

System Administration (2)

IT401

Prepared by
Dr. Ahmed El-Douh

Lecture 3



System Administration 2

Lecture 3

IT401

Access the command line & Manage files from the command line

Prepared by
Dr. Ahmed El-Douh

Objectives

Upon completion of this lecture, you will be able to:

- Access and use local help systems in Red Hat Enterprise Linux to troubleshoot and resolve system issues.
- Create, view, and edit text files using command-line tools and text editors.
- Manage file content and command output effectively to support system configuration and task automation.
- Maintain a smooth workflow while working with system-level tasks in a Linux environment.

Outlines

Part 1: Access the command line

- Log into a Linux system
- run simple commands using the shell.

Part 2: Manage Files from the Command Line

- Copy, move, create, delete, and organize files while working from the bash shell.

Chapter
03

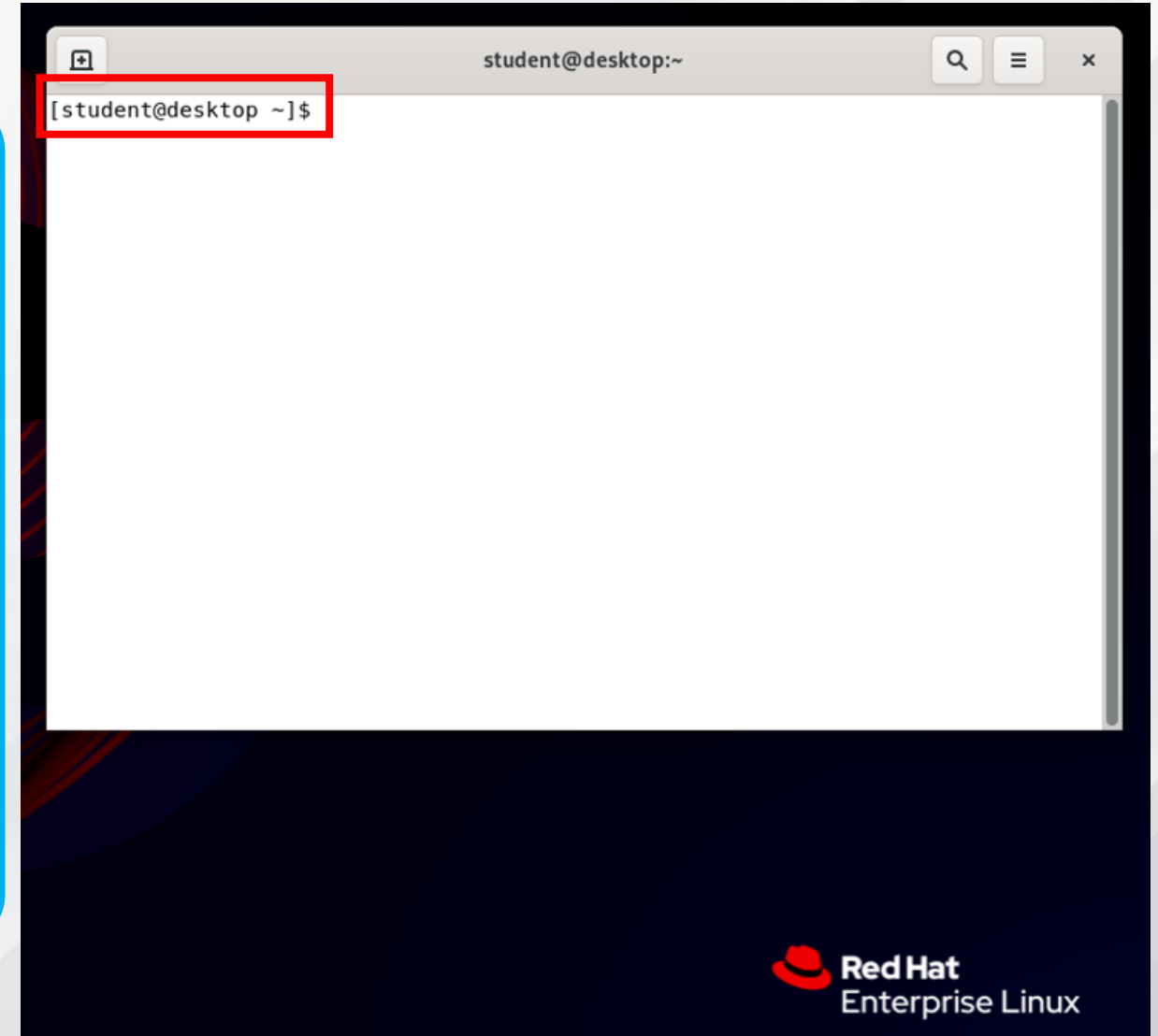
Access the command line



Linux Terminal

- **The Shell Prompt**

When you open a terminal, you'll see a line of text where you can start typing. This is called the **shell prompt**. It tells you that the system is ready for your commands.



