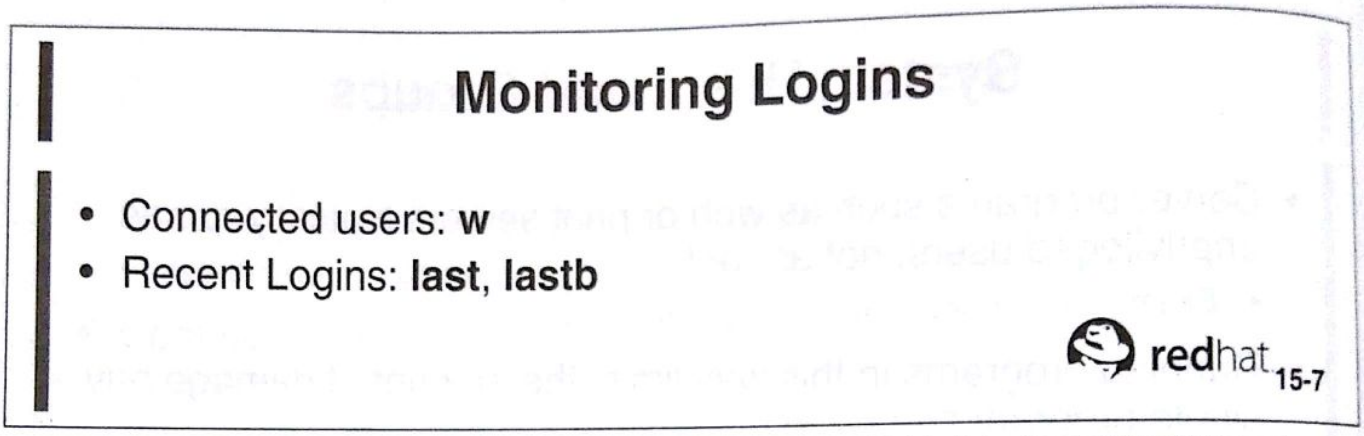
System Users and Groups

In addition to the ordinary user accounts and the `root` account for the superuser, a number of system users and groups exist. These accounts exist primarily so that server programs can run as non-privileged users or as particular groups.

System users and groups all have uid and gid numbers between 1 and 499. This excerpt from `/etc/passwd` shows several system users:

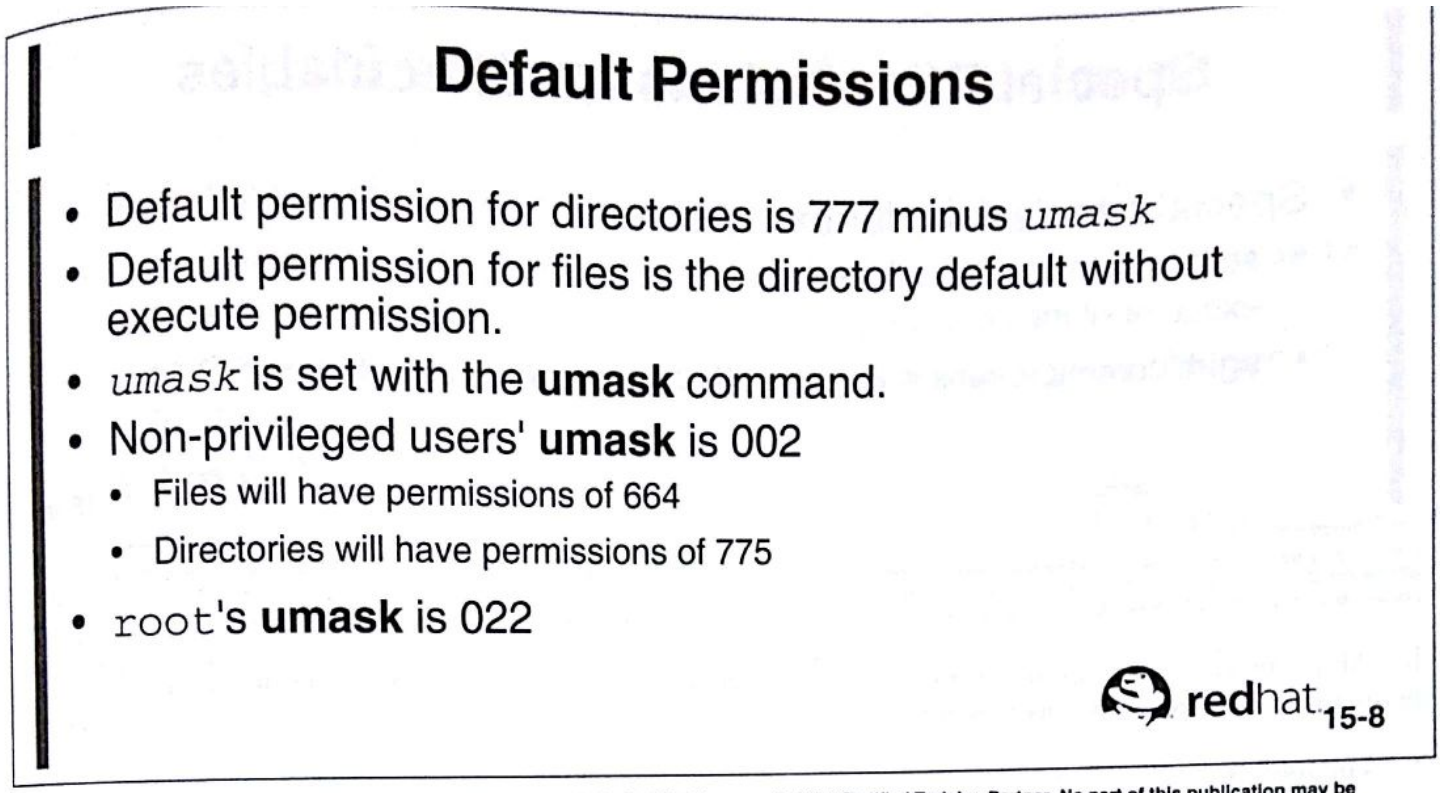
```
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
```

Check the `/etc/group` file for sample system groups.



The **w** command is used to display information about connected users and the processes they are running as long as they are attached to a TTY or in the background. Information on load average and other system statistics is also displayed.

last displays a connection and reboot history. In other words, it displays a list of the most recently logged-in users, as well as the times of all recent system reboots. You can append a username to **last** (eg **last root**) and it will display this information for that account. If you are only interested in failed attempts to access an account (*bad logins*), you can use **lastb** instead.



Default Permissions and umask

Without a umask in effect, any file created will have 666 permissions and any directory created will have 777 permissions. This means that anyone on the system will have read and write access to any newly-created file and similarly full permissions to any directories. In order to withhold permissions one can use a *umask*, which lists the permissions to withhold. A umask of 002 will result in files created with 664 permissions and directories with permissions 775.

