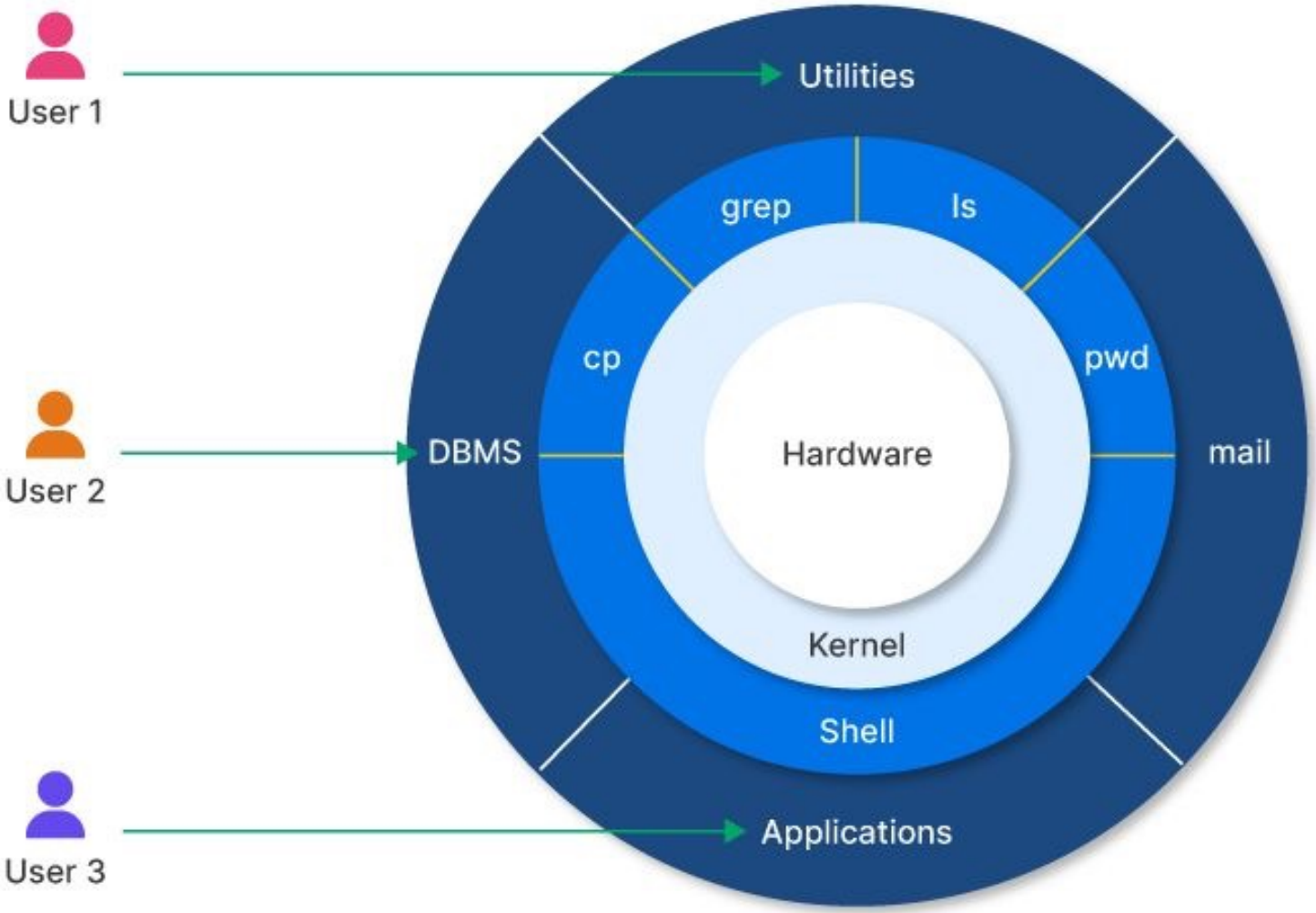
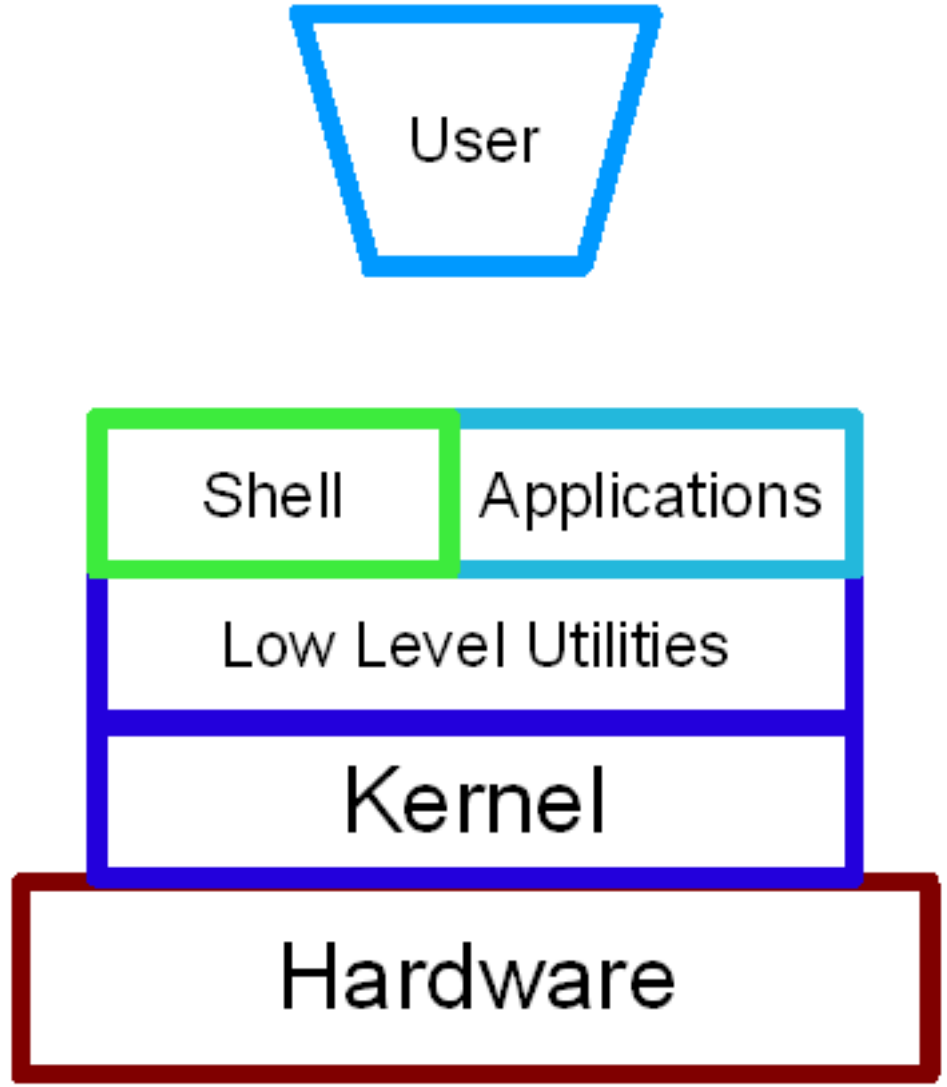
UNIT - 1

What is "the Shell"?

- Simply put, the shell is a program that takes commands from the keyboard and gives them to the operating system to perform.
- In the old days, it was the only user interface available on a Unix-like system such as Linux.
- Nowadays, we have *graphical user interfaces (GUIs)* in addition to *command line interfaces (CLIs)* such as the shell.
- On most Linux systems a program called bash (which stands for Bourne Again SHell, an enhanced version of the original Unix shell program, sh, written by Steve Bourne) acts as the shell program.
- Besides bash, there are other shell programs available for Linux systems. These include: ksh, tcsh and zsh.


User 1
User 2
User 3
Utilities
grep
ls
cp
pwd
DBMS
Hardware
mail
Kernel
Shell
Applications


User
Shell
Applications
Low Level Utilities
Kernel
Hardware

What is "the Shell"?

- To understand the Linux shell, it's important to first know what the kernel is.
- It is a core program of the Linux OS, responsible for managing many routine tasks and Linux resources like processes management, file handling, I/O operations, memory, devices, etc.
- The Linux shell is another integral part of the Linux OS, providing a command-line interface that allows users to directly interact with the operating system by accepting human-readable commands.
- Users type these human-readable commands in a terminal, where the Linux shell interprets these commands.

What is "the Shell"?

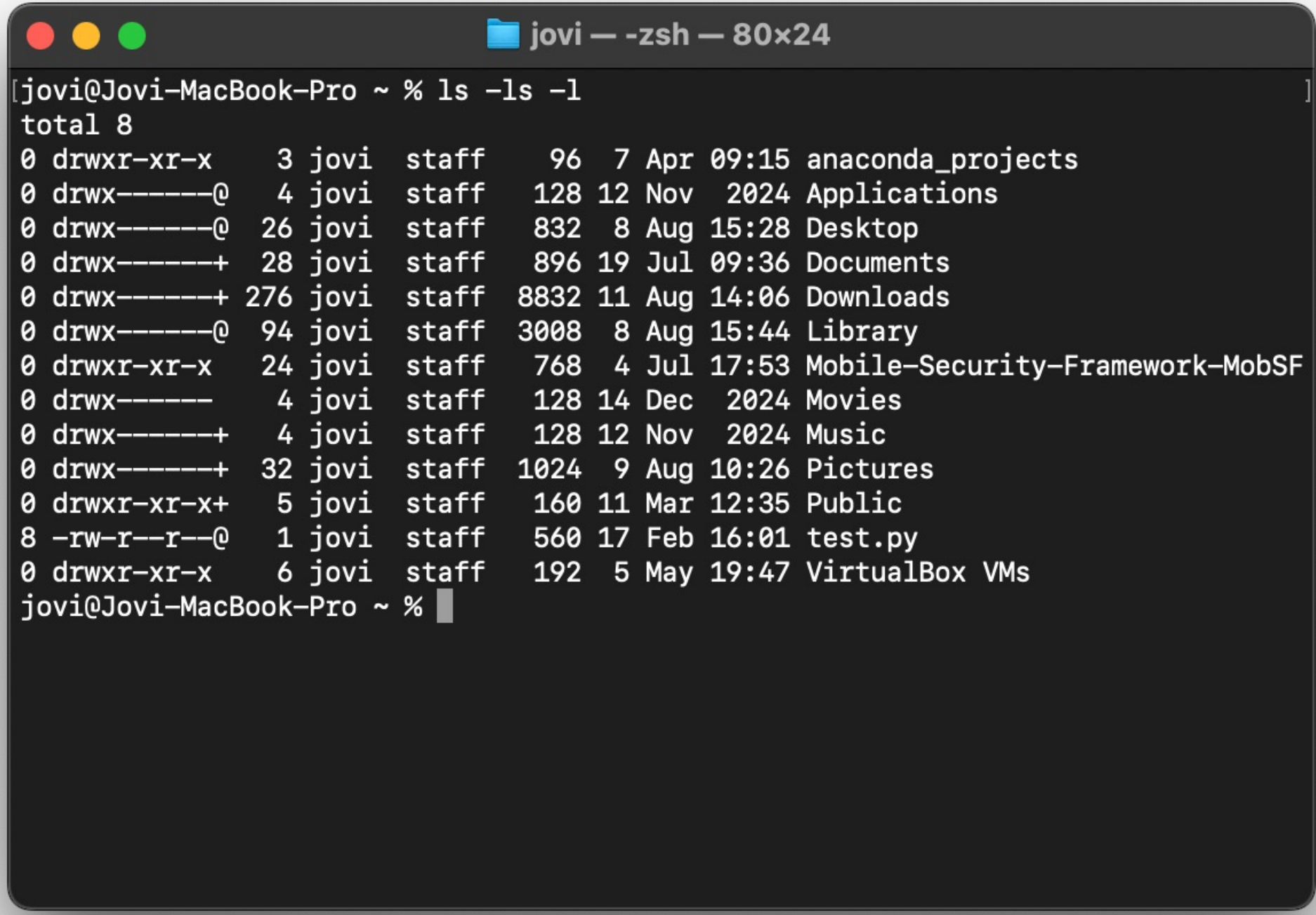
- The Linux terminal includes many built-in commands that users can utilize to perform various tasks.
- The Linux shell converts these commands for the kernel to understand and execute.
- All in all, the Linux shell serves as a bridge between the user and the kernel by facilitating the execution of human-understandable commands.
- It is the primary interface that provides users with a way to communicate with the Linux operating system at a more fundamental level than what is offered by a GUI.