Access the Command Line

The **Bash SHELL** A command line is: a text-based interface which can be used to input instructions to a computer system.

- The **Linux command line** is provided by a program called the **shell**.
- The **default shell** for users in Red Hat Enterprise Linux is the GNU Bourne-Again Shell (**bash**).
- **Bash** is an improved version of one of the most successful shells used on UNIX-like systems, the Bourne Shell (sh).
- This is called the shell prompt. When a **regular user** starts a shell, the default prompt ends with a **\$** character.

```
[student@desktopX ~]$
```

- The **\$** is replaced by a **#** if the shell is running as the super user, **root**. This makes it more obvious that it is a super user shell, which helps to avoid accidents and mistakes in the privileged account.

```
[root@desktopX ~]#
```

Access the Command Line

- # Using bash to execute commands can be powerful.
- The **bash shell** provides a scripting language that can support automation of tasks.
- The **shell** has additional capabilities that can simplify or make possible operations that are hard to accomplish efficiently with graphical tools.
- **VIRTUAL CONSOLES** Users access the bash shell through a terminal. A terminal provides a keyboard for user input and a display for output.
- On text-based installations, this can be the Linux machine's physical console, the hardware keyboard and display. Terminal access can also be configured through serial ports.
- Another way to access a shell is from a virtual console.
- A Linux machine's physical console supports multiple virtual consoles which act like separate terminals. Each virtual console supports an independent login session.

Access the Command Line

SHELL BASICS Commands entered at the shell prompt have three basic parts:

1. Command to run. 
2. Options to adjust the behavior of the command. 
3. Arguments, which are typically targets of the command The command is the name of the program to

```
[student@desktop ~]$ ls -l
total 8
drwxr-xr-x. 2 student student 6 Sep 15 22:02 Desktop
drwxr-xr-x. 2 student student 6 Sep 15 22:02 Documents
drwxr-xr-x. 2 student student 6 Sep 15 22:02 Downloads
drwxr-xr-x. 2 student student 6 Sep 15 22:02 Music
drwxr-xr-x. 2 student student 4096 Sep 17 12:44 Pictures
drwxr-xr-x. 2 student student 6 Sep 15 22:02 Public
drwx----- 15 student student 4096 Sep 17 21:19 student
drwxr-xr-x. 2 student student 6 Sep 15 22:02 Templates
drwxr-xr-x. 2 student student 6 Sep 15 22:02 Videos
```

run. 

```
[student@desktop ~]$ ls -l /home
total 8
drwx----- 14 eldouh eldouh 4096 Sep 15 22:07 eldouh
drwx----- 15 student student 4096 Sep 17 21:28 student
```

- It may be followed by one or more options, which adjust the behavior of the command or what it will do. Options normally start with one or two dashes (-a or --all, for example) to distinguish them from arguments.
- Commands may also be followed by one or more arguments, which often indicate a target that the command should operate on. For example, the command line `usermod -L morgan` has a command (`usermod`), an option (`-L`), and an argument (`morgan`).

```
[student@desktop ~]$ usermod -l Morgan
```

Access the Command Line

◆ Understanding Command Usage & Help

- Most commands support --help to show how to use them.

- This displays a **usage statement** showing:

- The command's **syntax**.
- Accepted **options** and **arguments**.

```
[student@desktop ~]$ ls --help
Usage: ls [OPTION]... [FILE]...
```

```
List information about the FILES (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
```

```
Mandatory arguments to long options are mandatory for short options too.
```

```
-a, --all                do not ignore entries starting with .
-A, --almost-all        do not list implied . and ..
--author                with -l, print the author of each file
-b, --escape             print C-style escapes for nongraphic characters
--block-size=SIZE        with -l, scale sizes by SIZE when printing them;
                        e.g., '--block-size=M'; see SIZE format below
-B, --ignore-backups     do not list implied entries ending with ~
-c                      with -lt: sort by, and show, ctime (time of last
                        modification of file status information);
```