Note that execute privilege is always denied a newly-created file, regardless of the umask in effect. Execute privilege always has to be explicitly granted to a file. Execute permission is given to a directory upon creation, unless explicitly denied by the umask.

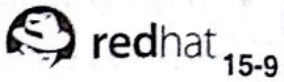
The default umask on a Red Hat Enterprise Linux system is 002. To change your umask to 022, use the **umask** command:

```
[student@stationX ~]$ umask 022
```

umask is typically set by scripts run at login time. As a result, the next time you log in your umask will be set back to your default unless you add the command to one of your startup files, such as `.bashrc`.

Special Permissions for Executables

- Special permissions for executables:
  - **suid**: command run with permissions of the *owner* of the command, not executor of the command
  - **sgid**: command runs with group affiliation of the group of the command



In addition to the user, group and other permissions, an additional set of permissions exist called the special permissions. The special permissions are:

- The *suid*, or *set user ID*, bit
- The *sgid*, or *set group ID*, bit
- The *sticky bit*

The suid and sgid permissions are effective for executable regular files; the sticky bit and the sgid permission are effective for directories.

To set the special permissions, use the **chmod** command, preceding the usual three digits with a digit representing the special permission or permissions that you wish to have set: 4 for suid, 2 for sgid, 1 for the sticky bit. Example:

chmod 3775 groupdir

This would set both the sgid permission and the sticky bit for the directory groupdir. You can also use Nautilus to set special permissions. To do this, right-click on a file or directory and choose Properties, then navigate to the Permissions tab. You will see a checkbox for each special permission under the Special Flags heading.

The SetUID Permission

When the suid special permission is set for an executable, it means that the command will run with the authority of the owner of the file, rather than the authority of the user running the command. An example is the **passwd** command. The **passwd** command changes a user's password, which is stored in the `/etc/shadow` file and is not writable by non-privileged users. However, since the **passwd** command is owned by `root` and runs with the suid permission, users running the command have the privilege of `root` while changing their passwords. Hence, they have permission to edit `/etc/shadow`.

In a long listing, the suid permission is displayed as a lower case "s" where the "x" would otherwise be located for the user permission (an upper case "S" would be present if the underlying executable permission was not set):

```
[student@stationX ~]$  chmod 4551 passwd

[student@stationX ~]$  ls -l passwd
 -r-sr-x--x   1 root    root     15368 May 28  2002 passwd
```

The Set GID Permission

Similarly, commands running with the sgid permission run with the group affiliation of the group of the command.

Special Permissions for Directories

- Special permissions for directories:
  - sticky bit: files in directories with the sticky bit set can only be removed by the owner and root, regardless of the write permissions of the directory
  - sgid: files created in directories with the sgid bit set have group affiliations of the group of the directory

15-10

The Sticky Bit

The sticky bit for a directory sets a special restriction on deletion of files: with the sticky bit set, only the owner of the file, and the superuser, can delete files within the directory. An example of a directory with the sticky bit set is `/tmp` (the "t" denotes that the sticky bit is set):

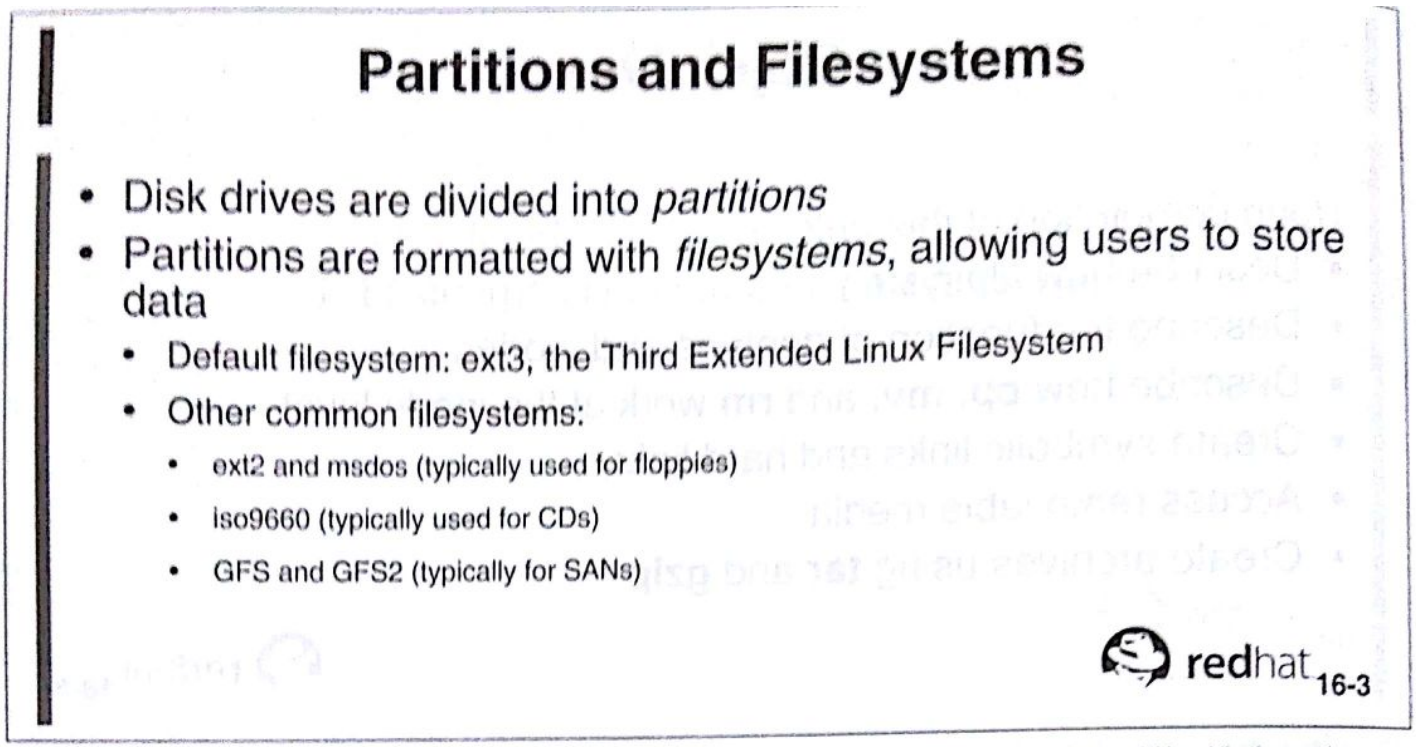
```
[student@stationX ~]$ chmod 1777 /tmp

[student@stationX ~]$ ls -ld /tmp
drwxrwxrwt   30 root     root      4096 Mar  9 10:25 /tmp
```

The sgid permission for a directory means that files created in the directory will inherit its group affiliation from the directory, rather than inheriting it from the user. This is commonly used on group directories:

```
[student@stationX ~]$ chmod 2770 GroupDir

[student@stationX ~]$ ls -ld GroupDir

drwxrws---    2 digby    penguin   4096 Mar  9 10:35 GroupDir
```

Often both the sticky bit and the sgid permission will be set on a group directory:

```
[student@stationX ~]$ chmod 3770 GroupDir

[student@stationX ~]$ ls -ld GroupDir
drwxrws--T    2 digby    penguin   4096 Mar  9 10:35 GroupDir
```



An overview of filesystems in Red Hat Enterprise Linux

On Red Hat Enterprise Linux systems, disk drives and other storage media typically are divided into partitions. Also typically, a partition is formatted with a filesystem. A filesystem is a data structure written to the media that allows users to store and access files.