What is Kernel?

- The kernel is a computer program that is the core of a computer's operating system, with complete control over everything in the system. It manages the following resources of the Linux system:
  - File management
  - Process management
  - I/O management
  - Memory management
  - Device management etc.
- Complete Linux system = Kernel + GNU system utilities and libraries + other management scripts + installation scripts.

Terminal Emulators

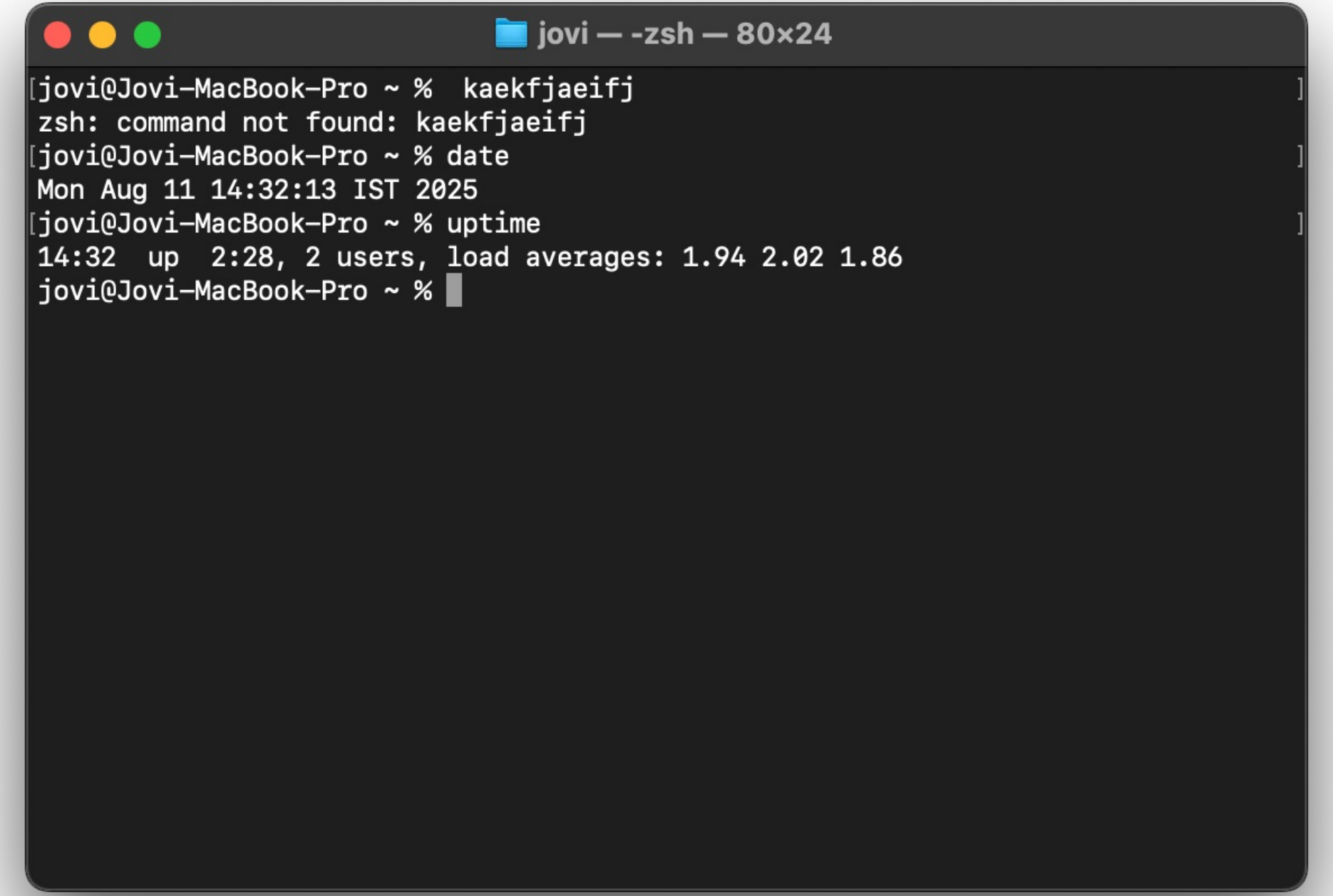
- When using a graphical user interface (GUI), we need another program called a terminal emulator to interact with the shell.
- If we look through our desktop menus, we will probably find one.
- KDE uses konsole and GNOME uses gnome-terminal, though it's likely called simply “terminal” on our menu.
- A number of other terminal emulators are available for Linux, but they all basically do the same thing; give us access to the shell.
- You will probably develop a preference for one or another terminal emulator based on the number of bells and whistles it has.


jovi — -zsh — 80×24
jovi@Jovi-MacBook-Pro ~ % ls -ls -l
total 8
0 drwxr-xr-x   3 jovi staff   96 7 Apr 09:15 anaconda_projects
0 drwx------@   4 jovi staff   128 12 Nov 2024 Applications
0 drwx------@ 26 jovi staff   832 8 Aug 15:28 Desktop
0 drwx------+ 28 jovi staff   896 19 Jul 09:36 Documents
0 drwx------+ 276 jovi staff 8832 11 Aug 14:06 Downloads
0 drwx------@ 94 jovi staff 3008 8 Aug 15:44 Library
0 drwxr-xr-x   24 jovi staff   768 4 Jul 17:53 Mobile-Security-Framework-MobSF
0 drwx------   4 jovi staff   128 14 Dec 2024 Movies
0 drwx------+   4 jovi staff   128 12 Nov 2024 Music
0 drwx------+ 32 jovi staff 1024 9 Aug 10:26 Pictures
0 drwxr-xr-x+   5 jovi staff   160 11 Mar 12:35 Public
8 -rw-r--r--@   1 jovi staff   560 17 Feb 16:01 test.py
0 drwxr-xr-x   6 jovi staff   192 5 May 19:47 VirtualBox VMs
jovi@Jovi-MacBook-Pro ~ %

Making Your First Keystrokes

```
[me@linuxbox ~]$
```

- This is called a shell prompt and it will appear whenever the shell is ready to accept input.
- While it may vary in appearance somewhat depending on the distribution, it will typically include your username@machinename, followed by the current working directory (more about that in a little bit) and a dollar sign.
- If the last character of your shell prompt is # rather than $, you are operating as the *superuser*.
- This means that you have administrative privileges.
- This can be dangerous, since you are able to delete or overwrite any file on the system.


jovi — -zsh — 80×24
jovi@Jovi-MacBook-Pro ~ % kaekfjaeifj
zsh: command not found: kaekfjaeifj
jovi@Jovi-MacBook-Pro ~ % date
Mon Aug 11 14:32:13 IST 2025
jovi@Jovi-MacBook-Pro ~ % uptime
14:32 up 2:28, 2 users, load averages: 1.94 2.02 1.86
jovi@Jovi-MacBook-Pro ~ %

Making Your First Keystrokes

- If we press the up-arrow key, we will see that the previous command kaekfjaeifj reappears after the prompt.
- This is called command history.
- Most Linux distributions remember the last 1000 commands by default.
- Press the down-arrow key and the previous command disappears.
- We can end a terminal session by either closing the terminal emulator window, by entering the exit command at the shell prompt, or pressing Ctrl-d.

Navigation

- The first thing we need to learn (besides how to type) is how to navigate the file system on our Linux system.
- Here introduce the following commands:
  - **pwd** – Print name of current working directory
  - **cd** – Change directory
  - **ls** – List directory contents

Understanding the File System Tree

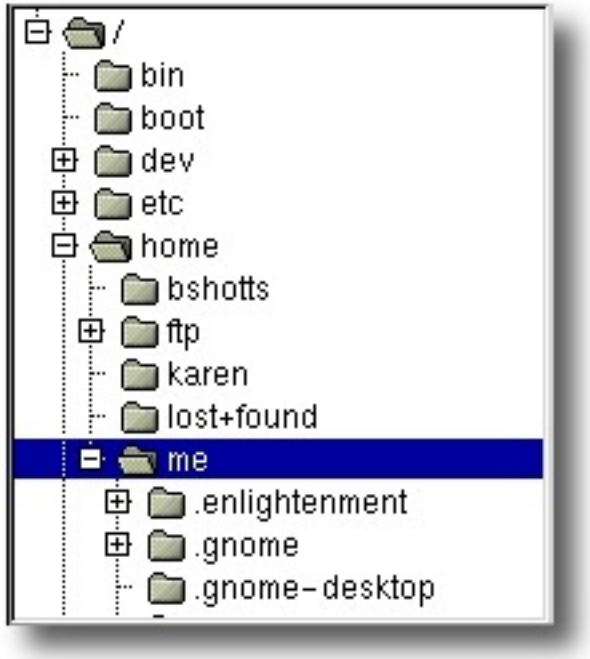
- Like Windows, a Unix-like operating system such as Linux organizes its files in what is called a hierarchical directory structure.
- This means they are organized in a tree-like pattern of directories (sometimes called folders in other systems), which may contain files and other directories.
- The first directory in the file system is called the root directory.
- The root directory contains files and subdirectories, which contain more files and subdirectories and so on.

Understanding the File System Tree

- Note that unlike Windows, which has a separate file system tree for each storage device, Unix-like systems such as Linux always have a single file system tree, regardless of how many drives or storage devices are attached to the computer.
- Storage devices are attached (or more correctly, mounted) at various points on the tree according to the whims of the system administrator, the person (or people) responsible for the maintenance of the system.

Understanding the File System Tree

- Most graphical environments include a file manager program used to view and manipulate the contents of the file system.
- Often we will see the file system represented like this:



Understanding the File System Tree

- However, the command line has no pictures, so to navigate the file system tree we need to think of it in a different way.
- Imagine that the file system is a maze shaped like an upside-down tree and we are able to stand in the middle of it.
- At any given time, we are inside a single directory and we can see the files contained in the directory and the pathway to the directory above us (called the parent directory) and any subdirectories below us.

The Current Working Directory

- The directory we are standing in is called the current working directory.
- To display the current working directory, we use the **pwd** (print working directory) command.

```
[me@linuxbox ~]$ pwd
/home/me
```

- When we first log in to our system (or start a terminal emulator session) our current working directory is set to our home directory.
- Each user account is given its own home directory and it is the only place a regular user is allowed to write files.

Listing the Contents of a Directory

- To list the files and directories in the current working directory, we use the ls command.

```
[me@linuxbox ~]$ ls
```

```
Desktop  Documents  Music  Pictures  Public  Templates  Videos
```

- Actually, we can use the ls command to list the contents of any directory, not just the current working directory, and there are many other things it can do as well. (Can you tell me?)

Changing the Current Working Directory

- To change our working directory (where we are standing in our tree-shaped maze) we use the **cd** command.
- To do this, type cd followed by the pathname of the desired working directory.
- A pathname is the route we take along the branches of the tree to get to the directory we want.
- We can specify pathnames in one of two different ways; as absolute pathnames or as relative pathnames.
- Let's look at absolute pathnames first.