◆ Common Usage Statement Conventions

- [] → Optional items
- ... → Repeatable items
- A | B → Choose one only
- < > → Variable input (e.g., <filename>)
- UPPERCASE → Placeholder names (e.g., FILENAME)

Access the Command Line

```
[student@desktopX ~]$ date --help  
date [OPTION]... [+FORMAT]
```

◆ What This Means:

- The date command can accept:
 - A list of **optional options** → [OPTION]...
 - An **optional format string** → [+FORMAT] (starts with a +)
- The format string defines **how** the date and time should be displayed (e.g. +%Y-%m-%d)
- **Both are optional:**
 - If you run date with no options or format, it will still work and print the current date and time in the **default format**.

```
[student@desktop ~]$ date  
Wed Sep 17 09:59:48 PM +03 2025
```

Access the Command Line: **LOCAL CONSOLE ACCESS TERMS**

DESCRIPTION	TERM
The interpreter that executes commands typed as strings.	Shell
The visual cue that indicates an interactive shell is waiting for the user to type a command.	Prompt
The name of a program to run.	Command
The part of the command line that adjusts the behavior of a command.	Option
The part of the command line that specifies the target that the command should operate on.	Argument
The hardware display and keyboard used to interact with a system.	Physical console
One of multiple logical consoles that can each support an independent login session.	Virtual console
An interface that provides a display for output and a keyboard for input to a shell session.	Terminal

Access the Command Line: **EXECUTING COMMANDS USING THE BASH SHELL**

- Each word typed into the shell is separated from each other with spaces. Commands are the names of programs that are installed on the system. Each command has its own options and arguments.
- The **Enter key** is pressed when a user is **ready to execute a command**. Each command is typed on a separate line and the output from each command displays before the shell displays a prompt.
- If a user wants to type more than one command on a single line, a **semicolon, ;**, can be used as a command separator. A **semicolon** is a member of a class of characters called meta characters that has special meanings for bash.

Access the Command Line: **EXECUTING COMMANDS USING THE BASH SHELL**

- The `password` command changes a user's own password. The original password for the account must be specified before a change will be allowed. By default, password is configured to require a strong password, consisting of lowercase letters, uppercase letters, numbers, and symbols, and is not based on a dictionary word.
- The super user can use the `password` command to change other users' passwords.
- Password: all authentication tokens updated successfully.
- Linux does not require file name extensions to classify files by type. The `file` command scans the beginning of a file's contents and displays what type it is. The files to be classified are passed as arguments to the command.

```
[student@desktopX ~]$ passwd
Changing password for user student.
Changing password for student.
(current) UNIX password: old_password
New password: new_password
Retype new password: new_password
```

Access the Command Line: EXECUTING COMMANDS USING THE BASH SHELL

```
[student@desktopX ~]$ file /etc/passwd
/etc/passwd: ASCII text
[student@desktopX ~]$ file /bin/passwd
/bin/passwd: setuid ELF 64-bit LSB shared object, x86-64, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.32,
BuildID[sha1]=0x91a7160a019b7f5f754264d920e257522c5bce67, stripped
[student@desktopX ~]$ file /home
/home: directory
```

- The **head** and **tail** commands display the **beginning and end** of a file respectively.
- By default, these commands display 10 lines, but they both have a -n option that allows a different number of lines to be specified. The file to display is passed as an argument to these commands.

Access the Command Line: EXECUTING COMMANDS USING THE BASH SHELL

```
[student@desktopX ~]$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
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
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
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:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
[student@desktopX ~]$ tail -n 3 /etc/passwd
gdm:x:42:42:./var/lib/gdm:/sbin/nologin
gnome-initial-setup:x:993:991:./run/gnome-initial-setup:/sbin/nologin
tcpdump:x:72:72:./:/sbin/nologin
```

Access the Command Line: EXECUTING COMMANDS USING THE BASH SHELL

```
[student@desktopX ~]$ wc /etc/passwd
 39   70 2005 /etc/passwd
[student@desktopX ~]$ wc -l /etc/passwd ; wc -l /etc/group
 39 /etc/passwd
 63 /etc/group
[student@desktopX ~]$ wc -c /etc/group /etc/hosts
843 /etc/group
227 /etc/hosts
1070 total
```

The **wc** command counts lines, words, and characters in a file. It can take a **-l**, **-w**, or **-c** option to display only the lines, words, or characters, respectively.