The default filesystem for Red Hat Enterprise Linux is the *Third Extended Linux Filesystem* or *ext3*. It is an enhanced version of ext2 that uses journaling to improve filesystem data integrity. ext2 has been in use on Linux since 1993 making it and its successor, ext3, very stable and reliable. ext2 and ext3 support a variety of advanced features which are not always present in other Unix filesystems such as extended attributes (EAs) and POSIX Access Control Lists (ACLs).

Red Hat Enterprise Linux supports over many different filesystem types. Other common filesystems are msdos, which is used for compatibility on floppy disks and iso9660 which is used on CDs. It is also common to format floppies with ext2 because it is more powerful and flexible than msdos and uses less disk space for its data structures than ext3. Newer filesystems include GFS and GFS2 which are used on SANs but can be used in other environments as well.

Inodes

- An *inode table* contains a list of all files in an ext2 or ext3 filesystem
- An *inode* (index node) is an entry in the table, containing information about a file (the *metadata*), including:
  - file type, permissions, UID, GID
  - the link count (count of path names pointing to this file)
  - the file's size and various time stamps
  - pointers to the file's data blocks on disk
  - other data about the file

What is an inode?

ext2 and ext3 file systems keep a list of the files they contain in a table called an *inode table*. An individual entry in the inode table is called an *inode*. The inode is referenced by its number, the *inode number*, which is unique within a file system.

The inode contains the metadata about files (remember that everything in Linux is a file; the term "file" is being used in this broad manner in the inode discussion and so can include a directory, for example). Among the data stored in the inode is:

- the file type (regular file, directory, etc.)
- file permissions
- link count: the number of file names associated with the inode number (more on this later)
- the user ID number of the file owner and the group ID number of the associated group
- time stamps, including last access time, last modification time, and last inode change time
- location of the data on the hard disk
- other metadata about the file

Directories

- The computer's reference for a file is the *inode number*
- The human way to reference a file is by *file name*
- A *directory* is a mapping between the human name for the file and the computer's inode number

What is a directory?

We commonly think of a directory as a container for files and other directories. In fact, a directory is a mapping between the file names that humans use to reference files and inode numbers used by the computer to reference files.

When a filename is referenced by a command or application, Linux references the directory in which the file resides, determines the inode number associated with the file name, looks up the inode information in the inode table, and, if the user has permission, returns the contents of the file.

The **ls -i** command displays the inode number:

```
[student@stationX ~]$ ls -il
total 28
80788 -rw-r-----   1 student  student     5120 Sep 18 11:26 myData
37777 drwxr-x---   2 student  student     4096 Sep 18 11:25 newStuff
80787 -rw-r-----   1 student  student     1536 Sep 18 11:26 notes
80789 -rw-r-----   1 student  student    11776 Sep 18 11:26 otherStuff
```

The inode number for the `myData` file is 80788.

Inodes and Directories

Name

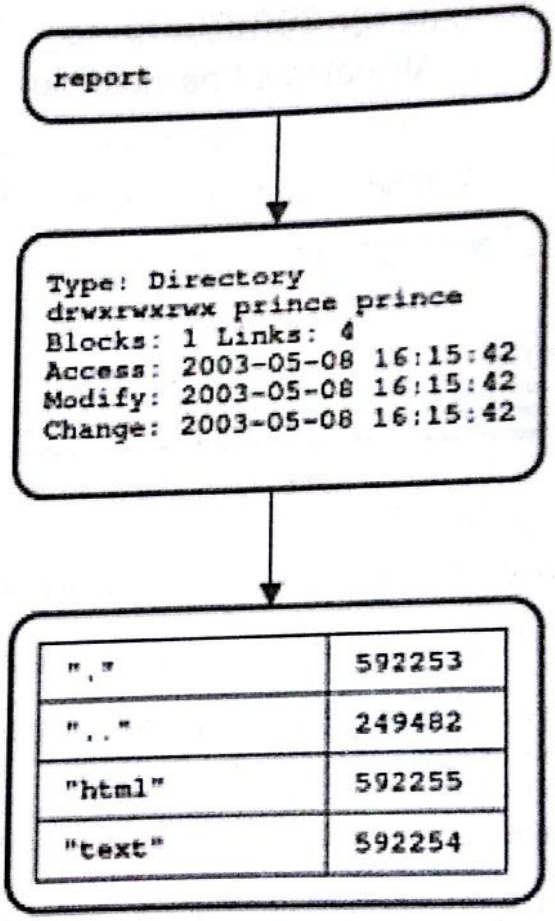
Associated with an inode by parent directory

Inode Metadata

Properties and a pointer to blocks on disk

Contents

For directories: name/inode list (shown)
For files: arbitrary data



cp and inodes

- The **cp** command:
  1. Allocates a free inode number, placing a new entry in the inode table
  2. Creates a dentry in the directory, associating a name with the inode number
  3. Copies data into the new file

cp in-depth

Consider the nature of the **cp**, **mv**, and **rm** commands in the context of the inode table.

When a file is copied to a new name in the same directory, the directory and the inode table get a new entry:

```
[student@stationX ~]$ ls -li penguin
246674 -rw-rw-r--    1 digby   digby         26 Sep 25 20:56 penguin

[student@stationX ~]$ cp penguin tux

[student@stationX ~]$ ls -li penguin tux
246674 -rw-rw-r--    1 digby   digby         26 Sep 25 20:56 penguin
246575 -rw-rw-r--    1 digby   digby         26 Sep 25 20:56 tux
```

mv and inodes

- If the destination of the **mv** command is on the same file system as the source, the **mv** command:
  1. Creates a new directory entry with the new file name
  2. Deletes the old directory entry with the old file name
- Has no impact on the inode table (except for a time stamp) or the location of data on the disk: no data is moved!
- If the destination is a different filesystem, **mv** acts as a copy and remove

16-8

mv in-depth

The name of the **mv** command works neatly with the metaphor that a directory is a container for files. But, as we now know that a directory is a mapping between file names and inode numbers, the **mv** command must be understood in these terms.