Absolute Pathnames

- An absolute pathname begins with the root directory and follows the tree branch by branch until the path to the desired directory or file is completed.
- For example, there is a directory on our system in which most of our system's programs are installed.
- The directory’s pathname is /usr/bin.
- This means from the root directory (represented by the leading slash in the pathname) there is a directory called "usr" which contains a directory called "bin".

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
[me@linuxbox bin]$ ls
...Listing of many, many files ...
```

Now we can see that we have changed the current working directory to /usr/bin and
that it is full of files. Notice how the shell prompt has changed? As a convenience, it is
usually set up to automatically display the name of the working directory.

Relative Pathnames

- Where an absolute pathname starts from the root directory and leads to its destination, a relative pathname starts from the working directory.
- To do this, it uses a couple of special notations to represent relative positions in the file system tree.
- These special notations are "." (dot) and ".." (dot dot).
- The "." notation refers to the working directory and the ".." notation refers to the working directory's parent directory.

Relative Pathnames

- Here is how it works. Let's change the working directory to /usr/bin again.

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

- Now let's say that we wanted to change the working directory to the parent of /usr/bin which is /usr.
- We could do that two different ways, either using an absolute path name

```
[me@linuxbox bin]$ cd /usr
[me@linuxbox usr]$ pwd
/usr
```

Relative Pathnames

- or, using a relative pathname.

```
[me@linuxbox bin]$ cd ..
[me@linuxbox usr]$ pwd
/usr
```

- Two different methods with identical results. Which one should we use? The one that requires the least typing!
- Likewise, we can change the working directory from /usr to /usr/bin in two different ways, either using an absolute pathname:

```
[me@linuxbox usr]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Relative Pathnames

- or, using a relative pathname.

```
[me@linuxbox usr]$ cd ./bin
[me@linuxbox bin]$ pwd
/usr/bin
```

- Now, there is something important to point out here. In almost all cases, we can omit the "./". It is implied. Typing:

```
[me@linuxbox usr]$ cd bin
```

- does the same thing. In general, if we do not specify a pathname to something, the working directory will be assumed.

Some Helpful Shortcuts

- Here we see some useful ways the current working directory can be quickly changed.

Shortcut	Result
`cd`	Changes the working directory to your home directory.
`cd -`	Changes the working directory to the previous working directory.
`cd ~`*`user_name`*	Changes the working directory to the home directory of *user_name*. For example, `cd ~bob` will change the directory to the home directory of user “bob.”

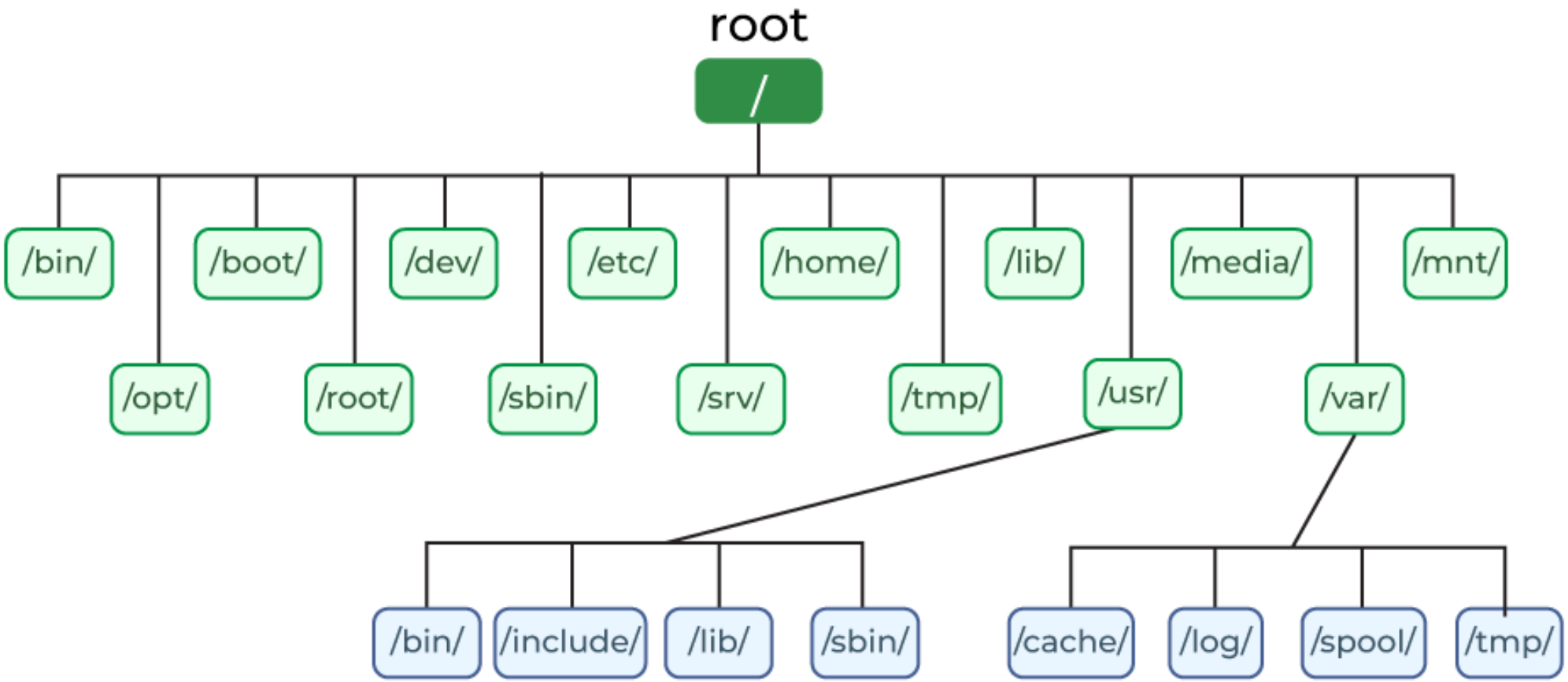
Important Facts About Filenames

On Linux systems, files are named in a manner similar to other systems such as Windows, but there are some important differences.

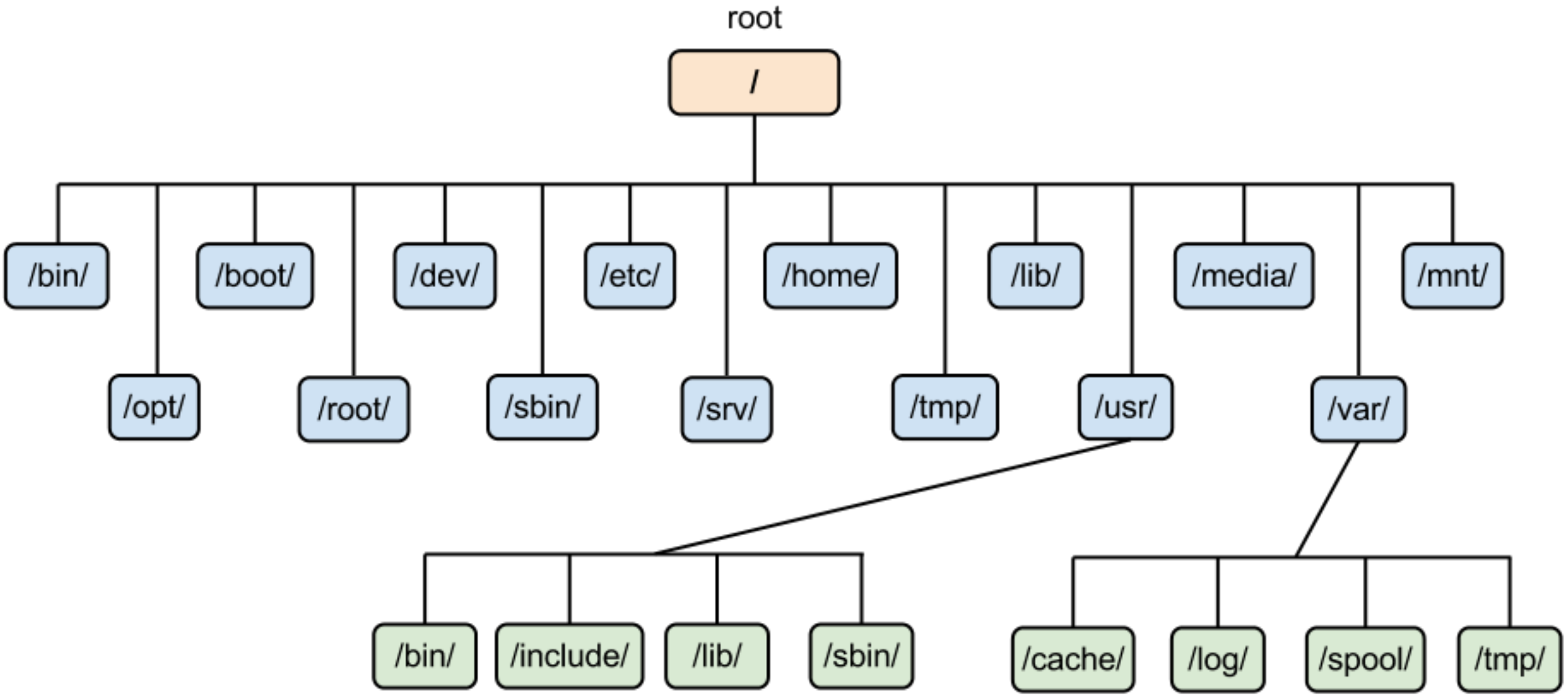
1. Filenames that begin with a period character are hidden. This only means that `ls` will not list them unless you say `ls -a`. When your account was created, several hidden files were placed in your home directory to configure things for your account. In Chapter 11 we will take a closer look at some of these files to see how you can customize your environment. In addition, some applications place their configuration and settings files in your home directory as hidden files.
2. Filenames and commands in Linux, like Unix, are case sensitive. The filenames “File1” and “file1” refer to different files.

3. Linux has no concept of a “file extension” like some other operating systems. You may name files any way you like. The contents and/or purpose of a file is determined by other means. Although Unix-like operating systems don’t use file extensions to determine the contents/purpose of files, many application programs do.
4. Though Linux supports long filenames that may contain embedded spaces and punctuation characters, limit the punctuation characters in the names of files you create to period, dash, and underscore. *Most importantly, do not embed spaces in filenames*. If you want to represent spaces between words in a file-name, use underscore characters. You will thank yourself later.

Unix File System Hierarchy



Linux File System Hierarchy



Understanding the Folders

- /bin Essential system binaries
- /etc Configuration files
- /home User directories
- /var Variable data (logs, caches)
- /usr User-installed applications
- /tmp Temporary files

Exploring the System

- Now that we know how to move around the file system, it's time for a guided tour of our Linux system.
- Before we start however, we’re going to learn some more commands that will be useful along the way.
  - ls – List directory contents
  - file – Determine file type
  - less – View file contents

Having More Fun with *ls*

- The ls command is probably the most used Linux command, and for good reason.
- With it, we can see directory contents and determine a variety of important file and directory attributes.
- As we have seen, we can simply enter ls to get a list of files and subdirectories contained in the current working directory.

```
[me@linuxbox ~]$ ls
Desktop  Documents  Music  Pictures  Public  Templates  Videos
```

Having More Fun with *ls*

- Besides the current working directory, we can specify a directory to list, like so:

```
me@linuxbox ~]$ ls /usr
bin  games  include  lib  local  sbin  share  src
```

- We can even specify multiple directories. In the following example, we list both the user's home directory (symbolized by the "~" character) and the /usr directory.

```
[me@linuxbox ~]$ ls ~ /usr
/home/me:
Desktop  Documents  Music  Pictures  Public  Templates  Videos

/usr:
bin  games  include  lib  local  sbin  share  src
```

Having More Fun with *ls*

- We can also change the format of the output to reveal more detail.

```
[me@linuxbox ~]$ ls -l
total 56
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Desktop
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Documents
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Music
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Pictures
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Public
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Templates
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Videos
```

- By adding “-l” to the command, we changed the output to the long format.