Access the Command Line: **TAB COMPLETION**

```
[student@desktopX ~]$ pas<Tab><Tab>
passwd      paste      pasuspender
[student@desktopX ~]$ pass<Tab>
[student@desktopX ~]$ passwd
```

- **Tab completion** allows a user to quickly complete commands or file names once they have typed enough at the prompt to make it unique. If the characters typed are not unique, pressing the Tab key twice displays all commands that begin with the characters already typed.

Access the Command Line: **TAB COMPLETION**

```
[student@desktopX ~]$ pas<Tab><Tab>
passwd      paste      pasuspender
[student@desktopX ~]$ pass<Tab>
[student@desktopX ~]$ passwd
```

- **Tab completion** allows a user to quickly complete commands or file names once they have typed enough at the prompt to make it unique. If the characters typed are not unique, pressing the Tab key twice displays all commands that begin with the characters already typed.

```
[student@desktopX ~]$ ls /etc/pas<Tab>[student@desktopX ~]$ ls /etc/passwd<Tab>
passwd  passwd-
```

- **Tab completion** helps in quickly finding and completing file names or paths

Access the Command Line: **TAB COMPLETION**

```
[root@desktopX ~]# useradd --<Tab><Tab>
--base-dir      --groups        --no-log-init   --shell
--comment       --help          --non-unique    --skel
--create-home   --home-dir      --no-user-group --system
--defaults      --inactive      --password      --uid
--expiredate    --key           --root          --user-group
--gid           --no-create-home --selinux-user
[root@desktopX ~]# useradd --
```

- Arguments and options can be matched with tab completion for many commands. The `useradd` command is used by the superuser, root, to create additional users on the system. It has many options that can be used to control how that command behaves. **Tab completion** following a partial option can be used to complete the option without a lot of typing.

Access the Command Line: **COMMAND HISTORY**

```
[student@desktopX ~]$ history
...Output omitted...
23 clear
24 who
25 pwd
26 ls /etc
27 uptime
28 ls -l
29 date
30 history
[student@desktopX ~]$ !ls
ls -l
total 0
drwxr-xr-x. 2 student student 6 Mar 29 21:16 Desktop
```

```
[student@desktopX ~]$ !26
ls /etc
abrt                hosts                pulse
adjtime             hosts.allow          purple
aliases             hosts.deny            qemu-ga
...Output omitted...
```

The **history** command displays a list of previously executed commands prefixed with a command number.

The exclamation point character, **!**, is a metacharacter that is used to expand previous commands without having to retype them.

!string expands to the most recent command that begins with the string specified.

!number expands to the command matching the number specified.

Access the Command Line: **Editing Command Line**

- The arrow keys can be used to navigate through previous command lines in the shell's history.
- **Up Arrow** edits the previous command in the history list.
- **Down Arrow** edits the next command in the history list. Use this key when the Up Arrow has been pressed too many times. Left Arrow and Right Arrow move the cursor left and right in the current command line being edited.
- The **Esc+.** key combination causes the shell to copy the last word of the previous command on the current command line where the cursor is. If used repeatedly, it will continue to go through earlier commands.
- **EDITING THE COMMAND LINE** When used interactively, bash has a command line-editing feature. This allows the user to use text editor commands to move around within and modify the current command being typed.
- **Using the arrow keys** to move within the current command and to step through the command history was introduced earlier in this session. More powerful editing commands are introduced in the following table.

Access the Command Line: **Editing Command Line**

Useful command line-editing shortcuts

SHORTCUT	DESCRIPTION
Ctrl+a	Jump to the beginning of the command line.
Ctrl+e	Jump to the end of the command line.
Ctrl+u	Clear from the cursor to the beginning of the command line.
Ctrl+k	Clear from the cursor to the end of the command line.
Ctrl+Left Arrow	Jump to the beginning of the previous word on the command line.
Ctrl+Right Arrow	Jump to the end of the next word on the command line.
Ctrl+r	Search the history list of commands for a pattern.

There are several other command line-editing commands available, but these are the most useful commands for beginning users. The other commands can be found in the **bash(1)** man page.

BASH COMMANDS AND KEYBOARD SHORTCUTS

DESCRIPTION	SHELL COMMAND
Jump to the beginning of the previous word on the command line.	Ctrl+Left Arrow
Separate commands on the same line.	;
Clear from the cursor to the end of the command line.	Ctrl+k
Re-execute a recent command by matching the command name.	!<i>string</i>
Shortcut used to complete commands, file names, and options.	Tab
Re-execute a specific command in the history list.	!<i>number</i>
Jump to the beginning of the command line.	Ctrl+a
Display the list of previous commands.	history
Copy the last argument of previous commands.	Esc+.