When a file is moved, the underlying file, either as inode entry or as data on the hard disk, does not move. What moves is the entry in a directory. When a file is moved within the same file system, only the directory itself (or directories if moving between directories) are affected:

```
[student@stationX ~]$ ls -li tux
246575 -rw-rw-r--  1 digby  digby  26 Sep 25 20:56 tux

[student@stationX ~]$ mv tux fedora

[student@stationX ~]$ ls -li fedora
246575 -rw-rw-r--  1 digby  digby  26 Sep 25 20:56 fedora
```

The inode number remains the same. The data on the file system is not moved. The inode is not change, except that the inode change time is updated.

rm and inodes

- The **rm** command:
  1. Decrements the link count, thus freeing the inode number to be reused
  2. Places data blocks on the free list
  3. Removes the directory entry
- Data is not actually removed, but will be overwritten when the data blocks are used by another file

rm in-depth

The **rm** command has a broad effect. The entry in the directory is effectively deleted; the inode number is made available; the block locations that the file was using are placed on the free list.

The underlying data is not actually removed, however. The data will be overwritten eventually when the blocks are reused by some other file, but the data is otherwise unmodified.

Hard Links

- A hard link adds an additional pathname to reference a single file
  - One physical file on the filesystem
  - Each directory references the same inode number
  - Increments the link count
    - The **rm** command decrements the link count
    - File exists as long as at least one link remains
    - When the link count is zero, the file is removed
  - Cannot span drives or partitions
- Syntax:
  - **ln *filename* [*linkname*]**

Hard links

A hard link is one of the more difficult types of files to understand. A hard link is a path name that references an inode: that is, all files are hard linked at least once. The interesting twist in the Linux world is that a file can be hard linked more than once. Remember that an individual file is referenced by its inode number, the file name merely being a human convenience for referencing the inode number. Because the name of a file is separate from an inode (it is stored in a directory, not in the inode), it is possible to have multiple file names pointing to the same inode number.

To create an additional hard link to an existing file, use the **ln** command:

```
[student@stationX ~]$  ln fedora redhat

[student@stationX ~]$  ls -li fedora redhat
246575 -rw-rw-r--    2 digby   digby         26 Sep 25 20:56 fedora
246575 -rw-rw-r--    2 digby   digby         26 Sep 25 20:56 redhat
```

The most notable effect here is that the two files, fedora and redhat, have the exact same inode number: there is only one underlying file, but there are two entry points. Also note that they are both regular files: an additional hard link is not a separate file type. Finally, note the link count. The link count is the number after the permissions and before the user name. The link count has been incremented to two because two path names point to the same file. When one path name is deleted, perhaps using the **rm** command, the link count will decrement to one; when the final path name is deleted, the link count is decremented to zero and the file is removed.

Two restrictions to additional hard links should be noted: first, the two path names must be on the same filesystem: because they share an inode number and an inode table is unique to a file system, both must be on the same file system. Second, it is not possible to use the **ln** command to create additional hard links to directories.

Symbolic (or Soft) Links

- A symbolic link points to another file
  - **ls -l** displays the link name and the referenced file

```
lrwxrwxrwx 1 joe joe 11 Sep 25 18:02 pf -> /etc/passwd
```

  - File type: **l** for symbolic link
  - The content of a symbolic link is the name of the file that it references
- Syntax:
  - `ln -s filename` ***`linkname`***

16-11

Symbolic Links

A symbolic link (or symlink) is a file that points to another file. That is, when you read from the symbolic link using, perhaps, the **cat** or **less** command, you will actually receive the contents of the underlying file. Most actions, in fact, will be performed on the underlying file, with the exception of the **rm** command. Removing a symlink removes the actual link itself, not the underlying file.

To create a symbolic link, use the **ln** command with the **-s** option:

```
[student@stationX ~]$  ln -s /etc/passwd password
```

In this example the file passwd in the current directory points to `/etc/passwd`. The **ls -l** command lists the symlink and the file that is being referenced by the link:

```
[student@stationX ~]$ ls -li password  /etc/passwd
30338 -rw-r--r--  1 root    root      1729 Aug 24 11:43 /etc/passwd
33276 lrwxrwxrwx  1 digby   digby       11 Sep 26 09:33 passwd -> /etc/passwd
```

Note a few interesting things about this long listing. First, it has its own inode number: a symbolic link is a separate file from the original. Second, the first character of a long listing for a symlink is the letter l: a symlink is a special type of file and so it has its own file type indicator. Third, note the odd permissions of the symbolic link. The permissions on a symlink are irrelevant; the permissions set on the file pointed to by the symlink control access rights. Finally, note the size of the symbolic link. The size in this case is 11: there are 11 characters in `/etc/passwd`. The contents of a symlink is the path name that the symlink references. Therefore, the size of the symlink is always the number of characters in the path name.

The Seven Fundamental Filetypes

ls -l symbol	File Type
-	regular file
d	directory
l	symbolic link
b	block special file
c	character special file
p	named pipe
s	socket

16-12

Identifying basic file types

In the preceding pages, we have learned about three file types: regular files, directories, and symbolic links. In Red Hat Enterprise Linux, there are a total of seven file types, called the Seven Fundamental Filetypes of Linux and UNIX. Below is a list of the remaining file types and a brief explanation of each, preceded by the symbol indicating the file type in a long listing:

c *character special file*: in the overview that we stated that everything in Linux is a file, even hardware. Files referencing hardware are not regular files; they are one of two types of special files. Character special files are used to communicate with hardware one character at a time.

b *block special file*: used to communicate with hardware a block of data at a time: 512 bytes, 1024 bytes, 2048 bytes: whatever is appropriate for that type of hardware. Generally, block and character special files are located in the `/dev` directory. Run the following command to see a listing:

```
[student@stationX ~]$ ls -l /dev | less
...output omitted....
```

p *named pipe*: a file that passes data between processes. It stores no data itself, but passes data between one process writing data into the named pipe and another process reading data from the named pipe. A named pipe can be created using the **mknod** command:

```
[student@stationX ~]$ mknod mypipe p
```

s *socket*: a stylized mechanism for inter-process communications. It is extremely rare for a user or even a system administrator to explicitly create a socket.