Options and Arguments

- This brings us to a very important point about how most commands work.
- Commands are often followed by one or more options that modify their behaviour, and further, by one or more arguments, the items upon which the command acts.
- So most commands look kind of like this:

```
command -options arguments
```

Options and Arguments

- Most commands use options which consist of a single character preceded by a dash, for example, “-l”.
- Many commands, however, including those from the GNU Project, also support long options, consisting of a word preceded by two dashes.
- Also, many commands allow multiple short options to be strung together.
- In the following example, the ls command is given two options, which are the l option to produce long format output, and the t option to sort the result by the file's modification time.

```
[me@linuxbox ~]$ ls -lt
```

- We'll add the long option “--reverse” to reverse the order of the sort.

```
[me@linuxbox ~]$ ls -lt --reverse
```

Note that command options, like filenames in Linux, are case-sensitive.

Options and Arguments

- The ls command has a large number of possible options. The most common are listed

Option	Long Option	Description
`-a`	`--all`	List all files, even those with names that begin with a period, which are normally not listed (that is, hidden).
`-A`	`--almost-all`	Like the `-a` option above except it does not list `.` (current directory) and `..` (parent directory).
`-d`	`--directory`	Ordinarily, if a directory is specified, `ls` will list the contents of the directory, not the directory itself. Use this option in conjunction with the `-l` option to see details about the directory rather than its contents.
`-F`	`--classify`	This option will append an indicator character to the end of each listed name. For example, a forward slash (/) if the name is a directory.
`-h`	`--human-readable`	In long format listings, display file sizes in human readable format rather than in bytes.
`-l`		Display results in long format.
`-r`	`--reverse`	Display the results in reverse order. Normally, `ls` displays its results in ascending alphabetical order.
`-S`		Sort results by file size.
`-t`		Sort by modification time.

A Longer Look at Long Format

- As we saw earlier, the -l option causes ls to display its results in long format. This format contains a great deal of useful information.
- Here is the Examples directory from an early Ubuntu system:

```
-rw-r--r-- 1 root root 3576296 2017-04-03 11:05 Experience ubuntu.ogg
-rw-r--r-- 1 root root 1186219 2017-04-03 11:05 kubuntu-leaflet.png
-rw-r--r-- 1 root root   47584 2017-04-03 11:05 logo-Edubuntu.png
-rw-r--r-- 1 root root   44355 2017-04-03 11:05 logo-Kubuntu.png
-rw-r--r-- 1 root root   34391 2017-04-03 11:05 logo-Ubuntu.png
-rw-r--r-- 1 root root   32059 2017-04-03 11:05 oo-cd-cover.odf
-rw-r--r-- 1 root root  159744 2017-04-03 11:05 oo-derivatives.doc
-rw-r--r-- 1 root root   27837 2017-04-03 11:05 oo-maxwell.odt
-rw-r--r-- 1 root root   98816 2017-04-03 11:05 oo-trig.xls
-rw-r--r-- 1 root root  453764 2017-04-03 11:05 oo-welcome.odt
-rw-r--r-- 1 root root  358374 2017-04-03 11:05 ubuntu Sax.ogg
```

A Longer Look at Long Format

- Table here provides us with a look at the different fields from one of the files and their meanings.

Field	Meaning
`-rw-r--r--`	Access rights to the file. The first character indicates the type of file. Among the different types, a leading dash means a regular file, while a “d” indicates a directory. The next three characters are the access rights for the file's owner, the next three are for members of the file's group, and the final three are for everyone else. Chapter 9 "Permissions" discusses the full meaning of this in more detail.
`1`	File's number of hard links.
`root`	The username of the file's owner.
`root`	The name of the group that owns the file.
`32059`	Size of the file in bytes.
`2017-04-03 11:05`	Date and time of the file's last modification.
`oo-cd-cover.odf`	Name of the file.

A file’s number of hard links in Linux means how many different names or paths point to the *same actual data* on disk

Determining a File's Type with file

- As we explore the system it will be useful to know what kind of data files contain.
- To do this we will use the file command to determine a file's type.
- As we discussed earlier, filenames in Linux are not required to reflect a file's contents.
- While a file named “picture.jpg” would normally be expected to contain a JPEG compressed image,
- it is not required to in Linux. We invoke the file command this way:

```
file filename
```

Determining a File's Type with file

- When invoked, the file command will print a brief description of the file's contents.
- For example

```
[me@linuxbox ~]$ file picture.jpg
picture.jpg: JPEG image data, JFIF standard 1.01
```

- There are many kinds of files. In fact, one of the basic ideas in Unix-like operating systems such as Linux is that “everything is a file.”
- While many of the files on our system are familiar, for example MP3 and JPEG, there aremany kinds that are a little less obvious and a few that are quite strange.

Viewing File Contents with less

- The less command is a program to view text files.
- Throughout the Linux system, there are many files that contain human-readable text.
- The less program provides a convenient way to examine them.
- Why would we want to examine text files? Because many of the files that contain system settings (called configuration files) are stored in this format, and being able to read them gives us insight about how the system works.
- In addition, some of the actual programs that the system uses (called scripts) are stored in this format.

Viewing File Contents with less

- The less command is used like this:

```
less filename
```

- Once started, the less program allows us to scroll forward and backward through a text file.
- For example, to examine the file that defines all the system's user accounts, enter the following command:

```
[me@linuxbox ~]$ less /etc/passwd
```

Once the less program starts, we can view the contents of the file. If the file is longer than one page, we can scroll up and down. To exit less, press the q key.

Practical : Hands On!

- Now we are going to wander around the file system ourselves to see what makes our
- This will give us a chance to practice our navigation skills.
- One of the things we will discover is that many of the interesting files are in plain human-readable text. As we go about our tour, try the following:
    1. cd into a given directory
    2. List the directory contents with ls -l
    3. If you see an interesting file, determine its contents with file
    4. If it looks like it might be text, try viewing it with less
    5. If we accidentally attempt to view a non-text file and it scrambles the terminal window, we can recover by entering the reset command.

Information of Specific Directories Found on Linux Systems

Directory	Comments
`/`	The root directory. Where everything begins.
`/bin`	Contains binaries (programs) that must be present for the system to boot and run. Note that modern Linux distributions have deprecated `/bin` in favor of `/usr/bin` (see below).

Information of Specific Directories Found on Linux Systems

Directory	Comments
`/boot`	Contains the Linux kernel, initial RAM disk image (for drivers needed at boot time), and the boot loader. Interesting files: • `/boot/grub/grub.cfg` or `menu.lst`, which is used to configure the boot loader. • `/boot/vmlinuz` (or something similar), the Linux kernel
`/dev`	This is a special directory that contains *device nodes*. "Everything is a file" also applies to devices. Here is where the kernel maintains a list of all the devices it understands.
`/etc`	The `/etc` directory contains all of the system-wide configuration files. It also contains a collection of shell scripts that start each of the system services at boot time. Everything in this directory should be readable text. Interesting files: While everything in `/etc` is interesting, here are some all-time favorites: • `/etc/crontab`, on systems that use the `cron` program, this file defines when automated jobs will run. • `/etc/fstab`, a table of storage devices and their associated mount points. • `/etc/passwd`, a list of the user accounts.
`/home`	In normal configurations, each user is given a directory in `/home`. Ordinary users can only write files in their home directories. This limitation protects the system from errant user activity.
`/lib`	Contains shared library files used by the core system programs. These are similar to dynamic link libraries (DLLs) in Windows. This directory has been deprecated in modern distributions in favor of `/usr/lib`.
`/lost+found`	Each formatted partition or device using a Linux file system, such as ext4, will have this directory. It is used in the case of a partial recovery from a file system corruption event. Unless something really bad has happened to our system, this directory will remain empty.

Information of Specific Directories Found on Linux Systems

Directory	Comments
`/media`	On modern Linux systems the `/media` directory will contain the mount points for removable media such as USB drives, CD-ROMs, etc. that are mounted automatically at insertion.
`/mnt`	On older Linux systems, the `/mnt` directory contains mount points for devices that have been mounted manually.
`/opt`	The `/opt` directory is used to install “optional” software. This is mainly used to hold commercial software products that might be installed on the system.
`/proc`	The `/proc` directory is special. It's not a real file system in the sense of files stored on the hard drive. Rather, it is a virtual file system maintained by the Linux kernel. The “files” it contains are peepholes into the kernel itself. The files are readable and will give us a picture of how the kernel sees the computer. Browsing this directory can reveal many details about the computer’s hardware.

Information of Specific Directories Found on Linux Systems

`/root`	This is the home directory for the root account.
`/run`	This is a modern replacement for the traditional `/tmp` directory (see below). Unlike `/tmp`, the `/run` directory is mounted using the *tempfs* file system type which stores its contents in memory rather than on a physical disk.
`/sbin`	This directory contains “system” binaries. These are programs that perform vital system tasks that are generally reserved for the superuser. Note that modern Linux distributions have deprecated `/sbin` in favor of `/usr/sbin` (see below).
`/sys`	The `/sys` directory contains information about devices that have been detected by the kernel. This is much like the contents of the `/dev` directory but is more detailed including such things actual hardware addresses.
`/tmp`	The `/tmp` directory is intended for the storage of temporary, transient files created by various programs. Some distributions empty this directory each time the system is rebooted.

Information of Specific Directories Found on Linux Systems

Directory	Comments
`/usr`	The `/usr` directory tree is likely the largest one on a Linux system. It contains all the programs and support files used by regular users.
`/usr/bin`	`/usr/bin` contains the executable programs installed by the Linux distribution. It is not uncommon for this directory to hold thousands of programs.
`/usr/lib`	The shared libraries for the programs in `/usr/bin`.
`/usr/local`	The `/usr/local` tree is where programs that are not included with the distribution but are intended for system-wide use are installed. Programs compiled from source code are normally installed in `/usr/local/bin`. On a newly installed Linux system, this tree exists, but it will be empty until the system administrator puts something in it.
`/usr/sbin`	Contains more system administration programs.
`/usr/share`	`/usr/share` contains all the shared data used by programs in `/usr/bin`. This includes things such as default configuration files, icons, screen backgrounds, sound files, etc.
`/usr/share/doc`	Most packages installed on the system will include some kind of documentation. In `/usr/share/doc`, we will find documentation files organized by package.

Information of Specific Directories Found on Linux Systems

`/var`	With the exception of `/tmp` and `/home`, the directories we have looked at so far remain relatively static, that is, their contents don't change. The `/var` directory tree is where data that is likely to change is stored. Various databases, spool files, user mail, etc. are located here.
`/var/log`	`/var/log` contains *log files*, records of various system activity. These are important and should be monitored from time to time. The most useful ones are `/var/log/messages` and/or `/var/log/syslog` though these are not available on all systems. Note that for security reasons, some systems only allow the superuser to view log files.

Directory	Comments
`~/.config` and `~/.local`	These two directories are located in the home directory of each desktop user. They are used to store user-specific configuration data for desktop applications.

Symbolic Links

- As we look around, we are likely to see a directory listing (for example in /usr/lib) with an entry like this:

```
lrwxrwxrwx 1 root root   11 2007-08-11 07:34 libc.so.6 -> libc-2.6.so
```

- Notice how the first letter of the listing is “l” and the entry seems to have two filenames?
- This is a special kind of a file called a symbolic link (also known as a soft link or symlink).
- In most Unix-like systems it is possible to have a file referenced by multiple names.
- While the value of this might not be obvious, it is really a useful feature.