Chapter
03

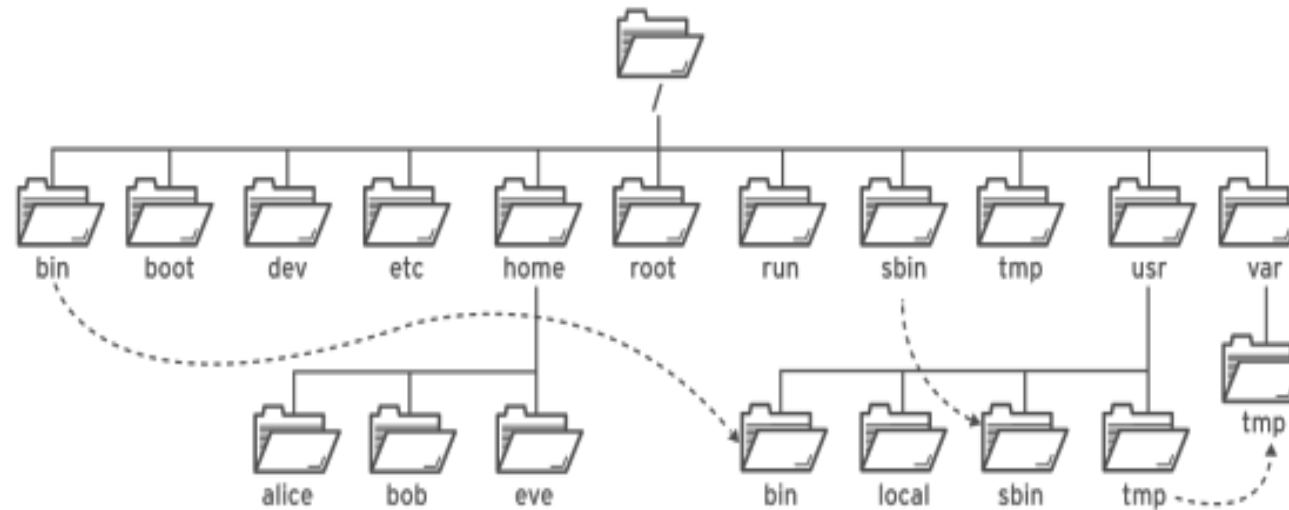
Manage Files from the Command Line



Managing Files from the Command Line

THE FILE SYSTEM HIERARCHY:

- All files on a Linux system are stored on file systems which are organized into a single inverted tree of directories, known as a file system hierarchy.
- This tree is inverted because the root of the tree is said to be at the top of the hierarchy, and the branches of directories and subdirectories stretch below the root.



Significant file system directories in Red Hat

Managing Files from the Command Line

- The **directory /** is the **root directory** at the top of the file system hierarchy.
- The **/** character is also used as a directory separator in file names. For example, if etc is a subdirectory of the **/** directory, we could call that directory **/etc**.
- Likewise, if the **/etc** directory contained a file named issue, we could refer to that file as **/etc/issue**.
- **Subdirectories** of **/** are used for standardized purposes to organize files by type and purpose. This makes it easier to find files. For example, in the root directory, the subdirectory **/boot** is used for storing files needed to boot the system.

Linux File System Hierarchy: **Benefits**

Key Benefits:

- Ensures consistency across different Linux distributions
- Provides predictability for software developers and system administrators
- Facilitates interoperability between different Unix-like systems
- Standardizes locations of important files and directories

File System Hierarchy

DIRECTORY PURPOSE	LOCATION
This directory contains static, persistent system configuration data.	/etc
This is the system's root directory.	/
User home directories are located under this directory.	/home
This is the root account's home directory.	/root
This directory contains dynamic configuration data, such as FTP and websites.	/var
Regular user commands and utilities are located here.	/usr/bin
System administration binaries, for root use, are here.	/usr/sbin
Temporary files are stored here.	/tmp
Contains dynamic, non-persistent application runtime data.	/run
Contains installed software programs and libraries.	/usr