Checking Free Space

- **df** - Reports disk space usage
  - Reports total kilobytes, kilobytes used, kilobytes free *per file system*
  - **-h** and **-H** display sizes in easier to read units
- **du** - Reports disk space usage
  - Reports kilobytes used *per directory*
  - Includes subtotals for each subdirectory
    - **-s** option only reports single directory summary
  - Also takes **-h** and **-H** options
- Applications->System Tools->Disk Usage Analyzer or **baobab** - Reports disk space usage graphically

*Getting usage summaries with **df***

The **df** and **du** commands are used to determine how much disk space is used. The **df** command reports on a per filesystem basis. It reports total disk space, disk space used, and disk space free. The disk space free column is often the most useful information:

```
[student@stationX ~]$ df /opt
Filesystem      1K-blocks      Used Available Use% Mounted on

/dev/hda2        16512384   5134496  10539108  33% /opt
```

With typically large file systems, the units above can be hard to read. The **-h** option used multipliers such as G and M for gigabytes and megabytes, respectively:

```
[student@stationX ~]$  df -h /opt
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda2              16G  4.9G   11G  33% /opt
```

*Getting detailed usage information with **du***

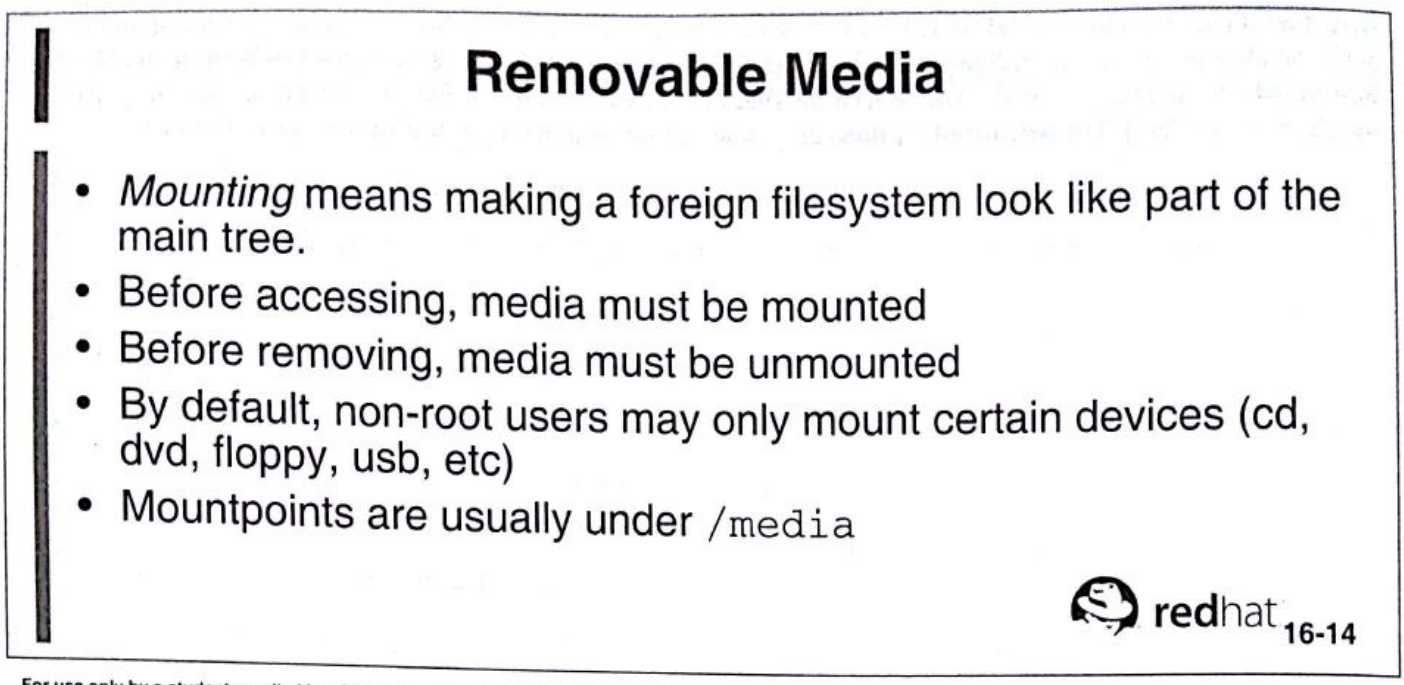
The **du** command reports the number of kilobytes contained by the items within a directory. By default, **du** will report on the given directory, plus all subdirectories. The **-s** option can be used to request only the summary directory information. The **-h** option is also available. Example:

```
[student@stationX ~]$  du -s /etc
62552   /etc
```

Displaying detailed disk usage graphically

The graphical Disk Usage Analyzer, which can be called using the **baobab** command, or can be called through the menu selection Applications->System Tools->Disk Usage Analyzer, displays disk usage by directory. From the

Directory Tree tab, select **Folder** to determine how disk space is used within a directory, or select **Filesystem** to get a complete breakdown of disk usage for the entire filesystem tree per directory. From the **File Search** tab, select Edit->Find... to find files by name. The **Search for file:** field takes filenames or filename generation search symbols such as *, ?, or [and]. The **Advanced Options** drop down allows searches by modification date or file size.



Handling removable media

Before you can access the data on newly inserted removable media (floppy disk, CD, zip disk), the filesystem(s) on the media must first be mounted. Although mounting filesystems is generally a system administrator-only function, non-privileged users can mount removable media in a number of ways.

Non-privileged users logged into the console are permitted to mount and unmount removable media using commands such as:

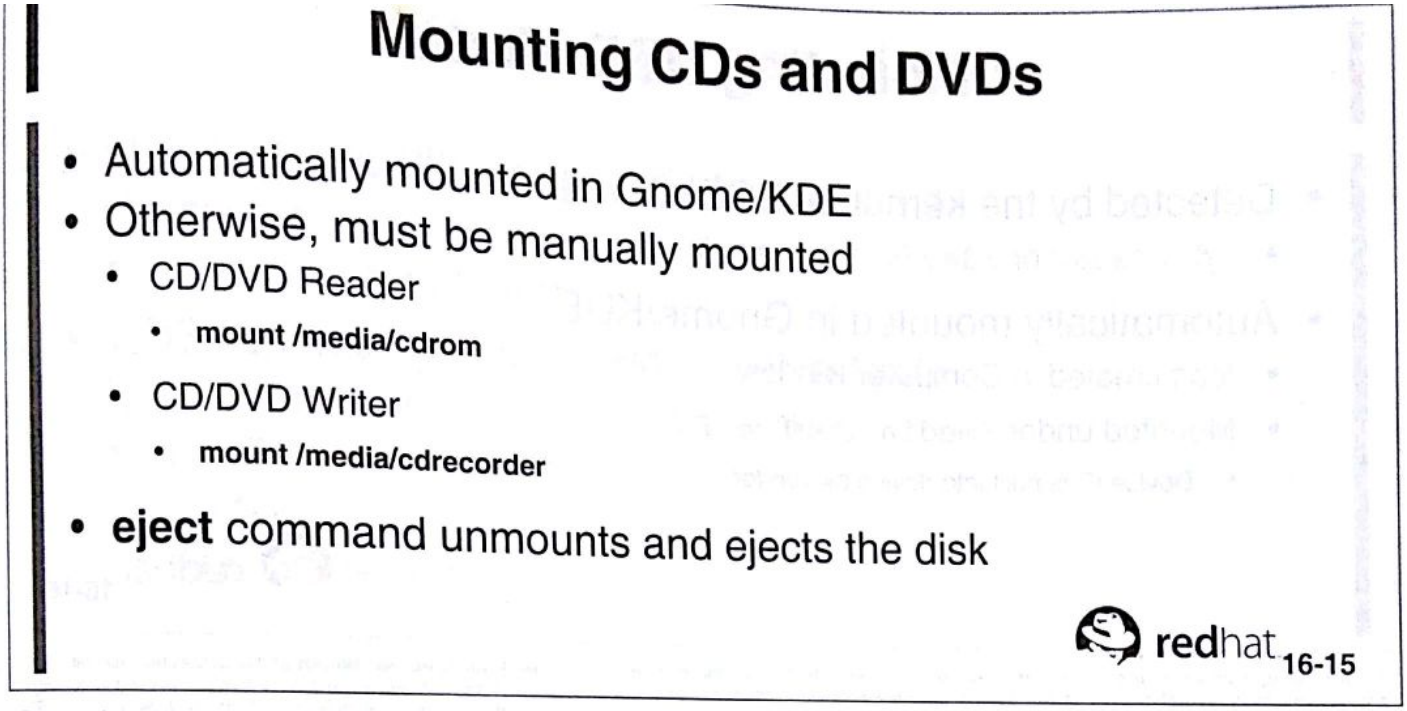
```
[student@stationX ~]$ mount /media/floppy          mount /media/cdrom
umount /media/floppy
umount /media/cdrom
```

Note that if you have a cd writer it will be mounted under `/media/cdrecorder` instead of `/media/cdrom`.

An alternative to mounting disks is to use the **mtools** commands. The mtools commands mimic standard DOS commands. The a: drive is generally mapped to your floppy. You can list, copy, format, and otherwise manipulate removable media, particularly floppies, using DOS commands preceded by an "m":

```
[student@stationX ~]$ mdir a:
...output omitted...
[student@stationX ~]$ mcopy myfile a:
```

These command only work on DOS formatted floppies. For a complete list of the mtools commands, run **mtools**.

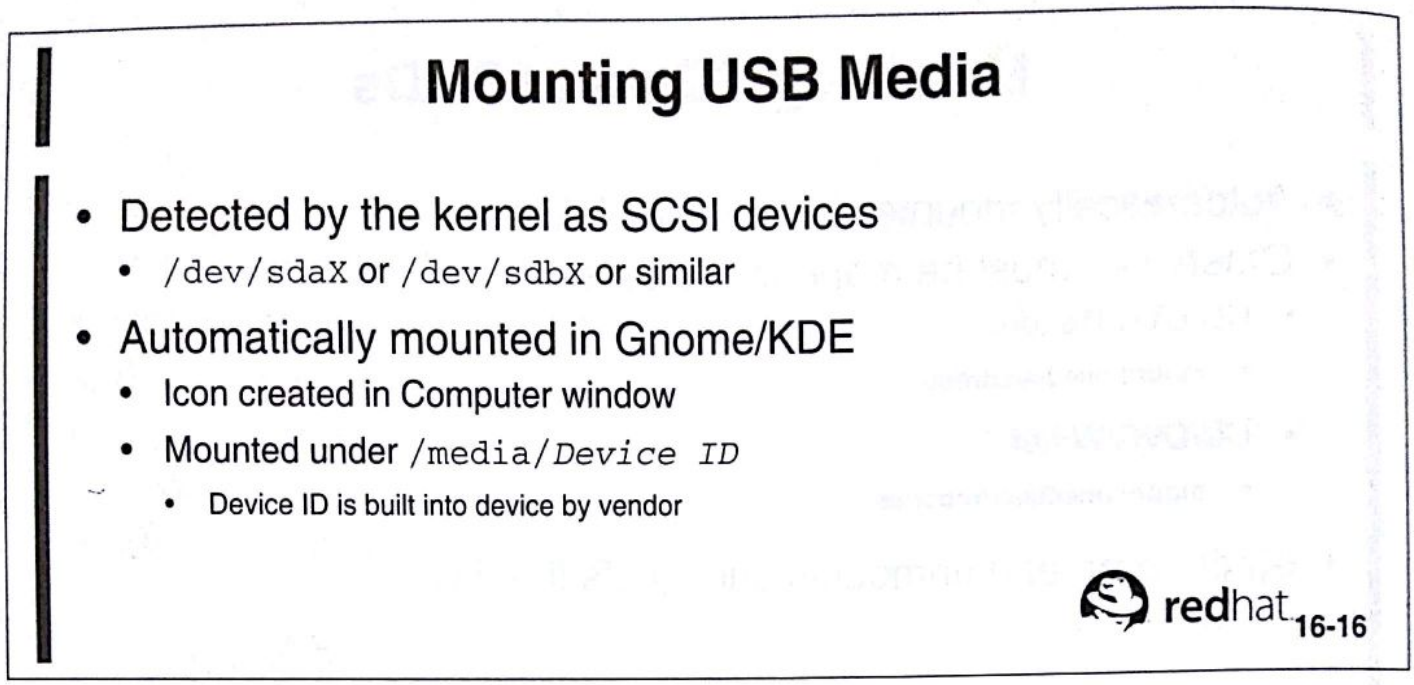


Accessing CDs and DVDs

When using the X Window System, inserting a CD into the drive automatically mounts the CD and adds an icon to the desktop.

The mountpoint associated with a device, which you will need to know when mounting it manually, depends on whether you are using a CD/DVD reader or or writer. Readers are mounted under `/media/cdrom` and writers under `/media/cdrecorder`.