Symbolic Links

- Picture this scenario: A program requires the use of a shared resource of some kind contained in a file named “foo,” but “foo” has frequent version changes.
- It would be good to include the version number in the filename so the administrator or other interested party could see what version of “foo” is installed.
- This presents a problem. If we change the name of the shared resource, we have to track down every program that might use it and change it to look for a new resource name every time a new version of the resource is installed.
- That doesn't sound like fun at all.

Hard Links

- While we are on the subject of links, we need to mention that there is a second type of link called a hard link.
- Hard links also allow files to have multiple names, but they do it in a different way.
- Hard links enable a file to have multiple names (in the same filesystem/partition), accessible from different directories.
- Regardless of which name is used, reading or editing the file changes the underlying data, making all names for a file always up to date.

Manipulating Files and Directories

- Now we will introduce the following commands:
  - cp – Copy files and directories
  - mv – Move/rename files and directories
  - mkdir – Create directories
  - rm – Remove files and directories
  - ln – Create hard and symbolic links
- These five commands are among the most frequently used Linux commands.
- They are used for manipulating both files and directories.

Manipulating Files and Directories

- Simple file tasks (copy, move, delete) are easier with a graphical file manager using drag-and-drop and visual menus.
- Command-line tools offer more power and flexibility for complex operations compared to graphical interfaces.
- Complicated tasks—such as copying only certain types of files (e.g., only HTML) or conditionally based on modification dates—are easy to automate with the command line but difficult using a graphical file manager.
- The command line excels where precision, automation, and advanced options are required, making it a preferred choice for technical and repetitive tasks.
- While a GUI is intuitive for beginners and suitable for basic actions, the command line is best for users seeking efficiency in complex scenarios.

Manipulating Files and Directories

- For example, how could we copy all the HTML files from one directory to another but only copy files that do not exist in the destination directory or are newer than the versions in the destination directory?
- It's pretty hard with a file manager but pretty easy with the command line.

```
cp -u *.html destination
```

Wildcards

- Before we begin using our commands, we need to talk about a shell feature that makes these commands so powerful.
- Since the shell uses filenames so much, it provides special characters to help us rapidly specify groups of filenames.
- These special characters are called wildcards.
- Using wildcards (which is also known as globbing) allows us to select filenames based on patterns of characters.

Wildcards

Wildcard	Meaning
*	Matches any characters
?	Matches any single character
[*characters*]	Matches any character that is a member of the set *characters*
[!*characters*] or [^*characters*]	Matches any character that is not a member of the set *characters*
[[:*class*:]]	Matches any character that is a member of the specified *class*

Wildcards

- Commonly Used Character Classes

Character Class	Meaning
`[:alnum:]`	Matches any alphanumeric character
`[:alpha:]`	Matches any alphabetic character
`[:digit:]`	Matches any numeral
`[:lower:]`	Matches any lowercase letter
`[:upper:]`	Matches any uppercase letter

Wildcards

- Using wildcards makes it possible to construct sophisticated selection criteria for filenames.
- Here are some examples of patterns and what they match.
- Wildcards can be used with any command that accepts filenames as arguments

Pattern	Matches
`*`	All files
`g*`	Any file beginning with “g”
`b*.txt`	Any file beginning with “b” followed by any characters and ending with “.txt”
`Data???`	Any file beginning with “Data” followed by exactly three characters
`[abc]*`	Any file beginning with either an “a”, a “b”, or a “c”
`BACKUP.[0-9][0-9][0-9]`	Any file beginning with “BACKUP.” followed by exactly three numerals
`[[:upper:]]*`	Any file beginning with an uppercase letter
`[![:digit:]]*`	Any file not beginning with a numeral
`*[[:lower:]123]`	Any file ending with a lowercase letter or the numerals “1”, “2”, or “3”

Copying Files

- The cp program copies files and directories. In its simplest form, it copies a single file:

Command	**Results**
`cp` *file1 file2*	Copies the contents of *file1* into *file2*. If *file2* does not exist, it is created; **otherwise, *file2* is silently overwritten with the contents of *file1*.**
`cp –i` *file1 file2*	Like above however, since the "-i" (interactive) option is specified, if *file2* exists, the user is prompted before it is overwritten with the contents of *file1*.
`cp` *file1 dir1*	Copy the contents of *file1* (into a file named *file1*) inside of directory *dir1*.
`cp –R` *dir1 dir2*	Copy the contents of the directory *dir1*. If directory *dir2* does not exist, it is created. Otherwise, it creates a directory named *dir1* within directory *dir2*.

Examples of the cp command

Moving Files and Directories

- The mv command moves or renames files and directories depending on how it is used.
- It will either move one or more files to a different directory, or it will rename a file or directory.

Command	**Results**
`mv` *file1 file2*	If *file2* does not exist, then *file1* is renamed *file2*. **If *file2* exists, its contents are silently replaced with the contents of *file1*.**
`mv –i` *file1 file2*	Like above however, since the "-i" (interactive) option is specified, if *file2* exists, the user is prompted before it is overwritten with the contents of *file1*.
`mv` *file1 file2 dir1*	The files *file1* and *file2* are moved to directory *dir1*. If *dir1* does not exist, **`mv`** will exit with an error.
`mv` *dir1 dir2*	If *dir2* does not exist, then *dir1* is renamed *dir2*. If *dir2* exists, the directory *dir1* is moved within directory *dir2*.

Examples of the mv command

Removing Files and Directories

- The rm command removes (deletes) files and directories.
- Using the recursive option (-r), rm can also be used to delete directories:

Command	**Results**
`rm` *`file1 file2`*	Delete *file1* and *file2*.
`rm –i` *`file1 file2`*	Like above however, since the "-i" (interactive) option is specified, the user is prompted before each file is deleted.
`rm –r` *`dir1 dir2`*	Directories *dir1* and *dir2* are deleted along with all of their contents.

Examples of the rm command

Removing Files and Directories

- Be careful with rm!
- Linux does not have an undelete command. Once you delete something with rm, it's gone. You can inflict terrific damage on your system with rm if you are not careful, particularly with wildcards.
- ***Before you use rm with wildcards, try this helpful trick:*** construct your command using ls instead.
- By doing this, you can see the effect of your wildcards before you delete files.
- After you have tested your command with ls, recall the command with the up-arrow key and then substitute rm for ls in the command.

Creating Directories

- The mkdir command is used to create directories. To use it, you simply type:

```
[me@linuxbox ~]$ cd
[me@linuxbox ~]$ mkdir playground
```

- Using Commands with Wildcards : Since the commands we have covered here accept multiple file and directories names as arguments, you can use wildcards to specify them. Here are a few examples:

Creating Directories

Command	**Results**
`cp *.txt text_files`	Copy all files in the current working directory with names ending with the characters ".txt" to an existing directory named *text_files*.
`mv dir1 ../*.bak dir2`	Move the subdirectory *dir1* and all the files ending in ".bak" in the current working directory's parent directory to an existing directory named *dir2*.
`rm *~`	Delete all files in the current working directory that end with the character "~". Some applications create backup files using this naming scheme. Using this command will clean them out of a directory.

Command examples using wildcards

Hard Links and Symbolic Links

- First lets create a playground

```
[me@linuxbox ~]$ cd
[me@linuxbox ~]$ mkdir playground
```

```
[me@linuxbox ~]$ cd playground
[me@linuxbox playground]$ mkdir dir1 dir2
```

- Notice that the mkdir command will accept multiple arguments allowing us to create both directories with a single command.

Hard Links and Symbolic Links

- Using the cp command, we'll copy the passwd file from the /etc directory to the current working directory.

```
[me@linuxbox playground]$ cp /etc/passwd .
```

Shorthand for the current working directory, the single trailing period.

- So now if we perform an ls, we will see our file.

```
[me@linuxbox playground]$ ls -l
```

Hard Links and Symbolic Links

- Now, the name passwd doesn't seem very playful and this is a playground, so let’s change it to something else.

```
[me@linuxbox playground]$ mv passwd fun
```

Let's pass the fun around a little by moving our renamed file to each of the directories and back again. The following moves it first to the directory `dir1`:

```
[me@linuxbox playground]$ mv fun dir1
```

The following then moves it from `dir1` to `dir2`:

```
[me@linuxbox playground]$ mv dir1/fun dir2
```

Hard Links and Symbolic Links

Finally, the following brings it back to the current working directory:

```
[me@linuxbox playground]$ mv dir2/fun .
```

Next, let's see the effect of `mv` on directories. First we will move our data file into `dir1` again, like this:

```
[me@linuxbox playground]$ mv fun dir1
```

Then we move `dir1` into `dir2` and confirm it with `ls .`

Hard Links and Symbolic Links

```
[me@linuxbox playground]$ mv dir1 dir2
[me@linuxbox playground]$ ls -l dir2
total 4
drwxrwxr-x 2 me  me   4096 2025-01-11 06:06 dir1
[me@linuxbox playground]$ ls -l dir2/dir1
total 4
-rw-r--r-- 1 me  me   1650 2025-01-10 16:33 fun
```

Note that since `dir2` already existed, `mv` moved `dir1` into `dir2`. If `dir2` had not existed, `mv` would have renamed `dir1` to `dir2`. Lastly, let's put everything back.

```
[me@linuxbox playground]$ mv dir2/dir1 .
[me@linuxbox playground]$ mv dir1/fun .
```

Hard Links and Symbolic Links

Now we'll try some links. We'll first create some hard links to our data file like so:

```
[me@linuxbox playground]$ ln fun fun-hard
[me@linuxbox playground]$ ln fun dir1/fun-hard
[me@linuxbox playground]$ ln fun dir2/fun-hard
```

So now we have four instances of the file `fun`. Let's take a look at our playground directory.

Hard Links and Symbolic Links

```
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me   me   4096 2025-01-14 16:17 dir1
drwxrwxr-x 2 me   me   4096 2025-01-14 16:17 dir2
-rw-r--r-- 4 me   me   1650 2025-01-10 16:33 fun
-rw-r--r-- 4 me   me   1650 2025-01-10 16:33 fun-hard
```

One thing we notice is that both the second fields in the listings for `fun` and `fun-hard` contain a `4` which is the number of hard links that now exist for the file. Remember that a file will always have at least one link because the file's name is created by a link. So, how do we know that `fun` and `fun-hard` are, in fact, the same file? In this case, `ls` is not very helpful. While we can see that `fun` and `fun-hard` are both the same size (field 5), our listing provides no way to be sure. To solve this problem, we're going to have to dig a little deeper.

Hard Links and Symbolic Links

- When thinking about hard links, it is helpful to imagine that files are made up of two parts.
  1. The data part containing the file's contents.
  2. The name part that holds the file's name.
- When we create hard links, we are actually creating additional name parts that all refer to the same data part.
- The system assigns a chain of disk blocks to what is called an inode, which is then associated with the name part.
- Each hard link therefore refers to a specific inode containing the file's contents.
- The ls command has a way to reveal this information. It is invoked with the -i option.

Hard Links and Symbolic Links

```
[me@linuxbox playground]$ ls -li
total 16
12353539 drwxrwxr-x 2 me   me   4096 2025-01-14 16:17 dir1
12353540 drwxrwxr-x 2 me   me   4096 2025-01-14 16:17 dir2
12353538 -rw-r--r-- 4 me   me   1650 2025-01-10 16:33 fun
12353538 -rw-r--r-- 4 me   me   1650 2025-01-10 16:33 fun-hard
```

In this version of the listing, the first field is the inode number and, as we can see, both `fun` and `fun-hard` share the same inode number, which confirms they are the same file.