Managing Files from the Command Line

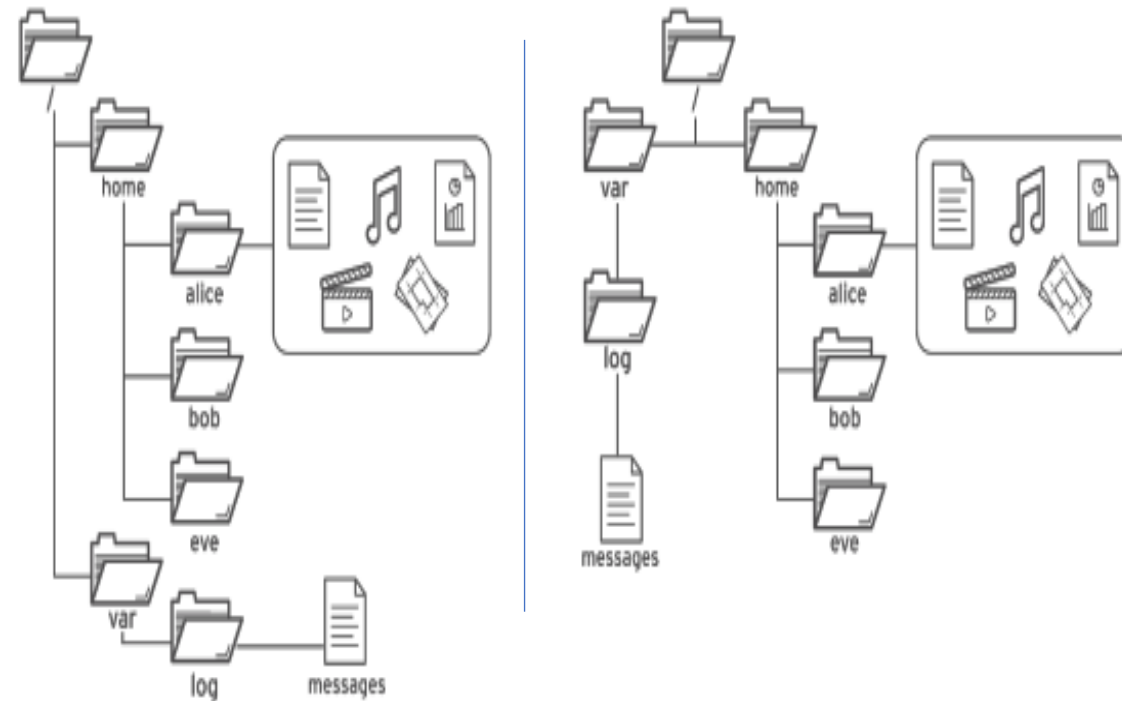
Important Red Hat Enterprise Linux directories

LOCATION	PURPOSE
/usr	Installed software, shared libraries, include files, and static read-only program data. Important subdirectories include: - /usr/bin: User commands. - /usr/sbin: System administration commands. - /usr/local: Locally customized software.
/etc	Configuration files specific to this system.
/var	Variable data specific to this system that should persist between boots. Files that dynamically change (e.g. databases, cache directories, log files, printer-spoiled documents, and website content) may be found under /var .
/run	Runtime data for processes started since the last boot. This includes process ID files and lock files, among other things. The contents of this directory are recreated on reboot. (This directory consolidates /var/run and /var/lock from older versions of Red Hat Enterprise Linux.)
/home	<i>Home directories</i> where regular users store their personal data and configuration files.
/root	Home directory for the administrative superuser, root.
/tmp	A world-writable space for temporary files. Files which have not been accessed, changed, or modified for 10 days are deleted from this directory automatically. Another temporary directory exists, /var/tmp , in which files that have not been accessed, changed, or modified in more than 30 days are deleted automatically.
/boot	Files needed in order to start the boot process.
/dev	Contains special <i>device files</i> which are used by the system to access hardware.

LOCATING FILES BY NAME

❖ ABSOLUTE PATHS AND RELATIVE PATHS

- The path of a file or directory specifies its unique file system location.
- Following a file path traverses one or more name subdirectories, delimited by a forward slash (/), until the destination is reached.
- Standard file behavior definitions apply to directories (also called folders) the same as other file types.