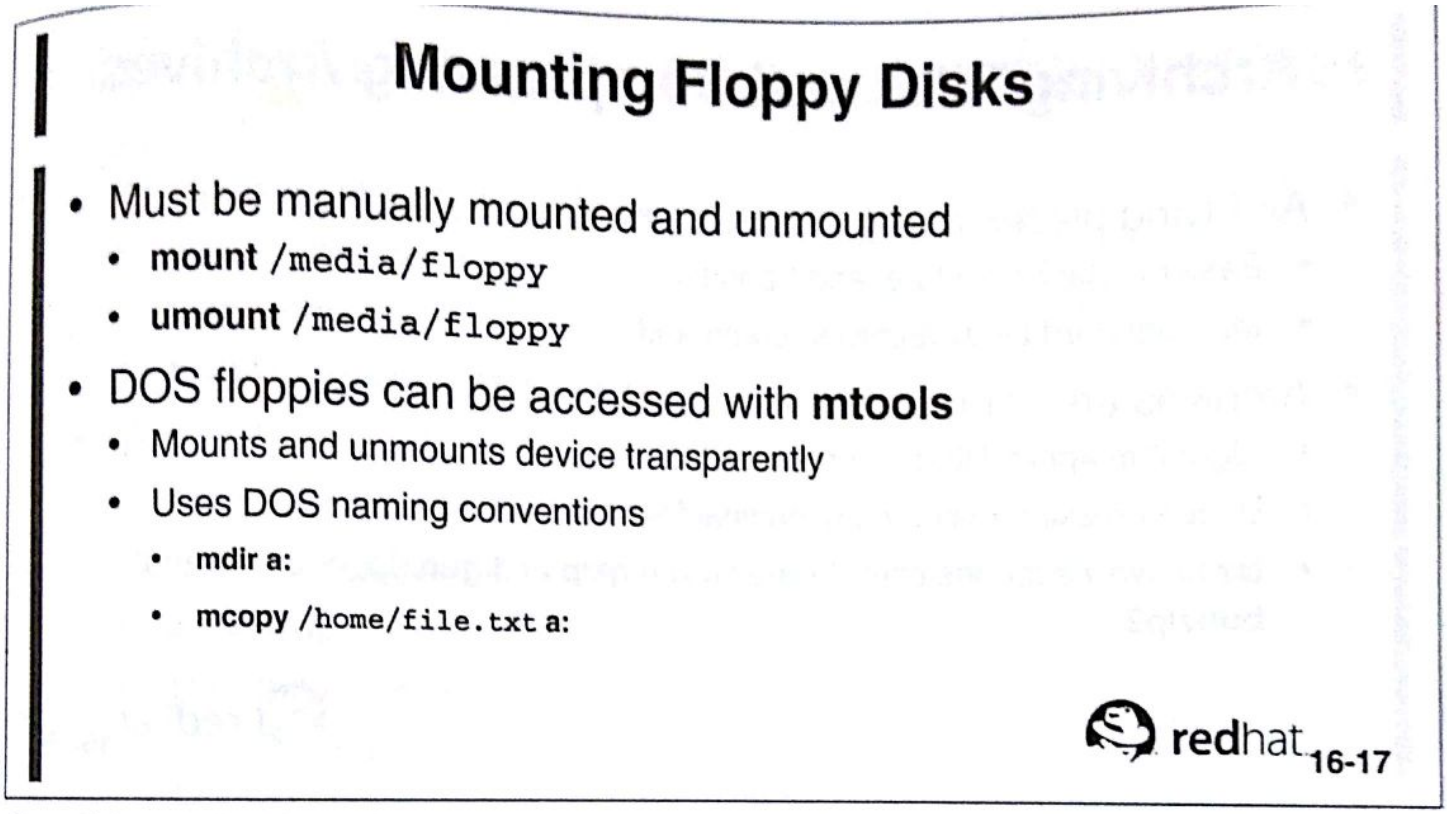
When you are done using the disk you can eject it by either right-clicking on the desktop icon and selecting "Eject" or running the **eject** command from a prompt. If you have a single device, running eject by itself is sufficient. If you have multiple devices you will have to include the appropriate device node as an argument.



Accessing USB storage

Most vendors give USB disks a label. **fstab-sync** reads that label and automatically puts an entry in `/etc/fstab` for it, mounting it in `/media/label`.

USB disks are treated as SCSI devices, thus they are referenced as `/dev/sda`, `/dev/sdb`, etc. Normally, USB memory sticks use the first partition (e.g., `/dev/sda1`), but they may use other partition--`/dev/sda4` is common. Check your logs for information regarding USB devices and partitions after you plug in the USB disk.



*Accessing floppies with **mtools***

An alternative to mounting disks is to use the **mtools** commands. The **mtools** commands mimic standard DOS commands. While **mtools** can be configured to recognize any device, the only one they recognize automatically is the floppy drive which is mapped to "a:"". You can list, copy, format, and otherwise manipulate the media, using DOS commands preceded by an "m" (**mcopy**, **mdir**, **mdel**, etc). For a complete list of the **mtools** commands, run **mtools** and/or consult the **mtools** info page. The **mtools** commands only work on DOS-formatted floppies.

Archiving Files and Compressing Archives

- Archiving places many files into one target file
  - Easier to back up, store, and transfer
  - **tar** - standard Linux archiving command
- Archives are commonly compressed
  - Algorithm applied that compresses file
  - Uncompressing restores the original file
  - **tar** natively supports compression using **gzip** and **gunzip**, or **bzip2** and **bunzip2**

redhat 16-18

The value of file archives

Archiving files is the process of incorporating a copy of those files into a single archive file. This permits you to easily distribute many files by distributing the single archive.

Archiving files is generally a good idea if you want to back up a few directories or transfer many files over a network. Originally, **tar** was used to create archives on tape devices, hence its name - which stands for *tape archive*. While **tar** is seldom used today to back up entire filesystems, it is often used to bundle together related files before moving or compressing them.

tar archives filenames are usually created with `.tar` filename extensions, although this is not required.

Smaller archives are easier to distribute than larger archives, and so it is common to use compression algorithms to compress archives. The **tar** command can automatically compress and uncompress files, giving the appropriate options. Common compression tools include **gzip** and **gunzip** commands, and the newer **bzip2** and **bunzip2** commands.

Creating, Listing, and Extracting File Archives

- Action arguments (one is required):
  - **-c** create an archive
  - **-t** list an archive
  - **-x** extract files from an archive
- Typically required:
  - **-f *archivename*** name of file archive
- Optional arguments:
  - **-z** use **gzip** compression
  - **-j** use **bzip2** compression
  - **-v** be verbose

The **tar** command requires one option to indicate the action to be taken. This could be **-c** to create a tar file, **-t** to list the table of contents of a tar file, or **-x** to extract files from the tar file. Typically, when creating a tar file or extracting contents from a tar file, you will want to see a list of files archived or extracted: use the **-v** option to see this. Finally, to specify the tar file, use the **-f** option.

For example, to create an archive of the `/etc` directory:

```
[root@stationX ~]# tar -cvf /tmp/etc.tar /etc
[output omitted...]
```

List the contents of the tar archive:

```
[root@stationX ~]# tar -tf /tmp/etc.tar | less
[output omitted...]
```

And finally, extract the contents of the archive into another directory:

```
[root@stationX ~]# mdir /tmp/NewETC
[root@stationX ~]# cd /tmp/NewETC
[root@stationX ~]# tar -xvf /tmp/etc.tar
[output omitted...]
```

One unusual thing about the **tar** command: in the examples above, the **tar** command would have worked the same way without the leading dash before the options.