Hard Links and Symbolic Links

- Symbolic links were created to overcome the two disadvantages of hard links.
  1. Hard links cannot span physical devices.
  2. Hard links cannot reference directories, only files.
- Symbolic links are a special type of file that contains a text pointer to the target file or di-
- Creating symbolic links is similar to creating hard links.

```
[me@linuxbox playground]$ ln -s fun fun-sym
[me@linuxbox playground]$ ln -s ../fun dir1/fun-sym
[me@linuxbox playground]$ ln -s ../fun dir2/fun-sym
```

Hard Links and Symbolic Links

- The first example is pretty straightforward; we simply add the “-s” option to create a symbolic link rather than a hard link.
- But what about the next two? Remember, when we create a symbolic link, we are creating a text description of where the target file is relative to the symbolic link.
- It's easier to see if we look at the ls output shown here:

```
[me@linuxbox playground]$ ls -l dir1
total 4
-rw-r--r-- 4 me   me   1650 2025-01-10 16:33 fun-hard
lrwxrwxrwx 1 me   me      6 2025-01-15 15:17 fun-sym -> ../fun
```

Hard Links and Symbolic Links

The listing for `fun-sym` in `dir1` shows that it is a symbolic link by the leading `l` in the first field and that it points to `../fun`, which is correct. Relative to the location of `fun-sym`, `fun` is in the directory above it. Notice too, that the length of the symbolic link file is 6, the number of characters in the string `../fun` rather than the length of the file to which it is pointing.

When creating symbolic links, we can either use absolute pathnames, as shown here:

```
[me@linuxbox playground]$ ln -s /home/me/playground/fun dir1/fun-sym
```

or relative pathnames, as we did in our earlier example. In most cases, using relative pathnames is more desirable because it allows a directory tree containing symbolic links and their referenced files to be renamed and/or moved without breaking the links.

In addition to regular files, symbolic links can also reference directories.

```
[me@linuxbox playground]$ ln -s dir1 dir1-sym
[me@linuxbox playground]$ ls -l
```

Hard Links and Symbolic Links

```
total 16
drwxrwxr-x 2 me   me   4096 2025-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2025-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2025-01-15 15:17 dir2
-rw-r--r-- 4 me   me   1650 2025-01-10 16:33 fun
-rw-r--r-- 4 me   me   1650 2025-01-10 16:33 fun-hard
lrwxrwxrwx 1 me   me      3 2025-01-15 15:15 fun-sym -> fun
```

I/O Redirection

- In this lesson, we will explore a powerful feature used by command line programs called *input/output redirection*.
- As we have seen, many commands such as ls print their output on the display.
- This does not have to be the case, however.
- By using some special notations we can *redirect* the output of many commands to files, devices, and even to the input of other commands.

I/O Redirection

- **Standard Output** : Most command line programs that display their results do so by sending their results to a facility called *standard output*.
- By default, standard output directs its contents to the display.
- To redirect standard output to a file, the ">" character is used like this:

```
[me@linuxbox me]$ ls > file_list.txt
```

I/O Redirection

- In this example, the ls command is executed and the results are written in a file named file_list.txt. Since the output of ls was redirected to the file, no results appear on the display.
- Each time the command above is repeated, file_list.txt is overwritten from the beginning with the output of the command ls.
- To have the new results *appended* to the file instead, we use ">>" like this:

```
[me@linuxbox me]$ls >> file_list.txt
```

When the results are appended, the new results are added to the end of the file, thus making the file longer each time the command is repeated. If the file does not exist when we attempt to append the redirected output, the file will be created.

I/O Redirection

- **Standard Input** : Many commands can accept input from a facility called *standard input*.
- By default, standard input gets its contents from the keyboard, but like standard output, it can be redirected.
- To redirect standard input from a file instead of the keyboard, the "<" character is used like this:

```
[me@linuxbox me]$ sort < file_list.txt
```

I/O Redirection

- In the example before, we used the sort command to process the contents of file_list.txt.
- The results are output on the display since the standard output was not redirected. We could redirect standard output to another file like this:

```
[me@linuxbox me]$ sort < file_list.txt > sorted_file_list.txt
```

- As we can see, a command can have both its input and output redirected.
- Be aware that the order of the redirection does not matter.
- The only requirement is that the redirection operators (the "<" and ">") must appear after the other options and arguments in the command.

I/O Redirection

- **Pipelines** : The most useful and powerful thing we can do with I/O redirection is to connect multiple commands together to form what are called *pipelines*.
- With pipelines, the standard output of one command is fed into the standard input of another. Here is a very useful example:

```
[me@linuxbox me]$ ls -l | less
```

- In this example, the output of the ls command is fed into less. By using this "| less" trick, we can make any command have scrolling output.
- By connecting commands together, we can accomplish amazing feats. Here are some examples to try:

I/O Redirection

Command	What it does
`ls -lt | head`	Displays the 10 newest files in the current directory.
`du | sort -nr`	Displays a list of directories and how much space they consume, sorted from the largest to the smallest.
`find . -type f -print | wc -l`	Displays the total number of files in the current working directory and all of its subdirectories.

Examples of commands used together with pipelines

I/O Redirection

- **Filters** : One kind of program frequently used in pipelines is called a *filter*.
- Filters take standard input and perform an operation upon it and send the results to standard output.
- In this way, they can be combined to process information in powerful ways.
- Here are some of the common programs that can act as filters:

Program	What it does
sort	Sorts standard input then outputs the sorted result on standard output.
uniq	Given a sorted stream of data from standard input, it removes duplicate lines of data (i.e., it makes sure that every line is unique).
grep	Examines each line of data it receives from standard input and outputs every line that contains a specified pattern of characters.
fmt	Reads text from standard input, then outputs formatted text on standard output.
pr	Takes text input from standard input and splits the data into pages with page breaks, headers and footers in preparation for printing.
head	Outputs the first few lines of its input. Useful for getting the header of a file.
tail	Outputs the last few lines of its input. Useful for things like getting the most recent entries from a log file.
tr	Translates characters. Can be used to perform tasks such as upper/lowercase conversions or changing line termination characters from one type to another (for example, converting DOS text files into Unix style text files).
sed	Stream editor. Can perform more sophisticated text translations than **tr**.
awk	An entire programming language designed for constructing filters. Extremely powerful.

Common filter commands

Performing tasks with pipelines

Printing from the command line. Linux provides a program called **lpr** that accepts standard input and sends it to the printer. It is often used with pipes and filters. Here are a couple of examples:

```
cat poorly_formatted_report.txt | fmt | pr | lpr

cat unsorted_list_with_dupes.txt | sort | uniq | pr | lpr
```

In the first example, we use **cat** to read the file and output it to standard output, which is piped into the standard input of **fmt. fmt** formats the text into neat paragraphs and outputs it to standard output, which is piped into the standard input of **pr. pr** splits the text neatly into pages and outputs it to standard output, which is piped into the standard input of **lpr. lpr** takes its standard input and sends it to the printer.

The second example starts with an unsorted list of data with duplicate entries. First, **cat** sends the list into **sort** which sorts it and feeds it into **uniq** which removes any duplicates. Next **pr** and **lpr** are used to paginate and print the list.

Performing tasks with pipelines

Viewing the contents of tar files Often you will see software distributed as a *gzipped tar file*. This is a traditional Unix style tape archive file (created with **tar**) that has been compressed with **gzip**. You can recognize these files by their traditional file extensions, ".tar.gz" or ".tgz". You can use the following command to view the directory of such a file on a Linux system:

```
tar tzvf name_of_file.tar.gz | less
```

Expansion

- Each time we type a command line and press the enter key, bash performs several processes upon the text before it carries out our command.
- We have seen a couple of cases of how a simple character sequence, for example “*”, can have a lot of meaning to the shell.
- The process that makes this happen is called expansion. With expansion, we type something and it is expanded into something else before the shell acts upon it.
- To demonstrate what we mean by this, let's take a look at the echo command.
- echo is a shell builtin that performs a very simple task. It prints out its text arguments on standard output:

```
[me@linuxbox me]$ echo this is a test
this is a test
```

Expansion

- That's pretty straightforward. Any argument passed to echo gets displayed. Let's try another example:

```
[me@linuxbox me]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

- We recall from wildcards, the “*” character means match any characters in a filename, but what we didn't see in our original discussion was how the shell does that.
- The simple answer is that the shell expands the “*” into something else (in this instance, the names of the files in the current working directory) before the echo command is executed.
- When the enter key is pressed, the shell automatically expands any qualifying characters on the command line before the command is carried out, so the echo command never saw the “*”, only its expanded result.
- Knowing this, we can see that echo behaved as expected.

Pathname Expansion

- The mechanism by which wildcards work is called *pathname expansion*.
- If we try some of the techniques that we employed in our earlier lessons, we will see that they are really expansions.
- Given a home directory that looks like this:

```
[me@linuxbox me]$ ls
Desktop
ls-output.txt
Documents Music
Pictures
Public
Templates
Videos
```

Pathname Expansion

we could carry out the following expansions:

```
[me@linuxbox me]$ echo D*
Desktop Documents
```

and:

```
[me@linuxbox me]$ echo *s
Documents Pictures Templates Videos
```

or even:

```
[me@linuxbox me]$ echo [[:upper:]]*
Desktop Documents Music Pictures Public Templates Videos
```

and looking beyond our home directory:

```
[me@linuxbox me]$ echo /usr/*/share
/usr/kerberos/share /usr/local/share
```

Tilde Expansion

As we recall from our introduction to the **cd** command, the tilde character ("~") has a special meaning. When used at the beginning of a word, it expands into the name of the home directory of the named user, or if no user is named, the home directory of the current user:

```
[me@linuxbox me]$ echo ~
/home/me
```

If user "foo" has an account, then:

```
[me@linuxbox me]$ echo ~foo
/home/foo
```

Arithmetic Expansion

The shell allows arithmetic to be performed by expansion. This allow us to use the shell prompt as a calculator:

```
[me@linuxbox me]$ echo $((2 + 2))
4
```

Arithmetic expansion uses the form:

```
$((expression))
```

where expression is an arithmetic expression consisting of values and arithmetic operators.

Arithmetic Expansions

Arithmetic expansion only supports integers (whole numbers, no decimals), but can perform quite a number of different operations.

Spaces are not significant in arithmetic expressions and expressions may be nested. For example, to multiply five squared by three:

```
[me@linuxbox me]$ echo $(($((5**2)) * 3))
75
```

Single parentheses may be used to group multiple subexpressions. With this technique, we can rewrite the example above and get the same result using a single expansion instead of two:

```
[me@linuxbox me]$ echo $(((5**2) * 3))
75
```

Arithmetic Expansions

Here is an example using the division and remainder operators. Notice the effect of integer division:

```
[me@linuxbox me]$ echo Five divided by two equals $((5/2))
Five divided by two equals 2
[me@linuxbox me]$ echo with $((5%2)) left over.
with 1 left over.
```

Brace Expansion

Perhaps the strangest expansion is called *brace expansion*. With it, we can create multiple text strings from a pattern containing braces. Here's an example:

```
[me@linuxbox me]$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back
```

Patterns to be brace expanded may contain a leading portion called a *preamble* and a trailing portion called a *postscript*. The brace expression itself may contain either a comma-separated list of strings, or a range of integers or single characters. The pattern may not contain embedded whitespace. Here is an example using a range of integers:

```
[me@linuxbox me]$ echo Number_{1..5}
Number_1 Number_2 Number_3 Number_4 Number_5
```

Brace Expansion

A range of letters in reverse order:

```
[me@linuxbox me]$ echo {Z..A}
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

Brace expansions may be nested:

```
[me@linuxbox me]$ echo a{A{1,2},B{3,4}}b
aA1b aA2b aB3b aB4b
```