The common file browser view (left) is equivalent to the top-down view (right)

Managing Files from the Command Line: **Absolute Path**

- **Absolute paths** : An absolute path is a fully qualified name, beginning at the root (/) directory and specifying each subdirectory traversed to reach and uniquely represent a single file.
- Every file in a file system has a **unique absolute path** name, recognized with a simple rule: A path name with a forward slash (/) as the first character is an absolute path name.
- For example, the absolute path name for the system message log file is /var/log/messages. Absolute path names can be long to type, so files may also be **located relatively (relative paths)**.
- When a user logs in and opens a command window, the initial location is normally the user's home directory. System processes also have an initial directory.
- Users and processes navigate to other directories as needed; the terms working directory or current working directory refer to their **current location**.

Absolute Path vs. Relative Path

An **absolute path** specifies the complete location of a file or directory from the root directory (/). It always starts with a forward slash.

A **relative path** specifies the location of a file or directory relative to [your current working](#) directory. It does NOT start with a forward slash.

Managing Files from the Command Line: NAVIGATING PATHS

NAVIGATING PATHS The **pwd** command displays the full path name of the current location, which helps determine appropriate syntax for reaching files using relative path names.

- The **ls** command lists directory contents for the specified directory or, if no directory is given, for the current directory.

```
[student@desktopX ~]$ pwd
/home/student
[student@desktopX ~]$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
[student@desktopX ~]$
```

Managing Files from the Command Line: NAVIGATING PATHS

The **cd** command stands for "Change Directory" and is used to navigate between directories in the Linux file system.

cd	# Takes you to your home directory
cd ~	# Same as above (~ represents home)
cd /	#Go to the root directory
cd ..	# Go up one directory level
cd -	# Go back to previous directory
cd /path/to/directory	# Go to specific directory

```
[student@desktop /]$ cd /home  
[student@desktop home]
```

Managing Files from the Command Line

The **touch** command is used to create empty files and update file timestamps in Linux.

If the file already exists, touch updates its access and modification times to the current time

```
[student@desktop ~]$ touch Documents/file1  
[student@desktop ~]$ ls Documents  
file1
```

- This is useful for creating empty files, which can be used for practice, since "touching" a file name that does not exist causes the file to be created. Using touch, practice file are created in the Documents subdirectory.

Managing Files from the Command Line

Create directories The **mkdir** command

- **creates one or more directories or subdirectories**, generating errors if the file name already exists or when attempting to create a directory in a parent directory that doesn't exist.
- The **-p** parent option creates missing parent directories for the requested destination. Be cautious when using **mkdir -p**, since accidental spelling mistakes create unintended directories without generating error messages.
- In the following example, a user attempts to use **mkdir** to create a subdirectory named **Watched** in the existing **Videos** directory but mistypes the directory name.

```
mkdir directory_name    # Create single directory
mkdir dir1 dir2 dir3    # Create multiple directories
mkdir -p dir3/dir4      # This will create all missing directories
                        i.e. dir3 and dir4
```

```
[student@desktopX ~]$mkdir Video/Watched
mkdir: cannot create directory `Video/Watched': No such file or directory
```

The **mkdir** failed because **Videos** was misspelled and the directory **Video** does not exist. If the user had used **mkdir** with the **-p** option, there would be no error and the user would end up with two directories, **Videos** and **Video**, and the **Watched** subdirectory would be created in the wrong place.

Managing Files from the Command Line

The provided Figure demonstrates a sequence of commands in the command line for organizing files. It begins with creating a subdirectory Watched inside Videos using **mkdir**, then changes the directory to Documents with **cd** to create new folders (ProjectX, ProjectY, and Thesis) using **mkdir** with the **-p** option. The process concludes by using the **ls -R** command to display the entire directory structure, confirming the successful creation of all the folders.

```
[student@desktopX ~]$mkdir Videos/Watched[student@desktopX ~]$cd
Documents[student@desktopX Documents]$mkdir ProjectX ProjectY[student@desktopX
Documents]$mkdir -p Thesis/Chapter1 Thesis/Chapter2 Thesis/
Chapter3[student@desktopX Documents]$cd[student@desktopX ~]$ls -R Videos Documents
Documents:
ProjectX ProjectY Thesis thesis_chapter1.odf thesis_chapter2.odf
```