It is typical, but not required, to compress a tar archive. The **tar** command can do this for you, using either the **gzip** compression tool with the **-z** option or the **bzip2** compression tool using the **-j** option.

When using these compression options, typical suffixes are `.tar.gz` or `.tgz` when using the **gzip** option, or `.tar.bz2` when using the **bzip2** option.

Examples:

```
[root@stationX ~]# tar -czvf /tmp/etc.tar.gz /etc
[output omitted...]

[root@stationX ~]# tar -tzf /tmp/etc.tar.gz | less
[output omitted...]

[root@stationX ~]# mdir /tmp/NewETC
[root@stationX ~]# cd /tmp/NewETC
[root@stationX ~]# tar -xzvf /tmp/etc.tar.gz
[output omitted...]

[root@stationX ~]# tar -cjvf /tmp/etc.tar.bz2 /etc
[output omitted...]

[root@stationX ~]# tar -tjf /tmp/etc.tar.bz2 | less
[output omitted...]

[root@stationX ~]# mdir /tmp/NewETC
[root@stationX ~]# cd /tmp/NewETC
[root@stationX ~]# tar -xjvf /tmp/etc.tar.bz2
[output omitted...]
```

Note that whenever using compression options, you must use it for creating, listing, and extracting commands.

Creating File Archives: Other Tools

- **zip** and **unzip**
  - Supports **pkzip**-compatible archives
  - Example:

```
zip etc.zip /etc
unzip etc.zip
```

- **file-roller**
  - Graphical, multi-format archiving tool

redhat 16-20

Other archiving tools include **zip** and **unzip**, which are compatible with **pkzip** archives.

The **file-roller** command is a graphical tool that can be used to view archives in any of a number of formats, including any of the formats, compressed or uncompressed, discussed in this unit. It acts as a file browser, looking into archives and permitting you to extract individual files. The file-roller command can also create archives.

Invoke **file-roller** either by executing **file-roller** on the command line or by selecting Applications->Accessories->Archive Manager. Then, select **Open** to look into an archive. A new window will pop up showing the contents of the archive. Navigate the archive and select a file as you would with any file browser. Move a file to the desktop by dragging and dropping with the left mouse button.

To create an archive, select **New**. Type the filename of the desired archive in the **Name:** field; select a destination folder in the **Save in folder:** field; and select a file type in **Archive type:** field. A window will pop up. Select **Add** to add files and directories to the archive.

Managing Services

- What is a service?
- Graphical Interface to Service Management
  - **system-config-services**
- Command Line Interface to Service Management
  - **service**
  - **chkconfig**

17-5

A service is a linux application that runs in the background listening for requests, or to carry out some specific task(s).

Examples include:

- Apache Web Server, Secure Shell, Sendmail, etc...

Services are managed by System V scripts, the `init` process, `xinetd` super server, or manually using graphical or command-line tools.

- Graphical tool **system-config-services** is accessed from the menu by selecting: System->Administration->Services
- Command-line tools **service** and **chkconfig** are used to start and stop a service, and to ensure persistence across reboots.
  - Running **chkconfig *--list service*** will list which runlevels the Secure Shell server is scheduled to run in:

```
[student@stationX ~]$ /sbin/chkconfig --list sshd
sshd                0:off   1:off   2:on    3:on    4:on    5:on    6:off
```

  - Running **chkconfig *service sshd on | off*** will turn the Secure Shell server on or off in the designated runlevels as indicated in the System V startup script for the Secure Shell server

    Note: You must be root to run the next command.

  - Running **service** without an argument will display basic usage syntax to the standard output:

    Note: You must be root to run the next command.

```
[root@stationX ~]# service
Usage: service < option > | --status-all | [ service_name [ command | ↙
 --full-restart ] ]
```

  - Commands include **start**, **stop**, **status**, **restart**, **reload** and others.
  - So running **service sshd status** as root will display the status (running or stopped) of the Secure Shell server.

Managing Software

- Software is provided as RPM packages
  - Easy installation and removal
  - Software information stored in a local database
- Packages are provided by Red Hat Network
  - Centralized management of multiple systems
  - Easy retrieval of errata packages
  - Systems must be registered first
  - Custom package repositories may also be used

The RPM Package Manager

All software on a Red Hat Enterprise Linux system is divided into RPM packages which can be installed, upgraded, or removed using the RPM package manager.

Package installation is never interactive. In contrast to package management on some other platforms, RPM's design does not provide interactive configuration of software as part of the package load process. RPM can perform configuration actions as part of the installation, but these are scripted not interactive. It is common for packages to install with reasonable default configurations applying. On the other hand some software installs in an un-configured state.

Applies to all software. On some other common platforms, the package management system applies only to part of the installed software. The scope of RPM include core operating systems as well as services and applications. It is common and desirable to run Red Hat Enterprise Linux systems such that all software installed falls under the management of RPM. This is a great aid for management and configuration control, since one framework applies for the management of every installed file.

No such thing as a patch. It is common on other platforms to have operating system updates released as software objects (eg. "service packs") which represent incremental changes to a large number of installed component packages. RPM never does this. If part of any given software package is changed as part of an errata or bug fix, then that entire package will be re-released in its entirety at a new version. The implications are that the installed state of an RPM-managed system can be described as the version number of all the installed components.

Software to be installed using **rpm** is distributed through rpm package files, which are essentially compressed archives of files and associated dependency information. Package files are named using the following format:

`name-version-release.architecture.rpm`

The `version` refers to the open source version of the project, while the `release` refers to Red Hat internal patches to the open source code.

The Red Hat Network (RHN)

The Red Hat Network allows administrators to manage software installations and upgrades efficiently using a combination of your RHN account and the **yum** utility.

The Yum Package Management Tool

- Front-end to **rpm**, replacing **up2date**
- Configuration in `/etc/yum.conf` and `/etc/yum.repos.d/`
- Used to install, remove and list software
  - **yum *install packagename***
  - **yum *remove packagename***
  - **yum *update packagename***
  - **yum *list available***
  - **yum *list installed***

redhat 17-7

The command-line utility **yum** gives you an easy way to manage the packages on your system:

```
[root@stationX ~]# yum install firefox
```

The above command will search the configured repositories for a package named firefox, and if found will install the latest version, pulling in dependencies if needed.

```
[root@stationX ~]# yum remove mypackage
```

The above command will try to remove the package named mypackage from your system. If any other package depends on mypackage **yum** will prompt you about this, giving you the option to remove those packages as well.

```
[root@stationX ~]# yum update [mypackage...]
```

If any packages are specified on the command-line **yum** will search the configured repositories for updated versions of those packages and install them. When no packages are specified **yum** will search for updates to all of your currently installed packages.

```
[root@stationX ~]# yum list available
```

The above command will list all packages in the yum repositories available to be installed.

See **man yum** file for additional options.