Brace Expansion

So what is this good for? The most common application is to make lists of files or directories to be created. For example, if we were a photographer and had a large collection of images we wanted to organize into years and months, the first thing we might do is create a series of directories named in numeric “Year-Month” format. This way, the directory names will sort in chronological order. we could type out a complete list of directories, but that's a lot of work and it's error-prone too. Instead, we could do this:

```
[me@linuxbox me]$ mkdir Photos
[me@linuxbox me]$ cd Photos
[me@linuxbox Photos]$ mkdir {2017..2019}-{01..12}
[me@linuxbox Photos]$ ls
2017-01 2017-07 2018-01 2018-07 2019-01 2019-07
2017-02 2017-08 2018-02 2018-08 2019-02 2019-08
2017-03 2017-09 2018-03 2018-09 2019-03 2019-09
2017-04 2017-10 2018-04 2018-10 2019-04 2019-10
2017-05 2017-11 2018-05 2018-11 2019-05 2019-11
2017-06 2017-12 2018-06 2018-12 2019-06 2019-12
```

Parameter Expansion

- It's a feature that is more useful in shell scripts than directly on the command line.
- Many of its capabilities have to do with the system's ability to store small chunks of data and to give each chunk a name.
- Many such chunks, more properly called *variables*, are available for our examination.
- For example, the variable named “USER” contains our user name. To invoke parameter expansion and reveal the contents of USER we would do this:

```
[me@linuxbox me]$ echo $USER
me
```

Parameter Expansion

To see a list of available variables, try this:

```
[me@linuxbox me]$ printenv | less
```

With other types of expansion, if we mistype a pattern, the expansion will not take place and the echo command will simply display the mistyped pattern. With parameter expansion, if we misspell the name of a variable, the expansion will still take place, but will result in an empty string:

```
[me@linuxbox me]$ echo $SUER
[me@linuxbox ~]$
```

Command Substitution

Command substitution allows us to use the output of a command as an expansion:

```
[me@linuxbox me]$ echo $(ls)
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

A clever one goes something like this:

```
[me@linuxbox me]$ ls -l $(which cp)
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Here we passed the results of **`which cp`** as an argument to the **`ls`** command, thereby getting the listing of of the **`cp`** program without having to know its full pathname. We are not limited to just simple commands. Entire pipelines can be used (only partial output shown):

Command Substitution

```
[me@linuxbox me]$ file $(ls /usr/bin/* | grep bin/zip)
/usr/bin/bunzip2:
/usr/bin/zip:       ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripped
/usr/bin/zipcloak: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripped
/usr/bin/zipgrep:  POSIX shell script text executable
/usr/bin/zipinfo:  ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripped
/usr/bin/zipnote:  ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripped
/usr/bin/zipsplit: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripped
```

In this example, the results of the pipeline became the argument list of the file command.

Quoting

Now that we've seen how many ways the shell can perform expansions, it's time to learn how we can control it. Take for example:

```
[me@linuxbox me]$ echo this is a     test
this is a test
```

or:

```
[me@linuxbox me]$ [me@linuxbox ~]$ echo The total is $100.00
The total is 00.00
```

In the first example, word-splitting by the shell removed extra whitespace from the echo command's list of arguments. In the second example, parameter expansion substituted an empty string for the value of "$1" because it was an undefined variable. The shell provides a mechanism called *quoting* to selectively suppress unwanted expansions.

Double Quotes

- The first type of quoting we will look at is double quotes.
- If we place text inside double quotes, all the special characters used by the shell lose their special meaning and are treated as ordinary characters. The exceptions are “$”, “\” (backslash), and “ ` ” (back- quote).
- This means that word-splitting, pathname expansion, tilde expansion, and brace expansion are suppressed, but parameter expansion, arithmetic expansion, and command substitution are still carried out.
- Using double quotes, we can cope with filenames containing embedded spaces.
- Imagine we were the unfortunate victim of a file called two words.txt. If we tried to use this on the command line, word-splitting would cause this to be treated as two separate arguments rather than the desired single argument:

Double Quotes

```
[me@linuxbox me]$ ls -l two words.txt
ls: cannot access two: No such file or directory
ls: cannot access words.txt: No such file or directory
```

By using double quotes, we can stop the word-splitting and get the desired result; further, we can even repair the damage:

```
[me@linuxbox me]$ ls -l "two words.txt"
-rw-rw-r-- 1 me me 18 2020-02-20 13:03 two words.txt
[me@linuxbox me]$ mv "two words.txt" two_words.txt
```

There! Now we don't have to keep typing those pesky double quotes. Remember, parameter expansion, arithmetic expansion, and command substitution still take place within double quotes:

Double Quotes

There! Now we don't have to keep typing those pesky double quotes. Remember, parameter expansion, arithmetic expansion, and command substitution still take place within double quotes:

```
[me@linuxbox me]$ echo "$USER $((2+2)) $(cal)"
me 4
February 2020
Su Mo Tu We Th Fr Sa
                    1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29
```

We should take a moment to look at the effect of double quotes on command substitution. First let's look a little deeper at how word splitting works. In our earlier example, we saw how word-splitting appears to remove extra spaces in our text:

Double Quotes

We should take a moment to look at the effect of double quotes on command substitution. First let's look a little deeper at how word splitting works. In our earlier example, we saw how word-splitting appears to remove extra spaces in our text:

```
[me@linuxbox me]$ echo this is a     test
this is a test
```

By default, word-splitting looks for the presence of spaces, tabs, and newlines (linefeed characters) and treats them as delimiters between words. This means that unquoted spaces, tabs, and newlines are not considered to be part of the text. They only serve as separators. Since they separate the words into different arguments, our example command line contains a command followed by four distinct arguments. If we add double quotes:

```
[me@linuxbox me]$ echo "this is a     test"
this is a     test
```

Double Quotes

word-splitting is suppressed and the embedded spaces are not treated as delimiters, rather they become part of the argument. Once the double quotes are added, our command line contains a command followed by a single argument. The fact that newlines are considered delimiters by the word-splitting mechanism causes an interesting, albeit subtle, effect on command substitution. Consider the following:

```
[me@linuxbox me]$ echo $(cal)
February 2020 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
[me@linuxbox me]$  echo "$(cal)"
February 2020
Su Mo Tu We Th Fr Sa
                 1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29
```

In the first instance, the unquoted command substitution resulted in a command line containing thirty-eight arguments.

In the second, a command line with one argument that includes the embedded spaces and newlines.

Single Quotes

When we need to suppress all expansions, we use single quotes. Here is a comparison of unquoted, double quotes, and single quotes:

```
[me@linuxbox me]$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 me
[me@linuxbox me]$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/*.txt {a,b} foo 4 me
[me@linuxbox me]$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

As we can see, with each succeeding level of quoting, more and more of the expansions are suppressed.

Escaping Characters

Sometimes we only want to quote a single character. To do this, we can precede a character with a backslash, which in this context is called the *escape character*. Often this is done inside double quotes to selectively prevent an expansion:

```
[me@linuxbox me]$ echo "The balance for user $USER is: \$5.00"
The balance for user me is: $5.00
```

It is also common to use escaping to eliminate the special meaning of a character in a filename. For example, it is possible to use characters in filenames that normally have special meaning to the shell. These would include “$”, “!”, “&”, “ “, and others. To include a special character in a filename we can to this:

```
[me@linuxbox me]$ mv bad\&filename good_filename
```

Escape Character	Name	Possible Uses
\n	newline	Adding blank lines to text
\t	tab	Inserting horizontal tabs to text
\a	alert	Makes our terminal beep
\\	backslash	Inserts a backslash
\f	formfeed	Sending this to our printer ejects the page

Permissions

- The Unix-like operating systems, such as Linux differ from other computing systems in that they are not only *multitasking* but also *multi-user*.
- What exactly does this mean? It means that more than one user can be operating the computer at the same time.
- While a desktop or laptop computer only has one keyboard and monitor, it can still be used by more than one user.
- For example, if the computer is attached to a network, or the Internet, remote users can log in via ssh (secure shell) and operate the computer.
- In fact, remote users can execute graphical applications and have the output displayed on a remote computer.

Permissions

- The multi-user capability of Unix-like systems is a feature that is deeply ingrained into the design of the operating system.
- If we remember the environment in which Unix was created, this makes perfect sense.
- Years ago before computers were "personal," they were large, expensive, and centralized.
- A typical university computer system consisted of a large mainframe computer located in some building on campus and *terminals* were located throughout the campus, each connected to the large central computer.
- The computer would support many users at the same time.
- In order to make this practical, a method had to be devised to protect the users from each other.
- After all, we wouldn't want the actions of one user to crash the computer, nor would we allow one user to interfere with the files belonging to another user.

Commands involved in Permissions

- chmod - modify file access rights
- su - temporarily become the superuser
- sudo - temporarily become the superuser
- chown - change file ownership
- chgrp - change a file's group ownership

File Permissions

- On a Linux system, each file and directory is assigned access rights for the owner of the file, the members of a group of related users, and everybody else.
- Rights can be assigned to read a file, to write a file, and to execute a file (i.e., run the file as a program).
- To see the permission settings for a file, we can use the ls command.
- As an example, we will look at the bash program which is located in the /bin directory:

File Permissions

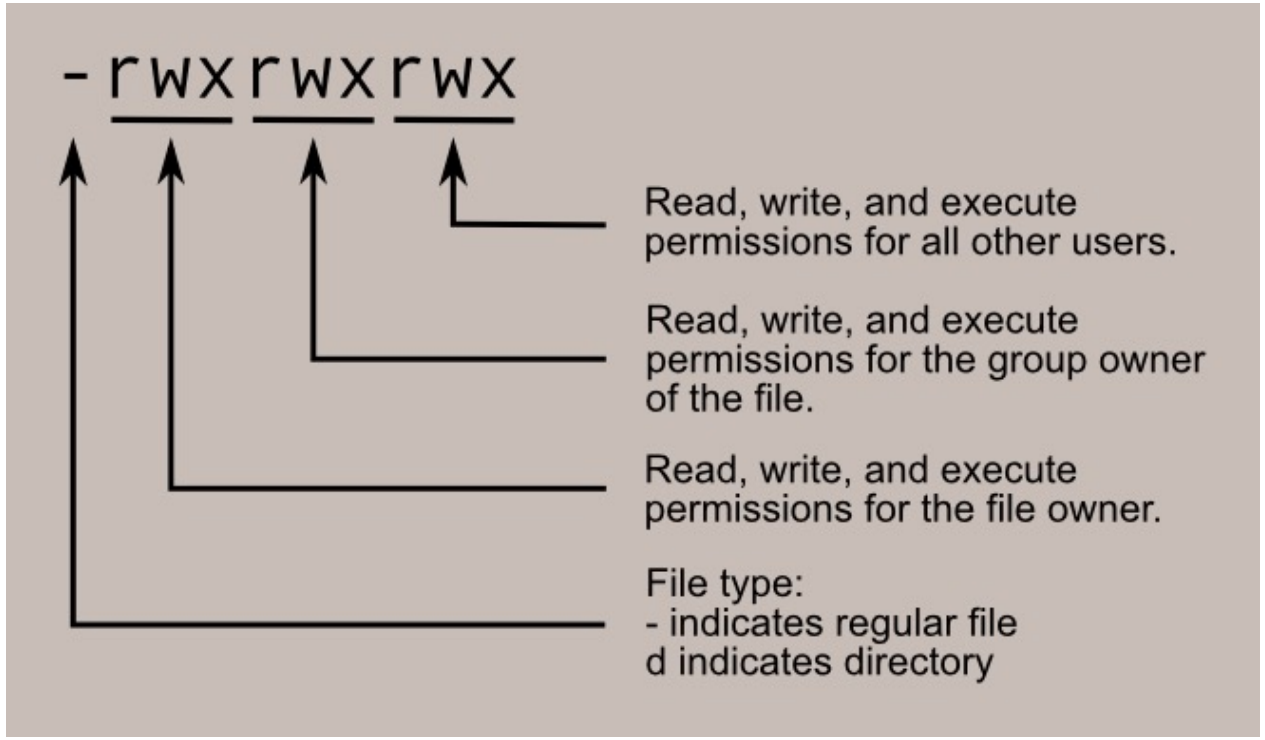
```
[me@linuxbox me]$ ls -l /bin/bash
-rwxr-xr-x 1 root root 1113504 Jun  6  2019 /bin/bash
```

Here we can see:

- The file "/bin/bash" is owned by user "root"
- The superuser has the right to read, write, and execute this file
- The file is owned by the group "root"
- Members of the group "root" can also read and execute this file
- Everybody else can read and execute this file

File Permissions

- Lets see how the first portion of the listing is interpreted.
- It consists of a character indicating the file type, followed by three sets of three characters that convey the reading, writing and execution permission for the owner, group, and everybody else.



chmod

- The chmod command is used to change the permissions of a file or directory.
- To use it, we specify the desired permission settings and the file or files that we wish to modify.
- There are two ways to specify the permissions.
- In this lesson we will focus on one of these, called the *octal notation* method.
- It is easy to think of the permission settings as a series of bits (which is how the computer thinks about them).

chmod

- Now, if we represent each of the three sets of permissions (owner, group, and other) as a single digit, we have a pretty convenient way of expressing the possible permissions settings.
- For example, if we wanted to set some_file to have read and write permission for the owner, but wanted to keep the file private from others, we would:

```
rwx rwx rwx = 111 111 111
rw- rw- rw- = 110 110 110
rwx --- --- = 111 000 000

and so on...

rwx = 111 in binary = 7
rw- = 110 in binary = 6
r-x = 101 in binary = 5
r-- = 100 in binary = 4
```

```
[me@linuxbox me]$ chmod 600 some_file
```

chmod

Value	Meaning
777	**(rwxrwxrwx)** No restrictions on permissions. Anybody may do anything. Generally not a desirable setting.
755	**(rwxr-xr-x)** The file's owner may read, write, and execute the file. All others may read and execute the file. This setting is common for programs that are used by all users.
700	**(rwx------)** The file's owner may read, write, and execute the file. Nobody else has any rights. This setting is useful for programs that only the owner may use and must be kept private from others.
666	**(rw-rw-rw-)** All users may read and write the file.
644	**(rw-r--r--)** The owner may read and write a file, while all others may only read the file. A common setting for data files that everybody may read, but only the owner may change.
600	**(rw-------)** The owner may read and write a file. All others have no rights. A common setting for data files that the owner wants to keep private.

Directory Permissions

- The chmod command can also be used to control the access permissions for directories.
- Again, we can use the octal notation to set permissions, but the meaning of the r, w, and x attributes is different:
  - **r** - Allows the contents of the directory to be listed if the x attribute is also set.
  - **w** - Allows files within the directory to be created, deleted, or renamed if the x attribute is also set.
  - **x** - Allows a directory to be entered (i.e. cd dir).
- Here are some useful settings for directories:

Directory Permissions

Value	Meaning
777	**(rwxrwxrwx)** No restrictions on permissions. Anybody may list files, create new files in the directory and delete files in the directory. Generally not a good setting.
755	**(rwxr-xr-x)** The directory owner has full access. All others may list the directory, but cannot create files nor delete them. This setting is common for directories that you wish to share with other users.
700	**(rwx------)** The directory owner has full access. Nobody else has any rights. This setting is useful for directories that only the owner may use and must be kept private from others.

Becoming the Superuser for a Short While

- It is often necessary to become the superuser to perform important system administration tasks, but as we know, we should not stay logged in as the superuser.
- In most distributions, there is a program that can give you temporary access to the superuser's privileges.
- This program is called su (short for substitute user) and can be used in those cases when you need to be the superuser for a small number of tasks.
- To become the superuser, simply type the su command. You will be prompted for the superuser's password:

```
[me@linuxbox me]$ su
Password:
[root@linuxbox me]#
```

Becoming the Superuser for a Short While

- After executing the su command, we have a new shell session as the superuser.
- To exit the superuser session, type **exit** and we will return to your previous session.
- In most modern distributions, an alternate method is used.
- Rather than using su, these systems employ the sudo command instead.
- With sudo, one or more users are granted superuser privileges on an as needed basis.
- To execute a command as the superuser, the desired command is simply preceded with the sudo command.
- After the command is entered, the user is prompted for the their **own password rather than the superuser's**:

Becoming the Superuser for a Short While

```
[me@linuxbox me]$ sudo some_command
Password for me:
[me@linuxbox me]$
```

In fact, modern distributions don't even set the root account password thus making it impossible to log in as the root user. A root shell is still possible with **sudo** by using the "-i" option:

```
[me@linuxbox me]$ sudo -i
Password for me:
root@linuxbox:~#
```

Changing File Ownership

- We can change the owner of a file by using the **chown** command. Here's an example: Suppose we wanted to change the owner of some_file from "me" to "you". We could:

```
[me@linuxbox me]$ sudo chown you some_file
```

- Notice that in order to change the owner of a file, we must have superuser privileges. To do this, our example employed the **sudo** command to execute **chown**.
- chown works the same way on directories as it does on files.

Changing Group Ownership

- The group ownership of a file or directory may be changed with **chgrp**. This command is used like this:

```
[me@linuxbox me]$ chgrp new_group some_file
```

- In the example above, we changed the group ownership of some_file from its previous group to "new_group".
- We must be the owner of the file or directory to perform a chgrp.
- To add a new group use **sudo groupadd new_group_name**

Job Control

- We have seen some of the implications of Linux being a multi-user operating.
- Now we will examine the multitasking nature of Linux, and how it is controlled with the command line interface.
- As with any multitasking operating system, Linux executes multiple, simultaneous processes.
- Well, they appear simultaneous, anyway.
- Actually, a single processor core can only execute one process at a time but the Linux kernel manages to give each process its turn at the processor and each appears to be running at the same time.

Job Control

- There are several commands that are used to control processes. They are:
- ps - list the processes running on the system
- kill - send a signal to one or more processes (usually to "kill" a process)
- jobs - an alternate way of listing your own processes
- bg - put a process in the background
- fg - put a process in the foreground

A Practical Example

While it may seem that this subject is rather obscure, it can be very practical for the average user who mostly works with the graphical user interface. Though it might not be apparent, most (if not all) graphical programs can be launched from the command line. Here's an example: there is a small program supplied with the X Window system called **xload** which displays a graph representing system load. We can execute this program by typing the following:

```
[me@linuxbox me]$ xload
```

Notice that the small **xload** window appears and begins to display the system load graph. On systems where **xload** is not available, try **gedit** instead. Notice also that our prompt did not reappear after the program launched. The shell is waiting for the program to finish before control returns. If we close the **xload** window, the **xload** program terminates and the prompt returns.

Putting a Program into the Background

- ow, in order to make life a little easier, we are going to launch the xload program again, but this time we will put it in the background so that the prompt will return.
- To do this, we execute xload like this:

```
[me@linuxbox me]$ xload &
[1] 1223
[me@linuxbox me]$
```

- In this case, the prompt returned because the process was put in the background.

Putting a Program into the Background

- Now imagine that we forgot to use the "&" symbol to put the program into the background.
- There is still hope.
- We can type Ctrl-z and the process will be suspended.
- We can verify this by seeing that the program's window is frozen. The process still exists, but is idle.
- To resume the process in the background, type the bg command (short for background).
- Here is an example:

```
[me@linuxbox me]$ xload
[2]+ Stopped xload

[me@linuxbox me]$ bg
[2]+ xload &
```

Listing Running Processes

Now that we have a process in the background, it would be helpful to display a list of the processes we have launched. To do this, we can use either the **jobs** command or the more powerful **ps** command.

```
[me@linuxbox me]$ jobs
[1]+ Running xload&

[me@linuxbox me]$ ps
PID  TTY   TIME     CMD
1211 pts/4 00:00:00 bash
1246 pts/4 00:00:00 xload
1247 pts/4 00:00:00 ps

[me@linuxbox me]$
```

Killing a Process

- Suppose that we have a program that becomes unresponsive; how do we get rid of it? We use the kill command, of course.
- Let's try this out on xload.
- First, we need to identify the process we want to kill.
- We can use either jobs or ps, to do this.
- If we use jobs we will get back a job number.
- With ps, we are given a *process id* (PID).
- We will do it both ways:

Killing a Process

```
[me@linuxbox me]$ xload &
[1] 1292

[me@linuxbox me]$ jobs
[1]+ Running xload&

[me@linuxbox me]$ kill %1

[me@linuxbox me]$ xload &
[2] 1293
[1] Terminated xload

[me@linuxbox me]$ ps
PID  TTY   TIME     CMD
1280 pts/5 00:00:00 bash
1293 pts/5 00:00:00 xload
1294 pts/5 00:00:00 ps

[me@linuxbox me]$ kill 1293
[2]+ Terminated xload

[me@linuxbox me]$
```

A Little More About kill

While the **`kill`** command is used to "kill" processes, its real purpose is to send *signals* to processes. Most of the time the signal is intended to tell the process to go away, but there is more to it than that. Programs (if they are properly written) listen for signals from the operating system and respond to them, most often to allow some graceful method of terminating. For example, a text editor might listen for any signal that indicates that the user is logging off, or that the computer is shutting down. When it receives this signal, it could save the work in progress before it exits. The **`kill`** command can send a variety of signals to processes. Typing:

```
kill -l
```

will print a list of the signals it supports. Many are rather obscure, but several are handy to know:

A Little More About kill

Signal #	Name	Description
1	**SIGHUP**	Hang up signal. Programs can listen for this signal and act upon it. This signal is sent to processes running in a terminal when you close the terminal.
2	**SIGINT**	Interrupt signal. This signal is given to processes to interrupt them. Programs can process this signal and act upon it. We can also issue this signal directly by typing `Ctrl-c` in the terminal window where the program is running.
15	**SIGTERM**	Termination signal. This signal is given to processes to terminate them. Again, programs can process this signal and act upon it. This is the default signal sent by the **`kill`** command if no signal is specified.
9	**SIGKILL**	Kill signal. This signal causes the immediate termination of the process by the Linux kernel. Programs cannot listen for this signal.

A Little More About kill

- Now let's suppose that we have a program that is hopelessly hung and we want to get rid of it. Here's what we do:
- Use the ps command to get the process id (PID) of the process we want to terminate.
- Issue a kill command for that PID.
- If the process refuses to terminate (i.e., it is ignoring the signal), send increasingly harsh signals until it does terminate.

A Little More About kill

```
[me@linuxbox me]$ ps x | grep bad_program
PID  TTY   STAT TIME COMMAND
2931 pts/5 SN   0:00 bad_program

[me@linuxbox me]$ kill -SIGTERM 2931

[me@linuxbox me]$ kill -SIGKILL 2931
```

- In the example above we used the ps command with the x option to list all of our processes (even those not launched from the current terminal).
- In addition, we piped the output of the ps command into grep to list only list the program we are interested in.
- Next, we used kill to issue a SIGTERM signal to the troublesome program.

A Little More About kill

- In actual practice, it is more common to do it in the following way since the default signal sent by kill is SIGTERM and kill can also use the signal number instead of the signal name:

```
[me@linuxbox me]$ kill 2931
```

Then, if the process does not terminate, force it with the SIGKILL signal:

```
[me@linuxbox me]$ kill -9 2931
```