Managing Files from the Command Line

Copy files

- The **cp** command copies one or more files to become new, independent files. Syntax allows copying an existing file to a new file in the current or another directory or copying multiple files into another directory.
- In any destination, new file names must be unique. If the new file name is not unique, the copy command will overwrite the existing file.

```
[student@desktopX ~]$cd Videos[student@desktopX Videos]$cp blockbuster1.ogg
blockbuster3.ogg[student@desktopX Videos]$ls -l
total 0
-rw-rw-r--. 1 student student  0 Feb  8 16:23 blockbuster1.ogg
-rw-rw-r--. 1 student student  0 Feb  8 16:24 blockbuster2.ogg
-rw-rw-r--. 1 student student  0 Feb  8 19:02 blockbuster3.ogg
drwxrwxr-x. 2 student student 4096 Feb  8 23:35 Watched
```

Managing Files from the Command Line

- To protect users from accidentally overwriting directories with contents, multiple file **cp** commands ignore directories specified as a source.
- Copying non-empty directories, with contents, requires the -r recursive option.

```
[student@desktopX Videos]$cd ../Documents[student@desktopX Documents]$cp
thesis_chapter1.odf thesis_chapter2.odf Thesis ProjectX
cp: omitting directory `Thesis'
[student@desktopX Documents]$ cp -r Thesis ProjectX
[student@desktopX Documents]$ cp thesis_chapter2.odf Thesis/Chapter2/
[student@desktopX Documents]$ ls -R
.:
ProjectX ProjectY Thesis thesis_chapter1.odf thesis_chapter2.odf

./ProjectX:
Thesis thesis_chapter1.odf thesis_chapter2.odf

./ProjectX/Thesis:

./ProjectY:

./Thesis:
Chapter1 Chapter2 Chapter3

./Thesis/Chapter1:

./Thesis/Chapter2:
thesis_chapter2.odf

./Thesis/Chapter3:
[student@desktopX Documents]$
```

Managing Files from the Command Line

Move files

- The **mv** command renames files in the same directory or relocates files to a new directory.
- File contents remain unchanged. Files moved to a different file system require creating a new file by copying the source file, then deleting the source file. Although normally transparent to the user, large files may take noticeably longer to move.

```
[student@desktopX Videos]$cd ../Documents[student@desktopX Documents]$ls -l
total 0
-rw-rw-r--. 1 student student    0 Feb  8 16:24 thesis_chapter1.odf
-rw-rw-r--. 1 student student    0 Feb  8 16:24 thesis_chapter2.odf
[student@desktopX Documents]$ mv thesis_chapter2.odf thesis_chapter2_reviewed.odf
[student@desktopX Documents]$ mv thesis_chapter1.odf Thesis/Chapter1
[student@desktopX Documents]$ ls -lR
.:
total 16
drwxrwxr-x. 2 student student 4096 Feb 11 11:58 ProjectX
```

Managing Files from the Command Line

Remove files

- Remove files and directories Default syntax for **rm** deletes files, but not directories. **Deleting a directory, and potentially many subdirectories and files below it, requires the -r recursive option.**

There is no command-line undelete feature, nor a trash bin from which to restore.

```
[student@desktopX Documents]$pwd
/home/student/Documents
[student@desktopX Documents]$ rm thesis_chapter2_reviewed.odf
[student@desktopX Documents]$ rm Thesis/Chapter1
rm: cannot remove `Thesis/Chapter1': Is a directory
[student@desktopX Documents]$ rm -r Thesis/Chapter1
[student@desktopX Documents]$ ls -l Thesis
total 8
drwxrwxr-x. 2 student student 4096 Feb 11 12:47 Chapter2
drwxrwxr-x. 2 student student 4096 Feb 11 12:48 Chapter3
[student@desktopX Documents]$ rm -ri Thesis
rm: descend into directory `Thesis'? y
rm: descend into directory `Thesis/Chapter2'? y
rm: remove regular empty file `Thesis/Chapter2/thesis_chapter2.odf'? y
```

Managing Files from the Command Line

Remove files

- After **rm** failed to delete directory, the **-r recursive** option succeeded. The last rm command parsed into each subdirectory first, individually deleting contained files before removing each now-empty directory. Using **-i** will interactively prompt for each deletion.
- This is essentially the opposite of **-f** which will force the deletion without prompting the user. The **rmdir** command deletes directories only if empty. Removed directories cannot be undeleted.

```
[student@desktopX Documents]$pwd
/home/student/Documents
[student@desktopX Documents]$ rmdir ProjectY
[student@desktopX Documents]$ rmdir ProjectX
rmdir: failed to remove `ProjectX': Directory not empty
[student@desktopX Documents]$ rm -r ProjectX
[student@desktopX Documents]$ ls -lR
.:
total 0
[student@desktopX Documents]$
```



Thank You